The Remote Socket Architecture:
A proxy based solution for TCP over wireless

vorgelegt von
Diplom-Informatiker
Morten Schläger

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

Promotionsausschuss:

Vorsitzender : Prof. Dr.-Ing. Kurt Geihs
Berichter : Prof. Dr.-Ing. Adam Wolisz
Berichter : Prof. Dr.-Ing. Lars Wolf

Tag der wissenschaftlichen Aussprache: 24.05.2004

Berlin 2004

D 83

For the ones I love

Acknowledgements

You need more time;
and you probably always will.

A couple of years ago on a holiday trip I was traveling through Scotland by bike with a few friends when I suggested to take a footpath as a shortcut from Loch Ness to Laggan. My estimation was a four hour trip through the mountains. We started in the morning, and when the daylight was nearly gone we had to pitch our tents. It took us another second (rainy) day to finally reach civilization late at night.

This, however, was nothing compared to my PhD Thesis. While traveling the footpath I was always certain that we would reach our destination. Regarding my thesis this was only true on account of the support of my wife, dear friends, and colleagues.

Working at TKN was a pleasureful experience. I always enjoyed the warm working atmosphere, the expertise of my colleagues, the excellent working conditions, and the liberty to organize my work according to my own ideas. During my nine years at TKN I developed a deep friendship with my colleagues Jean-Pierre Ebert, Berthold Rathke, and Andreas Willig. We walked a long way together, starting at the very beginning of TKN. Working together, as well as going out after work was always fun. Jean-Pierre deserves special thanks for organizing many TKN parties and for sharing my fate of finishing my PhD thesis over Christmas and New Years Eve.

From the numerous excellent students I met, Stefan Bodenstein and Tobias Poschwatta deserve special acknowledgment. Stefan implemented the first ReSoA prototype for a fast proof of my concepts and ideas, while Tobias developed the current ReSoA implementation and carried out countless measurements. They both contributed to my work in many fruitful discussions.

I would also like to thank Heike Klemz and Adrian Schrepfer for proofreading this thesis. Especially Adrian put much effort into improving the readability of this thesis.

I am indebted to Professor Lars Wolf, TU Braunschweig for being my second advisor. I especially enjoyed my one day visit at his group.

I am grateful for having had the opportunity to work for and with Professor Adam Wolisz over the past nine years. We had many in-depth technical discussions which were especially valuable for my thesis. His tremendous motivation, brilliant ideas and critical assessment were a key factor for the success of my thesis.

Finally I would like to thank my wife Caroline. Not only did she have to travel the Scottish footpath with me. She also accompanied me with my Studienarbeit, diploma thesis and now my PhD thesis. Without her support, understanding and patience I would not have been able to master any of these tasks.

Zusammenfassung

Internet-Zugang unabhängig von Ort und Zeit ist, insbesondere auf Grund von Funknetzen, heute fast schon Realität. Andererseits haben gerade Funknetze einige Eigenschaften, wie beispielsweise ein hohe Fehlerrate, mit denen die klassischen Internet-Protokolle, vor allem TCP, nicht effizient umgehen können. Daher wird derzeit diskutiert, ob Proxy-basierte Ansätze, die mit der Ende-zu-Ende Philosophie des Internets brechen, nicht eine Alternative zu dem klassischen Internet-Zugang sind, bei dem jeder Rechner einen TCP/IP-Protokollstack und eine IP-Adresse besitzen muß.

Die *'Remote Socket Architecture (ReSoA)'*, die im Rahmen dieser Arbeit entwickelt und spezifiziert wird, realisiert einen Proxy-basierten Ansatz. ReSoA wurde durch die Beobachtung, das für Applikationen nicht das verwendete Protokoll entscheidend ist, sondern die Schnittstelle zwischen Applikation und Protokoll-Stack sowie der erwartete Dienst, motiviert und realisiert eine verteilte Implementierung der weit verbreiteten BSD-Socket-Schnittstelle.

Für den Entwurf von ReSoA wurde die Semantik jeder einzelnen Socket-Funktion analysiert und in ReSoA nachgebildet. Um zu überprüfen, ob ReSoA semantisch äquivalent zur BSD-Socket-Schnittstelle ist, wurden zum einen Tests mit existierenden Applikationen durchgeführt und zum anderen wurde eine SDL-Spezifikation entwickelt, die es erlaubt, interessante Fälle unabhängig von einer Implementierung zu untersuchen. Alle Untersuchungen haben ergeben, daß die Existenz von ReSoA für Applikationen transparent ist.

Die Arbeit wird durch eine systematische Leistungsbewertung abgerundet, wobei sowohl Messungen als auch Simulationen zum Einsatz kommen. Gegenstand der Leistungsbewertung ist ein Vergleich zwischen TCP und ReSoA unter unterschiedlichen Netzwerkeigenschaften wie Bitrate, Fehlermodell und Verzögerung mit dem Ziel, Faustregeln zu finden, unter welchen Umständen der Einsatz eines Proxies besonders lohnend ist. Die Messungen in einer drahtlosen LAN-Umgebung (IEEE 802.11b) haben gezeigt, daß ReSoA aufgrund seiner auf den Halbduplex-Kanal angepaßten Protokolle selbst dann einen deutlich besseren Durchsatz bietet als TCP, wenn das drahtlose Endgerät an einer guten Position in der Funkzelle steht (Paketverluste treten nicht auf). Dieser Effekt wird an schlechten Positionen noch gesteigert. Die Simulationen haben gezeigt, daß ReSoA insbesondere dann von Vorteil ist, wenn die Verzögerung im Zugangsnetzwerk groß ist, und wenn Verluste im Internet auftreten.

Abstract

As wireless technology evolves, Internet access at any time, at any place today is close to becoming reality. Classic Internet protocols like TCP, however, cannot efficiently handle certain properties of wireless networks like high bit error rates. This is the cause for an ongoing discussion whether or not proxy based approaches that break the end-to-end philosophy of the Internet are an alternative to the classic Internet access, where each computer necessitates a TCP/IP protocol stack and an IP address.

The *Remote Socket Architecture (ReSoA)* that is developed and specified in the scope of this thesis, pursues a proxy based approach. ReSoA was motivated by the observation that it is not the underlying protocol, but rather the interface between application and protocol stack, and the expected service that are important from the application point of view. ReSoA is a distributed realization of the widely deployed BSD socket interface.

Each BSD socket function was analyzed and emulated for the design of ReSoA. To verify that ReSoA is semantically equivalent to the BSD socket interface, tests with existing applications were performed and an SDL specification was developed. The SDL specification permits the investigation of ReSoA independent of an implementation. All investigations confirmed that ReSoA is transparent to applications.

A systematic performance evaluation, incorporating measurements as well as simulations, is performed as part of this thesis. The goal of this performance evaluation is the comparison of TCP to ReSoA with different network properties like bitrate, error model, and delay, in order to determine rules of thumb that can help to decide under which circumstances the deployment of a proxy is especially useful. The measurements in a wireless LAN environment (IEEE 802.11b) have shown that ReSoA achieves a performance gain over TCP even for good positions due to its protocols especially tailored for half-duplex channels (no packets are lost). This gain increases for bad positions. The simulations showed that ReSoA is especially beneficial for access networks with long delays, and if packet losses occur in the Internet.

Contents

List of Tables

List of Figures

Part I.

Basics

Chapter 1.

Introduction

1.1. Motivation

One of the design goals of the Internet protocol suite, and especially its transport protocols, has been the independence of the underlying technology. Looking back at the history of the Internet it has to be admitted that this design goal has been reached. The Internet protocols, which were designed over 25 years ago, still provide their communication services on top of technologies that had not even been invented when they were designed. Although several facets of certain protocols, especially TCP, required improvement, the core architecture of the Internet is still the same. In [45] the development of the Internet is compared with constantly renewing individual streets and buildings of a city rather than razing the city and rebuilding it.

Besides the independence of communication technology, the Internet has also shown stability against changing usage patterns. Starting as a military and scientific network mainly used by a small number of scientists for applications like file transfer and electronic mail, it has evolved to a major economical factor during the last decade. Today applications that did not exist when the Internet protocols were designed, operate on top of the Internet protocol suite. The interest in the Internet has been growing exponentially due to the exploding popularity of the World Wide Web and the availability of communication equipment. This equipment not only enables private users and small companies to access the Internet, but especially allows Internet access everywhere using wireless technologies.

The key to the success story of the Internet is a single network layer protocol, the IP protocol with its hardware independent addressing format making no assumptions about the underlying communication technology or service. IP simply provides a connectionless datagram service. Packets are delayed, lost, or delivered out of order. On top of the network layer, different transport protocols are responsible for providing the service required by different kinds of applications (see Section 2.1 for details). Until today two transport protocols were sufficient to support a variety of applications. TCP provides a reliable stream oriented service and UDP only adds application level demultiplexing to the service offered by IP.

However, despite the flexibility the Internet has proven over the last decades, new extensions (applications or communication technologies) always challenge its design principles. This is especially the fact regarding the increased usage of wireless technologies during the past years. Wireless technologies introduce challenges in two respects. First, if only the cable is replaced by radio communication (without allowing any mobility), the Internet protocols will be confronted with high and time varying error rates, and often with a reduced throughput compared to tethered networks (see Section 3.2 for details). Second, wireless technologies allow the user to move while communicating whereas the Internet Protocol (IP) was designed for fixed hosts.

Hence the introduction of wireless technologies raises two questions: how can mobility be supported and how will the transport protocols deal with the new communication characteristics[1]? This thesis deals with the second question. A discussion of the first question can be found in [71, 70].

The different characteristics of wireless communication technologies should not be a problem because

[1] There are other important issues that are not further discussed, like security and power efficiency.

the Internet makes no assumptions about the communication technology. TCP however is designed to avoid congestion. Congestion control is essential for the stability of the Internet. Packet losses are used as an indication of an overload situation. Upon a loss TCP has to reduce its sending rate. However in the case of wireless technologies, packet losses are rather caused by transmission errors than by congestion. Therefore if a packet is lost due to a transmission error, a fast retransmission without reducing the sending rate would be the appropriate reaction. Different researchers have shown that TCP's behavior leads to poor performance if the communication path includes an error prone channel. Best to our knowledge papers by DeSimone et al. [55] and by Caćeres et al. [43] are the first papers which address these issues, both dating back to 1993 (see Section 3.3 on page 25 for a discussion of these papers).

In [152] we present systematic measurements of TCP with early wireless LAN products (non-compliant with IEEE 802.11) at different physical positions. These measurements show that TCP has severe performance problems over wireless networks. The positions can be classified into there groups. Positions where communication is nearly impossible, positions with a high throughput variance and positions with good throughput and low variance. Especially we observed that bad positions exist even close to the base-station. This means that the performance problems cannot be solved by just installing additional base-stations.

To remedy TCP's problem over wireless technologies many different solutions have been suggested (see Section 3 for a discussion). Provocatively speaking, these solutions divide the research community into two groups. The conservative group suggests solving the problem by carefully adding new functionality to TCP (e.g. Eifel Algorithm[115]) in addition to using a reliable link layer protocol; the reformers advocate the deployment of performance enhancing proxies.

At a first glance the conservative approach has the advantage that it is inline with the Internet architecture but requires a flow differentiation on the link layer. With the deployment of a fully reliable link layer protocol one of the main features of the Internet is lost. The Internet is able to support all kinds of applications, since the IP protocol provides the smallest common denominator regarding its service. Delay sensitive applications (like voice over IP) are supported by the Internet, since IP does not try to retransmit lost packets (retransmissions translate losses into delay). A fully reliable link layer protocol could obviously be harmful for these applications. Thus, in order to support different flows the link layer has to support different protocols and has to be able to assign incoming packets to the appropriate protocol. This assignment becomes complicated (or even impossible if IPSec is used) if the differentiation goes beyond recognizing TCP and UDP packets. A fine grained differentiation requires the analysis of transport protocol and application level header fields.

The proxy based approaches generally require modifications of the Internet architecture. Instead of having all functionality implemented in the end systems as recommended by Saltzer et al. in [149], Performance Enhancing Proxies require the implementation of functionality in the network. This is especially a problem if the Performance Enhancing Proxy (PEP) violates the fate-sharing principle of the Internet. On the other hand a PEP based approach is very promising since it allows to divide networks with totally different characteristics into two halves, and to use a tailored protocol in the access network.

Optimized protocols for a certain part of a communication path are especially appealing considering the increasing heterogeneity of the Internet. Since heterogeneity is increased in different directions regarding applications, end systems and communication technologies, it has become more difficult to find an efficient single solution. The optimization goals depend on the field of application. In the case of light weight end systems an energy efficient solution might be more appropriate.

In [152] we advocate the usage of ReSoA, a performance enhancing proxy based on a split implementation of the BSD socket interface (see Section 5 for details). Measurements indicate that ReSoA can even improve performance in situations where other approaches fail. Because the first investigations of ReSoA were promising and the Internet community admits (see RFC 3135[34]) that a PEP should be used if it provides significant advantages over an end-to-end solution, we decided to refine our solution and to investigate for which network properties or traffic patterns the usage of ReSoA is beneficial.

1.2. Contributions/Goal

The usage of a performance enhancing proxy for Internet access raises two major questions. The first question concerns the functional equivalence of the legacy approach and the PEP based approach. Is the service provided by the PEP functionally equivalent to the service provided by the classical Internet protocol stack? This not only means that any existing Internet application should be able to run on top of a PEP without recognizing the difference. It also means that the preconditions that were made when the application was implemented are still maintained. For instance, when a data request is completed does this mean that the data was successfully transmitted or only that it was accepted by the transport protocol for delivery. Thus, the question is whether it is possible to decouple two networks with incompatible (very different) characteristics by a PEP without changing the communication service expected by the service users (i.e. the application).

The second question deals with performance benefits, where performance can have different meanings like throughput, energy efficiency, or protocol overhead. The question is to which extent does ReSoA improve the performance as a function of different network parameters like error rate or Round Trip Time (RTT) ?

To investigate these questions, we designed a performance enhancing proxy based on the idea of an exported interface. Our proxy provides a distributed realization of the widely deployed Berkeley socket API, called ReSoA. The design of a split socket implementation maintaining the semantics of the socket interface made it necessary to thoroughly analyze the semantics of the socket interface.

In order to show that our ReSoA approach is functionally equivalent to a local (classic) socket implementation we used two different approaches. First we performed a prototype implementation of ReSoA for Linux and ran tests using existing applications which make extensive use of the socket interface (partially in an unexpected manner), like netscape and ftp, and we used test scenarios written by ourselves, which test certain aspects of the socket interface (e.g. operation with linger option set). Second we specified ReSoA using SDL and compared our system with an SDL specification of the BSD socket interface derived from the description of the socket interface. However, a formal proof of the semantic equivalence between the two systems is not given, because the focus of this thesis is on performance issues.

To investigate the performance issue we used the implementation to perform measurements in an IEEE 802.11b wireless LAN testbed as well as simulations. The purpose of the measurements is twofold. First, we wanted to investigate the performance of ReSoA in a real environment. Second, we used measurement results to validate our simulation model.

In order to perform simulations we developed a simulation model for ReSoA using the Network Simulator (ns-2 [123, 38]). The simulation model allowed us to compare ReSoA with an end-to-end approach using a large parameter space. The goal of the simulation was to find rules of thumb for which networks (or network properties) a proxy like ReSoA should be used. The focus of the performance comparison is on throughput seen by the application.

1.3. Structure

This thesis deals with wireless Internet access and advocates the usage of a specific kind of proxy namely ReSoA. The focus of this thesis is on TCP over wireless access. We do not consider the UDP case although a similar proxy could be used for the User Datagram Protocol (UDP) . This thesis is divided into three parts: Basics, 'The Remote Socket Architecture' and 'Performance Evaluation'.

The 'Basic' part provides fundamentals important to the understanding of the motivation of our approach as well as its design. The next chapter describes the Internet architecture as well as its Transmission Control Protocol, because both are needed to understand the difficulties with wireless TCP access. In Section 3 the problems of wireless Internet access are discussed with a focus on TCP. In Section 3.1

we introduce the network architecture which is assumed throughout this work. Section 3.2 discusses the problems which TCP faces over wireless networks. In Section 3.3 we present an overview of current approaches aiming at TCP's problems over wireless networks. In Section 4 we provide an overview of selected network programming issues. In Section 4.1 we introduce the Berkeley socket interface and in Section 4.2 we present a short introduction to Remote Procedure Calls.

The second part of this thesis deals with the design and specification of ReSoA. In Chapter 5 we introduce the design of our Remote Socket Architecture. Here we describe how the socket interface is split and discuss the implementation of every socket call in order to demonstrate that the semantics are maintained. In Section 6 the specification of ReSoA is presented. Chapter 7 deals with the semantic question. In Section 7.1 we describe the semantic question in detail. In Section 7.2 we describe the used methods. In Section 7.3 we use our implementation of ReSoA in order to show that ReSoA provides the same syntax and semantics as the BSD socket interface. Finally we compare the behavior of an SDL specification of the socket interface with the behavior of ReSoA, looking at test cases which we identified as crucial during the design of ReSoA in Section 7.4.

In the third part of this thesis we investigate the performance of ReSoA and provide a comparison with end-to-end TCP as reference. In Section 8 we describe our methods and define the set-up and configurations that are investigated. In Section 9 we show measurement results of ReSoA in a wireless LAN context. In Chapter 10 we present the results of a systematic performance evaluation based on simulations. In Chapter 11 we summarize our results.

Chapter 2.

The Internet Architecture and Protocols

This thesis advocates the usage of a new architecture for wireless Internet access, especially if the Transmission Control Protocol is used. In order to understand the design principles behind our architecture, as well as the problems TCP faces with wireless links, an understanding of the Internet architecture and protocols, especially the Transmission Control Protocol, is required. This chapter provides the required background. In Section 2.1 we give a description of the Internet architecture and Section 2.2 describes the Transmission Control Protocol. At the end of this chapter we discuss the definition of Internet access. Although this definition seems to be obvious at first glance, this section shows that there exist quite different views.

2.1. Architecture and Principles

The basic Internet architecture is depicted in Figure 2.1. It has evolved to its current form around 1977. One of the design goals of the Internet was the ability to support applications with quite different service requirements while operating on top of arbitrary communication technologies. This is achieved by having a single network layer protocol making no assumptions about the underlying communication technology, only offering a connectionless, best-effort service to the higher layer protocols. As illustrated in Figure 2.1, different higher layer protocols are responsible for providing an enhanced service for different types of applications. Thus from an architectural point of view, the integration of wireless technologies into the Internet should be straightforward. However, in Chapter 3 we show that although it is indeed possible to incorporate wireless networks, performance might suffer due to the peculiarities of the wireless link.

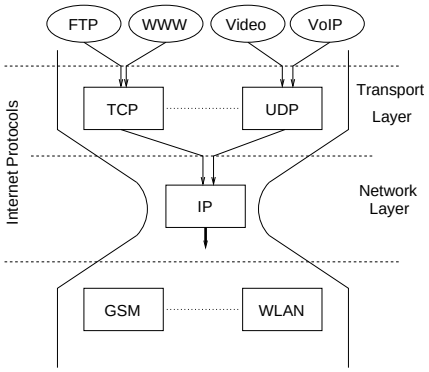


Figure 2.1.: The Internet architecture

The advantage of having a single network protocol making no assumptions about the underlying network and offering only a minimal service (just forwarding packets) is that the architecture is robust concerning new technologies or applications. On the other hand the multiplexing of all transport protocol flows into a single flow prevents a flow selective treatment of packets by the deployed communication technology. Although differentiation was not necessary when the Internet architecture was designed, it has become important with the increasing heterogeneity of the Internet, regarding both network properties and applications.

The fundamental structure of the Internet is a packet switching communication facility consisting of heterogenous networks connected together via routers implementing a store and forward packet forwarding mechanism[49]. The network layer protocol of the Internet is IP (specified in RFC 791[141]). Its main tasks are node addressing and packet forwarding. IP provides a connectionless, packet switched, best effort service. This means that packets can be lost, delivered out of order, duplicated or heavily delayed. Global connectivity is achieved by introducing a technology independent addressing scheme[1].

Currently two transport protocols are standardized. TCP[142] offers a reliable connection oriented service (see Section 2.2). UDP[140] offers a connectionless datagram service[2]. UDP only adds demultiplexing functionality and a data checksum to the service offered by IP. Both transport protocols use port numbers (a 16-bit integer value) to demultiplex packets to different applications. Thus an Internet connection or flow is identified by a quadruple consisting of source and destination IP addresses and source and destination port numbers.

The design of the Internet protocols has been driven by the belief that end-to-end functions can best be realized end-to-end protocols[149]. This argument together with the goal of designing a network which is robust against failures led to an architecture where most of the functionality (e.g. error control or flow control) and especially all states are implemented in the end system. If state in a network node is unavoidable, as it is the case for routing, it will have to be self-healing; adaptive procedures or protocols must exist to derive and maintain the state and to change it if the topology or activity of the network changes.

However, the architecture of the Internet is not final. New demands like the need for accounting or quality of service aspects might require changes. RFC 1958 states that the principle of constant change is perhaps the only principle of the Internet that should survive indefinitely[45]. Especially the end-to-end argument has been under discussion recently[127, 48, 147].

2.2. Transmission Control Protocol

The Transmission Control Protocol (TCP) is the dominant transport protocol of the Internet (up to 90% of the Internet traffic is based on TCP). It provides a reliable[3], connection oriented, byte stream service. Its behavior is defined in RFC 793[142]. Further details about how TCP should operate are given in RFC 1122, Section 4.2[36]. Some aspects of TCP were changed over the time and were updated by a number of RFCs[134, 74, 73, 9, 121].

The widespread deployment of TCP started in 1983 when the Department of Defense mandated that all computers connected to long-haul networks should use TCP/IP. About this time the University of Berkeley started to integrate their TCP/IP protocol implementation into BSD Unix. In 1988 the Tahoe TCP version was released. The main difference between TCP Tahoe and TCP Reno is the absence of the Fast Recoveryalgorithm. TCP Reno was released in 1990. TCP Reno is still used today, however with a number of extensions like selective acknowledgments.

[1] IP addresses refer to network interfaces. Every network interface has its own IP address. This works fine in static environments but introduces many problems in the case of mobility.

[2] RTP is not considered here because it operates on top of UDP.

[3] Reliable means that every byte is delivered to the service user exactly once in the correct order.

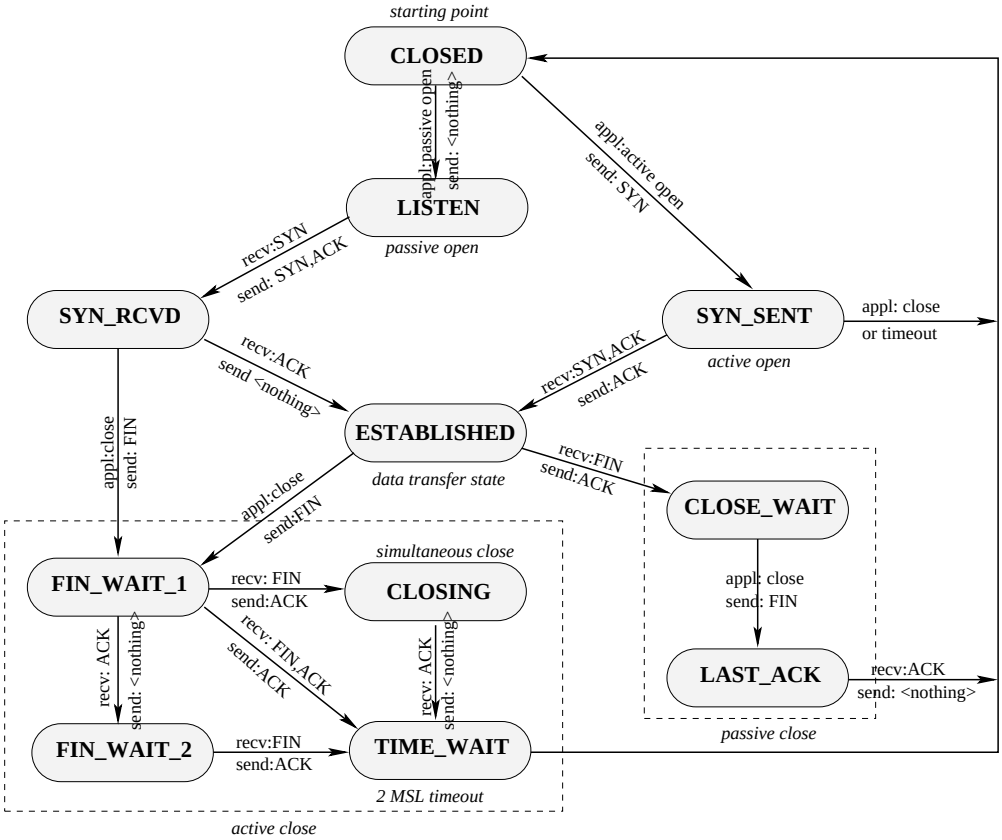


Figure 2.2.: TCP state transition diagram

The operation of TCP can be divided into three phases: connection establishment, data transfer, and connection termination. Figure 2.2 shows the state transition diagram of TCP. Since the data transfer phase (state: *ESTABLISHED*) is most important for this thesis we summarize connection establishment and termination below and discuss the data transfer phase in detail in the next section.

Connection Management

TCP uses a three way handshake to establish a connection. No data transfer is allowed during the connection establishment phase. The initiator of a connection chooses an initial sequence number ($\mathrm{seq\_init\_a}$) and sends a `SYN`-segment to the passive end of the connection which has to be in the `LISTEN`-State. Upon receiving the initial segment, the passive end sends a `SYN-ACK`-segment which includes its initial sequence number ($\mathrm{seq\_init\_b}$) and the initial sequence number of the initiator incremented by one ($\mathrm{seq\_init\_a} + 1$). The initiator considers a connection as established upon reception of the `SYN-ACK`-segment. The `SYN-ACK`-segment is acknowledged with an `ACK`-segment which has the sequence number $\mathrm{seq\_init\_a} + 1$ and the acknowledgment number $\mathrm{seq\_init\_b} + 1$. The callee enters the `ESTABLISHED`-State after the third packet is received. After the connection is established TCP provides a bidirectional channel.

The purpose of the three way handshake and the initial sequence numbers is to prevent that old packets

from previous connections are accepted by a new connection between the same hosts. Old packets might arrive after a connection was closed, since the Internet might duplicate and delay packets (see Section 6.2.2 in [155]).

During connection establishment the two TCP instances can negotiate parameters and options to be used for this connection. Options commonly used are, for example, the *timestamp option*, the *SACK option* and the *window scale option*. The Maximum Segment Size (MSS) can also be negotiated. The MSS is the maximum segment size a TCP instance is able to receive and should not be confused with the Maximum Transfer Unit (MTU) . The MTU is the maximum packet that can be transferred over the path between two communicating TCP instances without being fragmented.

Two performance problems are inherent to the connection establishment mechanism. First of all, the three way handshake takes some time to complete. This time mainly depends on the RTT between initiator and callee. Especially for short transfers, the set-up delay counts. Second, if one of the messages of the three way handshake gets lost, TCP will need several seconds to detect the failure. The problem is that at the beginning of a new connection TCP has no idea about the RTT of the path. In order to avoid the transmission of packets into a congested network, TCP uses a conservative default value to initialize its retransmission timer. This default value must be large enough to cover the RTT of arbitrary networks in order to avoid spurious retransmissions of the `SYN`-segment. Currently the default value is three seconds. This timeout value is doubled for every retransmission by TCP's backoff algorithm.

For example, if a single message of 1000 bytes has to be transferred over a path with a propagation delay of 100 ms and the first `SYN`-segment is lost, the throughput will be $\frac{8000\text{bits}}{3\text{s}+0.2\text{s}} = 312\text{bytes/s}$. This throughout is independent of the available bitrate[4].

The termination of a connection can be initiated by both ends. The interesting aspect of TCP's connection termination is that each end can only close its sending channel. Thus, terminating a connection means that no more data is sent but not that no more data is accepted. The termination of a connection is graceful. Before TCP closes a connection, it reliably transmits all unacknowledged data. In order to terminate a connection, TCP sends a segment with the `FIN`-bit set. This `FIN`-segment must be acknowledged by the peer entity. Similar to the connection establishment, TCP distinguishes between *active* and *passive* close. The TCP entity which closes its half of the connection first performs the active close. In order to close both halves of a connection, four packets are required.

Data transfer

As already mentioned, TCP provides a reliable service. To achieve this it uses sequence numbers, retransmission timers, flow control, and congestion control. Especially the congestion control algorithm is crucial for the operation of the Internet. The flow control and congestion control algorithms determine the achievable throughput.

After a connection is established, both ends are allowed to send data segments (bidirectional channel). However, the following discussion only describes a single direction. The other direction works identical. TCP's data transfer mechanisms are optimized to avoid congestion. Instead of sending at the maximum rate, TCP carefully tries to estimate the available capacity. The transmission rate is controlled by the rate of incoming acknowledgments. Therefore TCP's data transmission is called *Acknowledgment Clocked*. The number of bytes TCP is allowed to send without having to wait for an acknowledgment is controlled by two windows, whereby the smaller one is determinative (see Equation (2.1)). The *Advertised Window* is used to implement flow control between the two peers while the *Congestion Window* is used to protect the network from congestion. The Advertised Window is controlled by the receiving side (the receive buffer size can be set by the application) and the Congestion Window is controlled by the sender. The algorithms used to control the Congestion Window are discussed below. A connection is said to be either network,

[4] Assuming that the bitrate is not lower than the throughput.

application or receiver limited. In the first case the Congestion Window limits the transmission rate. In the second case the application data passed to TCP does not reach the available capacity. In the third case the Advertised Window is not large enough to fill the pipe. This means that a larger Advertised Window would increase the throughput.

$$cur\_wnd = min(cwnd, advertised\_wnd) \tag{2.1}$$

A peculiarity of TCP is the numbering of bytes instead of packets. The sequence number is increased with every byte sent. This is why TCP is said to provide a byte-stream service. From the application point of view packet boundaries are not preserved by TCP. The receiving application can read arbitrarily long chunks of data (if available) independent of the segment or block size used by the sender. From the protocol point of view the advantage of a byte stream is the possibility to combine multiple small packets into a single large packet. The drawback of this approach is a waste of sequence numbers. The sequence number field of TCP is 32 bits wide. This allows to uniquely number 2^{32} bytes before the sequence number wraps around. Although this seems to be a large sequence number space, it wraps around rather fast for high bitrates. Additionally in order to distinguish old packets from new ones the sequence number space must be at least twice the window size if selective repeat is used (see page 202 of [81] for a counter example). This is especially a problem for networks with a high bandwidth-delay product, which for example is the case for satellite networks. A separate RFC 1323[90] deals with problems introduced by so called fat pipes (LFN, pronounced elephan(t)). An LFN is a network with a large bandwidth-delay product. RFC 1323 introduces mechanisms to protect TCP against wrapped sequence numbers (PAWS), a Window scale option, and other mechanisms (SACK and timestamps).

TCP uses cumulative acknowledgments to acknowledge the reception of data. Each acknowledgment acknowledges all data up to the acknowledgment number. In order to reduce overhead, TCP only acknowledges every second received data segment. This mechanism is called *delayed acknowledgments*. To avoid deadlocks and unnecessary retransmission, an acknowledgment must be sent if a delayed acknowledgment timer expires before a second packet is received. The timeout value of this timer is not standardized but an upper limit is given. According to RFC 1122[36] an acknowledgment should not be delayed for more than 500 ms. A typical value found in some implementations is 200 ms. Every acknowledgment informs the sender about the receiver's receive buffer size (Advertised Window). In the case of bidirectional communication, acknowledgments as well as window updates can be piggybacked on data segments.

In order to detect losses, TCP protects the transmission of data by a retransmission timer. Whenever unacknowledged data is outstanding, a single retransmission timer is active. The retransmission timer is started when a packet is sent and no retransmission timer is currently running. If the retransmission timer expires, TCP will enter Go-Back-N mode. In this mode all unacknowledged packets are retransmitted according to TCP's congestion control mechanisms. If the timed packet is acknowledged and there is sent but unacknowledged data in the send buffer, then the retransmission timer will be restarted.

To determine a value for the retransmission timer, TCP measures the RTT of a connection. The measurement can be based on two different algorithms. The standard algorithm measures one RTT value per window. If a data packet is sent and no measurement is currently active, TCP will start a measurement. The measurement is stopped when the corresponding acknowledgment is received. The corresponding acknowledgment is any acknowledgment with an acknowledgment number higher than the sequence number of the timed data packet plus its length. The measurement is only completed if no retransmission has occurred, to avoid wrong RTT estimates (Karn's algorithm[98, 99]). This problem is known as TCP's acknowledgment ambiguity. If a packet is heavily delayed in the network, causing its retransmission, the TCP sender won't be able to distinguish whether the acknowledgment belongs to the original transmission or to the retransmission. Therefore it cannot calculate the RTT.

The second mechanism is based on timestamps as described in RFC 1323[90]. To enable this mode, both entities must negotiate the use of the timestamp option during connection establishment. In this

mode every data packet gets a timestamp when it is sent. The receiver echoes this timestamp with the corresponding acknowledgment[5]. Using this algorithm the sender can calculate the RTT for every received acknowledgment (even in the case of spurious retransmissions). As a result TCP can adapt faster to network delay changes. The costs of this approach are TCP option processing and 12 extra bytes of overhead for every TCP segment. This is especially a problem for slow links and small MTU sizes.

The computation of TCP's retransmission timer is defined in [88, 36] and further discussed in RFC 2988[134]. To compute the retransmission timer, TCP calculates the Smoothed Round Trip Time (SRTT) and the round trip time variation (RTTVAR)[6]. The initial value for the Retransmission Timeout (RTO) is three seconds. After the first RTT measurement is completed and the measured value is R, the following steps must be applied (G is the timer granularity).

$$\begin{aligned} \mathrm{SRTT} &\leftarrow R && (2.2)\\ \mathrm{RTTVAR} &\leftarrow \frac{R}{2} && (2.3)\\ \mathrm{RTO} &\leftarrow \mathrm{SRTT} + \max(G, 4 * \mathrm{RTTVAR}) && (2.4) \end{aligned}$$

When a subsequent RTT measurement R' is made, the following calculation is performed.

$$\begin{aligned} \mathrm{RTTVAR} &\leftarrow (1 - 0.25) * \mathrm{RTTVAR} + 0.25 * |\mathrm{SRTT} - R'| && (2.5)\\ \mathrm{SRTT} &\leftarrow (1 - 0.125) * \mathrm{SRTT} + 0.125 * R' && (2.6)\\ \mathrm{RTO} &\leftarrow \mathrm{SRTT} + \max(G, 4 * \mathrm{RTTVAR}) && (2.7) \end{aligned}$$

If the resulting RTO is smaller than one second, it will be rounded up to one second. If a maximum value for RTO is used, it will have to be at least 60 seconds.

Each time the retransmission timer expires the RTO value is doubled. The reasoning behind this backoff algorithm is the assumption that a loss indicates congestion. Hence, TCP should be conservative with pushing additional packets into the network, especially because the current RTT cannot be determined due to Karn's algorithm. After the retransmission timer has expired several times for the same segment, TCP gives up and resets the connection. According to RFC 1122, the threshold for giving up should be configurable by the application and should exceed 100 seconds. After the retransmission timer has expired, SRTT and RTTVAR may be initialized with the first valid new RTT measurement, since the old values might be outdated.

As the formulas and figures presented above show, the TCP retransmission timer is very conservative. A minimum retransmission timeout of 1 second is an eternity to modern high speed networks. The retransmission timer can always be expected to be larger than the current RTT. The reasoning behind this conservative algorithm is twofold. On one hand spurious retransmission should be avoided. Since the Internet makes no guarantees about the delay, it is better to assume a high variance. On the other hand congestion should be prevented. The assumption here is that the major cause for packet losses is queue overflows at routers. Thus, if a loss occurs it will be better to give the network some time to recover than to aggressively retransmit packets. Later we will see to which extent this strategy conflicts with error prone wireless links.

[5] The mechanism is a bit more complicate due to delayed acknowledgments and the like. However the given explanation should be sufficient to understand the core mechanisms.

[6] Jacobson's algorithm for incorporating the measured RTT variance is especially important on a low-speed link, where the natural variation of packet sizes causes a large RTT variation. One vendor was able to increase link utilization on a 9.6 kb/s line from 10% to 90% as a result of implementing Jacobson's variance algorithm.

Since the retransmission timer of TCP has to be conservative, a second mechanisms that is able to react faster on lost packets was introduced. If TCP receives three *duplicate acknowledgments*, it will assume that a packet is lost. A duplicate acknowledgment is an acknowledgment which is unchanged with respect to the previously received packet. The receiver generates a duplicate acknowledgment whenever it receives an unexpected segment which lies within its window. Since the underlying network can reorder packets, TCP does not react on the first duplicate acknowledgment but on the third. Basically the Fast Retransmitalgorithm is a kind of negative acknowledgment. In contrast to Go-back-N mode, which is entered after a timeout, the reception of three duplicate acknowledgments only triggers the retransmission of a single packet. The oldest unacknowledged packet is retransmitted. If more than one packet is lost within a window, TCP will need a timeout to recover. To improve performance, this is changed in new TCP versions by the SACK option or by TCP NewReno (see below).

Congestion Control

If a new connection is started, TCP will not know the state and capacity of the network path to the destination host. Since TCP operates on top of an unreliable store and forward service, it is designed to avoid congestion without support by the network layer. At the beginning of a new connection TCP has to slowly probe the network to determine the available capacity. Otherwise it might overload the network with an inappropriately large burst of data.

One of the most important features of TCP crucial for the operability of the whole Internet is its ability to adapt itself to the capacity of the network. For this purpose the transmission of data segments is clocked by the reception of acknowledgments. TCP only pushes new data into the network if it receives an acknowledgment, since an acknowledgment indicates that a packet has left the network. The number of segments which TCP is allowed to send upon reception of an acknowledgment is controlled by four algorithms, namely *Slow Start* , *Congestion Avoidance*, *Fast Retransmit* and *frecover*. These four mechanisms are used to control TCP's Congestion Window. The congestion algorithms were first defined by Van Jacobson in 1988[88] and were updated later in RFC 2581[9].

To implement the congestion control mechanism, two variables are maintained per connection and for each direction. The congestion window (*cwnd*) represents the maximum amount of data TCP is allowed to push into the network and the Slow Start Threshold(*ssthresh*) attempts to dynamically estimate the correct window size for a connection. The correct window size maximizes throughout without congesting the network.

Slow Start is used to probe the network at the beginning of a new connection, after a long idle period, or after the retransmission timer has expired. This algorithm is used to reach the equilibrium state. TCP is in Slow Start mode as long as `cwnd` is less than `ssthresh`. The initial Congestion Window is set to one[7]. This means that at the beginning of a connection TCP is only allowed to push a single packet into the network. Each time an acknowledgment is received, `cwnd` is incremented by one[8]. Thus, after the first acknowledgment is received TCP is allowed to send two segments.

If the network capacity is reached, packets will be lost due to queue overflow. If the loss is detected by a timeout, TCP will have to restart the Slow Start phase. `cwnd` will be set to one even if the initial window is larger than one. `ssthresh` is updated using Equation (2.8). To update `ssthresh`, TCP calculates the `FlightSize`. The `FlightSize` is the amount of outstanding data in the network. Earlier versions

[7] RFC 2581[9] allows to set the initial window to 2. RFC 2414[7] suggests to set the initial Congestion Window to $\min(4 * \mathrm{MSS}, \max(2 * \mathrm{MSS}, 4380\mathrm{bytes}))$. However, the status of this RFC is experimental.

[8] Actually, the time it takes to open to a given window is $R \log_2 W$ where R is the RTT and W is the window in packets. If the receiver sends one acknowledgment for every second segment, this estimate should be $R \log_{1.5} W$. It is generally agreed that during Slow Start it is appropriate to increase the window size by one MSS per acknowledgment, even if the acknowledgment acknowledges more or less than one MSS of data.

of TCP used the minimum of the current value of the Congestion Window and the Advertised Window instead of the `FlightSize`. However, this is not correct, since it is possible that the transmission rate is application limited rather than network or receiver limited.

$$\text{ssthresh} = \min(\frac{\text{FlightSize}}{2}, 2 * \text{SMSS}) \tag{2.8}$$

If `cwnd` reaches `ssthresh`[9], TCP will enter the Congestion Avoidancephase. In this phase `cwnd` is increased by one segment per window. Thus, the exponential increase of the Congestion Window during the Slow Start phase is slowed down to linear increase. During the Congestion Avoidancephase TCP carefully tries to reach the capacity of the network. If the capacity is reached, often only a single packet is lost.

If three duplicate acknowledgments are received, TCP enters Fast Retransmitfollowed by Fast Recovery. Fast Retransmitmeans that the oldest unacknowledged packet is retransmitted. In Fast Recoverymode `ssthresh` is updated using Formula (2.8). Contrary to the Slow Start phase `cwnd` is set to `ssthresh` plus three. The argumentation behind this is that the network is still operable since some packets have made it through to the receiver. Because three duplicate acknowledgments were received, three new packets are allowed to be pushed into the network. However, the loss of packets indicates that the network is operating at its limit, therefore the sending rate has to be reduced. During Fast Recoverythe Congestion Window is incremented by one for every received acknowledgment. The Fast Recoveryphase ends when a new acknowledgment is received or when the retransmission timer expires. In the first case the Congestion Window is set to the Slow Start Thresholdand TCP continues its operation in Congestion Avoidancemode. In the second case Slow Start is entered. The four modes are illustrated in Figure 2.3. This figure shows the sending of data segments and the reception of acknowledgments on the left side. The right side shows the development of the Congestion Window and the Slow Start Threshold. In order to show the mechanisms we configured NS to drop an entire window of packets at around 0.4 seconds and to drop a single segment at 1.2 seconds. Before the first drop event occurs the Congestion Window develops according to the Slow Start mechanism. It increases from a single packet up to 16 packets. Since all 16 packets are lost, the retransmission timer expires and TCP restarts Slow Start. The slow start threshold is set to eight. When the Congestion Window reaches eight, Congestion Avoidanceis entered. At 1.2 seconds a packet is lost. This loss is detected by the reception of the third duplicate acknowledgment at 1.38 seconds. The Congestion Window is halved and then inflated to seven. The Slow Start Thresholdis set to four. Then Fast Recoveryis started. During Fast Recoverythe Congestion Window is increased by one for every received acknowledgment. The Congestion Window is set to the Slow Start Thresholdand TCP enters Congestion Avoidanceif the acknowledgment for the lost packet is received.

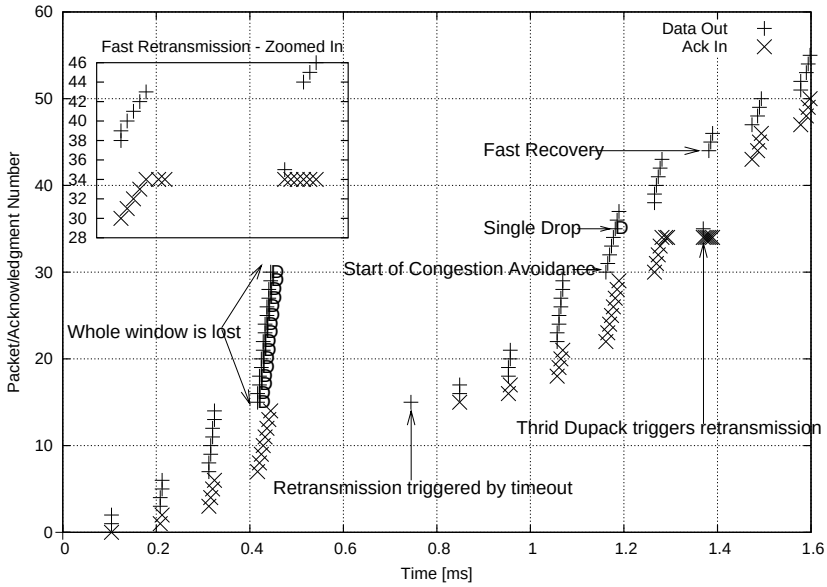
Recommended extensions to TCP

This section describes extensions to the TCP standard which are recommended by the Internet Engineering Task Force (IETF) and are part of most of the current TCP implementations. For each extension we discuss which problems are mitigated.

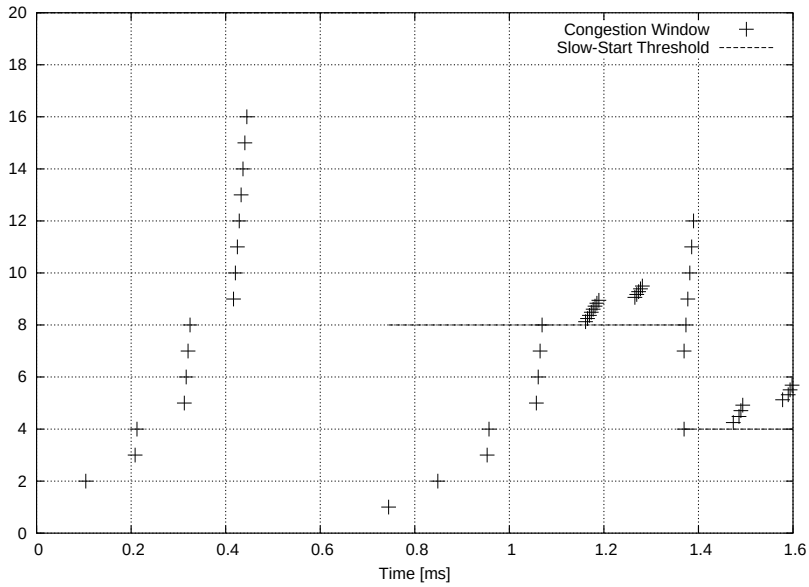
SACK[121]

Early versions of TCP implement only a Go-back-N algorithm in the case of packet loss. This was improved later by retransmitting a single packet upon reception of three duplicate acknowledgments. However, the cumulative acknowledgments of TCP do not explicitly provide information to the sender about

[9] A TCP implementation can enter the Congestion Avoidancephase either if `cwnd` equals `ssthresh` or if it is larger than `ssthresh`.



(a) Time-Sequence Chart



(b) Congestion Window

Figure 2.3.: TCP's Slow Start, Congestion Avoidance, Fast Retransmitand Fast Recovery. Plot is taken from a simulation performed with NS. Configuration: One hop network, 2 Mbit/s bitrate, and 50 ms RTT.

the reception of out-of-order segments. The sender has to deduce this information by interpreting duplicate acknowledgments. As this information is incomplete (e.g. how many packets were lost) any solution must be sub-optimal[10]. Furthermore Fast Retransmitand Fast Recoverycan only recover from a single lost segment per window. If multiple packets are lost, TCP will not fill the network and hence the self-clock mechanism will not be preserved. In this case TCP needs a timeout followed by Slow Start to recover.

In order to combat these problems a Selective Acknowledgment (SACK) option is defined in RFC 2018. This option allows a TCP receiver to report the reception of non-contiguous blocks of data to the TCP sender using the option field of the TCP header.

If the receiver receives non-contiguous data, it will send a duplicate acknowledgment including the SACK option to inform the sender which segments were successfully received. Each block of continuous data is expressed in the SACK option using the sequence number of the first octet of data in the block and the sequence number of the octet just beyond the end of the block. In the new SACK option the first block is required to include the most recently received segment. Additional SACK blocks repeat previously sent SACK blocks to increase robustness if acknowledgments are lost.

RFC 2883[74] extends RFC 2018 to report the reception of duplicate segments. It does not define the actions to be taken if an acknowledgment reports reception of a duplicate segment. This extension to the SACK option allows the TCP sender to infer the order of packets received at the receiver, allowing the sender to infer whether it has unnecessarily retransmitted a packet. A TCP sender can use this information for more robust operation in an environment of reordered packets[8], ACK loss, packet replication and/or early retransmit timeouts.

After a retransmission timeout the TCP sender has to cancel all SACK information, because it is possible that the data receiver has discarded segments (due to resource limitations) which have already been acknowledged by a selective acknowledgment. Thus, in the case of a timeout the data sender must retransmit the segment at the left edge of the window whether it was included in a SACK block or not. A segment is not removed from the send buffer until the left window edge has passed it.

In order to use the selective acknowledgment option both TCP instances have to support it and must negotiate the usage of SACKs during connection establishment. If one of the two TCP instances does not support this option it won't be used. In this case TCP NewReno, which is defined in RFC 2582[73], can be applied to recover from multiple losses per window during Fast Recovery. The NewReno algorithm modifies Fast Recoveryto retransmit the first unacknowledged packet upon reception of partial acknowledgment. A partial acknowledgment is an acknowledgment which does not acknowledge all data sent before Fast Recoverymode was entered.

SACK is a method for data recovery while TCP performs Fast Recovery(i.e. after three duplicate acknowledgments). Another issue in this state is to maintain the principle of conservation of packets. TCP Reno fails to estimate the number of packets in flight during recovery because it can only use the duplicate acknowledgments to estimate this value. The Forward Acknowledgment (FACK[122]) approach realizes the idea to decouple the congestion control algorithm from the data recovery algorithms. The FACK algorithm uses the information provided by the SACK option to achieve a more accurate estimate of the packets in flight. For this estimation it introduces two new variables (*seq_fack*) which is set to the forward-most data held by the receiver (hence the name FACK) and a counter which counts the number of retransmitted packets. The forward-most data is the segment with the highest sequence number seen by the receiver. The number of outstanding packets is simply calculated as $\mathrm{awmd} = \mathrm{snd\_nxt} - \mathrm{snd\_fack} + \mathrm{retrans\_data}$. A performance comparison presented in [122] shows that the FACK algorithm performs better than SACK alone if multiple packet losses occur per window. Today FACK is included in many TCP implementations (e.g. Linux).

[10] In[65] Fall and Floyd show that without selective acknowledgments TCP implementations are constrained to either retransmit at most one dropped packet per RTT, or to retransmit packets that might have already been successfully delivered.

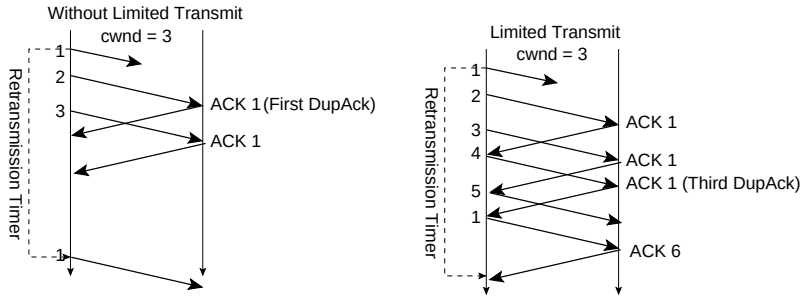


Figure 2.4.: Limited Transmit example

Further Extensions to TCP

Besides the extensions recommended by the IETF and/or found in most current TCP implementation there exist a number of suggestions which are either currently under discussion or deal with a specific improvement.

One of TCP's problems is that it will hardly be able to recover fast if the Congestion Window is small, since in this case the Fast Retransmitalgorithm is not triggered. To remedy this problem the *Limited Transmit Algorithm* described in RFC 3042[5] calls for sending a new data segment in response to each of the first two duplicate acknowledgments that arrive at the sender. The transmission of these two new segments increases the probability of receiving three duplicate acknowledgments and hence TCP can recover from a single lost segment using the Fast Retransmitalgorithm rather than using a costly retransmission timeout. Limited Transmit can be used both in conjunction with and in the absence of the TCP selective acknowledgment (SACK) mechanism. The general algorithm is illustrated in Figure 2.4.

After long idle times of an application or if the application does not have sufficient data to send to reach the current Congestion Window, the Congestion Window might have become invalid. In RFC 2861[82] an algorithm to validate the current setting is specified. Contrary to RFC 2581 which recommends (the should form is used) to set the Congestion Window to its initial window after a long idle period, RFC 2861 suggests to decay the Congestion Window. It particularly suggests halving the Congestion Window with every RTT during which the flow remains inactive. Further it suggests to increase the Congestion Window only if the send buffer is not empty.

TCP-Reno, despite all improvements (e.g. SACK), implements a reactive approach to congestion control. The sending rate is increased unless the sender perceives packet loss. In contrast to this, TCP-Vegas[37, 111] aims at decreasing losses and increasing throughput by anticipating the onset of congestion. It also uses mechanisms like Slow Start and Congestion Avoidancebut differs from Reno in the way it updates its Congestion Window.

The idea is that with increasing sending size (Congestion Window) the throughput also increases but not beyond the available bandwidth. Hence TCP-Vegas compares the expected throughput, defined as $\mathrm{Expected} = \frac{\mathrm{cwnd}}{\mathrm{MinRTT}}$, to the actual throughput. During Slow Start TCP-Vegas allows exponential growth only every other RTT, so that a valid comparison of the expected and the actual rates can be made. If the actual rate falls below the expected rate, TCP changes from Slow Start mode to Congestion Avoidance. During Congestion Avoidancetwo thresholds ($\alpha < \beta$) are used to determine whether the Congestion Window should increase linearly, stay constant or decrease linearly. If the difference between expected and actual sending rate is less than α the window is increased. If the difference is larger than β the window is decreased.

2.2.1. TCP's User Interface

RFC 793 also functionally characterizes the interface that TCP should provide to the user (application). However, the described interface is not mandatory. The interface description consists of a set of calls, much like the operating system calls provided to an application process for manipulating files. For example, there are calls to open and close connections as well as to send and receive data on established connections. Although the interface is described using the function call paradigm, the standard does not determine whether the interface must be synchronous or asynchronous. Later in Section 4.1 we will see how the BSD socket interface implements this description.

The `open`-function is used to create a new TCP instance. The caller must specify the local port, the foreign socket (address of the peer entity), a flag which indicates whether this is an active or passive open, a timeout and additional parameters that are not important in the scope of this thesis. If the flag indicates passive mode, the new TCP instance enters the listing mode, waiting for connection requests to arrive. If the flag is set to active mode, the newly created TCP instance will initiate TCP's three way handshake. The timeout, if present, permits the caller to set a time limit during which all data submitted to TCP should be delivered. If TCP fails to deliver the data, the connection will be aborted. The `OPEN` function returns a local connection name that can be used by the caller as a short hand term for all succeeding function calls related to this connection.

The `send`-function is used to transmit data. The caller has to specify the local connection name, the buffer containing the data and the number of bytes in this buffer. Further it can specify an urgent flag, a push flag, and a timeout. If the connection has not been opened, the `send`-function will be considered as an error. The RFC does not define the semantics of the `send`-call, for instance that a call to the `send`-function must not return before all data is acknowledged by the peer TCP entity. Instead it discusses different possible alternatives. For performance reasons the RFC suggests that the `send`-function should return an immediate local acknowledgment, even if the segment sent has not been acknowledged by the distant TCP entity. We emphasis this semantic point here, since it is important for the design of the remote TCP interface.

The `receive`-function is used to consume received data. Similar to the `send`-function the caller has to specify the local connection name, a buffer and the number of requested bytes. The `receive`-function returns either if the number of requested bytes is available or if a segment with the push flag set is received.

The `close`-function is used to terminate a connection. The meaning of a close is that the caller has no more data to send but not that the caller is not willing to receive additional data. Furthermore the close operation is graceful in the sense that outstanding data (data that has been passed to TCP by the application but has not been sent or acknowledged yet) is reliably transmitted.

The `status`-function is used to query a local communication endpoint. The caller has to specify the local connection name as parameter. The return values of this call are values of the connection control block. However, it is not specified which values should be included in the report. The RFC only makes some suggestions.

The `abort`-function is used to abort a connection. In this case all pending sends and receives are canceled and a `reset`-segment is sent.

Besides these function calls it is assumed that TCP can deliver signals (messages) asynchronously to the user. The purpose of such signals is for example, to inform the user about a state change or that new data has arrived. The RFC does not specify which signals have to be implemented.

2.3. User Datagram Protocol

UDP offers a connectionless unreliable datagram service. Basically it just adds payload checksumming and application addressing to the functionality of IP.

2.4. Definition of Internet Access

Although the definition of Internet access seems to be obvious at a first glance, there are different views. In [167] Wolisz et al. distinguish between three different definitions:

- Sending and receiving IP packets.
- Access to popular services like E-Mail and WWW.
- Access to the service of the Internet protocol stack.

The first definition describes the classical case where the wireless end system includes the entire Internet protocol stack. The second definition considers solutions which allow end systems to access popular Internet services without using IP. Usually a special network node is used to translate between different protocols. A well known candidate of this class is the Wireless Application Protocol (WAP)[11]. Finally the third definition offers full Internet access without having the Internet protocol stack on the wireless end system. In our opinion this is the most appealing approach and the idea we pursue with the Remote Socket Architecture.

[11] www.wapforum.org

Chapter 3.

TCP over Wireless

In the previous chapter we stressed that the Internet protocols are able to operate on arbitrary technologies. Hence replacing a cable by wireless technologies should not introduce any problems. On the other hand following the one size fits all idea requires either an environment that is homogeneous at least to some extent or compromises regarding performance issues. Since wireless networks have many characteristics that increase the Internet's heterogeneity (e.g. high error rate, time-varying nature, disconnectivity,...), it is questionable whether the Internet protocols, especially TCP, are able to provide the available bitrate to applications. Thus, the question is not whether the Internet protocols are able to operate over wireless networks, but rather what the performance penalty is if they are used. Since TCP is the most complex protocol of the Internet protocol suite and is especially impaired by the peculiarities of wireless communication, the discussion in this chapter is limited to TCP.

The goal of this chapter is to explain why TCP might fail in the case of wireless networks and to summarize the most popular solutions which try to solve the described problems. For this purpose we start with defining the network architecture which we assume throughout this work. Please note that a different network architecture might lead to additional problems (e.g. we do not consider network path asymmetry here, which for example is common for satellite networks). Then we proceed with explaining the effects that the peculiarities of wireless communication have on TCP's operation. We end this chapter with a discussion of different approaches that try to solve the described problems.

3.1. Assumed Wireless Internet Architecture

Throughout this thesis we assume a single wireless cell which is either the first or last hop of the path between source and destination. The wireless end system is connected to the Internet using an access point that connects the wired and wireless world. The technology of the wireless access network is arbitrary (e.g.: Bluetooth, 802.11, GSM, UMTS). The possible technologies differ in parameters like cell size and bitrate as they have different fields of application. A detailed description of GSM, GPRS, UMTS and IEEE 802.11 can be found in [146, 160, 84] and [129], respectively.

As commonly anticipated, we assume that the access network is composed of different technologies. For example global connectivity might be provided by 2.5G (e.g. GPRS) or 3G (e.g UMTS) cellular networks. Whereas for locations with high bandwidth demands wireless LAN (e.g. IEEE 802.11) or bluetooth hot-spots will be employed. The motivation behind the usage of different technologies is that different available technologies have quite different fields of application. Cellular networks like 2.5G and 3G networks promise ubiquitous Internet access while wireless LANs and bluetooth promise higher data rates and allow ad-hoc networking. Since wireless LAN cannot offer ubiquitous Internet access and cellular systems cannot offer the throughput of wireless LANs, it can not be expected that a single technology will replace all others. Ongoing research is working towards architectures that allow roaming between different technologies without losing the current state (e.g. see [76, 148, 35]). However, this is out of the scope of this thesis.

Although IEEE 802.11 and Bluetooth enable the establishment of ad-hoc networks, we preclude this type of networks in this work. Ad-hoc networking introduces many specific problems like routing issues. Furthermore in the case of ad-hoc networks the wireless path will often span multiple hops, whereas we assume a single hop. Problems of ad-hoc networking are discussed in [95].

One appealing new feature of wireless networking is mobility. Regarding mobility, layer two and layer three mobility have to be distinguished. In the case of layer two mobility the wireless end system does not change its ISP. The handover between different access points is transparent for higher layers. In the case of layer three mobility the end system moves to a new (sub)network. This means that the IP address of the end-system becomes topologically incorrect. Since the focus of this work is on problems that arise from replacing the cable by radio communication, and since IP does not inherently support mobility, we preclude layer three mobility from the discussion. Problems that come along with layer three mobility are a separate topic and are discussed elsewhere (e.g. [27, 171]). Problems that arise from layer two mobility are included in the discussion because it is transparent to the higher layers but might lead to an abrupt change of network conditions.

3.2. Problem Description

In this section we explain why TCP might suffer in the case of wireless links. Although the problems introduced by radio communication affect different aspects like security, energy efficiency, health, and performance, we focus on performance issues.

TCP provides a reliable service on top of the connectionless and unreliable IP service (see chapter 2.1 on page 7). Thus, TCP must be able to deal with packet loss, duplication, reordering and varying delay. TCP is also responsible for estimating the capacity of the network in order to avoid a congestion collapse. In the following we discuss how these requirements conflict with the peculiarities of wireless networks.

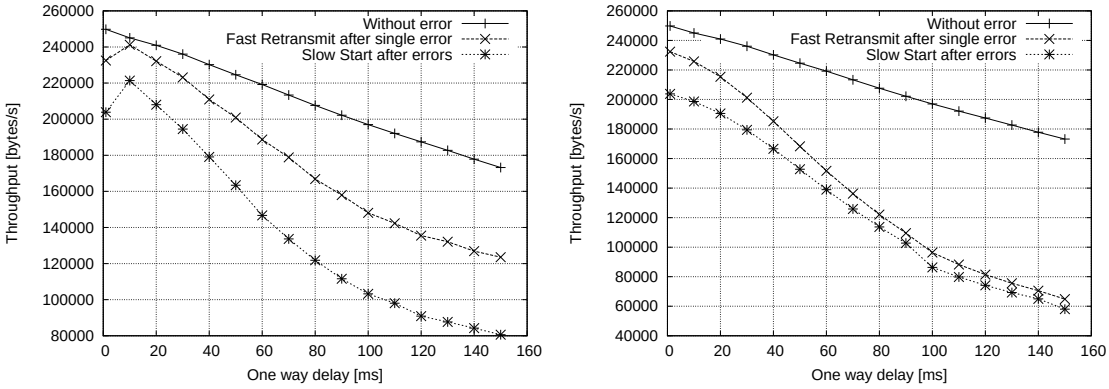
3.2.1. Congestion Losses vs. Transmission Losses

The fundamental problem that TCP has with radio links is the increased error probability. In tethered networks the bit error probability generally is very low ($< 10^{-6}$). Thus, in the Internet, packet losses are caused by queue overflows rather than by transmission errors. This is totally different for wireless systems. Wireless channels are often error-prone. For example measurements in the 2.4 GHz Industrial Scientific and Medical band (ISM) showed error rates as bad as $\approx 10^{-2\ldots-3}$[33, 56, 59].

At the beginning of the Internet it turned out that congestion avoidance is essential for the operability of the Internet. Hence TCP is designed to avoid congestion. If a TCP sender detects a packet loss either by a timeout or by receiving three duplicate acknowledgments, it has to follow the Congestion Avoidancerules, resulting in a reduced sending rate. This is also true if the loss is not caused by queue overflow (congestion) but by transmission errors. The problem is that TCP cannot determine the reason of a loss and hence has to assume congestion. Unfortunately both error sources require opposite reactions. In the case of a packet loss due to congestion TCP must reduce its sending rate, while in the case of a packet loss due to transmission errors a retransmission should follow. Unfortunately all attempts trying to determine the cause of a loss without modifying or enhancing the protocol have failed (e.g. see [31, 30]).

The mechanisms that TCP uses to adapt its sending rate depend on the loss indication (timeout or three duplicate acknowledgments). In the case of a retransmission timeout TCP re-enters Slow Start. This means that the Congestion Window is reduced to a single segment (even if the connection was started with a larger initial Congestion Window). If the loss event is detected by reception of three duplicate acknowledgments, TCP will halve its Congestion Window and enter Fast Recovery.

Independent of the loss event TCP corrects its estimation of the network path capacity by setting the Slow Start Thresholdto half of the current Congestion Window. For multiple losses or if the first loss



(a) Loss occur after link capacity is reached

(b) Loss occur at beginning of connection

Figure 3.1.: Effects of losses on TCP performance

occurs at the beginning of a connection, the Slow Start threshold is likely to reach the minimum value of two segments. In this case TCP has to linearly increase the Congestion Window in order to utilize the available capacity, even if the current sending rate is significantly smaller than the current network capacity. The performance penalty mainly depends on the RTT.

The effects of losses are illustrated in Figure 3.1. We simulated the transfer of 1 MByte of data using TCP Reno[1]. For every simulation run we varied the delay between source and destination as shown on the x-axis. For each delay we ran three simulations, one without error (curve labeled *Without error*), one with a single packet loss (curve labeled *Fast Retransmitafter a single error*) and one with three packet losses (labeled *Slow Start after errors*). The y-axis shows the throughput in bytes/second. For Figure 3.1(a) the errors occurred after TCP had reached the link capacity while for Figure 3.1(b) the errors occurred at the beginning of the connection. The two figures allow three conclusions. First, even a low error rate reduces TCP's performance significantly (we only simulated a single loss event). Second, the performance penalty depends on the RTT. And third, losses at the beginning of a connection are more harmful because TCP enters Congestion Avoidancetoo early.

3.2.2. Error Control Mechanisms

The question is whether TCP is able to detect and repair losses efficiently. Losses are detected (assumed) by either the reception of three duplicate acknowledgments or by a timeout. Since TCP is an end-to-end protocol, the loss indication always occurs later than the loss event. In the case of three duplicate acknowledgments the loss indication is delayed by the time which is required to transmit at least three additional packets to the receiver plus the time the acknowledgments need to reach the data sender. In the case of a loss indication due to the expiration of the retransmission timer the delay depends on how accurately the retransmission timer reflects the current RTT. For the setting of the retransmission timer a trade-off between being too aggressive or too conservative has to be found. In the former case the timer might expire prematurely leading to spurious retransmissions. In the latter case the performance would

[1]NS and the TCP Reno agent.

suffer from long idle periods. Since TCP is designed to avoid congestion, it uses a conservative mechanism. The retransmission timer has (according to RFC 2988) a minimum value of 1 second[2] and is doubled after every retransmission. If the real RTT is smaller than the minimum retransmission timer value, it will take TCP too long to detect a loss event caused by bit errors.

The second question is which packets should be retransmitted after a loss is assumed. If the retransmission timer expires, TCP will perform Go-back-N. This means that all unacknowledged packets are retransmitted starting with the oldest. Thus, even if the receiver receives out of order packets, those packets will be retransmitted. This is even true if TCP uses the SACK option. According to RFC 2018, a TCP sender must discard all SACK information upon a timeout, since selective acknowledgments are only informational. The receiver might discard packets it has already selectively acknowledged.

In the case of a Fast Retransmit, the TCP sender retransmits the oldest unacknowledged packet. In earlier versions the Fast Retransmitalgorithm could only correct a single packet loss per window and more than three packets must have been received by the receiver (which implies that the sender is allowed to and has sent more than three packets). The former problems are mitigated by the SACK option (RFC 2018) and New Reno (RFC 2582), and the latter problem is mitigated by the Limited Transmit algorithm (RFC 3042).

If for any reason the retransmission timer expires too early, the packets will be unnecessarily retransmitted. The major problem here is that TCP is unable to detect that the estimated RTT was too short. Therefore TCP goes through the entire Go-back-N cycle retransmitting every packet already received by the receiver and must also update its congestion control variables. The effects of spurious retransmission are shown in [115].

As a result neither the timer triggered retransmissions nor the Fast Retransmitalgorithm are able to handle high error rates efficiently. This is even true if TCP would be able to determine the cause of a packet loss and would not reduce the transmission rate upon non congestion-caused losses.

3.2.3. Estimation of the Retransmission Timer

TCP has to estimate the RTT to configure the retransmission timer. If the RTT variance is high, the retransmission timer will be set to a large value, since the RTO calculation includes the RTT variation multiplied by four (see Equation (2.7)). Thus, in this case TCP needs a long time to detect an error.

On the other hand, if the variance is low, the value of the retransmission timer will be close to the observed RTT. In this case it can happen that the timer expires spuriously. This can be especially observed if TCP operates on top of a link with a varying bit rate like GPRS (see e.g. [151, 170, 79, 86, 46]) or if long (with respect to the RTT) error bursts occur.

If TCP does not use the timestamp option it will only measure a single RTT per window. This measurement will not be completed if errors occur. Thus, in the case of an error prone link, TCP must live without an up to date estimation of the RTT.

TCP has no information about the RTT for the first packet of a connection. Hence, it must use a default value, which is currently three seconds. If this packet is lost, it takes much longer than required to recover. This is especially harmful if only a small amount of data should be transferred.

If the bit rate is low, leading to a high impact of the packet generation time on the RTT, a series of short packets will reduce the mean RTT. If the next long packet is sent, the retransmission timer might be too small leading to an unnecessary timeout.

3.2.4. Protocol Overhead

The ratio between payload and total amount of data sent becomes more important with decreasing bitrate or in the case of wireless devices. This ratio is negatively influenced by the protocol header size and the

[2] RFC 2988 uses the should form. This means that TCP implementations can choose a smaller minimum value.

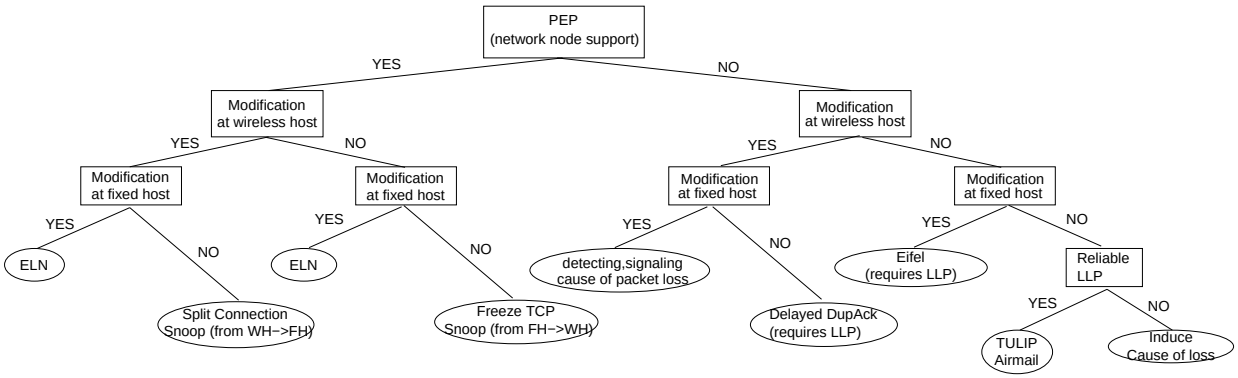


Figure 3.2.: Classification of solutions

number and size of control frames (e.g. acknowledgments), as well as the MTU size. TCP and IP each have a protocol header of at least 20 bytes. Since most protocol stacks support TCP Timestamp and SACK option today, the TCP header is usually between 32 bytes and 60 bytes long[3]. Thus, the overhead per TCP segment is around 52 bytes (neglecting the overhead of lower layer protocols). Furthermore TCP sends an acknowledgment for at least every second packet. If the timestamp option is enabled, the size of an acknowledgment is 52 bytes (without SACK blocks).

3.3. Solutions - State of the Art

Many papers by different research groups deal with the problems of wireless Internet access. Best to our knowledge papers by DeSimone et al. [55] and by Caćeres et al. [43] are the first papers that address such issues, both dating back to 1993. DeSimone et al. investigate the performance of reliable transport protocols over wireless LANs. We discuss this paper in more detail in Section 3.3.1. Caćeres et al. deal with mobility implications on reliable end-to-end connections. They use the Fast Recoverymechanism of TCP to shorten the delay after a mobile end system has changed the base-station. Altogether we found over 120 references in conference proceedings and journals and the number of publications concerning this topic still increases. The discussion of the various approaches is structured according to the ideas behind an approach and not in chronological order.

The main reason for performance degradation is TCP's inability to distinguish congestion losses from transmission errors and to deal with high error rates. Hence most research groups have contributed solutions for these problems. Most solutions only consider data transfer from a fixed host (i.e. server) to a wireless host (i.e. client). This is motivated by the observation that most data flows downstream (e.g. WWW traffic)[4]. Thus, the following discussion mainly deals with approaches that combat the aforementioned problem. Solutions for other problems mentioned in Section 3.2 are summarized in Section 3.3.3.

Figure 3.3 shows the classification scheme for the various approaches. This classification is used in the following discussion. The classification is represented as a tree in which the nodes provide information

[3]The maximum option size is 40 bytes. The size of the SACK option depends on the number of SACK blocks. An SACK option that specifies n blocks has a size of $8 * n + 2$.

[4]New network applications (e.g. Peer-to-Peer networks) have invalidated this assumption. However, when most of the discussed approaches were designed this assumption was justified.

on where modifications are required and the leaves present examples of each class. At the top level we separate solutions that are based on a PEP from those that are not. Hence the root of our classification tree is the question whether an approach requires support from a network node or not. The right sub-tree of Figure 3.3 shows solutions that are not based on PEPs. Such solutions are generally in line with the current Internet philosophy. The left sub-tree represents solutions that require a modification of the end-to-end principle of the Internet. The outer right path represents the classical Internet protocol stack as it is found in todays end systems. The outer left path represents solutions that require modifications at both end systems and within the network. Solutions that require modifications at fixed hosts (standard Internet hosts) have the drawback that they require changes on all existing TCP implementations and installations. In contrast to this, solutions that require modifications at the wireless end system are more feasible, since this affects a smaller number of hosts and does not affect legacy hosts.

A different classification scheme is to put all solutions that hide the peculiarities of the wireless link from the sender into one class and all solutions that try to inform TCP about the cause of a packet loss into a second group. This classification puts solutions that are in line with the Internet design principle and PEP based approaches in the same group (e.g. LLP and split connection approaches, both hide the peculiarities of the wireless link). Therefore we decided to use this classification only as a first coarse categorization. For each resulting group we used the scheme shown in Figure 3.3.

3.3.1. Solution Based on Hiding Link Characteristics

End-to-End solutions

This section describes the right sub-tree of Figure 3.3 (i.e. all solutions that are not based on a performance enhancing proxy).

Link Layer Protocols

We start the discussion about the usage of link layer protocols to remedy TCP's problems over wireless links with general aspects. At the end of this section we present AIRMAIL and TULIP as examples. Link layer protocols can be considered as the most obvious approach to mitigate the effects of high error rates on TCP performance because they fit naturally into the layered structure of network protocols, and because they deal with the error situation where it occurs. They have a much shorter control cycle than transport layer protocols and hence they can react on errors much more efficiently than TCP. In order to reduce the error rate, link layer protocols either use Forward Error Correction (FEC) schemes, Automatic Repeat Request (ARQ) schemes or combine both approaches (e.g. [26, 110]). FEC schemes can be used to correct a certain number of errors. The advantage of FEC is that it introduces an overhead that is independent of the instantaneous error-rate. Thus, the link layer protocol does not add to the RTT variance observed by the TCP sender. On the other hand the same overhead is incurred independent of the channel state, which wastes bandwidth during good channel states. In [60] Eckhardt et al. show how adaptive schemes can reduce the overhead. Since FEC is mostly deployed at the physical layer in mobile communication systems, we do not further discuss it here.

Link level ARQ schemes retransmit a packet on the local link if a loss is detected (e.g. negative acknowledgment, timeout). The advantage of link layer retransmissions in contrast to end-to-end retransmissions is that the link-layer can react faster to errors than the transport layer. It is easier to estimate the local round trip time if only a single link is involved. The link-layer receiver can also use negative acknowledgments to report losses upon reception of an out-of-order packet, because packets are not reordered on a single link. Further on the retransmission cost only occurs on the local link where the packet was lost and not on the end-to-end path. The latter is especially important if the packet has already traversed the bottleneck link. Another advantage of local error control compared to end-to-end error control is that the local entity

might be able to estimate the current channel state and hence is able to use more sophisticated mechanisms to handle bad channel states (e.g. [28]).

Unlike FEC, ARQ only produces overhead if a packet is retransmitted. This saves bandwidth during error free periods, but packet losses are translated into delay. Thus, in contrast to FEC schemes, ARQ schemes increase the delay variance seen by the TCP sender. Furthermore local error control mechanisms can potentially lead to three different adverse interactions with TCP's error control mechanism. First, the two reliable protocols on different layers set their retransmission timers independently, which might lead to redundant retransmissions. Second, if the link layer protocol does not preserve the packet order, a fast retransmission might be triggered due to duplicate acknowledgments. Third, a higher mean RTT and an increased variance might lead to a long and conservative retransmission timeout at the transport layer. This means that it takes longer for the transport protocol to react to congestion based losses.

Besides these interactions between TCP and link layer protocols, link layer protocols introduce two additional shortcomings. The first problem is called head of line blocking. If a reliable link layer protocol tries to deliver a packet in a bad channel state, all other packets (possibly destined for other receivers) are queued until this packet has been successfully delivered. Since the quality of the wireless channel depends on the location of the receiver, it is possible that some receivers are reachable while others are not. Thus, it is possible that packets destined for a different receiver or even shorter packets to the same receiver could be transmitted successfully. While the sender tries to deliver a packet over the bad channel, bandwidth and energy are wasted and the outgoing queue can overflow. One possible solution is link state dependent scheduling as described by Bhagwat et al.[28].

The second problem is that not all applications would benefit from a reliable link layer service. In the case of delay sensitive data (e.g. voice over IP) a packet loss is often more acceptable than additional delay due to excessive retransmissions. Unfortunately, the link layer cannot distinguish between TCP and UDP packets without looking into the IP header. A further differentiation (e.g. guessing the kind of application) is even more complicated. The IETF's PILC working group has summarized recommendations for the design of subnetworks in RFC 3150[126], RFC 3155[52], and RFC 3366[64].

One of the first papers that deals with the effects of combining error recovery on the transport layer and the link layer is [55]. The authors investigate the performance of reliable transport protocols over wireless LANs. They show that performance and efficiency (goodput over throughput) critically depend on the transport level retransmission timer. If the timeout value is too high, the system will suffer from poor performance, as errors are detected late. If the value is too small, the system will suffer from poor performance since too many unnecessary retransmissions waste scarce resources. They conclude that a link layer protocol should only be used if the error rate exceeds a certain threshold.

Since then different researchers have shown that spurious retransmissions are not really an issue. In [20] it is shown that interference is unlikely if the RTO granularity is large and the time required for link level retransmissions is small compared to TCP's RTO. As discussed in Section 2.2 TCP's minimum RTO value is twice the timer granularity and should be at least 1 second. For a 2 Mbit/s wireless LAN, an RTO value of 1 second will allow more than 150 retransmissions if the packet length is 1500 bytes, neglecting the access delay. In [113] it is pointed out that the timeout value for a segment is often larger than the RTO value. The retransmission timer is restarted if a new acknowledgment is received that does not acknowledge all outstanding packets. Therefore the next packets get an additional RTO before the timer goes off.

One example for a link layer protocol designed for wireless networks is *AIRMAIL*[11]. Although its design is not directly related to TCP, it has some interesting features that differentiate it from classical link layer protocols. AIRMAIL is asymmetric to reduce the processing load at the mobile host. Reliability is achieved by a combination of ARQ and FEC. Furthermore it supports handovers by transferring AIRMAIL-state from one access point to the next access point.

Another link layer protocol explicitly designed to improve TCP over wireless links is *TULIP*[132, 131]. TULIP is tailored for half-duplex radio links. Its timers rely on the maximum propagation delay over the

link rather than performing a RTT estimate of the channel delay. It does not provide a reliable service for TCP acknowledgments. The reasoning behind this is that subsequent cumulative acknowledgments supersede the information of the lost one. Although this is true regarding error control, lost acknowledgments negatively effect the transmission rate of the TCP sender. Packets are delivered in order. To achieve the reliable service, a continuous ARQ approach is used. The receiver sends a bit-vector with every acknowledgment informing the sender about missing packets.

IETF's Performance Implications of Link Characteristics (PILC) working group summarized the research results about link layer ARQ in RFC 3366[64]. This RFC states that a single TCP connection would benefit from persistent ARQ. However, if many flows share the same link, the additional delay might be an issue, and ARQ protocols with a reduced persistence might be preferable. Furthermore, the RFC points out that effects such as increased delays are cumulative. If flow classification is impossible, then persistent ARQ should not be used.

TCP add ons

Although link layer protocols are able to hide an error-prone link from a TCP sender, they are generally unable to guarantee in-order delivery of TCP segments. Since TCP repeats the last acknowledgment upon reception of an out of order segment, a large number of out of order segments can lead to three duplicate acknowledgments which in turn trigger the sender to enter Fast Retransmission Mode. Unfortunately the resulting retransmissions and reduction of the Congestion Window are spurious, since no packet was lost. In order to overcome this behavior, Vaidya et al. suggest the introduction of a delayed duplicate acknowledgment mechanism in [125, 159]. In this approach a TCP receiver delays the third and all later duplicate acknowledgments for an interval d. This interval gives the link layer protocol time to retransmit the missing packets. If the timer expires before all missing packets have been received, all duplicate acknowledgments will be sent in a burst. The drawback of this scheme is that the third duplicate acknowledgment is also delayed if losses are caused by congestion. Thus, the TCP sender does not recognize congestion based losses until the delayed dupack interval is over. This can lead to performance degradation. This approach is also unable to hide delay spikes from the TCP sender (e.g. long outage periods, reduction of bitrate). Thus, it is possible that TCP's retransmission timer expires, leading to spurious retransmissions. The authors also admit that the determination of the interval d is an open question.

The Eifel algorithm addresses the problems of spurious retransmissions[115] although the author of this paper considers this a rare event. The idea is that a link layer protocol is sufficient to deal with the peculiarities of the wireless link. Negative interactions between TCP's error control mechanism and the link layer protocol occur only during outages that are longer than the current RTO setting. In this case TCP spuriously times out and retransmits all unacknowledged packets. This leads to performing Slow Start as well as a Go-Back-N retransmission of all unacknowledged packets. To prevent this, the author suggests eliminating the acknowledgment ambiguity (the TCP sender cannot distinguish between an acknowledgment for the original packet and an acknowledgment for a retransmitted packet) by either the timestamp option or by introducing a flag. If the sender receives an acknowledgment that was triggered by the original packet during Slow Start, it will know that the retransmission was spurious. In this case the TCP sender restores the state it had prior to the timeout and continues with normal operation.

TCP-Probing by Tsaoussidis and Badr is another approach based on TCP options[157]. The goal of the authors is not only to improve performance but also to be more energy efficient. To achieve this goal, TCP-Probing will not enter Slow Start or Fast Recoveryif communication problems are detected. Instead a probe cycle is entered during which only probe packets are sent. The probe mode is abandoned after the sender was able to perform two successive RTT measurements. If the two RTT measurements are between the best RTT seen so far and the last RTT (before the probe cycle was started), immediate recovery will be applied. Otherwise the sender enters Slow Start regardless whether a timeout or three duplicate acknowledgments triggered the probe cycle.

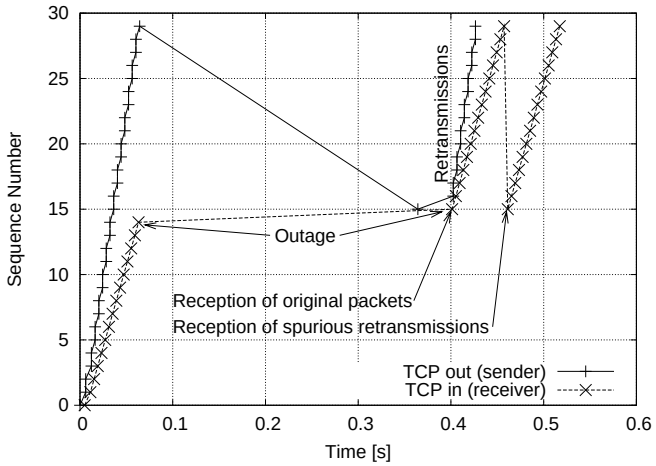


Figure 3.3.: Results of example simulation showing effects of a spurious timeout on the behavior of the TCP sender. The simulated scenario is a file transfer between two directly connected hosts. As file size we have chosen 30 KB arbitrarily. During the transfer we simulated an outage period. The figure shows that no TCP segment is lost (every segment sent is received by the receiver) and that, in spite of this, the TCP sender enters retransmit mode performing Go-Back-N.

In [100] Keshav and Morgan propose a new retransmission strategy called *Simple Method to Aid Retransmissions (SMART)*. SMART combines features of Go-Back-N and selective retransmission. In SMART every acknowledgment carries two numbers: a cumulative acknowledgment and the sequence number of the packet that triggered the transmission of the acknowledgment. Thus, the sender can deduce which packets were lost. Contrary to TCP, SMART follows an aggressive retransmission policy. This means that any out of order delivery is interpreted as a loss indication and hence followed by retransmissions. If the retransmission itself is lost, SMART will retransmit two copies of the missing packet after one RTT.

Proxy based solutions

This section summarizes approaches that require support by a network node in order to hide the peculiarities of the wireless network.

Protocol Booster

According to RFC 3135, protocol boosters are a special sub class of PEPs. RFC 3135 defines a protocol booster as a PEP variant that is transparent to the end systems and that neither modifies the syntax nor the semantics of the end-to-end protocol (e.g. TCP). A general description of protocol boosters as well as some examples can be found in [69].

A well known example of a protocol booster is the *Snoop* approach[21, 22]. It implements a TCP aware link layer protocol. The motivation behind Snoop is the observation that link layer protocols introduce additional overhead (e.g. protocol header) and often lead to out of order delivery of TCP segments. Since out of order segments can trigger Fast Recoveryat the sender (as already mentioned) they can impair to

TCP's performance. Snoop mitigates these two problems by reusing TCP's header format and messages, and by filtering duplicate acknowledgments if the missing packet was lost on the wireless hop.

Snoop does not require any modifications of the end systems[5] and only requires a soft-state at the access point. The heart of the Snoop approach is an agent that is placed at the access point. The operation of this agent depends on the traffic flow direction. For traffic from the Internet to a wireless terminal the Snoop agent looks at the TCP header and stores all segments for later recovery. If an acknowledgment flows back to the TCP sender, the Snoop-agent will clear its cache. If it receives a duplicate acknowledgment for a packet that is stored in its cache, it will discard the duplicate acknowledgment and retransmit the missing packet locally. After the first retransmission a retransmission timer is started. All further duplicate acknowledgments for this packet are discarded as long as the missing packet is in the cache. If a duplicate acknowledgment is received for a packet that is not cached, the duplicate acknowledgment will simply be forwarded to the TCP sender.

Obviously the caching approach is not helpful for the upstream direction. The Snoop proposal has two different mechanisms to improve data transfer from the wireless end system into the Internet. Either a negative acknowledgment approach (based on SACK) or an ELN scheme is used. In both cases the access point is responsible for examining the traffic flow from the wireless end system in order to detect gaps in the flow. Packets that are lost on the wireless link are noted in a list. If the ELN approach is used and a duplicate acknowledgment indicates a loss of a segment that is on the list, the Snoop agent will set an ELN bit in the TCP header. If the TCP instance at the wireless host receives such an acknowledgment, it will retransmit the packet without performing any congestion control mechanism. The advantage of this is that only a soft-state is required at the access point. The disadvantage is that the mobile host needs an entire RTT to react to packet loss on the local wireless link. If the negative acknowledgment scheme is used, the Snoop agent at the access point sends an SACK if it detects a gap in the traffic flow. Although the access point inserts a TCP segment (namely an SACK) into a connection between two end systems, this does not violate the end-to-end semantics, since SACKS are only advisory (see Section 2.2).

The major drawback of Snoop is the layer violation. The link layer protocol has to be able to interpret and generate TCP segments. Furthermore Snoop is unable to shield the TCP sender completely from the peculiarities of the wireless link. Thus, the TCP sender sees a higher and more variable RTT which might lead to spurious retransmissions. Snoop is also unable to support data transfer from the wireless host without modifications of the wireless end system.

The *WTCP* approach by Ratman et al. performs local recovery similar to the Snoop approach [144, 145]. Like Snoop, WTCP maintains TCP's end-to-end semantics, since acknowledgments are sent by the TCP receiver. The difference between Snoop and WTCP is that WTCP uses a more aggressive retransmission strategy and that it hides the delay introduced by the local recovery from the TCP sender. WTCP hides the delay variation as well as the delay introduced by the wireless network from the TCP sender. Thus, the RTT estimation of the TCP sender does not include the wireless link. The advantage of this scheme is that the TCP sender can react faster to congestion caused losses. The drawback is that the TCP sender is more susceptible to spurious retransmissions since the delay and variance of the last hop are not included in the RTO calculation, although present in the network path. WTCP requires that both TCP sender and TCP receiver support the TCP timestamp option. An agent at the base-station modifies the timestamps in a way that the delay introduced by the wireless network is not included. As most approaches, WTCP only deals with the data path from the fixed host to the wireless host.

[5] This is only true for downstream communication. For upstream the wireless end system must support an Explicit Loss Notification (ELN) -bit or the TCP SACK extension.

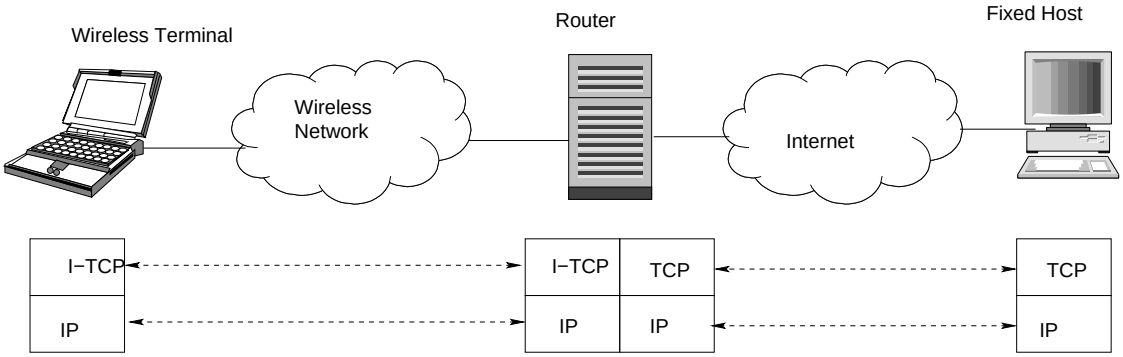


Figure 3.4.: The principle of transport protocol splitting

Split Connection approaches

A split connection approach is an approach that splits the end-to-end path into at least two parts. According to the OSI reference model, the cut can be made at any layer above the network layer. However, in general the transport layer is used to split a connection.

The idea of splitting the end-to-end path to improve the quality of wireless Internet access was first mentioned in [12]. In this paper the authors discuss how wireless networks benefit from a split at different layers. They argue that mobility must be made explicit to as many protocol layers as needed. At the transport layer the split is useful to hide the error-prone link from the sender and by keeping the end-to-end RTT small. At the session layer the switching point should be used to perform tasks like service look-up and registration after a handover on behalf of the wireless client. A split at the presentation layer could be used to encrypt data in the case of an untrusted territory. And finally at the application layer, optimized versions of the application level protocol could be used. Later the Indirect TCP (I-TCP) approach was invented by the same group[13, 14]. Figure 3.4 shows the general concept behind split connection approaches.

The advantage of Indirect transport protocols are:

1. Separation of protocol mechanisms like flow control, congestion control and error control of two networks with vastly different characteristics.
2. A separate transport protocol for the access link can be tailored to the peculiarities of that link. This can include specific notifications like disconnection or registration with a new base-station, or freezing timers during outage periods.
3. Different classes of reliability can be supported. For example a semi-reliable service for applications that would use an unreliable protocol over the Internet, but would suffer from the error-prone wireless link.
4. Local Recovery of errors results in faster recovery as it is the case with link layer approaches.
5. Backward compatibility with the existing wired network protocols. Thus, the protocol stack at unrelated fixed hosts does not need to be modified.
6. Allow the access point to manage most of the communication overhead for a mobile host.

The major drawback of indirect approaches is that they change the end-to-end principle of the Internet. It is possible that the data receiver receives an acknowledgment before the corresponding data packet has reached its final destination. Furthermore they require maintaining a hard-state at the intermediate station.

Thus, split connection approaches are more susceptible to system failure (violate the fate-sharing principle of the Internet) and introduce additional processing and resource overhead that can lead to scalability problems if a high number of connections are active. For mobility they might require the transfer of the hard-state from access point to access point while a host is moving. Furthermore they either do not work in conjunction with IP security or must be part of a trusted entity.

In I-TCP a connection between two communicating applications is established by two independent TCP connections – one over the wireless medium and one over the fixed network. A special network node called *Mobile Support Router (MSR)* relays data between these two connections. If a wireless end systems changes its current cell, the MSR will also be changed. For the fixed host the existence of a second connection is completely transparent. The transport protocol connection between the wireless terminal and the MSR is established by a tuned TCP variant. This variant resets the congestion related parameters of both endpoints of the wireless link upon a handover.

In [169] Yavatkar et al. pursue the same idea of two independent transport protocol connections. However, contrary to I-TCP they compare two different transport protocols for the connection between MSR and wireless terminal. On one hand they use MTCP, a TCP variant like I-TCP, and on the other hand they use a selective repeat protocol tailored for the wireless link. The authors show that both approaches outperform an end-to-end TCP connection. They also show that it is worth investing in the design of a new tailored protocol since their selective repeat protocol performs better than the TCP variant.

Haas introduces a split connection approach called *Mobile-TCP*[80]. The difference to I-TCP is that he uses a tailored transport protocol over the wireless link that is based on the assumption that the protocol between MSR (called Mobile Gateway) and wireless terminal only covers a single hop. Similar to AIRMAIL, this protocol is asymmetric, trying to reduce processing load of the wireless end system. Mobile-TCP uses smaller headers over the wireless hop and has no congestion control mechanisms. Error control is different for both directions. Go-Back-N is used for retransmissions from the MSR to the wireless end systems because this simplifies the processing at the end-system. Selective repeat is used for the opposite direction. The usage of Go-Back-N is justified by the observation that the single hop network cannot buffer many packets and that if packets are lost, the probability will be high that multiple packets are lost in a burst. Retransmissions are directly triggered if an out-of-order packet is received. The wireless end system does not use timers for error control. All timers are located at the MSR. The MSR has two timers on behalf of a wireless terminal. The first timer is responsible for protecting negative acknowledgments. Whenever the MSR receives an out-of-order packet, it sends a negative acknowledgment and starts the timer. If it does not receive the missing packet during this timer interval, it will retransmit the negative acknowledgment and restart the timer. The second timer is a persistence timer. If the MSR has not received data from a wireless end system during this interval, it will poll the end system. The flow control between the end systems is simply achieved by an ON/OFF scheme. If the amount of data in the receive buffer exceeds a certain threshold, the end system is stopped. If the amount of data falls below a low threshold the wireless terminal is restarted. The flow control in the opposite direction is performed by the MSR. The wireless end system sends information about its receive buffer with every packet using only two bits. The MSR estimates the available buffer space using its knowledge of the link characteristics and the amount of data it has sent during the last RTT.

Another difference between I-TCP and Mobile-TCP is that the Mobile-TCP maintains TCP's end-to-end semantics. This is achieved by delaying acknowledgments until the data packets have in fact been received and acknowledged by the wireless end system. However, this feature eliminates the decoupling of the two worlds, which is one of the major advantages of split connection approaches.

In [162] a *Mobile-End Transport Protocol (METP)* based on the split connection idea is suggested. Similar to the previous solutions it is based on a simple protocol that reduces the protocol overhead (e.g. smaller headers). METP is tuned to operate on top of 802.11 wireless LANs. After a packet is transmitted, it waits for the reception of an immediate acknowledgment before the next packet is transmitted. METP

will only retransmit a packet if the MAC fails to deliver it. In addition, a coarse-grained acknowledgment scheme is supported to implement flow control. Furthermore METP does not only support TCP but also UDP.

M-TCP by Brown et al. is another split connection approach that maintains TCP's end-to-end semantics[42]. Brown follows the idea of a three level hierarchy where multiple cells (base-stations) are managed by a supervisor host[41]. M-TCP does not operate at the base-station (called Mobile Support Station), but at the supervisor host. Thus, the number of required context transfers will be significantly reduced if the mobile switches cells. Besides M-TCP the architecture also supports UDP[40]. The idea behind M-TCP is not to overcome the problem of high error rates (which is not an issue in the opinion of the authors of the paper) but to deal with frequent and possibly long periods of disconnection. In M-TCP, acknowledgments are sent by the mobile host. The supervisor host forwards the acknowledgment, but reduces the acknowledgment number by one. This allows the supervisor host to put the TCP sender into persist mode by acknowledging the final byte and advertising a zero window size during disconnection periods[6]. After the mobile host is reconnected, a duplicate acknowledgment is used to increase the Advertised Window. M-TCP requires modifications at the supervisor host and at the mobile end system. Another feature of M-TCP is data compression if the bandwidth of the wireless link is not sufficient for handling the data. This approach has several drawbacks: The TCP sender might need to re-segment its packets, complicating the processing. TCP sender and receiver have a different view of the TCP state (the sender receives acknowledgments that the receiver has never sent), and the RTT estimation of the TCP sender includes the disconnect times of the wireless links, which can lead to long idle periods after congestion based losses. Furthermore shrinking the window leads to the retransmission of all packets that are moved out of the window. This creates three problems. First, bandwidth is wasted, second, TCP sends these packets in a burst that might overwhelm a router in the path to the mobile host, and finally the network condition might change during the disconnected time. Therefore Slow Start would be the appropriate behavior. Besides this, the supervisor host must decide when to shrink the window. The window shrink messages should not be sent too often since this would lead to unnecessary retransmissions if the window is re-opened. The shrink message also must not be sent too late since this would result in retransmissions followed by Slow Start. The transmission of a shrink message is triggered by a timer bound to the RTO estimate of the TCP sender. Since the supervisor hot does not know this estimate it uses its own RTT estimate. This approach has two shortcomings. It assumes that the network path is symmetrical for both directions and that data flows in both directions.

In [101, 4, 102] Kojo et al. describe a new architecture for connecting mobile end systems to the Internet via GSM called *Mobile Office Workstations using GSM Links (Mowgli)*. Mowgli follows the split connection approach and replaces the client-server paradigm by a client-mediator-server paradigm. In Mowgli the mediator is called Mobile Connection Host. In contrary to the split connection approaches discussed so far, Mowgli does not split the transport protocol. It is a socket level solution in which the MCH is a socket level gateway. It introduces a new communication architecture that includes all protocol layers. The *Mowgli Socket* is the core of the architecture. A Mowgli Socket extends the classical Berkeley socket interface. All function calls to the interface are encapsulated at the mobile node and are transferred to the MCH where they are executed. Since Mowgli Sockets are compatible with Berkeley sockets, all legacy applications can operate on top of Mowgli without modifications. Unfortunately no details are given how the different socket calls are implemented. Details are only given for the `connect`-call. In contrast to the socket `connect`-call, data can be transferred before the connection is established in order to hide the delay of the GSM link. New services introduced by the Mowgli Socket for example are priorities and automatic recovery from unexpected link-level disconnections. Each Mowgli Socket is implemented by an agent-proxy pair. The agent is located at the end system and the proxy is located at the MCH. A master agent intercepts all BSD-socket calls and forwards them to a master proxy at the MCH. Then either a

[6] RFC 1122 requires that window shrinking is accompanied by a new acknowledgment.

generic agent-proxy pair (for legacy applications) or an application specific agent-proxy pair is created. This pair is mainly responsible for data relaying. The service can be arbitrarily extended for application specific agent-proxy pairs, e.g. to provide data compression. Or it can be customized for specific applications (e.g. see [109] for wireless web browsing in Mowgli). The MCH dynamically allocates a virtual network interface for each mobile node connected to it and assigns the IP address of the mobile to this interface.

Agent and proxy use the service of the *Mowgli data channel service (MDCS)* to communicate with each other. The MDCS is a transport service tailored for GSM transparently replacing standard TCP/IP protocols in the access network. The MDCS is the Mowgli component responsible for the performance of the architecture. The major design goal of MDCS is the support of different parallel flows over the slow wireless (GSM) link and to hide temporary disconnections. Therefore it provides priority based multiplexing of data over the wireless link. In order to hide link breaks Mowgli supports automatic and transparent reconnection. If the link level connection breaks and is reestablished later, the mobile node will be able to decide whether to reestablish the old session (including old IP address) or a new session. In the first case, synchronization packets must be sent for all open channels to inform the peer about the last acknowledged byte and so on. Furthermore a number of flags allows the application to fine tune its usage of the wireless link (e.g. time until disconnect).

The MDCP implements the service of the MDCS. It supports two types of packets: control packets and data packets, whereas the control packets have a higher priority than data packets. In order to save the scarce wireless bandwidth, MDCP makes economical use of control information. It has a small header size and minimizes the number of acknowledgments. It can operate in two modes. In default mode the MDCP expects a reliable service. In this mode it makes no attempt to detect or correct lost frames. In the second mode, called error monitoring mode, the MDCP provides a reliable channel, however assuming that the underlying protocols only provide a moderate error rate. A third mode called window recovery mode based on selective acknowledgment is also mentioned but not further discussed. For both modes acknowledgments are only used as permits to implement a credit based flow control. Permits are rarely sent, hence they always open the window by several packets.

3.3.2. Solutions Based on Determining the Loss Cause

If TCP knew the cause of a packet loss, it could react accordingly (e.g. not reduce the Congestion Window in the case of non congestion based losses). In order to inform the transport protocol (here: TCP) about the state of the network, explicit notifications could be introduced into the network or link layer.

Generally, explicit notification schemes vary in what to signal (e.g. congestion or loss), who sends the notification, how to know if a notification should be sent and how the sender should react upon a notification. Most schemes based on explicit notification have to be supported by the network in order to detect the cause of a loss. One example of an approach not based on a PEP is TCP-Hack (see below). Almost all schemes require the sender to be modified since it must be able to react properly upon the notification. One approach that does not require a modification of the TCP sender is to manipulate the Advertised Window field of the TCP header when a packet passes a router. This approach is for example used by M-TCP discussed above and by Freeze-TCP discussed below. For some schemes it is necessary to modify the TCP receiver.

End-to-End solutions

The idea behind *TCP-HACK* is to enable the TCP sender to distinguish between congestion and corruption caused losses[23]. To achieve this, two new TCP options are introduced. The first option is used to include a checksum that is calculated over the TCP header and the IP pseudo header contained in the TCP header. If the TCP receiver receives a data segment with an invalid checksum but correct header checksum (in the

option field) it is possible to determine the correct TCP connection and a negative acknowledgment can be sent to the sender, indicating that the corresponding packet is lost due to corruption. The second option is used to implement a negative acknowledgment. A TCP segment with this option set is interpreted by the TCP sender as a negative acknowledgment. This option includes the sequence number of the packet that triggered the transmission of the negative acknowledgment. If the TCP sender receives a negative acknowledgment, it will retransmit the missing packet without further processing the acknowledgment (e.g. entering Slow Start). This proposal requires modification of TCP sender and receiver. To avoid this, the authors propose use of TCP-tunnels over error prone networks to make the use of TCP-HACK transparent to the communicating end systems. Furthermore they assume that corrupted packets are delivered to the receiving TCP instance and are not discarded by the hardware as it normally is the case. Last but not least the negative acknowledgment must reach the TCP sender, which might be not the case (e.g. bad channel state, congestion losses).

The *Freeze-TCP* approach by Goff et al. is a pro-active solution[78]. The idea behind Freeze-TCP is to signal an impending disconnection to the TCP sender. The authors assume that a wireless terminal can monitor the signal strength and predict periods of disconnection (especially handoffs). To signal the disconnection to the data sender, Freeze-TCP utilizes TCP's possibility to shrink the Advertised Window to zero, although this behavior is discouraged by RFC 1122[36]. If a TCP sender receives an Advertised Window of zero, it should freeze all retransmit timers and should enter a persist mode. In this mode the TCP sender sends probe packets until the window opens up again. Since the probe interval is exponentially increased until it reaches one minute, the TCP receiver sends three duplicate acknowledgments when the disconnection period is over, as proposed in [43]. Since the retransmission timers are frozen, the congestion mechanisms of TCP are not triggered during disconnection. Thus, in Freeze-TCP the TCP sender can continue to send with the old state after a disconnect period. The advantage of this scheme is that it is in fact a real end-to-end solution requiring modifications only at the wireless end system. For handovers with a long disconnection time (compared to TCP RTT) an improved performance can be expected compared to standard TCP. One drawback of this scheme is that, depending on the environment, Slow Start would be the correct TCP reaction to a handoff, since the network properties might have changed. A second problem is that the client has to be able to forecast the outage period. Furthermore Freeze-TCP only considers the data path to the wireless station.

Biaz and Vaidya investigate whether heuristics could be used to guess the cause of a packet loss. In [31] they used simple statistics on RTT and throughput as loss predictors. The predictors are based on congestion avoidance techniques found in TCP-Vegas[37], introduced by Wang et al.[163] and Jain[93] respectively. Whenever these algorithms would suggest to reduce the Congestion Window, the loss predictor deduces that the next loss is congestion based. A simulation based study has shown that these predictors do not perform well.

In [30] Biaz and Vaidya investigate schemes that let the receiver decide on the cause of a loss. They consider a fixed host sending data to a wireless host. In addition they assume that only the last hop is the bottleneck and error-prone. Thus, packets are queued at the access point and are sent back to back on the wireless link. The back to back characteristic is the key for the heuristic. If the inter-packet gap is larger than the transmission time of a packet, a wireless loss is deduced. If the inter-packet gap is small but the packets are not in order, a congestion based loss is deduced. Via simulation the authors show that this simple heuristic leads to results that are nearly identical to an ideal TCP sender, that (artificially) knows the loss cause. This is especially true if the loss rate is small. Unfortunately this scheme is not feasible for a shared medium or if different flows share the same outgoing queue.

Although TCP-Westwood[120] does not really try to determine the loss cause, it is discussed here because it implicitly makes assumptions about the loss cause. It only requires modifications of the TCP sender and is similar to TCP-Vegas, based on a new algorithm to compute the Congestion Window. It continuously

measures the bitrate used by a connection at the TCP-sender via monitoring the rate of returning acknowledgments. A low-pass filter is used to average sampled measurements. If the sender assumes a loss, this estimate is used to calculate the new values of the Slow Start Thresholdand the Congestion Window without changing the fundamental Congestion Window dynamics during Slow Start and Congestion Avoidance. The slow start threshold is not set to half of the Congestion Window but to $\frac{\mathrm{BitrateEstimate} \times \mathrm{RTTMin}}{\mathrm{SegSize}}$. If the loss is detected by the reception of three duplicate acknowledgments, then the Congestion Window is set to $\min(\mathrm{cwnd}, \mathrm{ssthresh})$. If the loss is detected by a timeout, then the Congestion Window is set to 1. Since the estimation of the sending rate is not sensitive to random loss (if the loss rate is not too high), the Congestion Window and the Slow Start Thresholdare not necessarily decreased for losses due to transmission errors.

Proxy Based Solutions

Explicit Notification

Two fundamentally different types of explicit notification schemes exist. Either the occurrence of congestion is signaled or a packet loss caused by transmission errors is indicated. The first is not related to wireless Internet access, but a mechanism to improve congestion control in the Internet. Today, *Explicit Congestion Notification (ECN)* [72, 143] is deployed in the Internet. Hence it could be assumed that ECN is sufficient to determine the cause of a packet loss. If a TCP sender detects a loss and has not received a congestion notification, it is able to conclude that the loss was caused by bit errors. Unfortunately this conclusion is wrong since the congestion notification might have been lost as well. Furthermore not all routers support ECN. Thus, ECN does not help to improve wireless Internet access but explicit loss notifications are needed.

In contrast to ECN, which is implemented at the network layer and the transport layer, some of the explicit loss notification schemes are placed at the link layer. Approaches that are based on an explicit loss notification are *ELN* [17], *Partial Acknowledgment[29]*, *Explicit Transport Error Notification (ETEN)[105]* and *Explicit Bad State Notification[16]*.

ELN complements the Snoop approach (see Section 3.3.1). It is used to improve the performance of data transfers originated at the wireless host. The Snoop agent maintains a list of missing packets of the flow from the wireless host to the fixed host by inspecting TCP header fields. If duplicate acknowledgments sent by the fixed host traverse this agent and the missing packet is included in the list, the agent sets an ELN-bit in the TCP header. Upon reception of a duplicate acknowledgment with the ELN-bit the TCP instance at the wireless host retransmits the missing packet without reducing the congestion control variables.

The Partial Acknowledgment approach by Biaz et al. [29] tries to improve the performance for the downstream direction by introducing a *partial acknowledgment* as a TCP option. A partial acknowledgment is sent by a base-station and carries information about which packets have reached the base-station and which packets are missing (i.e. lost due to congestion). The reception of a partial acknowledgment is treated by the TCP sender as an explicit notification, but not as a regular TCP acknowledgment that would advance the window. The partial acknowledgment informs the TCP sender that the access point has taken over the responsibility to deliver the acknowledged packets. Upon the reception of a partial acknowledgment the TCP sender increases the retransmission timeout in order to avoid spurious timeouts. If the channel state is bad, the packets are buffered at the base station and are locally retransmitted. This scheme requires modifications at the base-station and at the TCP sender.

Bakshi et al. propose a similar approach in [16]. They suggest resetting the TCP retransmission timer when the wireless channel is in a bad state. For this purpose the base-station has to send an explicit bad state notification to the TCP sender if it was unable to deliver a packet to the wireless end system after a certain number of retransmissions. The proposed scheme works together with a reliable link layer protocol and avoids unnecessary retransmissions by delaying the retransmission timeout at the sender.

In [105] Krishnan et al. investigate the general performance gain due to Explicit Transport Error Notification (ETEN) schemes. Contrary to the ELN approach discussed above, lost packets are not detected by gaps in the sequence number space, but by the reception of corrupted packets. The authors assume that the lower layers deliver corrupted packets and that it is possible to recover sufficient information to determine the connection a packet belongs to. Since this assumption is very unlikely to hold, the authors introduce cumulative ETEN (CETEN). In CETEN cumulative error rates over various flows are observed instead of the separate packets. The observed error rate is signaled to the sender, which heuristically guesses the reason for lost packets. The authors looked at three different notification schemes, namely an optimal approach (called Oracle-ETEN), a backward ETEN scheme and a Forward ETEN scheme. Oracle-ETEN assumes every loss to be detected and instantaneously reported to the TCP sender. In Backward ETEN the router directly sends the notification to the TCP sender. In Forward ETEN the loss information is conveyed to the TCP receiver, which sends the information to the TCP sender. The presented simulation results show that the Oracle TCP goodput is up to seven times higher than the TCP Reno throughput in the case of a single, uncongested link. As long as the error rate is not prevalent, the Backward ETEN scheme achieves nearly the same performance. With increasing error rate the gain by Backward ETEN diminishes, since the notification packets also are lost. If congestion based losses are included in the simulation, the performance gain will be insignificant. The cumulative schemes did not reach the performance of the per packet schemes, but are more attractive to implement.

In [164] Wenqing et al. suggest replacing the SNOOP agent (see Section 3.3.1) with an ELN-ACK agent. The goal of this approach is to improve the performance of the data path between a fixed host and a wireless host. The scheme requires modifications to the TCP receiver, the TCP sender, and the access point. The agent that is located at the access point maintains a list of packets that were lost in the Internet. The TCP receiver sends an ELN-ACK that contains information about the newest missing segment. If the access point sees the last missing packet, it assumes that it was lost on the wireless link due to bit errors and therefore clear the ELN bit of the ELN-ACK. If the TCP sender receives an ELN-ACK with a cleared ELN bit, it retransmits the corresponding packet without taking congestion control actions.

SYNDROME by Chen et al. uses explicit loss notification sent by the destination TCP instance[47]. To determine whether a packet was lost in the wired network or the wireless network, the authors use the following approach: The base-station counts all packets per flow that it forwards to the wireless end system. The counter is attached to every segment. The TCP instance at the wireless end system uses the sequence number, the counter, and a set of lemmas to determine the cause of a packet loss, if any. If it detects a loss caused by corruption it sends an explicit loss notification to the TCP sender. This scheme requires modifications at both TCP sides, albeit backward compatibility is achieved at the base-station.

Cobb et al. [50] propose that the last hop router (e.g. access point) sends a special acknowledgment whenever it receives a TCP segment destined for a wireless host. The meaning of this acknowledgment is that the segment has reached the end of the wired (error-free) network but not its final destination. If the TCP sender does not receive a standard TCP acknowledgment for a packet, it deduces that the packet was lost due to corruption. The same mechanism is applied if the wireless end system sends data, but this time the first router sends the special acknowledgment. If the sender does not receive this acknowledgment, it assumes a corruption loss and retransmits the packet.

3.3.3. Further Approaches

Ludwig et al. investigate the performance of TCP over GSM in various papers[112, 116, 113, 114]. In [112] they investigate whether the fixed frame size chosen for RLP is optimal and how much performance could be gained by applying an adaptive frame length scheme. To perform this investigation, the authors have collected over 500 minutes of block erasure traces. The collected traces reflect three scenarios, namely stationary with good receiver signal strength, stationary with bad receiver signal strength and mobile. The

obtained results show that the performance can be increased by up to 25% if the frame size is increased.

In [16] Bakshi et al. show that the packet size has effects on the end-to-end throughput. They point out that wireless networks often have a smaller packet size than wired networks. In this case either the smallest MTU is used for the entire end-to-end path, or the link layer protocol must fragment packets. In the first case the overhead is increased, while in the second case a drop of a single fragment requires the retransmission of the complete packet. The authors present a performance evaluation that shows that the throughput can be improved by 30% if the optimal packet size is chosen.

In [150] Samaraweera and Fairhurst point out that TCP has no appropriate mechanism to retransmit already retransmitted packets. Second and subsequent retransmissions of the same packet are usually triggered by a timeout, whereas the value of the retransmission timeout increases exponentially. The authors therefore suggest the *Retransmission Packet Loss Detection* algorithm. The algorithm records a timestamp for each packet sent during Fast Recovery. If a packet with a newer timestamp is acknowledged by a selective acknowledgment three times the algorithm deduces that the other packets were lost and retransmits them.

In [57] Dutta et al. propose a proxy based architecture for networks with variable bandwidth. The authors show via simulation that TCP is not able to handle oscillating bandwidth appropriately and suggest employing a proxy at the edge router to the TCP-unfriendly network. This router is responsible for generating premature[7] acknowledgments if the rate of TCP is slower than the available bandwidth and to delay acknowledgments if the sending rate is slower than the available bandwidth. A simulation based performance comparison shows that the proxy improves the performance.

MSOCKS by Maltz et al. is an architecture for Transport Layer Mobility[118]. The idea behind this architecture is to hide the movement of the wireless end system from the corresponding host and to allow the wireless end systems to use an arbitrary network interface. The authors anticipate that mobile end system have multiple network interfaces in order to make use of overlay networks. Especially, it should be possible to change the interface for active connections, for example for handoffs either between cells of the same technology or different technologies. To achieve this, the authors use a proxy that must be implemented at a top level router of the access network. The connection between the wireless end system and the corresponding host is split into two connections. A special module located at the same machine as the proxy splices the two TCP connections together so that the end-to-end semantics are maintained. This module is called *TCP-Splicer*. TCP splicing has been developed by the authors to improve the performance of application layer proxies[119]. Normally application layer proxies split a single connection into two connections. This requires the data to be processed twice by the transport protocol at the proxy machine and the data to be copied from kernel to user space and back. To overcome this performance penalty, the authors suggest splicing the two connections together within the kernel. As a result, the communicating end systems see a single connection and no transport protocol processing is required at the proxy. Since the transport protocol messages are exchanged end-to-end, the semantics of the transport protocol are preserved. In order to splice two connections together, the splicing module must modify the headers of incoming TCP segments.

The protocol used for the communication between the wireless end system and the proxy is based on the SOCKS protocol which is defined by RFC 1928[108]. MSOCKS extends SOCKS only by a connection identifier and a reconnect message. A module at the wireless client intercepts socket calls to TCP and replaces the socket calls by MSOCKS calls. Since the intention of MSOCKS is not to improve performance but to enable mobility, the authors provide no performance comparison with other approaches. However, they show via experiments that the overhead due to the proxy and the splicing technique is small enough to serve a high number of end systems as long as bit rates of only several megabits are used.

SOCKS[8] provides a framework that allows application level protocols to transparently and securely

[7] An acknowledgment sent by the proxy before the data has reached the receiver.

[8] We discuss SOCKS here, although it was not designed to overcome the problems introduced by wireless networks.

traverse a firewall. Firewalls are systems that effectively isolate the internal network structure of an organization from an exterior network. To use SOCKS, a client must establish a TCP connection to a SOCKS server. After the authentication is completed it must send a relay request, including the destination address of the server the client wants to connect to. The SOCKS server can either accept or deny the relay request. If it accepts it, it establishes the TCP connection to the server on behalf of the client and send a reply message to the client. Usually the creation of the TCP connection to the SOCKS server and the relay request are hidden from the application by a shim-library. This library intercepts the `connect`-call from the application and replaces it by a `connect`-call to the SOCKS server followed by relay request.

Another approach to improve the performance of TCP over slow (wireless) links is to reduce the protocol overhead by applying header compression[89, 53, 54][9]. Header compression is motivated by the observation that most fields of the TCP/IP header are static and that the dynamic fields can be coded as the difference to the previous header (*delta encoding*). Utilizing this knowledge, the TCP header can be reduced down to three to six bytes. The problem of delta encoding is that the compression state stored at the decompresser changes with every header. Thus, in error prone environments it is very likely that the compressor and decompressor have inconsistent states due to packet loss. In this case the TCP checksum algorithm detects the bad decompressed packet and drops it (in the case of a packet with a wrong checksum no acknowledgment is sent). Later, if the TCP sender retransmits the packet, the compressor detects the retransmission (by peeking into TCP segments) and resynchronizes the decompressor by sending a full TCP header. To remedy this performance penalty, Degermark et al. proposed two algorithms[53]. The *twice algorithm* assumes that the states are inconsistent due to a single packet loss and that all packets are full sized. Therefore it applies the delta twice and tests whether the TCP checksum is now correct. If the *twice algorithm* fails, a full header request is sent to the compressor. However, Degermark proposes to only use this algorithm for the acknowledgment stream.

3.3.4. Discussion and Comparison

As the discussion in the previous sections show, many different kinds of solutions exist to overcome TCP's performance problems with error prone links. All of these solutions are able to improve TCP's performance. On the other hand many of these solutions require changes to the existing TCP/IP protocol suite, which makes them unfeasible for broad deployment. In addition, most solutions only deal with the downstream direction.

All solutions that optimize TCP performance over error prone links with the costs of TCP's congestion control mechanisms will hardly survive. Solutions that require modifications to all Internet hosts pose the difficulty to change all existing TCP implementations and to update all installations. Modifications of existing TCP implementations are only feasible if they can be done incrementally and if they are backwards compatible with legacy implementations. TCP-Eifel is one candidate that might be deployed.

Recent investigations show that a reliable link layer protocol is in most cases sufficient to overcome TCP's performance problems. Hence, the current trend is to use a reliable link layer protocol, possibly in addition with the Eifel algorithm. Unfortunately the deployment of reliable link layer protocols requires the possibility of distinguishing flows at the link layer. This is necessary since not all applications benefit from a reliable link layer. Currently flow classification is impossible without analyzing higher layer protocol headers or even the payload. This introduces a layer violation and eliminates the advantage that link layer protocols can be used in conjunction with IPSec. Furthermore link layer protocols can only hide an error-prone link from the TCP instances. In the case of long outage periods (longer than the current RTO setting) or in the case of an oscillating bitrate, link layer protocols are unable to hide the peculiarities of the wireless network from the TCP sender.

It is however on the idea similar to our approach introduced in Section 5.

[9] [53] also discusses UDP and IPv6 header compression.

Proxy based solutions offer the highest flexibility. They are in general able to support both data flow directions and can hide the wireless link completely from TCP. Furthermore most proxy based solutions do not require a modification to the TCP implementation at the distant (corresponding) host. Unfortunately these benefits come along with the costs that PEPs are in conflict with the end-to-end principle of the Internet. Hence, a careful investigation whether the performance gain outweighs the loss of the end-to-end principle is required. Even if the performance gain by a proxy is convincing, not all users will like the idea of using a PEP for all applications. Therefore it is necessary that a user can decide whether a PEP should be used or not.

From the existing PEP approaches those that terminate the TCP connection at the proxy in our opinion are the most promising. All other proxies have the drawback that they either only support the downstream direction, that they require modifications of the TCP sender (at the distant host), or that they are not able to shield the distant host from the peculiarities of the wireless network under all circumstances.

Current PEP approaches that terminate TCP connections within the network lack a clear description how this effects the semantics seen by the application. Most approaches split the end-to-end connection into two halves and admit that this breaks TCP's end-to-end semantics without regarding the meaning of a TCP acknowledgment or the semantics seen by the application. (M)SOCKs and Mowgli follow a different approach. Here the split is made at the interface level. Function calls to the socket interface are intercepted and transferred to the device where the TCP connection is terminated. However, (M)SOCKS is concerned with security and mobility issues and does not deal with performance issues. Mowgli concentrates on improving the performance of Internet access over GSM rather than dealing with the exact semantics of the different socket calls.

We believe that the interface between the application and the protocol stack is what matters. If the syntax and the semantics of the interface are maintained, most applications will accept that the TCP acknowledgments are no longer exchanged between the end systems at which the applications are located. This is even proposed by the TCP RFC. On page two RFC 703 says: 'The TCP specification describes an interface to the higher level protocols which appears to be implementable even for the front-end case, as long as a suitable host-to-front end protocol is implemented'. Therefore in Chapter 5 we introduce our Remote Socket Architecture, which is based on a careful analysis on the semantics of the Berkeley socket interface. The design of ReSoA is driven by the semantics of the socket interface and not by the requirements of a specific technology as it is the case with the Mowgli approach.

Chapter 4.

Selected Network Programming Issues

4.1. BSD Socket Interface

This chapter provides details about the socket interface that are required to understand the design of our ReSoA approach presented in Chapter 5. It is not a tutorial about socket programming (details about network programming can be found in [154]). Since the semantics of the socket interface are a key issue in the design of ReSoA, we focus on semantic issues in this section. For this reason every socket call is discussed separately concentrating on its semantics.

Sockets provide an addressing and protocol independent interface to the services of the protocol stack and offer a logical connection to applications for connectionless protocols. A socket corresponds to an SAP in the ISO/OSI terminology. It represents one end of a communication path and holds or points to the information associated with this connection. This information includes the protocol to use, state information for the protocol including source and destination addresses, queues of arriving connections, data buffers, and option flags. Networking connections represented by sockets are accessed like files through a descriptor that normally is a small integer. The networking and filesystem facilities are integrated in Unix systems. Therefore the classical I/O system calls such as `read` and `write` can be used. An abstract view at the different components of a socket as well as its interfaces is shown in Figure 4.1. This figure omits many details. Basically the socket interface is nothing else than a translator that maps protocol independent function calls to protocol specific service primitives. This mapping is created whenever a socket is created. Some state is required for a proper communication with the attached protocol. For example, the socket object decides when a service primitive of the protocol can be called. Although Figure 4.1 shows a send and receive buffer, the buffers are usually shared by the attached protocol (see the discussion of sending and receiving data for further details).

Table 4.1 shows some information associated with each socket. However, not all attributes are presented. We omit, for example, control structures that are used to queue incomplete connections, connections ready to be accepted, and the relation of sockets derived from a socket that is prepared to accept new connections. These details are discussed later when we deal with socket calls that are related to connection management.

The *type* attribute of a socket specifies the communication semantics to be supported by the socket and its associated protocol. The types *SOCK_STREAM* and *SOCK_DGRAM* are especially of interest regarding the Internet protocol suite. For each type only a single protocol is currently available within the Internet protocol family currently. TCP[142] is the protocol for the `SOCK_STREAM` class and UDP[140] is the protocol of the `SOCK_DGRAM` class. The discussion in this thesis focuses on the `SOCK_STREAM` class because it deals with TCP over wireless networks.

The *option* attribute summarizes a collection of flags that describe the behavior of a socket. The different options are summarized in Table 4.4. We discuss the option together with the function calls whenever an option is relevant. The *linger* field describes the time in clock ticks that a socket waits for data to be successfully delivered if the connection is closed. This field is important for the semantics of the socket interface and is therefore discussed in more detail in Section 4.1.2. The attribute *state* represents the internal state and additional characteristics of the socket. The possible states are summarized in Table 4.2.

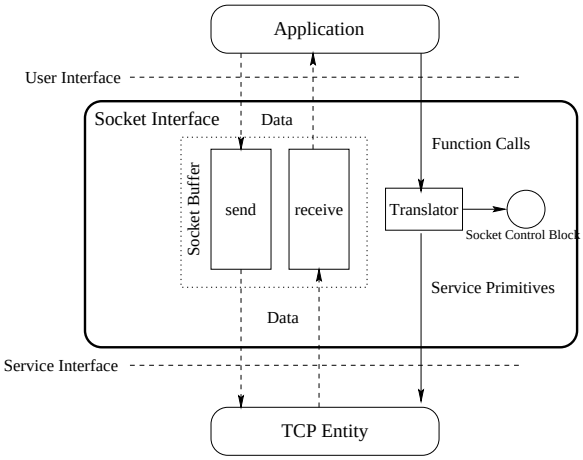


Figure 4.1.: Basic components and interfaces of a socket

Name	Description
type	Type of socket e.g. SOCK_STREAM, SOCK_DGRAM
option	Some configuration possibilities, see table 4.4
linger	time to linger while closing
state	internal state, see table 4.2
pcb	protocol control block
timeo	connection timeout
error	error affecting connection
send_buffer	Buffer for outgoing data
rcv_buffer	Buffer for incoming data

Table 4.1.: Information associated with a socket

SS_ASYNC	Socket should send asynchronous notification of I/O events
SS_NBIO	Socket operation should not block the process
SS_CANTRCVMORE	Socket cannot receive more information from peer
SS_CANTSENDMORE	Socket cannot send more data
SS_ISCONFIRMING	Socket is negotiating a connection request
SS_ISCONNECTED	Socket is connected to a foreign socket
SS_ISCONNECTING	Socket is connecting to a foreign socket
SS_ISDISCONNECTING	Socket is disconnecting from peer
SS_NOFDREF	Socket is not associated with a descriptor
SS_PRIV	Socket was created by a process with superuser privileges
SS_RCVATMARK	process has consumed all data received before the most recent out-of-band data was received.

Table 4.2.: Internal socket states

Generally the state of a socket changes implicitly either during the execution of a function call or driven by signals received from the attached protocol. However, the states *SS_ASYNC* and *SS_NBIO* can be explicitly set by the user. The SS_NBIO flag alters the socket mode from blocking (default) to non-blocking behavior. In blocking mode a process is blocked if required resources are not available. For example, if the process tries to read data and no data is available the process is blocked. Similarly if the process tries to send data, the kernel will block the process until buffer space becomes available. If the socket behavior is changed to non-blocking mode, a process will not be blocked but instead the error code EWOULDBLOCK is returned.

The *ASYNC* flag enables the kernel to send *SIGIO* signals to the process whenever the status of the socket changes due to one of the following reasons:

- A connection request was completed,
- a disconnect request was initiated,
- a disconnect request was completed,
- half of a connection was shut down,
- data arrived at a socket,
- data was sent from a socket (i.e. the output buffer has free space), or
- an asynchronous error occurred at a TCP or UDP socket.

The *pcb* field points to a protocol specific structure. This structure typically contains internal information about the state of the attached protocol. For TCP this is for example the next expected acknowledgment or the next packet to be sent. The timeout field (*timeo*) can be used to specify a number of ticks during which a request (e.g. connection set-up) has to be completed. The *error* field holds the error code of the last occurred error until it can be reported to a process as the return value of the next system call that references the socket at which the error occurred. Each socket uses two buffers, one for incoming and one for outgoing data. The usage and management of these two buffers is described in Section 4.1.2, where we deal with the exchange of data.

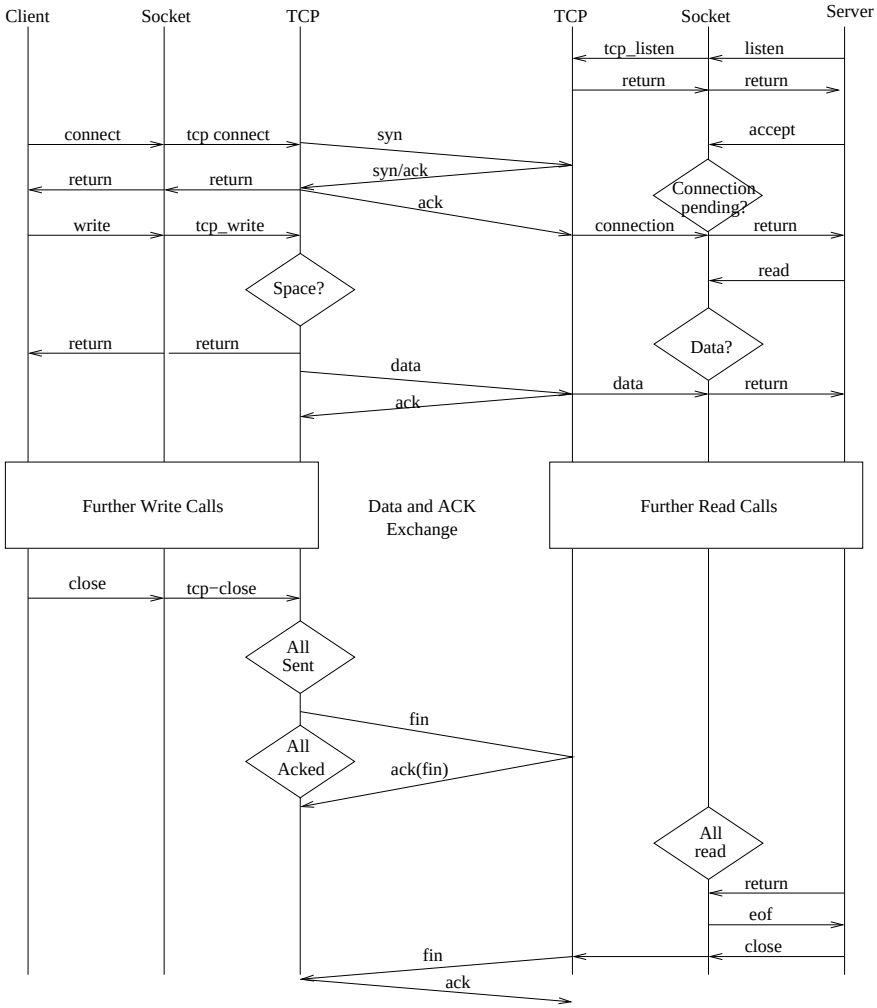


Figure 4.2.: Example communication scenario using the socket interface

4.1.1. Example Scenario

Single calls as well as call sequences (i.e. allowed call sequences) are important for discussing the socket interface. Therefore we start our discussion with a simple sample scenario, showing a typical client server communication. This scenario is used throughout this thesis to illustrate certain semantic issues. The scenario is illustrated in Figure 4.2.

The figure has three vertical time axes on each side, representing from left to right client application, socket object, TCP entity, and server TCP entity, socket object, and server application. The horizontal arrows between application, socket object, and TCP entities represent function calls. The arrows between the two TCP entities represent packet exchange. A labeled rhombus represents a condition. For example, the `tcp_write`-function will only return if buffer space is available. The same symbols are used for all interaction charts shown throughout this thesis.

In the figure we omitted the creation of a new socket on both sides as well as the `bind`-call used to set the local address of the communication endpoint. The first action is putting the server side TCP instance into listening mode. This prepares the TCP engine for accepting incoming connection requests (this is called passive open). The `listen`-function only changes the state of the local socket object and attached protocol. Thus, no TCP messages are exchanged. When the call to the `listen`-function returns, the application knows that TCP is ready to accept incoming TCP connections. The `accept`-call blocks until a connection indication is received, indicating that the three way handshake is complete.

The client (on the left side of the figure) opens the connection using the `connect`-call, performing an active open. As shown in the figure, the application is blocked until the connection is established. If TCP is unable to establish the connection, an error is returned.

After the `connect`-call or the `accept`-call has returned, the applications are allowed to pass data and to read data from the socket. In our example the client sends data to the server. After the client application has passed all data to the local protocol entity it closes the socket. After TCP has delivered all data successfully to the server it closes the connection. The server application closes its half of the connection after it has read all data.

4.1.2. Socket Function Calls

The function calls of the socket interface are summarized in Table 4.3. For each call purpose, preconditions (e.g. in which state the socket object has to be), post conditions (e.g. state change, protocol actions), and possible error conditions are given. As shown in the table, the system calls can be grouped into eight categories. The categories setup and administration can be summarized as management. The categories server, client, and termination as connection control, and input and output as data transfer.

For the following discussion we assume that the socket operates in blocking mode if not stated otherwise. We assume that none of the socket options like linger or async are set unless stated otherwise.

Management

The management group summarizes all socket functions that are used to create, release and configure sockets.

The `socket`-function creates a new socket and associates it with the protocol specified by the parameters. The function allocates a new descriptor that identifies the created socket for future system calls and returns either this descriptor or a negative value indicating an error to the calling process. The `socket`-function implements a part of the `open`-function regarding TCP's user interface (see Section 2.2.1). The descriptor corresponds to the local connection identifier, mentioned in the Chapter 2.2.1 describing TCP's user interface.

Category	Name	Function
Setup	socket	Create unnamed socket within specified communication domain.
	bind	Assign a local address to a socket.
Server	listen	Prepare a socket to accept incoming connections.
	accept	Wait for and accept a connection.
Client	connect	Establish a new connection.
Input	read	Receive data into a single buffer.
	readv	Receive data into multiple buffers.
	recv	Receive data specifying options.
	recvfrom	Receive data and address of sender.
	recvmsg	Receive data into multiple buffers, control information, receive address of sender; specify receive options.
Output	write	Send data from a single buffer
	writev	Send data from multiple buffers
	send	Send data specifying options
	sendto	Send data to specified address
	sendmsg	Send data from multiple buffers and control information to a specified address; specify send options
I/O	select	Wait for I/O conditions
Termination	shutdown	Terminate connection in on or both directions
	close	Terminate connection and release socket
Administration	fcntl	Modify I/O semantics
	ioctl	Miscellaneous socket operations
	setsockopt	Set socket or protocol options
	Getsockopt	get socket or protocol options
	getsockname	Get local address assigned to socket
	getpeername	Get foreign address of assigned socket

Table 4.3.: Networking system calls taken from [168]

Name	Description
SO_ACCEPTCONN	Socket accepts incoming connections
SO_BROADCAST	Socket can send broadcast messages
SO_SNDBUF	Send buffer high water mark
SO_RCVBUF	Receive buffer high water mark
SO_SNDLOWAT	Send buffer low water mark
SO_RCVLOWAT	Receive buffer low water mark
SO_SNDTIMEO	Send timeout
SO_RCVTIMEO	Receive timeout
SO_DEBUG	Record debugging option for this socket
SO_REUSEADDR	Socket can reuse a local address
SO_REUSEPORT	Socket can reuse a local port
SO_KEEPALIVE	Protocol probes idle connections
SO_DONTROUTE	Bypass routing tables
SO_BROADCAST	Socket allows broadcast messages
SO_USELOOPBACK	Routing domain sockets only; sending process receives its own routing messages
SO_OOBINLINE	Protocol queues out-of-band data inline
SO_LINGER	Socket lingers on close
SO_ERROR	Get error status and clear; `getsockopt` only
SO_TYPE	Get socket type; `getsockopt` only
other	ENOPROTOOPT returned

Table 4.4.: Socket options

After the socket control structures are initialized, the socket layer requests the creation of a new local communication endpoint from the associated communication protocol. It is the protocol's task to create and initialize its private control structures (e.g. TCP must create a new protocol control block). The `socket`-function fails if either sufficient resources are unavailable or the specified parameters are invalid. However, a `socket`-call requesting an Internet socket of type SOCK_STREAM (TCP) should hardly ever fail. The initial state of a socket is the state UNCONNECTED.

The `bind`-call assigns a local address to a socket. In our case this is an IP-address and a port-number. This function is usually used by servers listening to a well known address. If a process uses a socket before a local address was bound to the socket, the system implicitly assigns an address to the socket. Since addressing is a protocol specific function, the binding of addresses is performed by the associated protocol and not by the socket itself. The `bind`-call fails if the specified address is already in use. A socket must be in the `unconnected` state to allow the bind operation. After the socket is bound to an address, it is not possible to change the assigned address.

Three different calls are available for the modification of the socket behavior, namely `fcntl`, `ioctl` and `setsockopt`. The function `fcntl` is used to change the behavior from blocking (default) to non-blocking and to enable asynchronous I/O events. Table 4.4 lists the available socket options.

The `ioctl`-function is used for implementation specific configurations. To avoid such implementation specific functions, Posix defines new wrapper functions to replace ioctls (see Posix P1003.1g[87]). In BSD

systems, ioctls can be used to query whether the read-pointer currently points at out-of-band data to get or set the process group that receives SIGIO signals for the specified socket. Other ioctl calls can be used for interface related functions (e.g. to get a list of available network interfaces, to play with Address Resolution Protocol (ARP)). Due to its implementation specific behavior the ioctl-function is not further discussed here.

Connection Management

The listen and accept system calls are used for the passive side of a connection. The listen-call notifies a protocol that a process is prepared to accept incoming connections at the specified socket. For this purpose the socket should already be bound to an address and must be in the UNCONNECTED-State. The listen-call is only possible for sockets that are associated with a connection-oriented protocol. For TCP the listen-function triggers a state change of TCP's state machine from the CLOSED-State to the LISTEN-State (see Figure 2.2 on page 9). TCP does not accept any incoming connection requests until the corresponding protocol entity is in the LISTEN-State. If TCP receives a connection request (SYN-segment) when not in the LISTEN-State, it replies with a reset segment.

The listen-call has a single parameter that controls the behavior of the attached protocol when new connection requests are received. This parameter specifies the limit of connections that can be queued at the socket, waiting to be accepted by an application. Generally a TCP instance at the server side (passive open) has two queues for incoming connections. The first queue is for connections that have not completed the three way handshake yet. A new connection is enqueued in this queue when the initial syn-segment is received. The second queue is for connections that have completed the three way handshake. A connection is moved from the first queue to the second queue after the acknowledgment segment from the client (third packet of three way handshake) was received. The specified limit usually concerns the second queue, since otherwise denial of service attacks are easily possible. If the number of queued connections reaches the limit, the server ignores incoming acknowledgment (third packet) from the client. Thus, the server retransmits its SYN-segment according to the normal retransmission behavior of TCP. Usually a full queue of established connections should only be a transient state. Therefore the acknowledgment is ignored instead of sending a reset packet.

If the socket is in any other state than SS_UNCONNECTED, it will not be necessary to call the listen-function. After the listen-function has returned, the attached protocol accepts and handles incoming connection requests. The handling of incoming connection requests is independent of the socket interface, as long as the pending connection limit is not reached.

After calling listen, a process can wait for incoming connections by calling the accept-call. The accept-function is only allowed if the socket is in the LISTEN-State, otherwise an error is returned. If a new connection is established (i.e. three way handshake of TCP has completed), a new socket is created and accept returns a descriptor for this new socket. The original socket remains unconnected and ready to receive additional connections. The exact behavior (e.g. when the call returns) depends on the attached protocol. For TCP, the socket layer is notified if a connection has been established. For other protocols the process might be woken-up as soon as a connection request is received. The newly created socket inherits the configuration of the parent socket. The socket state of the new socket is set to SS_CONNECTED.

Contrary to the previous two system calls, the connect-function is used by the active end of a new connection (the side that initiates the set-up of a new connection). Figure 4.3 illustrates the message exchange between application, socket layer, and protocol entities. A call to the connect-function is only valid if the socket is in the SS_UNCONNECTED-State.

The socket state is changed to SS_ISCONNECTING and the calling process is put to sleep unless the socket is operating in non-blocking mode. After the three way handshake is complete, the calling process is waked up again and the socket state is changed to SS_ISCONNECTED. If the socket is operating in non-blocking mode, the connect-call returns without waiting for the three way handshake to complete. In this

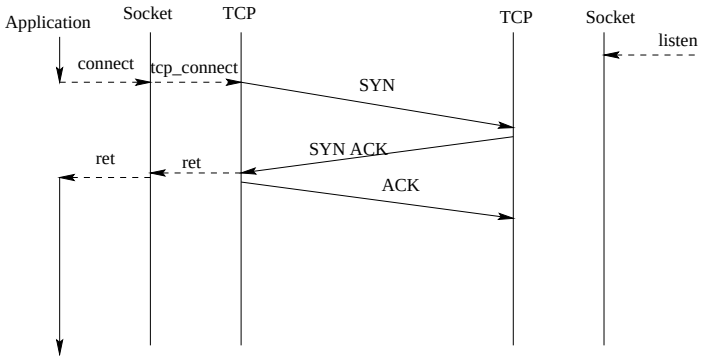


Figure 4.3.: `connect`-call

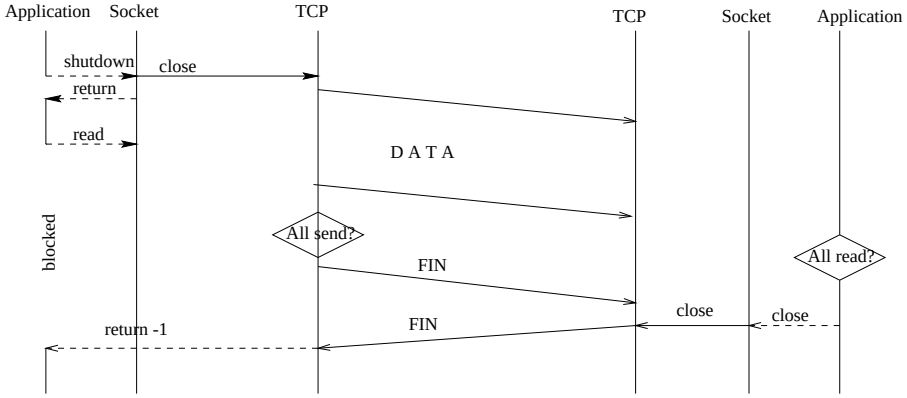


Figure 4.4.: `shutdown`-call

case the return code is `EINPROGRESS`. If a connection request is already pending at a non-blocking socket (the `connect`-function already was called), `EALREADY` is returned. If a socket is already connected, `EISCONN` is returned. `connect` must not be called for a socket that is already configured to accept incoming connections[1].

If a socket is not bound to a local address before the `connect`-call is used, a local address is automatically assigned to the socket. The connect call can fail if the specified destination address is not valid or if the attached TCP entity is unable to fulfill the connection request within a certain time.

The `shutdown`-call is used to close the write-half, read-half, or both halves of a connection without releasing the socket. If the read-half is closed, any data the process has not read yet and any data that arrives after the `shutdown`-call will be discarded. In this case TCP sends an `RST`-segment to its peer. If the write half is closed, the socket will not accept any further write calls, but it is still possible to read data. Since the close operation of TCP is graceful, TCP continues delivering all data in its send-buffer before it informs its peer about the end of the data stream.

[1]The state machine of TCP allows this. In this case the TCP state is changed from `LISTEN` to `SYN_SENT`. However, the socket interface does not support this.

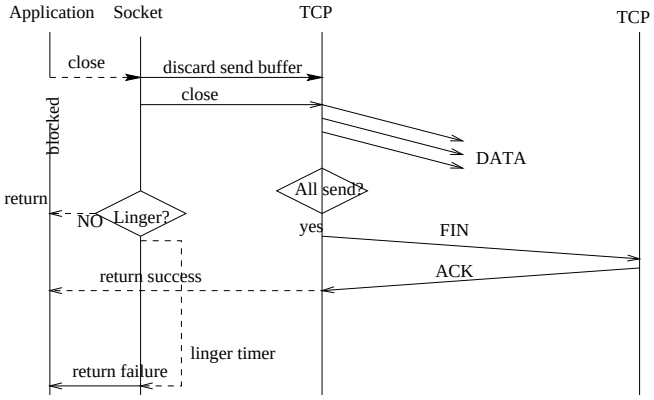


Figure 4.5.: `close`-call

The `shutdown`-function does not close a socket or free the resources allocated by the socket. It only affects the associated protocol and buffers.

By using the `shutdown`-function an application can make sure that all data has reached the destination application. How this can be achieved is illustrated in Figure 4.4. The figure shows a typical client server scenario where the server sends data to the client. In the figure the server is located on the left side and the client is located on the right side. After the server has passed all of its data to the socket (associated with TCP), it calls the `shutdown`-function, closing the write-half to inform the other end that it has no more data to send. The local TCP instance queues the close request until all packets are acknowledged, but the call to the `shutdown`-function returns immediately. Next the server uses the `read`-function to wait for incoming data. If the client application has read all data (a call to the `read`-function returns EOF), it closes the socket. If the TCP instance at the server receives a `FIN`-segment, the server (which is blocked waiting for data to read) is woken-up and the `read`-call will return with an appropriate message. Since the client has closed the socket, the server can be sure that all data was consumed. If the client closes the socket before it has read all data, an `RST`-segment is sent. In this case the `read`-function fails with an appropriate error message.

The `close`-function is called to shut a connection (both halves) down (if not already done) and to release the socket. After the `close`-function has returned, the application is no longer associated with the (transport) protocol end-point attached to this socket. All unconsumed data is silently discarded (TCP sends an `RST`-segment to its peer) and all connections that might be pending at this socket are aborted. If the receive buffer of a connection is empty (no reset packet is sent upon a close), TCP continues delivering all data in its send buffer. The data structures are not released until the protocol connection is also closed. Since the application has lost the association with the socket, it cannot be informed about the success or failure of data delivery. Normally a call to the `close`-function returns immediately. However, this behavior can be altered by using the `SO_LINGER` option. If this option is set, the `close`-function must wait for disconnect to complete or until a timer expires. In the first case the application is blocked until the peer TCP entity has acknowledged the `FIN`-segment. Although this guarantees that all data has reached the destination, there is no information from the receiving application (e.g. how many bytes were consumed). In the second case an error is returned to the calling application. A nonblocking socket never waits for a disconnect to complete.

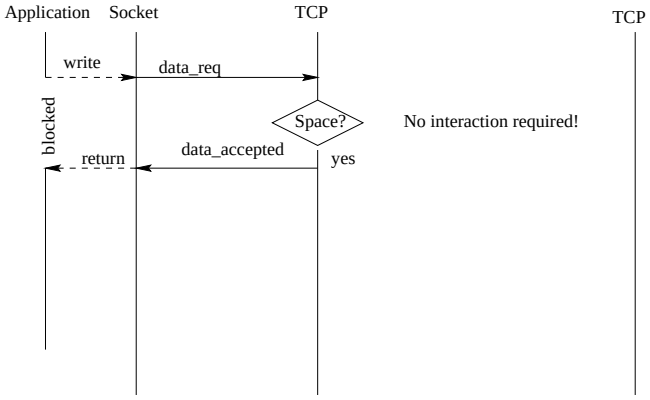


Figure 4.6.: `write`-call

Data Transfer

The role of the two buffers associated with each socket as well as the interaction with the attached protocol is important for the transmission and reception of data. Each socket has a send and a receive buffer associated with it as well as some attributes (control structures) to manage these buffers. Although it is the job of the protocol to access the buffers and fill them, the management is performed in a shared fashion. The attributes associated with each buffer are its total size (high water mark), the current state (e.g.: number of bytes buffered) and a low water mark (explained later). The main task of the socket object, besides buffer management, is to decide when a calling application should be blocked. The socket layer is responsible for blocking a process, if the buffer space is insufficient to hold the data of a `write`-call, or if a `read` -function is used and there is not enough data available while the socket is operating in blocking mode. If the socket is operating in non-blocking mode, appropriate error codes will be returned.

Sending Data

As can be seen in Table 4.3 there are four different system calls available for sending data. We refer to this group collectively as the *write* system call. Three of these calls are just simpler interfaces to the most general request, `sendmsg`.

The data passed via the socket interface using the write call is not stored in the send buffer by the socket object but is forwarded to the attached protocol. It is the protocol's responsibility and decision to store and remove the data from the send buffer. However, it is the sockets decision to forward the write request to the protocol layer. For this decision the attributes *high water mark* and *low water mark* are used.

In the case of TCP the send buffer holds unacknowledged and untransmitted data. The socket layer will not pass data to the attached protocol if the remaining space of the write buffer is insufficient (e.g. the new amount of data would reach the high water mark). The socket layer is able to pass the data to the protocol in smaller chunks, if the associated protocol operates on streams (as TCP does). Before any data is passed to the protocol, the amount of send buffer space must exceed the low water mark. The calling application is blocked until all data has been passed to the protocol. After the write call has returned with a positive return value, the calling application can be sure that the amount of data indicated by the return value has been accepted by TCP to be transmitted. If the return value is equal to the data length of the `write`-call, all data was accepted for delivery. The application is not informed about the final success of delivery. It relies on TCP which has the responsibility to reliably deliver the data upon accepting the data. However,

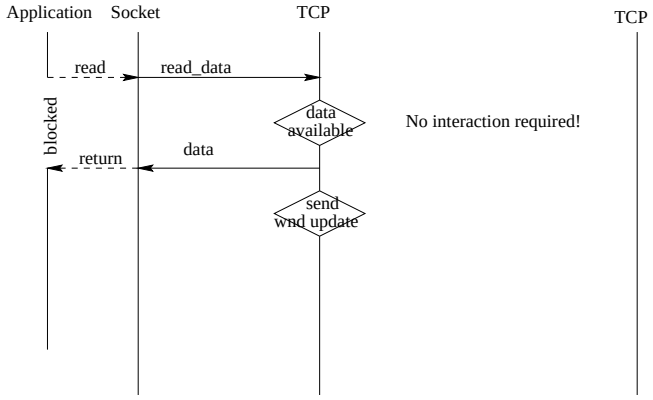


Figure 4.7.: `read` call

there are some possibilities for the client to get information about the state of the transmission. It can use the shutdown procedure as described in Section 4.1.2. It can also use the linger option so that the `close`-call does not return before all data was acknowledged by the peer TCP instance, or a timer has expired. Some operating interfaces support special ioctl commands, with which an application can query the state of the send buffer. In the case of Linux the command `TIOCOUTQ` can be used. The `write`-function does not necessarily trigger any interaction between the two TCP peer entities.

In blocking mode the calling application is blocked until sufficient memory is available to store the data. If the socket is set to non-blocking mode, the `write`-call returns `EWOULDBLOCK` if there is not enough space.

Receiving Data

As for sending data, there are four system calls available for receiving data. We collectively refer to these calls as *read* system calls. Any data received by a transport protocol is stored in the receive buffer of the associated protocol if space permits.

A process can either consume the data or it can retrieve it without consuming it. For the latter, the process must set the `MSG_PEEK` flag. However, if the MSG_PEEK option is used, only the first chunk of data is returned. Thus succeeding read calls return the same data.

If TCP receives a valid data segment, it stores the data in the receive buffer of the corresponding socket and informs the socket interface that new data is available. The Advertised Window of TCP is coupled with the receive buffer space. Thus, if new data is stored in the receive buffer, the Advertised Window is reduced and if the application consumes data, the Advertised Window is increased.

The application must explictly call one of the read functions to consume data. The application must specify a memory region where to copy the data and the amount of data it wants to consume. All read functions return the number of delivered bytes or an error code. If all data was read and the connection is closed by the remote host, a call to the `read`-function returns EOF (End of File).

As indicated by Figure 4.7, the functions of the read family do not necessarily trigger protocol interactions. Actions upon the reception of data, like the sending of acknowledgments, are part of protocol processing, not of the socket layer. There are no mechanisms supported that inform the sending side up to which byte/packet the receiving side has consumed data.

Although consuming bytes does not directly trigger any protocol actions, it increases the available buffer

space for more data. Thus, TCP will inform its peer about this fact. However, this can happen at any time (e.g. piggybacked with the next data message).

The semantics of the `read`-call depend on the mode of the socket and the associated protocol. In non-blocking mode the `read`-call returns immediately, even if no data is available. If the socket is operating in blocking mode, the process is blocked if one of the following conditions is true:

- There is no data in the receive buffer.
- There is not enough data to satisfy the entire read and `MSG_DONTWAIT`[2] is not set, the amount of available data is below the low water mark and more data can be added.
- `MSG_WAITALL` is set.

If case a stream oriented protocol like TCP is attached to the socket, the `read`-function can deliver any available amount of data but never more than requested.

The socket layer provides two alternatives to handle Out-of-Band (OoB) data. OoB data is either stored outside of the socket receive buffer or, if the socket option `SO_OOBINLINE` is set, the protocol places the data inline. In the first case a process must specify the flag `MSG_OOB` to receive out-of-band data. The interesting point here is that TCP does not really support OoB data. Although TCP supports the concept of urgent data, the urgent data is sent in-line. Only the notification that there is urgent data is sent to the receiving TCP instance immediately (even if the window is closed). Only a single byte can be sent as urgent data. If the receiving TCP instance receives a TCP segment with the urgent flag set, it informs the corresponding application. If the urgent byte is received, it is either stored in a separate buffer or in the socket receive buffer, depending on the setting of the `SO_OOBINLINE` option. In the first case the receiving application can read the urgent byte out-of-band using the `MSG_OOB` flag on a `read`-call.

Socket I/O

The `select`-call can be used if an application wants to perform different functions on the same socket (e.g. wait for available data to read and wait until space becomes available to send) or if the application needs the ability to perform I/O on multiple sockets simultaneously. Using non-blocking sockets is not an appropriate solution since this wastes resources by continuously polling all sockets.

A program can specify a list of descriptors to check for pending I/O with the `select`-function. The calling application is suspended until one of the descriptors in the list becomes ready to perform I/O or a timer expires. The `select`-function returns an indication of which descriptors are ready. Furthermore the application can specify a time limit. If no event is available within the specified timer interval, an error is returned.

4.1.3. Signals

Socket calls belong to the group of slow system calls and can be interrupted by signals. In this case an error is returned and the application is responsible for calling the last function again, if necessary. The `connect`-call cannot be restarted since the connection establishment is already in progress.

4.1.4. Failure Scenarios

Besides the operation of the socket interface it is essential to understand what happens if something goes wrong. The following error cases can be distinguished. Normally if the associated protocol detects an error (e.g. connection reset by peer), an appropriate error code is stored in the socket control block. The error message is passed to the application the next time it uses this socket.

[2] The `MSG_DONTWAIT` flag specifies non-blocking behavior for a single operation.

- End system crashes:

  Either of the two end systems of a TCP connection might crash. For the discussion we assume a client server communication where the client crashes. The discussion for a server crash is similar.

  Two possibilities exist after a crash. Either the client comes up again before the server detects the crash or not. In the first case, the TCP instance at the client sends reset packets if it receives any segments since it has neither any open connections nor any communication endpoints ready to accept new connections. It is TCP's responsibility to guarantee that no packets from the previous life cycle are accepted after a new connection is established.

  If the client does not come up again, the server will not detect the failure until either the retransmission threshold is exceeded or a destination unreachable ICMP packet is sent by a router. After detection of the failure the server closes the connection and reports an error to the application. If the server has no more data to send and the keep-alive option is not enabled, the crash is never detected. For the following discussion we assume that the client is not restarted. The discussion is identical for both cases.

  After a crash all data stored in the socket buffers and the socket state is lost. The peer entity does not know the state of the connection (e.g. which amount of data has been consumed by the application). This especially means that even data that has already been acknowledged by TCP can be lost. A crash cannot be signaled to the remote side. Thus, the peer entity has to detect the crash by itself. From the remote application (server) point of view, different cases must be distinguished. If the connection is idle (send buffer is empty), the application will not detect the crash (even if it would be blocked by a `read`-call). Since every dead connection wastes resources, most servers enable the keep-alive option. If this option is enabled, the failure will be detected after about two hours[3]. If the server has data to send, the failure stays undetected as long as the send-buffer is not full, because the local protocol entity can accept data for delivery. The local protocol entity will recognize the failure of the peer system if either its retransmission limit is reached, a reset packet is received, or if an ICMP message is received. In this case the error is signaled to the application using the return value of the next socket call. If the socket is operating in asynchronous mode, the failure is signaled to the application without waiting for the next function call.

  In the worst case scenario the application does not detect the failure at all. This can happen if the application closes the socket before the remote host crashes, since the `close`-call returns immediately without blocking until all data is sent. In this case the application wrongly assumes that all data was successfully delivered to the peer. To prevent this, the `LINGER` option must be set. If the linger option is set, the `close`-call is blocked until all data is acknowledged by the peer or a timer expires. In the latter case an error is returned to the application.

- Application is killed/crashed/terminated.

  The difference between a machine crash and an application crash (abrupt termination) is that the system including the socket and the protocol entities stays alive. Usually operating systems handle this case like a call to the `close`-function. This means that all socket objects and associated communication end-points are gracefully closed. For applications with a receive buffer that is not empty, an `RST`-segment is sent. Unfortunately the application cannot distinguish between a crashed application and a application that has closed a connection by itself.

- Router (intermediate host) crashes.

  If an intermediate host crashes, this is not critical for TCP connections as long as the intermediate host is not a single point of failure or comes up fast again. If an alternative path exists, IP simply

[3] TCP sends a keep-alive packet every two hours. If a keep-alive packet is not acknowledged, it will be retransmitted 10 times, with an inter packet gap of 75 seconds.

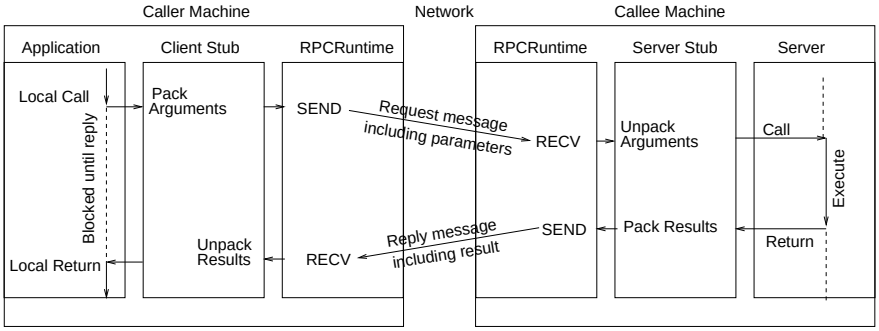


Figure 4.8.: General RPC call

reroutes the traffic.

If the crashed router is the only alternative and does not come up again, the end system is unable to distinguish this failure from the crash of an end system, as previously discussed.

4.2. Remote Procedure Calls

Remote Procedure Calls are a comfortable mechanisms to invoke an operation on a remote machine, as they appear like local function calls to the calling application. Figure 4.8 shows the components of an Remote Procedure Call (RPC) system and their interactions for a simple call. Basically RPCs introduce the idea of client-server computing to function calls. At the local machine (client) an RPC is realized as a stub procedure which marshals the parameters to be passed to the remote machine into a machine independent format and then sends the parameters and a request identifying the operation to be invoked to the remote machine. The operation is performed and the results are returned to the local stub procedure, which passes the results to the calling program. The RPC-Runtime module is responsible for providing the expected communication service. The expected service itself depends on the implemented call semantics as discussed below. As with local function calls the calling application of the client is blocked until either a return value is received or an exception is generated by the RPC-Runtime environment.

The illustrated RPC model only describes one possible model with a dormant server and only one of the two processes (client and server) being active at any time. Generally the idea of RPCs makes no restriction on the concurrency model implemented. For example, an implementation may choose to have asynchronous remote procedure calls so that the client can do useful work while waiting for the reply from the server. Although the basic idea behind RPCs is simple, there exists some design options that must be specified when an RPC system is defined. One early reference describing the design of an RPC system

by Birell and Nelson can be found in 1984[32]. The design options are summarized below and are shortly discussed in the following sections:

- Call semantics,
- stateful server vs. stateless server,
- server lifetime,
- concurrency model, and
- RPC-Runtime environment.

Although RPCs are meant to be syntactically and semantically equivalent to classical function calls, it is nearly impossible to achieve the same semantics. The difficulties in providing the same semantics are mainly based on system failures, either of client or server, or due to a communication failure.

For local function calls all states are simply lost if the system crashes. This is different for remote function calls. If the client crashes, the server state will survive and vice versa. Furthermore the communication network between client and server might be halted. In the case of a failure, an RPC system cannot necessarily guarantee how often a procedure call has been executed. On the other hand different applications can tolerate relaxed call semantics. The call semantics of an RPC system determine how often the remote procedure may be executed under fault conditions. The following call semantics are possible:

- Possible or May-be call semantics.

  These are the weakest semantics and only adequate for applications that do not really need the result of a function call (e.g. cyclic query of non-critical sensor). The motivation behind these semantics is preventing the caller from waiting indefinitely long for a response from the callee. To achieve this, the caller uses a timeout mechanism. If the timer expires, the caller continues its operation.
- Last-one call semantics

  In this case the RPC-Runtime environment is responsible for retransmitting the request if no response is received within a certain time. Thus, the call of the remote procedure by the caller, the execution of the procedure by the callee and the return of the result to the caller are eventually repeated until the result of the procedure execution is received by the caller. In the case of retransmissions the results of the last call should be returned to the caller.

  The difficulty in achieving last-one semantics is caused by orphan calls. An orphan call is a call whose parent (Caller) has expired due to a node crash. Orphans need to be terminated before repeating a request to achieve last-one semantics. Unfortunately this is difficult.
- At-least-once semantics

  Contrary to the previous call semantics these semantics only guarantee that the call is executed one or more times but does not specify which results are returned to the caller.
- Exactly once semantics

  These are the strongest and most desirable semantics. They eliminate the possibility that a procedure is executed more than once no matter how many times a call was retransmitted.

An RPC server can either be stateful or stateless. It can be dormant awaiting requests or created upon request. They can be further classified by the time duration they exist into instance-per-call, instance-per-transaction/session or persistent servers.

From a programmer's point of view, stateful servers are more convenient because they relieve the client from the task of keeping track of state information. Furthermore they are generally more efficient than stateless servers. On the other hand stateless servers have a distinct advantage over stateful servers in the event of a failure. The client of a stateful server must be designed to detect server crashes in order

to perform the required error handling activities. On the other hand a client might survive a crash of a stateless server without problems. If the stateless server comes up again fast enough, the client is able to continue its operation without any error handling. This is true if either no request was outstanding when the server crashed or if the operations are idempotent. In the first case rollback operations are required. The drawback of a stateless server is that the state has to be piggybacked in every request, if state is required.

Instance-per-call servers are created by the RPC runtime when a new message arrives and are deleted after the call was executed. They are not commonly used because they are inherently stateless and the creation and deletion is expensive in terms of time. In order to preserve the state between two succeeding calls, the state must be maintained by the client and appended to every request message.

Instance-per-session servers exist for the entire session during which a client and a server interact. With this method, there normally is a server manager for each type of service. All service managers are registered with a binding agent. If a client contacts the binding agent, the specified type of service is needed and the binding agent returns the server manager address of the desired type to the client. The client then contacts the concerned server manager, requesting the creation of a server. The server manager then spawns a new server and passes its address back to the client. The server is destroyed when the client informs the server manager of the corresponding type that it no longer needs the server.

Persistent servers generally remain in existence indefinitely and can be shared by clients. Each server independently exports its service by registering itself with a binding agent. A client communicates with a binding agent to obtain the address of a specific server. Persistent servers may also be used to improve the overall performance and reliability of the system. Therefore several persistent servers that provide the same type of service may be installed on different machines.

The service provided by the RPC-Runtime environment depends on the call semantics of the RPC system. If for example a distributed system can accept May-be Call semantics, the RPC-Runtime environment only needs to simply forward request and response messages once. If stronger semantics are required, the RPC-Runtime environment has to retransmit messages, reorder messages, and filter duplicates.

Another important difference between local and remote procedure calls is the response time and the performance of a system. RPCs have inherently longer response times than local function calls due to the communication delay. Furthermore the response time will vary heavily if the communication medium is unreliable or congested. From the caller (user) point of view this means that even if an RPC has the same results and side effects as the corresponding local function call, and even if the RPC system guarantees exactly-once semantics, the behavior of the RPC system will be different. In the worst case the caller interrupts the function call because it no longer expects the function to return successfully. In addition to that the client is unable to distinguish a long duration call from a communication failure or a server crash. To handle this issue, probe packets are required in order to periodically test whether the server is still working on a specified request.

The performance of an RPC system suffers mainly from the fact that succeeding RPCs are sequentially called, as shown in Figure 4.8. The client is blocked until it receives the response to the pending call. Especially in the case of a high bandwidth delay product this leads to a high performance penalty. Asynchronous RPCs can solve this problem. A summary of asynchronous RPCs can be found in[10].

Further points that must be considered if an RPC system is implemented, but are not important for the scope of this thesis, are summarized below:

1. different data representation on client and server machine,
2. binding (how a caller determines the location and identity of the callee),
3. the semantics of address-containing arguments (call by reference),
4. passing compound variables or lists, and
5. access to global variables.

The client server relationship considered so far is not appropriate for all distributed systems. Some distributed systems rather require a peer-to-peer paradigm. RPC systems that realize a peer-to-peer relationship are called *Callback RPC*. In a callback RPC system the RPC server is allowed to call functions at the RPC client (e.g. to request user input) while working on a request from the client.

Part II.

The Remote Socket Architecture

Chapter 5.

Preconsiderations for Designing ReSoA

In Section 3 we pointed out several reasons why TCP is not always suited if radio technologies are used as access network. Furthermore, if an end system is used in different environments, these different environments might require different configurations of the protocol stack (e.g. socket buffer sizes). One the other hand it is a matter of fact that Internet access is not dispensable and that the replacement of TCP is not an option since its congestion control mechanism is crucial for the operability of the Internet.

As discussed in Chapter 3.3, one class of solutions are PEPs. Unfortunately most PEPs have the drawback that they either violate TCP's end-to-end semantics or that they are unable to hide the peculiarities of the access network completely. In addition most approaches only deal with TCP and neglect UDP.

In this chapter we introduce our Remote Socket Architecture (ReSoA), which belongs to the class of PEPs. ReSoA is motivated by the observation that it is not the protocol but the Application Programming Interface (API) what matters. Applications are designed for a specific kind of interface and communication service but are independent of the implementation of the interface, and the protocol which delivers the service. In Section 4.2 we showed how RPCs are used to introduce the concept of client-server computing to function calls. Following this idea, it should be possible to have a special computer (e.g.: edge-router, access point) and grant all end systems access to its TCP/IP protocol stack by carefully designing a distributed implementation of the interface to the TCP/IP protocol stack. In ReSoA, function calls to the BSD socket interface are not executed at the local end system but in an RPC-like fashion at a ReSoA-server. To achieve this, the socket interface at the local end system as well as the TCP/IP protocol stack are replaced by the ReSoA modules. These modules intercept function calls to the socket interface like `socket`, `read`, `write`, `getsockname`, encapsulate them into ReSoA packets and deliver the ReSoA packet to a ReSoA-server, where the function calls are executed.

The idea of a remote interface to TCP is already considered by the TCP RFC 793. For example, on page 2 RFC 793 states: "The TCP specification describes an interface to the higher level protocols which appears to be implementable even for the front-end case, as long as a suitable host-to-front end protocol is implemented." and on page 8 it states "The mechanisms of TCP do not preclude implementation of the TCP in a front-end processor. However, in such an implementation, a host-to-front-end protocol must provide the functionality to support the type of TCP-user interface described in this document."[142].

The idea of providing a distributed implementation of an interface is not limited to the socket interface. It can be applied to an arbitrary interface. The BSD socket interface was chosen since it is widely deployed. However, for any interface the semantics of the interface must be carefully investigated in order to provide a functionally equivalent distributed implementation. For the BSD socket interface this is investigated in Chapter 6. Since our interest concentrates on the Internet domain, the export of the socket interface only deals with sockets belonging to the Internet family, focusing on TCP.

ReSoA's main field of application is wireless Internet access. In a wireless environment ReSoA can be used to overcome the performance problems of the Internet transport protocols over error prone links, or to implement new power saving schemes. Since the end-system is relieved from the TCP/IP protocol stack, a protocol tailored for the current environment and optimization goal can be used. However there are additional appealing features that make ReSoA interesting not only for wireless Internet access but

also for the general case. Additional advantages are:

- Internet access is provided even if an end-system (e.g. small sized) or the access network do not support IP.
- The end-system does not need its own globally valid IP address to communicate with an Internet host because it does not have a private TCP/IP stack. Thus, ReSoA can replace Network Address Translation (NAT), which is not in every case transparent to the application, and it can hide a local network behind a single access point (firewall-like behavior)[63].

5.1. Design Goals

The realization of ReSoA should meet the following design goals. The goals are listed in order of importance although a strict ordering is not always possible.

1. **Transparency:** An application that operates on top of ReSoA instead of a local implementation of the BSD socket interface should not recognize any difference. Neither the local nor the remote application (on the distant host) should be aware of the existence of ReSoA. Any application that is based on the Berkeley socket interface should be able to operate on top of ReSoA without any modification or recompilation. This especially means that there should not be any difference regarding syntax or semantics of the interface. Please note that this demand for transparency does not imply the user to be unaware that he is using ReSoA as required by the PEP RFC[34].
2. **Interoperability:** An end-system supporting only ReSoA (IP-free) should be able to communicate with every host supporting the TCP/IP protocol suite. This especially means that every ReSoA end-system can use every service offered by any Internet server or provide services to other hosts.
3. **IP free operation:** Access to the services of the Internet should also be possible if the end system or the access network do not support IP protocols.
4. **Performance:** Improvement of the throughput seen by an application using the service of TCP. This point targets the problem that TCP's mechanisms are in some aspects not suited for links with specific characteristics. Thus, we want to improve the utilization of the available link capacity, or in other words, aim at minimizing the time a link is idle although there is data to be transmitted and the link quality is good. There are situations where the performance cannot be improved. In these cases ReSoA should perform comparable to TCP.
5. **Response Time:** The response time of the exported interface should not be significantly increased compared to a local implementation. Normally RPC based function calls have a higher response time because the response time includes the time needed to transfer function calls and their return value over a network. As applications might protect socket calls with a timer, a longer response time cannot be tolerated for ReSoA. Higher response times can also lead to a reduced performance, especially in the case of small transfer sizes.
6. **Overhead:** The number of bytes sent over the access network should not be higher than the number of bytes for a local socket implementation.
7. **Technology independence:** ReSoA should be able to operate over a wide range of different technologies.
8. **TCP configuration:** The configuration of the used TCP installation should be tailored to the backbone characteristics. This means that the user should not be responsible for configuring TCP if he uses his laptop in a different environment.

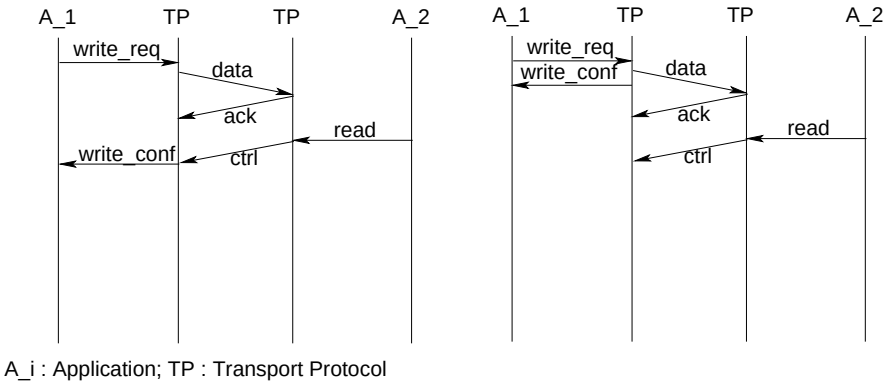


Figure 5.1.: Two possible semantics of a send request

Only the first three design goals are of general importance for the design of a distributed implementation of an interface. Design goal number four determines the optimization goal. Our focus is on performance issues. Besides performance, alternative optimization goals are possible, for example minimizing the complexity of the end systems or energy efficiency. The fifth design goal is related to performance issues and required to support transparency. The remaining design goals are nice to have but can adversely effect the optimization goal. For example, an aggressive retransmission strategy generally comes along with an increased overhead.

5.2. TCP Semantics versus Socket Semantics

The main advantage of PEPs compared to end-to-end approaches is that they can decouple two networks with different characteristics. Unfortunately, this main advantage also is the most frequently used argument against PEPs, since decoupling breaks TCP's end-to-end semantics. In order to understand why ReSoA can be used instead of a local TCP connection end-point while maintaining the semantics seen by an application, it is important to understand the difference between the semantics offered by TCP and the semantics seen by an application. The latter is a combination of the service of the transport protocol and the interface used to access this service.

With ReSoA we argue that not the protocol semantics are what matters but the interface semantics. The interface semantics determine what the communicating applications know about the state of their service requests. For example, what does the return of a data send request mean. Does this mean that the data was transmitted to the peer transport protocol entity, that the data was consumed by the peer application, or only that the data was accepted by the local transport protocol entity for delivery. To illustrate this, Figure 5.1 shows two possible semantics of a data-send-request. On the left side the data-send-request returns after the data was read by the peer application, while on the right side the data-send-request returns immediately after the data was stored in the local send buffer. Clearly in the case of an exported interface both cases must be treated differently.

In the case of TCP and the socket interface (the semantics of the different socket calls were discussed in Chapter 4.1) the semantics seen by the application correspond to a reliable data pipe. After the pipe is established, the sending application pushes data into the pipe, while the receiving application reads data from the pipe. The operation of the two applications is decoupled (asynchronously) by the sizes of the send and receive buffers. The sending application is not explicitly informed whether the transfer was successful,

or up to which byte the receiving application has consumed the available data. TCP acknowledgments only confirm the reception of data at the other end of the pipe but do not provide any information about whether the data was read by the receiving application or not.

The end-to-end semantics of TCP do not guarantee to a sending application that its data was read by the distant application. It is possible that a host crashes just after TCP has acknowledged all data including the FIN byte, but before the application has read the data. In this case the sending application would wrongly assume that the communication was successful, as long as it does not implement appropriate application level protocol mechanisms.

Even if we consider the worst case that the Remote Socket Server (RSS) crashes before all data that already was acknowledged by the TCP instance is delivered to the Remote Socket Client, this will not break the semantics seen by the application. The same is true for a local implementation of the socket interface, albeit with a lower probability. To improve the stability of the system, approaches of fault tolerant systems could be applied. However, this is out of the scope of this thesis.

The semantics of the socket interface depend on the configuration of a socket object (e.g. blocking mode vs. non-blocking). To preserve the semantics, every function call must be carefully investigated regarding its local effects as well as its effects on the interaction between the peer entities. The split implementation is *semantically equivalent* to a local socket interface if the interaction sequence between the application and the socket interface is not altered. This is discussed in more detail in Chapter 7 where we compare the semantics of ReSoA to the semantics of the BSD socket interface.

5.3. Architectural Considerations

In this section we develop the core architecture according to the design goals discussed in Section 5.1. Figure 5.2 provides a general overview about the components of the Remote Socket Architecture and their interfaces to other system components. As can be seen from the figure, the TCP/IP protocol stack and parts of the socket interface are removed from the end system and are replaced by the ReSoA modules. Basically ReSoA can be viewed as a client located at the end system and a server that is located at a network node (e.g. access point). These two modules together implement the functionality of the socket interface. The client intercepts the socket calls, encapsulates them and transmits them to the server. The server decapsulates the socket calls, executes them on behalf of the client and sends the result back to the client. The main task of the client and the server is to maintain the semantics of the socket interface. In the following we discuss the different design decisions that led to this architecture.

5.3.1. Alternatives for the Cut

The idea of ReSoA is to export the service to the Internet protocol stack by splitting the socket interface. There are different possibilities for this cut. The decision which functionality is implemented at the ReSoA-client and which at the ReSoA-server depends on the design goals. For example, if the major design goal was to reduce complexity, the cut might be at a different position than in the case where the major design goal is to improve performance. Our design goal is to improve performance, where performance mainly means throughput seen by the application.

Basically there are three alternatives for splitting the socket interface. Splitting is possible either at the top (library level or just after the call has entered the kernel), inbetween (splitting the socket object itself), or at the bottom (when the socket function calls have been translated to protocol dependent function calls) of the socket layer.

The most obvious place would be at the top. At this point the socket calls could be intercepted and

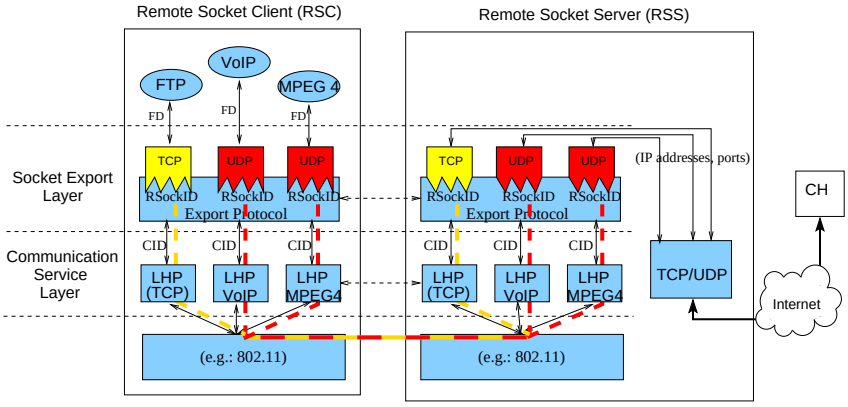


Figure 5.2.: Basic components of ReSoA

transmitted to the ReSoA-server following a synchronous RPC-based approach. In this approach the implementation of the socket interface at the end system would be replaced by a wrapper module that translates each socket call to a RPC call. The entire functionality of the socket interface would be implemented at the RPC-server. Thus, each time an application uses a function of the socket interface, it is blocked until the socket call is transferred to the ReSoA-server and the corresponding reply is received. Although such an implementation would relieve the client from complexity, it has many shortcomings.

First of all every request would have to be transmitted to the ReSoA-server which then decides whether the call is allowed in the current socket state. If the call is not allowed, an error message must be returned. Second, if only a single request can be outstanding at any time, the system will be forced to a send-and-wait behavior which is known to offer poor performance for links or networks with a high bandwidth-delay-product. Third, the response time of the system would depend on the underlying communication system and the current channel state. Anyway the response time would be longer than for a local implementation of the socket interface, which violates our requirements.

A split at the bottom is difficult since the socket layer and the protocol layer share common data structures like send and receive buffers. A split at this position would require the duplication of the data structures. This would automatically lead to a split in the middle of the socket layer.

To overcome the shortcomings discussed above, our approach follows the third alternative and splits the socket interface somewhere inbetween. This approach requires that some of the socket interface control structures are not only implemented at the ReSoA-server but also at the ReSoA-client. This permits the ReSoA-client to deal with some socket calls locally and thus, reduce the response time and increase the throughput. A detailed description of the duplicated functionality is given in Section 6.1.1 and Section 6.1.2.

5.3.2. Layering

To implement the functionality of the socket interface, the ReSoA-client and ReSoA-server must communicate with each other. The required functionality can be divided into two layers. First, a protocol for the exchange of messages between ReSoA-client and ReSoA-server is required. This protocol is principally independent of the underlying network characteristics. Second, the export of the socket interface requires a reliable communication service. An efficient realization depends on the peculiarities of the underlying communication technology, the service already offered by the protocol stack of the access network, and

the optimization goal (e.g., throughput vs. energy consumption).

Therefore we decided to use a two layered architecture on top of the protocol stack of the access network. The protocol of the upper layer is called Export Protocol. The Export Protocol (EP) is a simple connectionless request-response protocol. The protocols of the lower layer are called Last Hop Protocols. The service provided by an Last Hop Protocol (LHP) depends on the socket type and application and may be adapted to the properties of the applied technology. In the case of a TCP socket, an LHP must provide a fully reliable service. In the case of a UDP socket, an LHP may provide a semi-reliable service. In the latter case the LHP may behave application specific. Since the LHP depends on the technology, we decided to exclude the protocol specification of an LHP from the ReSoA specification. ReSoA only defines the service which it expects from an LHP as well as the interface an LHP has to provide. Thus, it is possible to use the same implementation of ReSoA-client, ReSoA-server, and EP on top of different LHPs.

5.3.3. Interface to TCP

The socket layer is closely intertwined with the attached transport protocol. For example, if an application reads data from the receive buffer, TCP will have more space available to store new data and might advertise a larger window to its peer. In the case of a split socket interface the interaction between the socket layer and the transport layer is more complicated. For example, TCP increases its Advertised Window whenever data is read from the receive buffer. In the case of ReSoA data is read from the receive buffer when the RSS forwards the data to the ReSoA-client. In order to maintain the socket semantics, this forwarding must not lead to an increase of the Advertised Window. In Section 6.3 we discuss which additional functionality is required to maintain the socket semantics.

5.3.4. Addressing

In the Internet world addressing at the application layer refers to the identification of transport protocol communication end-points (also called sockets). An application uses an integer descriptor to pass requests to a specific socket, and the network uses a quadruple consisting of the source and destination IP addresses and the source and destination port to find the correct transport protocol entity.

If ReSoA is used, two additional addressing problems need to be solved. On one hand an association of the two parts of a socket that are located at the ReSoA-server and the ReSoA-client respectively, has to be established. On the other hand the source address to be used for the transport protocol entity has to be determined. We refer to the first as *Access Network Addressing* and to the second as *Internet Addressing*.

Access Network Addressing

In our basic approach ReSoA-client and ReSoA-server are located within the same local network (broadcast domain). Therefore communication between both instances does not require any multi-hop capable addressing scheme such as Internet Protocol (IP) addressing.

Access Network Addressing defines how the two parts that implement a socket are glued together. According to the two layers of ReSoA, the addressing is performed in two steps. On the lower layer the LHP is responsible to address the machines that run ReSoA. On the upper layer the Export Protocol (EP) is responsible for delivering incoming packets to the correct socket object.

An LHP has to provide a communication channel between the ReSoA-client and ReSoA-server. The channel is created when ReSoA is started (see Section 5.4 on page 69). To establish a communication channel between two ReSoA nodes the LHP can chose an arbitrary addressing format as long as unique addresses are provided. Usually the LHP will use the addressing format of the underlying technology for this purpose. For instance, if ReSoA is deployed in an 802.x local network, 802.x addresses can be used. If the ReSoA domain contains different technologies, technology independent address formats can be used

(for instance, E.164). If the provided address format of the underlying technology is not sufficient to provide communication channels between ReSoA nodes, the LHP will have to add the missing functionality[1].

In order to map incoming messages to the correct socket object, a ReSoA-client assigns an integer identifier (called *rsockID*) to each created socket. This identifier is included in every EP message. The problem with this scheme is that the same identifier can be assigned to a socket on different ReSoA-clients. Thus, for the ReSoA-server the rsockID is not sufficient to map incoming packets to the correct socket. To overcome this problem, the EP must know the LHP connection on which the packet was received. Since the LHP provides a connection-oriented service, ReSoA-client and ReSoA-server maintain a connection identifier for each created LHP connection. This connection identifier is included in every indication from an LHP instance to the EP. The connection identifier together with the rsockID allows for uniquely addressing socket objects. It can be compared to the TCP/IP world where port numbers are not sufficient to identify a communication end-point. Instead the transport protocols use the IP address in addition to the port numbers. The difference here is that IP is connectionless and hence the entire source and destination addresses must be used, while the LHP utilizes the addressing format of the underlying technology and only uses a connection identifier for multiplexing[2].

The addressing formats of both levels are independent of each other. The EP uses the channel provided by the LHPs without knowing how this channel is established. This is possible in spite of the fact that the ReSoA-client and server must establish the LHP connection, which means that the ReSoA-client must know the address of the ReSoA-server's machine. This is achieved by looking at the address simply as a byte stream. The ReSoA-client must be configured with this byte stream (either manually or by some kind of look-up service), but will not interpret it.

IP Addressing

Due to the split of the socket interface and the movement of the TCP/IP protocol stack to a network node, ReSoA has the ability to support different alternatives for the IP address assignment to ReSoA-clients. An assignment scheme describes how the ReSoA-server maps globally valid IP addresses to its clients. The following alternatives exist:

- **1-1 mode:** The ReSoA-server provides a globally valid IP address for every ReSoA-client. It must be able to send and receive packets with this IP address. To implement this, the ReSoA-server can either use a pool of IP addresses, obtain an new IP address when a new client registers, or the ReSoA-client can pass its IP address[3] to the ReSoA-server during the registration process with the ReSoA-server.
- **n-1 mode:** The ReSoA-server has a single globally valid IP address that is used for all registered end systems. This is the most natural approach, since one possible ReSoA model is that all applications on all end systems are just processes that utilize the same interface (protocol stack) and IP addresses are assigned to interfaces and not to hosts. In this mode ReSoA replaces the need for Network Address Translation (NAT) devices.
- **n-m mode**: This is a combination of the previous two modes. The ReSoA-server has a set of IP addresses per interface. It is possible that some clients (e.g. those providing a service) have a separate IP address, while others share a common IP address.

The address mapping mode does not necessarily determine how the ReSoA-server obtains its IP addresses. In 1-to-1 mode it is possible for a client to register its IP address with the ReSoA-server or for the

[1] For example, in the case of a broadcast only the addressing scheme.

[2] All the usual constraints, e.g., lifetime of the identifier are easier to solve since an LHP operates in a local environment.

[3] This can be complicated if the end system also has a TCP/IP protocol stack that uses the same address.

ReSoA-server to use the Dynamic Host Configuration Protocol (DHCP) on behalf of the ReSoA-client to get a new IP address every time a new client is registered[4]. In n-to-1 mode the ReSoA-server can either use a fixed set of IP addresses or can also use DHCP, again on behalf of the client using its Medium Access Control (MAC) address.

ReSoA does not define which alternative should be used. Every approach listed above is possible. The selection of a specific scheme is the responsibility of the ReSoA-server provider. The used addressing scheme must be negotiated during the registration process (see Chapter 6.4 on page 96).

5.3.5. ReSoA-server Position

The ReSoA-server can be located anywhere in the network. However, we assume that the server is located close to the ReSoA-client at the border between different networks. Our architecture assumes that the server is placed between the access network and the Internet and not anywhere within the Internet. Although from the semantic point of view it seems to be possible to place the ReSoA-server deeper in the network, this would require a different design, at least regarding the LHP. We assume that ReSoA operates in a homogeneous environment.

According to the idea that an exported interface can be used to decouple networks with different (incompatible) characteristics, it is thinkable that the ReSoA-server also uses an exported interface of a second ReSoA-server and so on, building a chain of servers. We do not allow such an approach in our architecture and assume that there is a single client server relation. However, it is possible that both end systems use ReSoA to communicate with each other.

5.3.6. Co-existence with a Parallel Local TCP/IP Protocol Stack

Principally the purpose of ReSoA is to replace the local TCP/IP protocol suite. On the other hand the PEP RFC recommends that the usage of a proxy should be decided on an per application level[34]. Therefore ReSoA supports the parallel operation with a classical TCP/IP protocol stack. We call this mode *Dual Protocol Stack Mode*. If an end system only supports ReSoA, we call the operation mode *Pure ReSoA Mode*.

In pure ReSoA mode interception of socket calls is not a problem, since every socket call must be forwarded to the ReSoA-server. This is not the case for the dual protocol stack mode. Here for every socket interface call it must be decided whether the addressed socket belongs to a split socket or to a local one. Since we require ReSoA to be transparent to an application, a modification of an application (e.g. use of a different socket family for ReSoA) is not an option. However, this rather is an implementation issue we won't further discuss here.

A second problem of the dual protocol stack mode is the usage of IP addresses. If the ReSoA-client uses the n-1 mode, an application will see different IP addresses depending on whether ReSoA or the local socket interface is used. Thus, from an application point of view this configuration looks like an end system with two interfaces.

If the ReSoA-client uses the 1-to-1 mode and registers its own globally valid IP address with the ReSoA-server, then two different network interfaces would use the same IP address. This not only means that the ReSoA-server must intercept every packet with this destination address and determine the communication end-point of this packet, but also that two different hosts would answer the same ARP requests. Since this is not possible, the ReSoA-server must not assign a client's IP address to one of its interfaces and must

[4] To be able to register multiple addresses from a single DHCP server the ReSoA-server has to use the client's MAC address in the DHCP request; otherwise a standards-compliant DHCP server will not assign several IP addresses in the same subnet to the same MAC address.

send and intercept IP packets with a non-local IP address. This is implementation specific and therefore not discussed here[5].

5.4. Operation Example

The purpose of this chapter is to provide a better understanding of the operation of ReSoA. Therefore we go through the setup process of a ReSoA system and continue with an example session of a File Transfer Protocol (FTP) connection.

For this discussion we assume that ReSoA uses the n-to-1 mode (using only a single globally valid IP address for all registered clients) and operates in dual protocol stack mode.

First the ReSoA-client and server must be configured. The ReSoA-server and client must know which LHPs are used for TCP and UDP sockets. LHPs are specified by a provider identifier and a protocol number.

Next the ReSoA-server must be started. It prepares the local LHP instances to accept incoming connections and now waits for ReSoA-clients to register.

After initialization the RSS is ready to accept registration requests from a ReSoA-client. The Internet protocol stack at the ReSoA-server behaves in exactly the same way as it would if it were located at any end system and no socket was opened. The behavior of the ReSoA-server is passive until a ReSoA-client requests the creation of a new socket object. The functionality of this socket object corresponds to the functionality of a normal BSD socket object with three exceptions. First, it is triggered by incoming EP packets from a ReSoA-client and not directly by function calls from the application. Second, it must influence the protocol behavior of the associated protocol to some extent in order to maintain the semantics of the interface (see Section 6.3). And third, it must deal with error conditions that can not occur with the standard implementation (see Section 6.5). According to RPC-terminology, a ReSoA-server is a dormant and stateful server.

A ReSoA-client must be configured with the LHP address of the ReSoA-server. The binding mechanism (how a ReSoA-client finds a ReSoA-server and obtains the ReSoA-server's address) is not part of the architecture. We simply assume that the address is known (e.g. manually configured). The addressing format also depends on the used technology and is not part of ReSoA. The ReSoA-client is configured with a binary representation of the ReSoA-server address. This binary representation of the ReSoA-server address is passed to the local LHP instance in order to create a connection to the LHP instance at the RSS. For example, in an 802.x environment a ReSoA-client gets 6 bytes as the LHP address representing the MAC address of the ReSoA-server. The initial step when a ReSoA-client starts is to establish an LHP connection with the ReSoA-server and to register[6] itself at the server. The registration process concludes with providing the source address used for future connections via this ReSoA-server. This can be compared to configuring a local interface with an IP address. After the registration is completed, an application can access the TCP/IP protocol stack at the ReSoA-server without recognizing any difference from a local TCP/IP implementation.

For the further discussion let us consider an FTP session as shown in Figure 5.3 on page 71. In this session the FTP client runs at the ReSoA end system and wants to download a file from an FTP server in the Internet. The FTP client operates in active mode. This means that the FTP server establishes the data connection between FTP client and FTP server. Hence, the FTP client must send its IP address to the FTP server and has to offer a listening TCP endpoint.

Since ReSoA does not change the API between the application and the protocol stack, the application first has to create a new socket using the `socket`-function call. For performance reasons this call is not

[5] One possibility to implement this is to use the Linux netfilter framework.

[6] The registration process is out of the scope of this thesis and therefore will not be further discussed.

forwarded to the ReSoA-server immediately but delayed until the socket is used for the first time[7]. The ReSoA-client creates the local part of the socket object, assigns an `rsockID` to it, and returns control back to the application.

Next the application initiates the establishment of the transport protocol connection using the `connect`-call. From the state of the socket the ReSoA-client sees that the corresponding part of the socket at the ReSoA-server has not been created yet. Therefore it sends a request message to the server (using the previously opened LHP connection) that includes the `socket`-function call as well as the `connect`-function call. This request message includes the `rsockID` as explained in Section 5.3.4 on page 66. The calling application is blocked until the ReSoA-server sends the corresponding return value. The response of the ReSoA-server not only contains the result of the `connect`-call but also the port number of the local connection endpoint. This accelerates the fetching of address information (using `getsockname`).

Based on the LHP connection the ReSoA-server knows which ReSoA-client the request belongs to. It creates the corresponding socket part and assigns the `rsockID` and the LHP connection identifier of the LHP connection to the socket object. Then it executes the `connect`-function call. The local IP address of the ReSoA-server is used as source address for the TCP connection. Thus, every application—independent of the end system it is running on—uses the IP address of the server for its connections. If the TCP connection is established, the ReSoA-server sends a response message to the ReSoA-client. This response message again includes the `rsockID`.

Next the application sends a request to the FTP-server. Since the request must contain the source IP address of the TCP connection, the application first has to determine its IP address and port number. The application must query the socket using the function `getsockname`. Since the socket is associated with the address assigned by the ReSoA-server, this call returns the globally valid address pair (IP address and port) actually used by the connection[8].

When the `getsockname`-function call returns, the application encapsulates the results in its requests and asks the socket to deliver the data. The ReSoA-client forwards the request to the ReSoA-server and immediately returns control to the application (if the socket send buffer is large enough) without waiting for a response from the ReSoA-server.

In order to improve performance and to keep response times close to the response time of a local TCP/IP implementation, ReSoA is not based on a classical synchronous RPC mechanism. Instead the behavior of the socket stub depends on the socket call. Whenever possible the ReSoA-client deals with the user request locally and returns the control back to the calling application before it transfers the request to the ReSoA-server.

If the TCP instance at the ReSoA-server receives data (from the Internet), it uses the quadruple (IP addresses and ports) to map the data to a socket as usual using a callback. The socket is provided with information about the fact that it is part of a split socket object, the corresponding client (which LHP connection to use) and the socket identifier (`rsockID`). Thus, the data can be forwarded to the corresponding client without waiting for an explicit `read` data request. The notification includes the socket identifier used by the ReSoA-client to map the received data to the correct socket receive buffer.

5.5. Differences to other Solutions

ReSoA belongs to the class of PEPs. Hence, it shares the advantages of PEPs. However, the distinct difference between ReSoA and most other PEPs is that we do not split the TCP connection and then use a

[7] To limit the probability that the ReSoA-server has insufficient resources for the already allocated socket, each ReSoA-client is only allowed to have a certain number of parallel open sockets.

[8] Normally we would have to create a second socket, export it and put it into listening mode for the data connection of an FTP transfer. We left this out of the message sequence chart in Fig 5.3 on the next page for better readability.

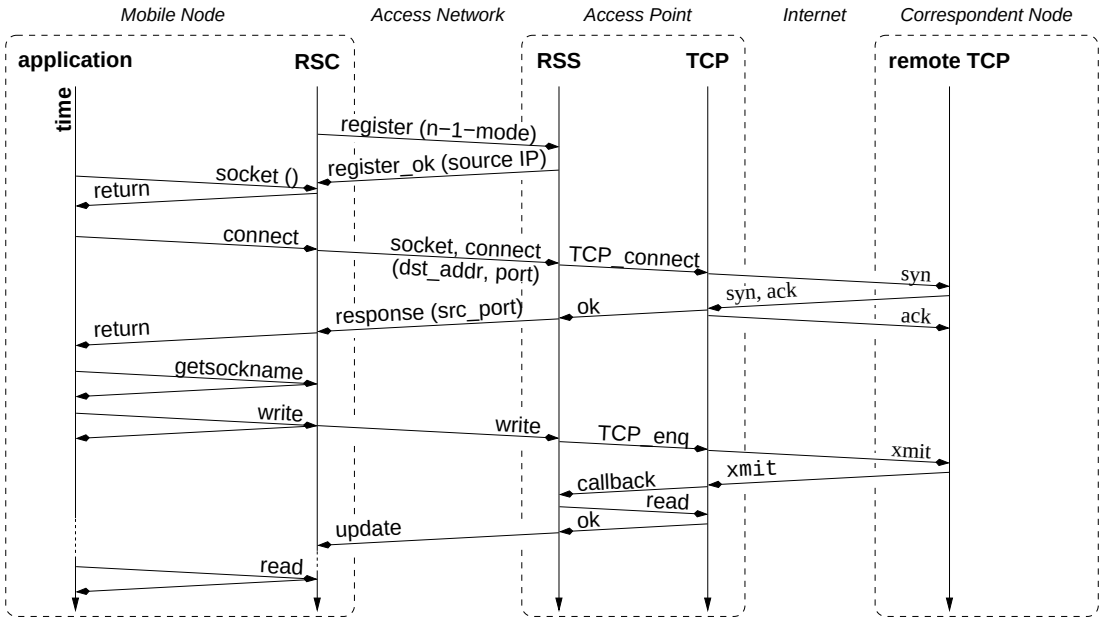


Figure 5.3.: Example message exchange resulting from an application using a ReSoA-client and a ReSoA-server configuration to access the Internet

kind of splicer or trick to maintain TCP's semantics. Instead we carefully designed a distributed implementation of the interface to the TCP/IP protocol stack. Our approach distinguishes between TCP and interface semantics. We discussed the meaning of a TCP acknowledgment and how different interface semantics effect the service seen by the application. ReSoA inherently supports a proxy based approach for the entire TCP/IP protocol stack, although we focus on TCP. The separation between the export functionality and provision of the communication service allows ReSoA to operate over a vast number of different technologies using optimized protocols. This is especially achieved by only including the specification of the required service and the interface to the LHP in ReSoA but not the LHP itself. This approach permits the use of flow specific LHPs without classification of IP packets. Theoretically it is possible to attach a different LHP to every socket.

Although the idea of a distributed socket interface implementation was already followed by the Mowgli and SOCKS approach, there are important differences. SOCKS is designed for security purposes and hence not comparable to the ReSoA approach. Mowgli focuses on the design of an efficient protocol stack for wireless Internet access using Global System for Mobility (GSM) . Although the authors mention that Mowgli could be used on top of arbitrary technologies, they do not believe that Mowgli is required in different scenarios (e.g. in the case of wireless LANs). The protocols are tuned for GSM despite Mowgli's two layered architecture in which the lower layer provides the communication service. However, in order to deal with the peculiarities of GSM, they also include specific optimizations in the upper layer. For example, the application is allowed to transmit data before the TCP connection is established.

Mowgli's support of the BSD socket interface is only included for compatibility reasons to support all legacy applications. The goal of Mowgli is rather to provide a new, more sophisticated interface for Mowgli (or GSM) aware applications than to maintain the semantics. Especially Mowgli does not include the special treatment of the TCP `FIN`-segment or the `LINGER` socket option as ReSoA does.

The Internet addressing modes provided by ReSoA are also different. With Mowgli each end system has its own IP address. ReSoA also supports a configuration where ReSoA-clients share one IP address.

Chapter 6.

Specification of ReSoA

As described in the previous chapter, ReSoA comprises two layers. The upper layer is responsible for the export of the socket interface, while the lower layer provides the appropriate communication service. This chapter specifies the components of the socket export layer as well as the interface and the expected lower layer communication service. The protocols of the lower layer are not part of ReSoA and therefore not described here.

Until we present the interface, we simply consider the lower layer as a reliable (semi-reliable for UDP) data pipe into which we can push packets on one end and remove packets at the other end. The specification of the interface between the two layers is given in Appendix C.1.

6.1. Splitting the Socket Interface

In Section 4.1.2 we discussed the socket interface. Basically the socket interface is a wrapper that not only translates protocol independent function calls to protocol specific calls, but also maintains a state (a communication endpoint) for every created socket. This state is modified by function calls and protocol messages.

Following the idea of RPCs, we use the client-server paradigm to implement a split implementation of the socket interface. Thus the socket interface is realized by a client (*ReSoA-client (RSC)*) which runs at the end system, a server (*ReSoA-server (RSS)*) which is located somewhere in the access network (e.g. access point or edge-router), and the *Export Protocol (EP)* used for the communication between ReSoA-client and ReSoA-server. In this chapter we specify these three components. Figure 6.1 depicts the socket split in an abstract way.

6.1.1. ReSoA-client

The ReSoA-client is located at the end system. It replaces the local implementation of the socket interface. From the application point of view the ReSoA-client looks exactly like the classical socket interface. Syntactical transparency is achieved by offering the same interface as the BSD socket interface does. All details of the socket export are hidden from the application by the ReSoA-client. According to RPC terminology the ReSoA-client implements a socket stub.

Internally the ReSoA-client is responsible for the communication with the ReSoA-server. However, as discussed in Section 5.3.1 due to performance issues it is not possible to simply forward socket calls to the ReSoA-server and to implement the entire socket functionality solely at the ReSoA-server. Instead, the ReSoA-client has to implement parts of the socket interface behavior by itself. It is responsible for testing whether a socket call is allowed in the current socket state, whether the application should be blocked and so on. Details are given in Section 6.2 where we discuss the implementation of the different socket calls. The main difference between the ReSoA-client and a local socket object is that the ReSoA-client does not translate socket function calls to service primitives of the attached protocol. Instead it transfers these calls to the ReSoA-server after it has performed a few tests.

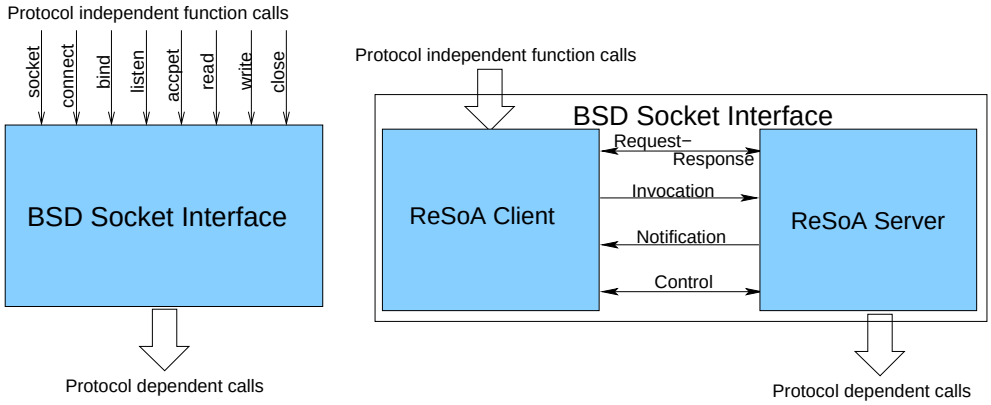


Figure 6.1.: Splitting the socket interface

In order to implement the local part of the split socket object, the ReSoA-client must maintain a global management state and a state for every created socket. The global state contains information about the address of the ReSoA-server, general state information about the association with a ReSoA-server, a list of all created sockets, the default lower layer connections to be used for the communication with the ReSoA-server (one reliable and one semi-reliable), and a watchdog timer. The timer is used to test whether the ReSoA-server still exists in the case that there has been no communication between the ReSoA-server and the ReSoA-client for some time. The settings for this timer are conservative for two reasons. First, this timer is only of interest if all applications are idle. Second, we expect the underlying layer to inform us if the connection to the ReSoA-server was lost.

The socket state contains all attributes that also exist for a local socket implementation (e.g. see Figure 6.2 and Figure 6.3). This means for example, that the ReSoA-client has a send and receive buffer for every socket as well as a socket state. How these attributes are maintained and synchronized with the corresponding state of the ReSoA-server is discussed in Section 6.2. The socket state is extended by the following information:

- **Reliable LHP connection**: The lower layer can support different communication services. This identifier refers to the reliable connection. This connection must be used for all TCP and UDP socket messages, except for socket calls used for reading and sending data. This identifier is initialized with the default connection of the global state.
- **Application-specific connection**: UDP socket application data is transferred using a semi-reliable connection. If the application is aware of ReSoA, it can register its own LHP and set this identifier accordingly.
- **Remote Socket Identifier**: To build a per socket association between the state of the ReSoA-client and the ReSoA-server.
- **Remote Send Space**: The available buffer space of the ReSoA-server for application data.

The motivation behind including connection identifiers in the per socket state is that this allows for using different lower layer connections for different sockets. This is especially important for UDP sockets, since in this case some socket calls must be delivered reliable to the ReSoA-server while socket calls including application data may get lost. In the current ReSoA version we only support two LHP connections. One for reliable data transfer and one that provides a semi-reliable service. The dynamic integration of additional protocols would require management functionality which is out of the scope of this thesis.

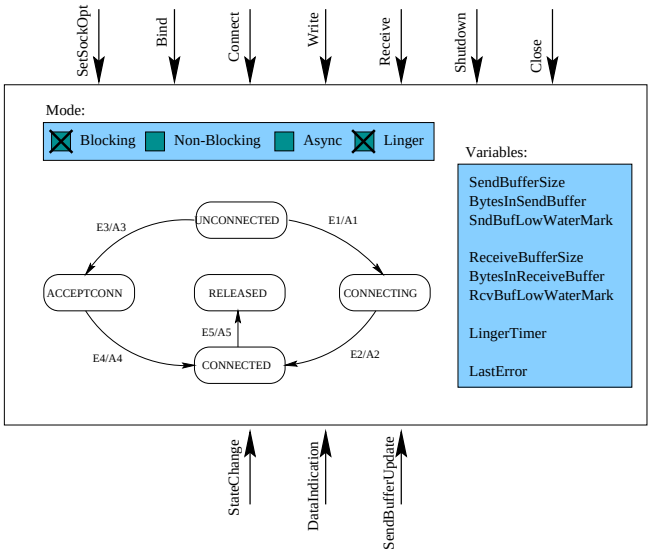


Figure 6.2.: BSD Socket model

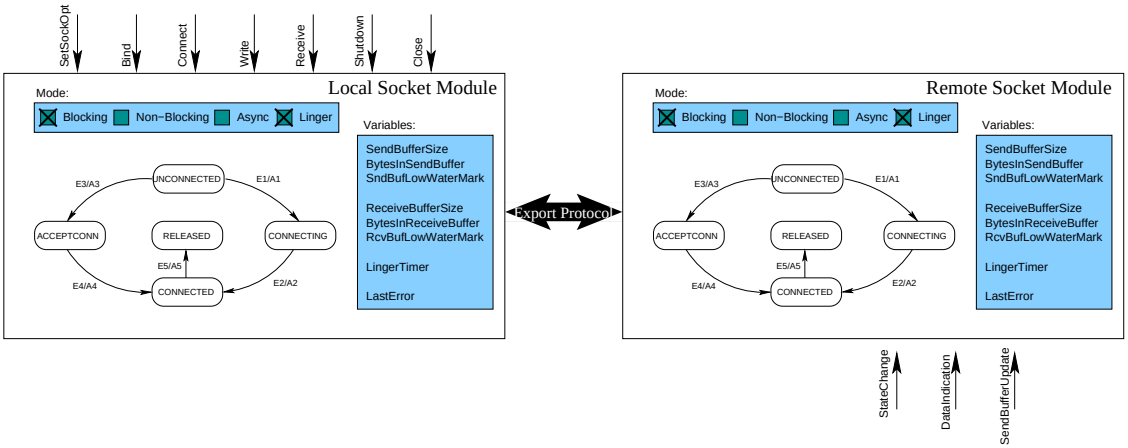


Figure 6.3.: Realization of a socket in ReSoA

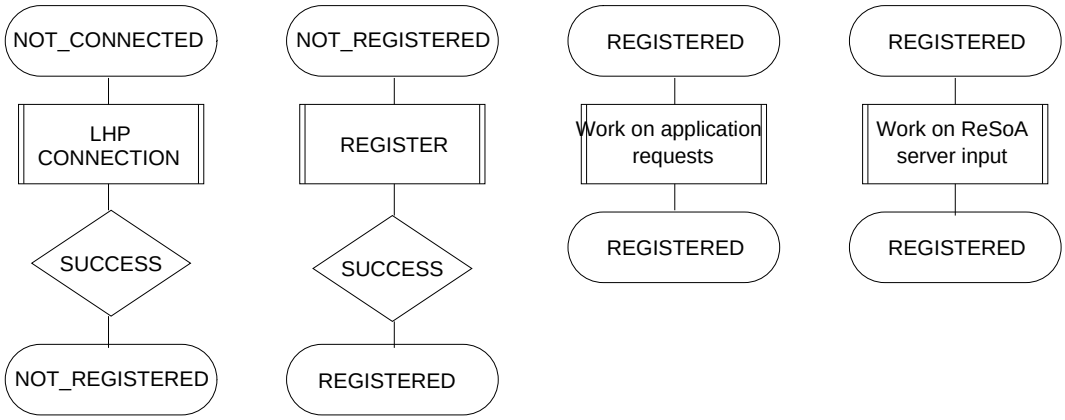


Figure 6.4.: High level flow chart of the ReSoA-client.

Figure 6.4 shows a high level flow chart of ReSoA's client's operation. The initial state of a ReSoA-client is `NOT_CONNECTED`. Whenever a ReSoA-client is instantiated, it must be configured with the LHP address of a ReSoA-server and the LHPs to use. The ReSoA-client initiates the establishment of the two LHP connections. Then it tries to register itself with the ReSoA-server using the reliable LHP connection. As soon as the ReSoA-client is in the `REGISTERED`-state, it is ready to accept input from applications and from the ReSoA-server. This is further discussed when we deal with socket calls in Section 6.2.

The ReSoA-client may not be terminated before all sockets were properly closed and the ReSoA-client has released its registration from the ReSoA-server.

6.1.2. ReSoA-server

The ReSoA-server is located at a network host, normally either at an access point or an edge router. It is persistent and must maintain a state for every registered ReSoA-client as well as for every open socket. The ReSoA-server must be instantiated before any ReSoA-client is started.

As with a local socket interface implementation the ReSoA-server implements the communication with the attached transport protocol. Thus, it translates protocol independent function calls to protocol specific service primitives. The interface to the attached protocol is implementation specific and therefore not discussed in this thesis.

In addition to the normal socket communication the ReSoA-server is responsible for the registration of new ReSoA-clients (see Section 6.4) and controling the behavior of the attached TCP instance. The latter is required to maintain the semantics of the socket interface. This is further discussed in Sections 6.2 and 6.3.

For every registered ReSoA-client the ReSoA-server must maintain a list of open sockets, a list of LHPs used for communication, and a watchdog timer to test whether an idle ReSoA-client is still alive. Similar to the keep-alive timer of TCP the setting of this timer can be conservative (e.g. test every hour).

Just as the ReSoA-client the ReSoA-server must keep a state for every created socket. The state corresponds to the state of the ReSoA-client. Thus, for every created socket the ReSoA-server maintains all attributes that can be found in a local implementation of the socket interface, as well as the additional attributes described in the ReSoA-client section (Section 6.1.1). The only difference is that the attribute `Remote Send Space` is replaced by an attribute `Remote Receive Space`. This attribute describes the amount of buffer space available at the ReSoA-client for application data.

The difference between a ReSoA-server and a classical RPC server is that in addition to events from the ReSoA-client, triggering inputs from the attached transport protocol entities are also handled. Furthermore ReSoA does not follow the strict paradigm of function calls. As shown in Section 6.2 there are requests that the ReSoA-server does not respond to. The ReSoA-server sends asynchronous indications to the ReSoA-client including received data or information about state changes.

Whenever a new ReSoA-server is created, it must at first set the used LHPs to listen mode so that they can accept new connections. After that the ReSoA-server is idle until a ReSoA-client registers. Whenever the ReSoA-server receives a message belonging to a socket, it first must test whether the client is registered and whether the socket exists. The requested function will only be executed if both tests succeed. If the client is not registered, the request is discarded silently. If the socket identifier is not valid or the requested function is not allowed, the ReSoA-server returns an error code. This is discussed in Section 6.5.

The detailed operation of the ReSoA-server is described in Section 6.2 where the implementation of the socket calls is specified.

6.1.3. Export Protocol

The EP is used for the communication between ReSoA-client and ReSoA-server respectively. It provides a connectionless, timer protected service that either has request-response or request-only semantics as described below. The EP is considered as a part of ReSoA-client and ReSoA-server. Hence we do not specify an upper interface to the EP. The lower interface is described in Section C.1.

The operational behavior of the EP is relatively simple since it operates on top of a reliable (semi-reliable) communication channel. For example, it does not include protocol mechanisms such as error control. It fully relies on the lower layer. The EP is mainly responsible for the encapsulation and coding (marshaling) of the socket calls and responses, as well as for the exchange of control-information and for demultiplexing. For the demultiplexing process a remote socket identifier is used that is assigned to each socket when it is created (see Section 6.2.1). Whenever the EP instance at the ReSoA-server receives a request message with an unknown remote socket identifier, this message must either request the creation of a new socket or it is invalid. The EP supports the following services:

- **REQUEST-RESPONSE**-Service: This service is used by the ReSoA-client whenever it wants to transmit a socket call requiring a response from the ReSoA-server (e.g. `bind` socket call). To transmit a request the EP uses an EP-PDU with the type field set to `REQUEST`. After the execution of the call at the ReSoA-server is complete the ReSoA-server responds with an EP-PDU with the type field set to `RESPONSE`.

  Although a request message must be acknowledged by the ReSoA-server with a response message, it must not be stored in a retransmission buffer since the lower layer is responsible for retransmissions. The EP never retransmits a message. However, the ReSoA-client must be able to detect whether a response message is missing in order to avoid deadlocks[1].

  The EP only permits a single outstanding request per socket (remote socket identifier). Whenever a ReSoA-client requests the transmission of a request message, the EP tests whether a request is already outstanding for this socket. If not, the remote socket identifier is stored in a table, a timer is started, and the request is sent to the ReSoA-server. Thus, the request-response service can be compared to some kind of acknowledged datagram service. The difficulty here is that the ReSoA-server does not immediately acknowledge the request upon reception. It does this only after it has processed the request in order to keep the network traffic low. Thus, determining a setting for the timer is a rather difficult task as the ReSoA-client cannot know how much time the ReSoA-server needs to complete a request. To overcome this problem the EP instance at the ReSoA-client

[1] The ReSoA-server might crash after it has received the request.

uses a very conservative timeout value which is at least a multiple of the RTT between the two communicating EP instances. If this timer expires before a response is received the EP first tests the state of the LHP and whether it is good. Then it sends a control message (`REQUEST_PENDING`) to the ReSoA-server. Upon reception of this control message the ReSoA-server checks whether it still has a pending request for this socket. If the LHP reports communication problems, or if a control message is sent, the timer is restarted. If the ReSoA-server does not reply with a control message indicating that it is still working on the request, the ReSoA-client gives up after it has reset the timer a certain number of times. This error situation is discussed in Section 6.5.

Whenever the EP at ReSoA-server receives a request, it uses the remote socket identifier to execute the request at the correct socket. After the execution of the request is complete an appropriate response message is returned.

- **CALL/STATUS**-Service: The call/status service is used for socket calls for which the client does not expect a response (e.g. a `write`-function call) and to inform the ReSoA-server about state changes of a socket (e.g. available buffer space).

  If the ReSoA-client uses this service the EP simply encapsulates the data into an appropriate frame and passes the frame to the LHP. In this case there is no need for a timer since the ReSoA-client does not expect a response message. If the LHP is unable to deliver the frame it informs the calling EP instance. Basically there are two situations where the LHP can send a negative response. First, the outgoing queue of the LHP can be full. In this case the LHP will inform the EP when it is ready to accept further requests. Second, the LHP was unable to deliver a frame after a number of retransmission attempts. The latter is an error situation and the ReSoA-client must inform the application while the first is only a temporary situation.
- **UPDATE**-Service: The update service is the ReSoA-server equivalent to the ReSoA-client call/status service. It is used by the ReSoA-server to inform the ReSoA-client about asynchronous events concerning a certain socket, like the reception of data. It is used to update the state of the corresponding socket (e.g. to pass the data to the receive buffer). As it is the case with call/status messages, the EP does not protect an `update` Protocol Data Unit (PDU) with a timer. It fully relies on the operation of the LHP.
- **CONTROL**-Service: The control-service is used by the ReSoA-client and the ReSoA-server to exchange control messages that are not directly related to socket functions. Examples for control messages are registration messages and queries about the server state. Table 6.1 lists the different control messages. All control messages sent by the ReSoA-client are protected by a timer if they require a reply.

6.2. Implementation of Socket Calls in ReSoA

This section describes the implementation of all socket calls shown in Table 4.3 on page 46. We point out where we break the TCP end-to-end semantics and how, despite this fact, the socket semantics can be maintained. The discussion shows that the close call is the most crucial call.

6.2.1. Creation of a New Socket

The `socket`-call is described in Section 4.1.2. Whenever an application requests the creation of a new socket it uses the `socket`-function and specifies: socket family, socket type, and protocol as function parameters. In our case the socket family is always `AF_INET` since we only deal with Internet protocols here. After the system has created the socket it returns a descriptor to the application. This descriptor is

Name	Code	Reply	Description
REGISTRATION_REQ	0x01	X	To register with a server
REGISTRATION_RES	0x01	-	Answer from the server
DEREGISTRATION_REQ	0x02	X	To de-register with a server
DEREGISTRATION_RES	0x02	-	Answer from the server
STILL_ALIVE	0x03	X	Testing whether the other end is still there. This message must be answered with a STILL_ALIVE message.
REQ_PENDING	0x05	X	Testing whether the server is still working on a request
REQ_PENDING_RES	0x05	-	Answer from server
ERROR	0x08	-	General error message.

Table 6.1.: Export Protocol control packets

used to address the socket for all later socket calls by the application. The application is blocked until the socket is created.

As described in Section 6.1 we need state information about each ReSoA-client and ReSoA-server socket to realize a socket in ReSoA. A first approach to realize this was to create the ReSoA-client instance first and then pass the function call to the ReSoA-server using the request-response service of the EP. The ReSoA-server then creates its instance of the socket as well as the protocol communication endpoint specified in the socket function call by the user. After the communication endpoint is created the ReSoA-server returns the return code to the ReSoA-client, which in turn passes the return code to the application.

With this `socket`-call implementation the application would be blocked until the request and the response have been transmitted over the communication network. The response time of the `socket`-call would depend on the environment, the current channel state, and the used technology. Due to the communication costs the response time would be higher than for a local socket interface. This unpredictable response time violates the design goal to keep the response time of ReSoA close to the response time of a local interface and is therefore inappropriate. This approach reduces the throughput especially for short connections since the total time increases.

A second approach is to return the control to the application after the ReSoA-client has created the local part of the socket. Here it is the responsibility of the ReSoA-client to test whether the parameters of the call are valid (e.g. whether the specified address family and protocol exist and are supported by ReSoA). The control is returned to the application at once after the local part of the socket is created or, in the case of invalid parameters, an appropriate error message is returned to the application. The corresponding instance of the ReSoA-server socket is then created in the background using the request-response service of the export protocol.

However, the second alternative still requires the exchange of two messages, while a local implementation of the socket interface does not generate network traffic. A further improvement of the second alternative is the delay of the creation of the socket instance at the ReSoA-server until the socket is used for the first time. This improvement avoids the two additional messages.

The advantage of the second and third approach is that the response time of ReSoA is comparable to the response time of a local socket interface implementation, since no messages need to be transferred over the wireless link and no second computer is involved in the socket creation. The third alternative saves two

messages[2].

The drawback of the second and third approach is that we additionally acknowledge the creation of a new communication endpoint before it actually has been created. The creation of a socket has two aspects. First, the parameters are checked whether the requested protocol family and protocol are supported. Second, the required resources are allocated. While the second and third approach can perform the first step they can only assume that the required resources are available for the ReSoA-server. Thus, it is possible that the ReSoA-server fails to create the socket, although the creation has already been acknowledged to the application. However, this should rarely occur since the creation of an end-point does not require many resources and nowadays memory is not a scarce resource.

We assume that the ReSoA-server has sufficient resources to create the requested sockets. To control the number of simultaneously open sockets per ReSoA-client the ReSoA-client is configured with an open socket limit. This limit can either be set manually (default limit) or during the registration process. With this assumption we decided to follow the third approach.

When the ReSoA-client receives the request to create a new socket from an application, it tests the call parameters and whether the number of open sockets exceeds the socket limit. If both tests are passed it creates a new local socket instance, assigns a remote socket identifier to it, and returns a socket descriptor to the application. If one of these tests fails an appropriate error code is returned.

The corresponding socket instance at the ReSoA-server is created if the application uses the socket the first time. If the application plays the active part this usually is the `connect`-call. In this case the EP-PDU must include two socket calls.

When the ReSoA-server receives the request to create a new socket it creates the socket object instance as well as the communication endpoints. If this fails it returns an error to the ReSoA-client. After the communication endpoint was created the next socket function included in the same request is executed. The response message is sent after the execution of this additional socket function is complete.

6.2.2. Configuration of a Socket

As explained in Section 4.1.2, a socket can operate in different modes and has attributes which define its behavior. If a socket is implemented in a split fashion it must be determined whether an option effects both socket parts or only one of them. If a socket option effects both parts or only the ReSoA-server the function call must be forwarded to the ReSoA-server. Whether the request-response service or the call service of the EP can be used depends on the socket option. In the following we discuss the different socket options/configurations whereby we define whether messages must be exchanged between the ReSoA-client and ReSoA-server, and which service of the EP should be used. The different options are summarized in table 4.4 on page 47.

Socket options, `ioctls` or `fcntls` (which are not described here) are simply forwarded to the ReSoA-server using the request-response service of the EP. The calling application at the ReSoA-client is blocked until the corresponding response message is received.

SO_KEEPALIVE

The `SO_KEEPALIVE` option is only valid for sockets of type `SOCK_STREAM`. It enables the sending of keep alive packets by the attached protocol for long idle periods. Although it can be used by a client and a server it is mostly used by servers to detect and remove connections with lost clients.

If the keep-alive option is set at a socket controlled by ReSoA, the request to enable this feature is simply forwarded to the ReSoA-server where it is passed to the attached TCP instance. The call service of the EP is used for this purpose. The keep-alive functionality is implemented between the TCP instance at

[2] Two messages do not sound a lot, but if an application opens many connections, it adds up.

the ReSoA-server and the TCP instance at the corresponding host. Different from a local Internet protocol stack the wireless link is not involved in the exchange of keep-alive messages. Thus, messages are saved and the wireless terminal can sleep (if no other connection is active). This is not really an issue because keep-alive messages are exchanged every 2 hours. Of course the ReSoA-server has to assure that its clients still exists.

If the keep alive functionality is enabled at the corresponding host (which does not know that the other end uses ReSoA), the application at the corresponding host (normally a server) wants to make sure that it does not allocate resources for clients that are not longer reachable. Thus, it is interested in whether the application at the wireless terminal is still running. It is not interested in whether there still are any TCP instances answering keep-alive requests[3].

There are different possibilities to achieve these semantics with ReSoA. The first option is to modify the TCP implementation at the ReSoA-server so that it forwards keep-alive packets to the ReSoA-server instead of answering them. The ReSoA-server can then test whether the corresponding application (socket) at the ReSoA-client is still alive, and trigger TCP to answer the keep-alive packet. This approach has the drawback that TCP needs to be changed and that the wireless link is involved in the communication without being included in the RTT estimation of the sending TCP.

In a different approach TCP at the ReSoA-server answers the keep-alive packets as if there were non ReSoA. It is the responsibility of ReSoA to kill or close the corresponding TCP endpoints if a client application at the ReSoA-client or the ReSoA-client itself vanishes. If the application is closed the ReSoA-client informs the ReSoA-server about the state change. If the ReSoA-client crashes, the LHP will inform the ReSoA-server (see also Section 6.5). Thus, in any case the TCP instance at the ReSoA-server will only answer TCP keep-alive segments if the client application still exists. Therefore ReSoA uses the second approach. Please note that due to the communication delay over the wireless link it is possible that a keep-alive is answered, although the client is already dead. However, this is not critical for two reasons. First, keep-alive packets are only used to free unused resources and the resource can also be freed two hours later. Second, even with a local Internet protocol stack it is possible that the client crashes just after a keep-alive request was answered. From the corresponding host point of view the two cases are not distinguishable.

SO_LINGER

This socket option is discussed in the context of the `close` socket call in chapter 6.2.3.

SO_SNDBUF and SO_RCVBUF

These socket options are discussed in the context of the read and write calls in Section 6.2.4. The socket options `SO_RCVLOWAT` and `SO_SNDLOWAT` are discussed in this section as well.

SS_ASYNC

Only the ReSoA-client needs to know that a socket is set to asynchronous mode, since the design of the socket export ensures that the ReSoA-client is automatically informed about all state changes.

SS_NBIO

This socket option sets a socket to non-blocking mode. Since this effects nearly every socket call, it is discussed in the context of the different socket functions.

[3] According to the end-to-end argument this functionality should be implemented by the applications. However, since such an option exists, we must deal with it in ReSoA.

6.2.3. Connection Management

`bind`-function

As discussed in Section 4.1.2 on page 45, the `bind`-call is used to assign a local address to a socket. Since addressing is a protocol issue, the `bind`-function has to be passed to the attached protocol which decides whether the requested address is valid and available.

For the realization of the `bind`-call in ReSoA this means that it must be implemented using the request-response service of the EP. The calling application is blocked until the ReSoA-client has received the response message. Although this behavior maintains the socket interface semantics it introduces the problem of a possibly long and unpredictable delay. As already mentioned before, the execution of a request-response pair includes the communication delay, which in turn depends on the channel state and capabilities of the used technology.

The alternative approach where the ReSoA-client decides whether the requested local address is valid is not an option. Although we only deal with Internet addresses here, the ReSoA-client could decide whether an address is syntactically correct. However it cannot decide whether an address is available. If multiple ReSoA-clients share a single Internet address it would be possible that applications on different hosts would bind a socket to the same port. This conflict must be prevented. Otherwise we would have to cancel already bound sockets later, as the conflicts are recognized. This would clearly violate the socket interface semantics. A local implementation of the `bind`-call would only be possible if the ReSoA-server provides a separate IP address for every client (1-to-1 mode). Since the realization of socket calls in ReSoA should be independent of the configuration of ReSoA such a configuration cannot be assumed.

Thus, in the case of the `bind`-call we have to weigh response time against socket semantics. However, the problem of increased response time can be relativized by two arguments. First, we can expect applications using ReSoA to implement the client-role, which means that they actively initiate the connection establishment. Often the active side of a connection uses an address assigned by the system instead of using the `bind`-function. This happens when the `connect`-function is executed. Second, a `bind`-call will normally be followed by a `connect`-call, or by a `listen`-call followed by an `accept`-call. Both the `connect`-call and the `accept`-call need some time to complete and the user cannot distinguish which call causes the delay.

Active Open - `connect`-function

At the beginning of the discussion we assumed that the socket operates in blocking mode. At the end of the section we consider non-blocking mode. In Section 4.1.2, describing the semantics of the socket interface, we showed that a `connect`-function call returns if either TCP was able to establish the connection, or after a timeout has occurred. In the first case a positive status is returned while in the second case a negative status is returned.

Two possible implementations of the `connect`-call in ReSoA are shown in Figures 6.5 and 6.6. The difference between these two approaches is that in the first approach the ReSoA-server acknowledges the reception of the request message and later the ReSoA-client acknowledges the response message from the ReSoA-server. In the second approach the ReSoA-client and ReSoA-server rely on the reliable service of the LHP. Thus, instead of four messages only two messages, namely the request message and the response message are exchanged. The advantage of the first approach is that it is easier to estimate the time during which a response should be received. However, we decided to minimize the number of packets exchanged over the air interface and therefore chose to follow the second approach.

By comparing the interaction diagrams for the `connect`-call of a local socket implementation and of ReSoA we can see that the semantics of the `connect`-call are maintained (compare Figure 4.3 on page 49 and Figure 6.6). In the case of ReSoA, the application calls the `connect`-function as it would do if the

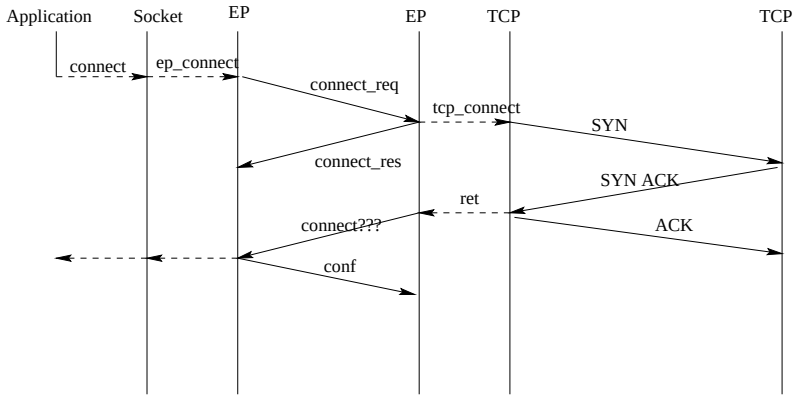


Figure 6.5.: ReSoA `connect`-call version 1

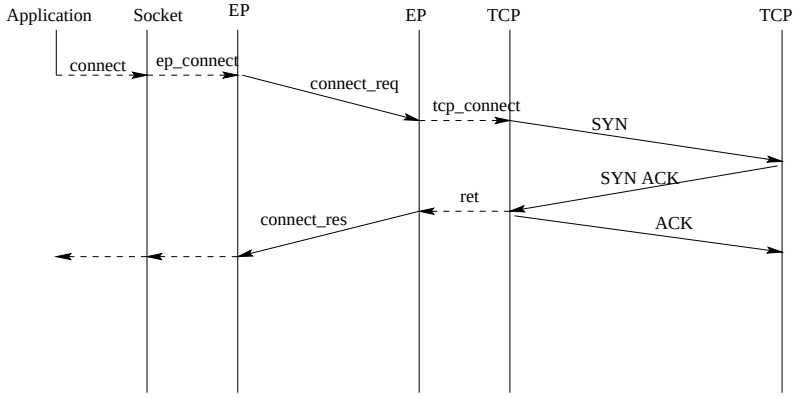


Figure 6.6.: ReSoA `connect`-call version 2

protocol stack were located at the local system. After calling connect the application is blocked until the TCP connection is established. The `connect`-call is transferred to the ReSoA-server which passes it to the associated protocol. Then the ReSoA-server waits for the protocol to confirm that the connection was established. The result of the confirmation is passed to the ReSoA-client which then returns control to the calling application.

The difference to a local TCP/IP stack is that the three way handshake of TCP is only performed between the ReSoA-server and the corresponding host. Since the local end system is not involved in the TCP connection establishment (after sending the request), it is possible that the ReSoA-client crashes after the ReSoA-server has sent the initial `SYN`-segment. In this case the ReSoA-server establishes the TCP connection on behalf of a crashed client. Later, when the ReSoA-server notices that the client has crashed, it must send a reset segment for this connection. Nevertheless the socket semantics are preserved. The three way handshake is performed independent of the service user after it was initiated by the application. The socket interface ensures that the call to the `connect`-function does not return a positive status before the connection is established. This is also true for the exported interface.

Next, let us consider the response time (the time the application is blocked). In the standard case the local host sends a `SYN`-segment which must travel over the wireless link and the Internet. The application is blocked until the corresponding `SYN-ACK`-segment is received. For ReSoA, request and response packets are exchanged over the wireless link (access network) instead of the two TCP packets. Thus, the response time should be identical or less if the initial `SYN`-segment is lost.

If the socket is operating in non-blocking mode the ReSoA-client returns the control to the application after it has sent the request message. Until the connection is established the ReSoA-client answers all further function calls to this socket as a local implementation would do.

Passive Open - `listen`-function

There are two alternative approaches to implement the `listen`-call in ReSoA. Either the request-response service of the EP or the call-service can be used. The latter is possible since the ReSoA-client has all information available to decide locally whether this call is allowed (e.g. it knows the current state of the socket). The first approach has the advantage that the application can be sure that the attached protocol is in the `LISTEN`-State when the socket function returns, but has an increased response time compared to a local socket interface implementation. The second approach has the advantage that the response time is identical to the local socket interface, but includes the possibility that, due to communication problems, the application wrongly assumes that the TCP instance is in the `LISTEN`-state. Another advantage of the second approach is that we only need to transfer a single message instead of two messages.

We decided to follow the second approach. If the `listen`-call could be transferred to the ReSoA-server, there is no reason why it should fail. If the LHP is unable to deliver the listen invocation to the server due to channel impairment, neither would any host be able to establish a TCP connection with this wireless terminal. Hence it is not important whether the local TCP implementation would already be ready to accept connections[4].

It is not possible to delay a call to the `listen`-function as we did with the `socket`-function since this causes TCP to reject incoming connection requests.

[4]There is a difference in what the peer TCP sees in the case of a link outage on the last hop. In the case of a local TCP implementation `SYN`-segments would be either dropped or buffered at the access point. The peer TCP would not recognize the link outage unless it reaches the maximum number of retransmissions. In the case of ReSoA, TCP might send reset segments because TCP runs on the ReSoA-server and might have not received the listen command yet. There are several possible workarounds for this problem. One possibility would be to change TCP at the ReSoA-server to not immediately send a reset segment, but to query the ReSoA-server. The ReSoA-server could then test whether the last hop is available. Since we expect this case to be rare, it is ignored.

Passive Open - `accept`-function

The connection establishment of TCP operates independent of interactions with the user/application. Whenever a new connection is ready to be accepted (TCP's three way handshake is complete), it is queued, unless the number of queued connections exceed the limit specified by the `listen`-call parameter. If ReSoA is used, the TCP instance that established the TCP connection runs on the ReSoA-server and the application which uses the connection runs on the wireless terminal. The complicated aspect of the `accept`-call is that a new socket (communication end-point) is created whenever TCP receives a new `SYN` packet. Since the TCP instance does not run on the end system in the case of ReSoA and the ReSoA-client is responsible for the assignment of remote socket identifiers, the creation of a new socket object initiated by the ReSoA-server does not fit into the addressing concept. A decision has to be made whether new but not yet accepted connections should be passed to the ReSoA-client immediately or only after the application has called the `accept`-function.

First, let us assume that all pending connections are only queued at the ReSoA-server. In this case a new socket half is created for each new TCP connection. This socket object is linked with the socket which is in `LISTEN`-State as it would be the case of a local socket implementation. However, the additional fields (like the remote socket identifier) are left empty. Whenever the application calls the `accept`-function, the function call is transmitted to the ReSoA-server using the request-response service of the EP. The application is blocked until the response message is received. The ReSoA-client sends a new remote socket identifier together with the request message. The ReSoA-server takes the first unaccepted socket from the list, assigns the remote socket identifier to this socket, and sends the information about this socket (e.g. destination address and port) to the ReSoA-client.

Unfortunately this approach has some shortcomings. First, the ReSoA-client sends the accept message to the ReSoA-server, not knowing whether there are any connections pending. Thus, it cannot determine when a response message will be received (servers can be idle for a long time). Therefore we would loose time compared to a local socket interface. In the case of a local interface the TCP connection would be pending at the end-system. Thus, after the `accept`-call the application could start at once consuming and sending data. In the case of ReSoA, the accept message and its response must first be transmitted over the access network. In the worst case the wireless link would be error-free if a TCP connection is established (thus it would be possible to deliver the information to the end-system), but interrupted (for some time) when the application calls the `accept`-function. Thus, this scheme might delay work although it could be started immediately.

To avoid the first problem, the ReSoA-server could send a notification message to the ReSoA-client if it has a new pending connection (but without including information about the connection). The ReSoA-client would only send the accept message if there is at least one pending connection. Otherwise the ReSoA-client would wait locally until such a notification is received. However, this does not help with the second problem.

In order to overcome the second problem, connections should be reported to the ReSoA-client at once after the three way handshake is complete. As mentioned above this introduces an addressing conflict. The remote socket identifier is assigned by the client. Therefore the ReSoA-server cannot know which identifier to use for the new socket half. One possible solution would be that the ReSoA-server first sends a notification informing the ReSoA-client that there is a new connection available. The client then sends a remote socket identifier to the ReSoA-server which in turn sends the information about the new socket to the ReSoA-client. However, the ReSoA-server cannot remove the socket from the queue of pending connections since this would violate the number of pending connections specified with the `listen`-call. Instead, the ReSoA-client must send an additional message to the ReSoA-server whenever an application has accepted one of the pending connections using the `accept`-call. Thus, we would need four export protocol messages to implement the `accept`-call.

In order to reduce the number of necessary messages we extend the addressing scheme for this case.

If a new TCP connection is established the ReSoA-server sends a notification message to the ReSoA-client. This message carries all information about the new socket. To identify the corresponding socket part the ReSoA-server uses the remote socket identifier of the originating socket plus an incarnation number which is incremented for each new connection. If the ReSoA-client receives such a notification it creates a new local socket half and assigns a new remote socket identifier to it. Whenever the application calls the `accept`-function, the pending socket is immediately returned to the application and the ReSoA-client sends an update message to the ReSoA-server. This message informs the ReSoA-server about the fact that a pending connection has been consumed and returns the new remote socket identifier. Thus, the `accept`-call is implemented using just two messages without introducing additional delay (compared to a local socket interface).

With this design of the `accept`-call, the handling of non-blocking sockets or sockets operating in asynchronous mode is implicit. The ReSoA-client knows whether there are connections pending and can therefore immediately decide whether a call to the `accept`-function should return with which status (new connection or error). It also is able to notify an application about the reception of a new connection.

Connection Termination

`close`-function

For the design of the `close`-call we have to consider four cases. Either the ReSoA-client or the corresponding host call the `close`-function first. In both cases the linger option may be set or not.

Let us start with the case where the application at the ReSoA-client calls the `close`-function and the linger option is not set. This means that the execution of the `close`-function does not depend on the state of the two communicating TCP instances. It returns immediately after it has performed the local (protocol dependent) actions[5]. Since the `close`-function will succeed if the socket descriptor is valid, it is not necessary to block the application until the close request is forwarded to the client. As with a local implementation of the socket interface, the `close`-function returns immediately in ReSoA.

The effects of the `close`-function depend on the state of the socket buffers. If the `close`-function is called while the receive buffer is not empty, the local TCP instance sends a reset segment instead of a `FIN`-segment. Unsent or unacknowledged data is discarded. If the receive buffer is empty while the send buffer is not, all data is transmitted before the connection is closed. If both buffers are empty, the `FIN`-segment will be sent immediately. For ReSoA this means that the state change of the socket must be signaled to the ReSoA-server immediately, even if there is data in the send buffer of the ReSoA-client that has not been forwarded to the ReSoA-server yet. It is necessary to inform the ReSoA-server about the state change since the TCP instance must send a reset segment if additional data is received. On the other hand, it is not possible to set the corresponding socket at the ReSoA-server to the `CLOSE`-State since this would mean that no further data from the ReSoA-client would be accepted. To overcome this problem a new socket state is introduced which we call `CLOSE_DATAPENDING`. When a socket object at the ReSoA-server is in this state, no more data will be accepted, and the TCP instance is triggered to send a reset segment if new data packets are received.

The socket object at the ReSoA-client is released if the receive buffer is not empty or if both buffers are empty. In this case no further messages regarding this socket are expected from the ReSoA-server. If the receive buffer is empty but the send buffer is not, the ReSoA-client has to inform the ReSoA-server when the last data message was forwarded. For this purpose the ReSoA-client sends a second close message with the the state field set to `CLOSE`. After this message is sent the resources related to this socket are freed. If the ReSoA-server receives data from the corresponding host while the socket is in the `CLOSE_DATAPENDING`-State it triggers the transmission of a TCP reset packet (as the protocol stack

[5] The `close`-function is normally implemented via the `shutdown` and `release`-function.

would do, if the state were closed). After this it sends an update message to the ReSoA-client indicating that the socket state has changed to the `CLOSE`-State. If the socket is in the `CLOSE`-State no further messages related to this socket are exchanged.

From the application point of view the socket is released when the `close`-function has returned. Thus, if the application tries to use the socket descriptor after calling close, it is the responsibility of the ReSoA-client to return the appropriate error code indicating that the used socket descriptor is invalid.

If the linger option is set, the call to the `close`-function must not return before all data is acknowledged by the TCP instance at the corresponding host or the linger timer has expired. Basically the operation is identical to the previous case. The only difference is that now the socket part at the ReSoA-client is not released before either the linger timer expires or the ReSoA-server has informed the ReSoA-client that all data has been acknowledged. To implement this, the socket at the ReSoA-client is set to a state called `CLOSE_LINGER`. The ReSoA-server sends a close update message to the ReSoA-client when all data has been acknowledged.

The linger timer is started by the ReSoA-client as soon as the application calls the `close`-function. Only the ReSoA-client has a linger timer. If the linger timer at the ReSoA-client expires before it receives the notification that all data has been acknowledged by the destination host, it returns an error message to the application and sends an update message to the ReSoA-server indicating the expiration of the linger timer. Furthermore, all data remaining in the send buffer is discarded as it is the case for a local protocol stack.

With this implementation of the `close`-function it is possible that all data has been acknowledged by the peer TCP instance but the ReSoA-server was unable to deliver the notification to the ReSoA-client during the linger interval. In this case the ReSoA-client returns an error to the application although all data was successfully delivered. However, under such network conditions the linger timer would also expire in the case of a local implementation of the Internet protocol stack. In this scenario TCP would either be unable to send its data over the wireless link or the access point would not be able to forward the acknowledgments. The difference is that TCP would not have received all acknowledgments, but from the calling application point of view this does not matter. The implementation of the `close`-function is summarized in Figure 6.7.

Now let us consider a scenario were the corresponding host invokes the `close`-function. The problem here is that the ReSoA-server cannot know whether the corresponding host has set the linger option or not. Thus, it must always assume that it is set.

If the `linger`-option is set, a `close`-call will not return until either all data (including the `FIN`-segment) is acknowledged by the peer, or a local timer expires. If the return value is positive the calling application can be sure that all of its data is (at least) stored in the receive buffer of the destination host. If the timer expires before all data was acknowledged a negative value is returned. To maintain these semantics ReSoA has to prevent that the `close`-call performed at the corresponding host returns before all data has reached the ReSoA-client, although the TCP instance at the ReSoA-server normally acknowledges all data segments before they are received by the ReSoA-client. This is achieved by a modification to the operation of TCP's behavior (see also Section 6.3). The TCP protocol machine is modified to delay the acknowledgment for the `FIN`-segment until the ReSoA-server allows it. The ReSoA-server triggers the transmission of the final acknowledgment if it is sure that all data has been received by the ReSoA-client. Section 6.3 gives more information about the modifications to the TCP protocol state machine. Since the `close`-call at the corresponding host does not return before all data including the `FIN`-segment is acknowledged, ReSoA preserves the semantics.

If TCP receives a segment with the `FIN`-flag set it informs the corresponding ReSoA-server module about this event. The ReSoA-server delivers all received data to the ReSoA-client and then sends a close-notification to the ReSoA-client. If the ReSoA-client receives the close-notification it can be sure that it has received all data since it operates on top of a reliable service. After the reception of this notification

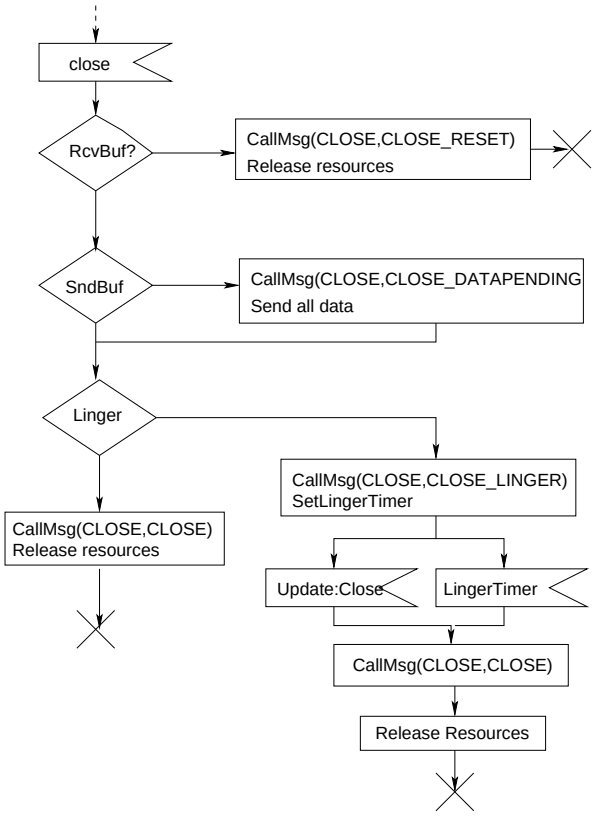


Figure 6.7.: Flow chart of close implementation at the ReSoA-client

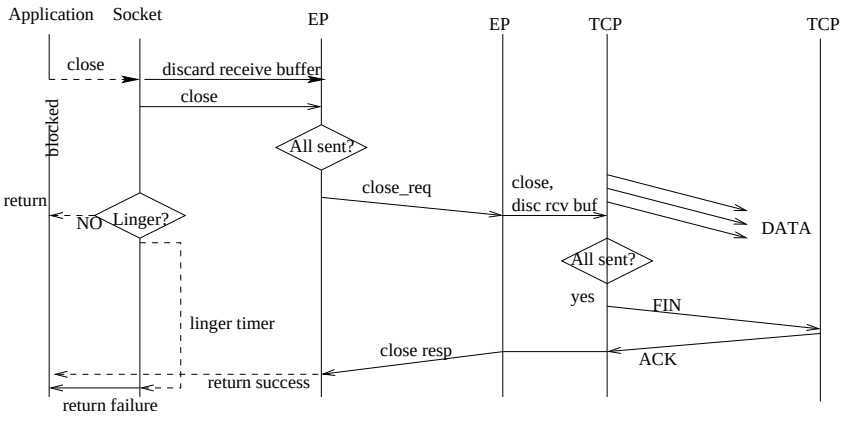


Figure 6.8.: `close`-call ReSoA - wireless host

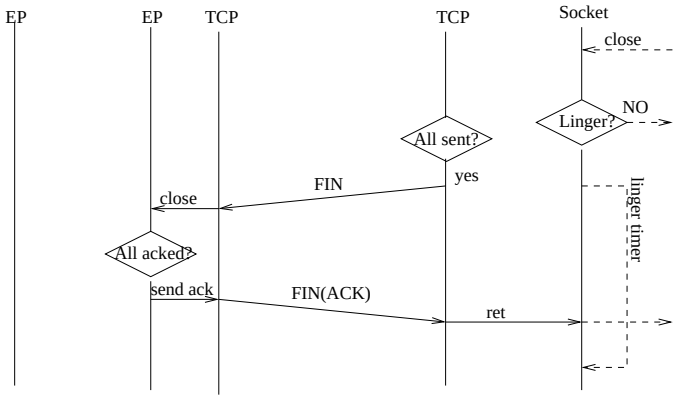


Figure 6.9.: `close`-call ReSoA - corresponding host

it sends a status packet to the ReSoA-server acknowledging the reception of all data. If the ReSoA-server receives this packet it triggers the transmission of the TCP acknowledgment packet. The corresponding socket objects at the ReSoA-client and ReSoA-server are released after the local application has called the `close`-function.

`shutdown`-function

The `shutdown`-function permits to close a connection without releasing the allocated resources. It is possible to select whether only the send, the receive, or both parts should be closed. Basically the `shutdown`-function behaves like to the `close`-function. The linger option is also valid for the `shutdown`-function. Since the ReSoA-server cannot distinguish whether an application at the corresponding host has called `close` or `shutdown` (in both cases as TCP `FIN`-segment is received) we only have to consider the case that the application at the ReSoA-client uses the `shutdown`-function.

Basically the realization of the `shutdown`-call is identical to the realization of the `close`-call. The only difference is that different commands are used whereas the commands depend on the usage of the `shutdown`-call and that the resources are not freed.

A possible usage of the `shutdown`-function is to test whether all data has been read by the destination application as shown in Section 4.1.2. In this chapter we show that ReSoA does preserve these semantics. For this discussion we consider two scenarios. First, we look at the case in which the corresponding host sends data to the ReSoA-client, and then we look at the reverse direction.

Let us start with the case where the application at the corresponding host (called server) sends data to the ReSoA-client. After the server has passed all of its data to the socket it calls the `shutdown`-function informing the peer that it has no more data to send. After the `shutdown`-function call has returned it calls the `read`-function, waiting for data from the client application.

The TCP instance at the ReSoA-server acknowledges the data and passes it to the ReSoA-client. The application at the ReSoA-client reads data until it reaches the end of file marker, indicating that it has consumed every byte up to the `FIN`-flag. If the client application has no own data to send it releases the socket using the `close`-function. This triggers TCP to send a FIN packet. Since the application at the remote host is blocked waiting for data, it will be woken up after receiving the FIN packet. To the application this indicates that all data must have been consumed, since otherwise the application at the other end would not have closed the connection. If the connection is closed before all data has been

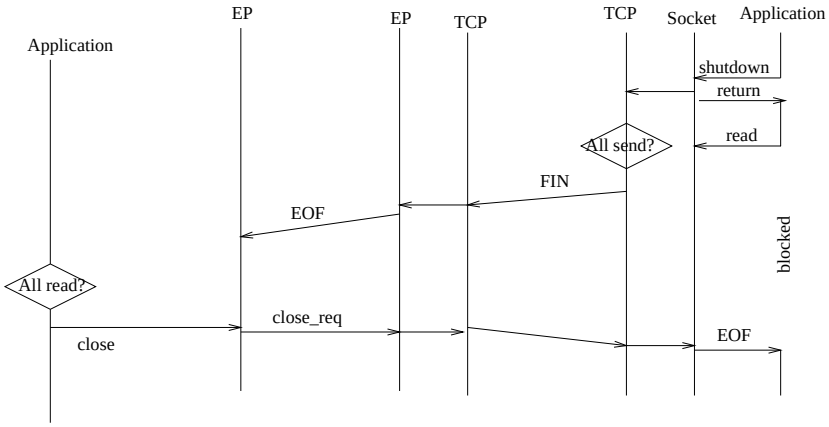


Figure 6.10.: `shutdown`-call on corresponding host when wireless host uses ReSoA

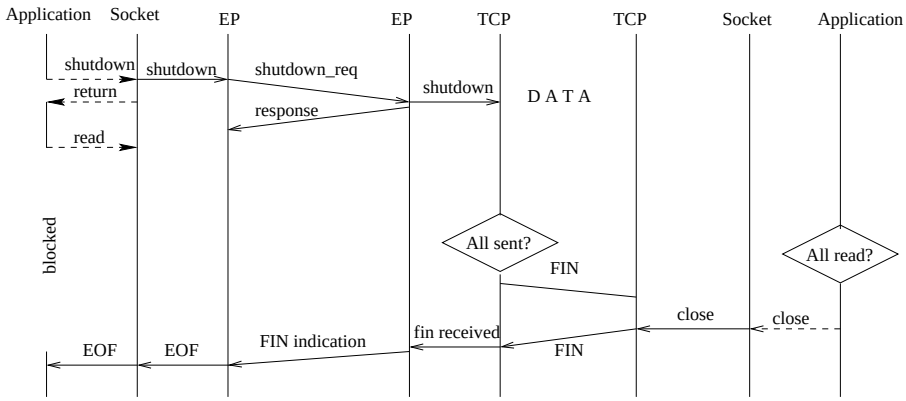


Figure 6.11.: `shutdown`-call on wireless host using ReSoA

read (e.g. the client crashes), the TCP instance at the ReSoA-server will send a reset packet. Since the TCP instance at the ReSoA-server will not acknowledge the `FIN`-segment until all data has reached the ReSoA-client, this is possible as long as it would be the case with a local TCP/IP protocol stack. Thus, the semantics of the `shutdown`-call are not changed by ReSoA.

Now let us consider the case where the application at the ReSoA-client is the data source and the corresponding host is the data sink. After the application has sent all of its data it calls the `shutdown`-function to close its half of the connection followed by a call to the `read`-function. All data is reliably delivered to the server. After the server has consumed all data it calls the `close`-function. If the TCP instance receives the `FIN`-packet it notifies the ReSoA-client which in turn wakes the application up, signaling that the connection was successfully closed. As for the TCP case the application can be sure that all data has been consumed by the destination application. If the application at the corresponding host closes the connection before it has read all data a reset segment is sent. Since the application is blocked by a `read`-call it will be informed about the abrupt termination of the connection.

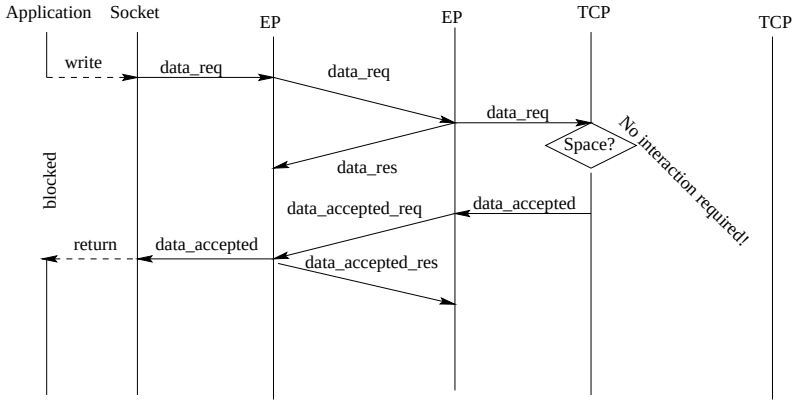


Figure 6.12.: `write`-call: version 1 using ReSoA

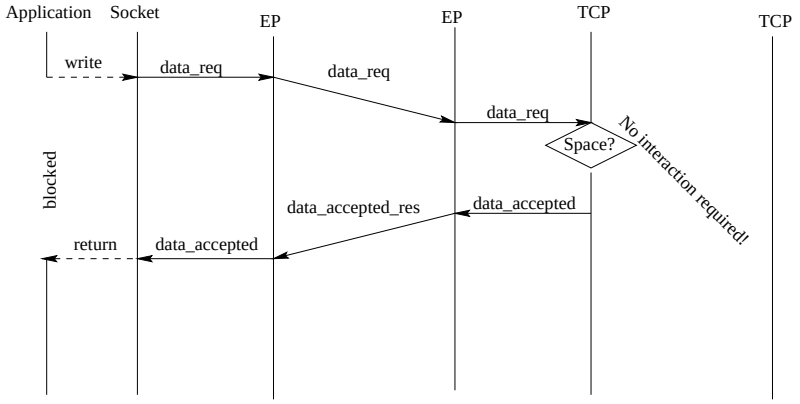


Figure 6.13.: `write`-call: version 2 - using ReSoA

6.2.4. Data Exchange

`write` calls

As discussed in Section 4.1 on page 41, the `write` functions operate asynchronous to the protocol operation. A `write`-call does not necessarily trigger an immediate interaction between the TCP instances. It will return if the data fits into the local send-buffer. This means that the `write`-functions return before the data is sent to the peer or even acknowledged by the peer. If the data does not fit into the send buffer, a blocking socket blocks the function call and a non-blocking socket returns a negative value (`-EWOULDBLOCK`). The application is not explicitly informed about success or failure of delivery. It optimistically assumes that everything functions properly as suggested by the interface description of RFC 793.

Because of the specific semantics of the `write` calls the design of an exported interface looks straight forward at first glance. The first design option is shown in Figure 6.12. In this approach the ReSoA-client simply forwards the write request to the ReSoA-server. If TCP can store the data in its send-buffer the

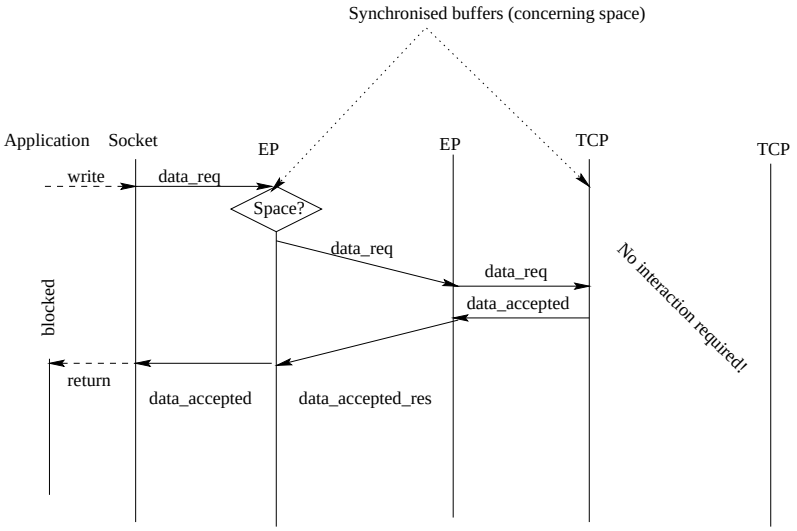


Figure 6.14.: `write`-call: version 3 - using ReSoA

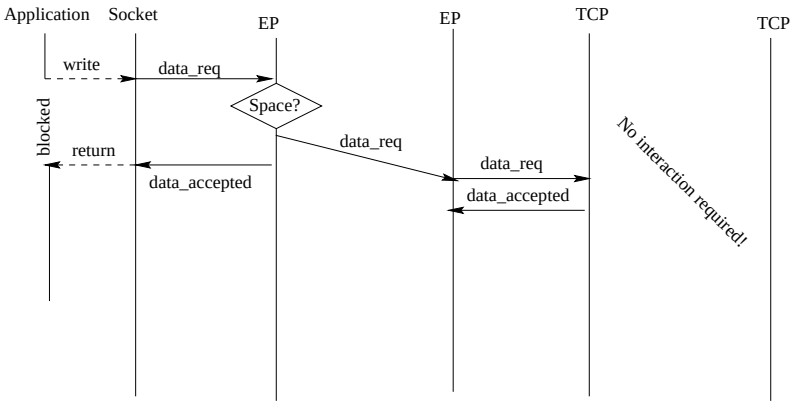


Figure 6.15.: `write`-call: version 4 - using ReSoA

ReSoA-server informs the ReSoA-client and the `write`-call returns.

Unfortunately, this strategy might lead to an increased response time seen by the application and hence would reduce performance. The response time increases since the `write`-call and the response are sent over the wireless link and therefore depend on the network properties as well as the current channel state. If the application protects a `write`-call with a timer it might assume that the write function has failed. The performance can be reduced, since this strategy leads to a Send-and-Wait like behavior on the channel between ReSoA-client and ReSoA-server. This results in poor performance for large RTT values.

Due to these drawbacks we decided to use a different approach. We equipped both ReSoA-client and ReSoA-server with a send-buffer for each created socket. The send-buffer at the ReSoA-client has the size requested by the application (the same size it would have in the case of a local socket implementation). The send-buffer of the ReSoA-server can have an arbitrary size[6]. The ReSoA-client not only manages its local send-buffer but also the send-buffer of the ReSoA-server.

For each `write`-call the ReSoA-client checks whether the data fits into the send-buffer of the ReSoA-server. If the ReSoA-server has sufficient resources the request is sent to the ReSoA-server using the call service of the EP. If the ReSoA-server has insufficient resources the ReSoA-client stores the data in its local send-buffer and returns the control to the application. In this case the `write`-call is delivered later to the ReSoA-server, probably together with succeeding `write`-calls[7]. If the local send-buffer is also full the calling application is blocked until space becomes available. With this approach a `write`-call returns as soon as the data was accepted by the ReSoA-client. Thus, the transmission of data (seen by the application) is decoupled from the transmission of data as it is the case if TCP is located at the end system. Another advantage of this approach is that the size of the send-buffer at the ReSoA-server can be used for flow-control between the two socket halves and to regulate the resource usage at the ReSoA-server.

In order to manage the send-buffer of the ReSoA-server, the ReSoA-client has to know the size of the ReSoA-server's send-buffer and has to maintain a counter reflecting the current state of the ReSoA-server's send-buffer (per socket). Whenever data is forwarded to the ReSoA-server the counter is increased by the number of bytes of the `write`-request. The counter is decreased if the corresponding TCP instance at the ReSoA-server receives a new acknowledgment. Upon reception of a new acknowledgment the ReSoA-server informs the ReSoA-client about the amount of data removed from its send-buffer using the `update`-message of the EP. All socket options related to the send-buffer only effect the attributes on the ReSoA-client and do not need to be sent to the ReSoA-server.

With this design we took advantage of the local semantics of the `write`-call. For the application, which relies on the reliable transport system, it is not important which reliable protocol is responsible. If the ReSoA-client is unable to deliver the data to the ReSoA-server neither is TCP.

Application at Corresponding Host Sends Data

If the application at the corresponding host sends data, ReSoA introduces the problem that the segments are acknowledged by a TCP instance at the ReSoA-server and not by a TCP instance at the end system. However, as pointed out in the previous section, the `write`-call of the socket interface is not coupled with the message exchange of TCP; it only makes sure that TCP has accepted the data for delivery, which is also true for ReSoA. As can be seen when comparing the interaction diagrams, the `write`-call returns in both cases before the data is sent to the receiving host. Thus, with or without ReSoA the application can falsely assume that its data has been read by the remote application. The TCP acknowledgment only means that data has reached the peer TCP instance but says nothing about whether the data was read by the application. As pointed out by Saltzer et al. in their paper about end-to-end arguments in system

[6] It must be large enough to fill the pipe between the two TCP instances, otherwise TCP's performance might suffer.

[7] This is only possible for TCP sockets since TCP establishes a byte stream between two applications. For UDP sockets the message boundaries must be preserved.

design[149], end-to-end reliability can only be implemented by the application or even the users. Even with a local implementation of the protocol stack it is possible that the machine crashes after having acknowledged some data but before the data was consumed by the application. By delaying the final acknowledgment (see Section 6.2.3) ReSoA guarantees that it can send a reset packet if an application closes a connection without consuming all data.

Reading Data

Generally the `read` functions only have local semantics. Packets are not necessarily exchanged if an application consumes data. As shown in Figure 4.7 on page 52, the acknowledgment belonging to a data segment is sent when the data is received (or a bit later, due to the delayed ack timer) and not when the data is consumed.

For the design of the `read` functions at least two alternatives exist. Either the ReSoA-server stores all received data until the client calls one of the `read` functions, or it forwards the received data immediately to the ReSoA-client without waiting for an explicit read request. In the first case the ReSoA-server will not pass more data to the ReSoA-client than requested or available. In order to keep the response time of the system comparable to a local socket interface and to use the wireless link as early as possible[8] we decided to follow the second approach. Every received packet is delivered to the ReSoA-client as early as possible using the update service of EP. Due to flow control (see below) it is possible that data is not immediately forwarded to the ReSoA-client upon reception.

One difficulty with this approach is the management of TCP's Advertised Window at the ReSoA-server. If data is consumed from the receive-buffer, which is the case when data is passed to the ReSoA-client, TCP increases its Advertised Window. However, in the case of ReSoA the data is just transferred from the ReSoA-server's receive-buffer to the ReSoA-client's receive-buffer and is not consumed by the application. Hence, the Advertised Window must not be increased.

To solve this problem the interaction between the socket layer and TCP needs to be modified. If data is received by the TCP instance of the ReSoA-server it is stored in the receive-buffer of the associated socket object of the ReSoA-server as it would be stored in the receive-buffer of a socket in the case of a local socket implementation. If flow control allows the ReSoA-server reads data from the receive-buffer and sends it to the ReSoA-client without modifying the byte count of the receive-buffer (hides the consumption of data from TCP)[9]. If the application at the ReSoA-client reads data the ReSoA-client sends an update packet to the ReSoA-server indicating how much data was read[10]. After reception of such an update packet the ReSoA-server informs the local TCP instance that receive-buffer space is available.

Similar to the `write`-calls we need one receive-buffer per socket at the ReSoA-server and the ReSoA-client. The receive-buffer at the ReSoA-server has to have the size requested by the application. Thus, all the socket options related to the receive-buffer are passed to the ReSoA-server using the request-response service of the EP. The receive-buffer at the ReSoA-client can have an arbitrary size. The size of the ReSoA-client's receive buffer can be used to control the flow of data between ReSoA-client and ReSoA-server.

6.2.5. `select`-function

An application can specify a list of descriptors (sockets) with the `select`-function to check for pending events. The `select`-function can wait for data to arrive, write-buffer space to become available (write would not block), and exceptions. Furthermore it supports a timeout mechanism.

[8] Due to the nature of wireless communication it might be the case that the channel is in good state (communication is possible) while data is received but not while the ReSoA-client request the data.

[9] How this can be achieved depends on the implementation of TCP.

[10] This information can be piggybacked.

To implement the `select`-function in ReSoA, two options are available. The `select`-function can either be implemented at the ReSoA-server or at the ReSoA-client. In the first case the `select`-function, including the lists of descriptors, must be forwarded to the ReSoA-server. The ReSoA-server has to inform the ReSoA-client whenever one of the specified events occurs. In the second case the ReSoA-client must be informed about all state changes.

ReSoA implements the second approach. The `select`-function is only executed at the ReSoA-client and not forwarded to the ReSoA-server. A transfer of the `select`-function to the ReSoA-server is not necessary because the ReSoA-client is automatically informed about all state changes required to implement this function. As described in Section 6.2.4, the ReSoA-server immediately transfers data received from the Internet to the ReSoA-client. Thus, the ReSoA-client knows whether a `read`-function call will block. The same is true for new connections, as described in the `accept`-call section (Section 6.2.3). Furthermore the ReSoA-client knows how much write buffer space is available for every socket. Any exception is passed to the ReSoA-client immediately. The other approach where the `select`-function is passed to the ReSoA-server would have had the drawback that an additional delay is introduced. The implementation of the timeout is complicated since the ReSoA-server cannot know how long it took the select message to reach it.

6.2.6. Host and Service Information

Two types of functions must be differentiated. The first group deals with name and address resolution of Internet hosts and addresses, while the second group deals with querying a specific socket.

The first group of function calls can be simply forwarded to the ReSoA-server where they are executed. This means that a ReSoA-client does not need to know a Domain Name System (DNS) or the correct Network Information Service (NIS) bindings.

For the second group again two options exist. Function calls like `getsockname` and `getpeername` can be forwarded to the ReSoA-server using a request PDU. The ReSoA-server replies with the result, or the function calls can be implemented locally. For a local implementation the ReSoA-client has to know the Internet addressing quadruple. Since this quadruple is sent to the ReSoA-client with the response message of the connect request (see Section 6.2.3), all information is available to answer the function calls locally. Therefore we decided to use the local implementation.

6.3. Interaction between ReSoA-server and TCP Protocol Machine

In the last section we pointed out that it is necessary to modify the behavior of the TCP instance at the ReSoA-server in order to maintain the socket semantics. In this section we discuss the required modifications and argue why they are transparent to the peer TCP entity.

The behavior of TCP was modified at three points. The first two points concern TCP's behavior if a `FIN`-segment is received. The last point deals with the management of the Advertised Window. Figure 2.2 shows the TCP state diagram and Figure 6.16 and Figure 6.17 show the required modifications. Two additional states are necessary and TCP's behavior in the `ESTABLISHED`-State has to be changed. The changes in the `ESTABLISHED`-State concern the flow control as discussed in Section 6.2.4. Since these modifications depend on the interface between the socket object and the TCP (protocol) implementation we cannot present further details here. Modifications in this direction do not modify TCP's behavior since TCP is expected to increase the Advertised Window whenever an application reads data. This solely depends on the application.

Figure 6.16 shows the necessary modifications for the case where a `FIN`-segment is received while TCP is in the `ESTABLISHED`-State. In this case the TCP entity at the ReSoA-server should not send an acknowledgment immediately as it would normally do. Instead it should delay the acknowledgment until all

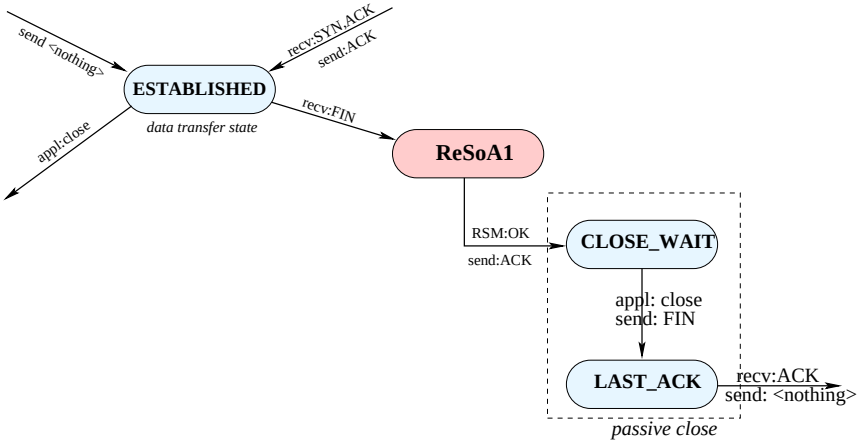


Figure 6.16.: Modification number 1 to TCP state transistion diagram

data from the correspondent host was successfully sent to the ReSoA-client. This modification is required to support the `LINGER` option at the distant TCP entity as described in Section 6.2.3. If the ReSoA-server sends the acknowledgment for the `FIN`-segment immediately, the application at the corresponding host could wrongly assume that the data has reached the other end. If the `LINGER` option is not used at the distant TCP entity it is not necessary to delay the acknowledgment. This is because the application assumes the successful delivery of all data, even if it is lost somewhere in the Internet. Unfortunately the ReSoA-server cannot determine whether the application at the far end uses the `LINGER` option or not. Therefore it always has to assume that this option is set and hence delays the final acknowledgment.

If the communication over the wireless link is slow (e.g. due to its error prone nature) the TCP entity at the far end will time out and repeat the `FIN`-segment. This is not really a problem because the `FIN`-segment or the corresponding acknowledgment could have gotten lost on its path between the communicating TCP entities. Thus, the other end cannot distinguish between a delayed final acknowledgment and a loss or a delay in the Internet. Due to this ambiguity this modification does not change the behavior of TCP observed by its peer entity.

The second modification effects the active connection termination as illustrated in Figure 6.17. The reasoning is the same as for the first case. If a `FIN`-segment is received the corresponding acknowledgment should not be sent before all data has been sent to the ReSoA-client.

6.4. System Initialization and (De)-Registration

Looking at a local implementation of the socket interface, the kernel (e.g. Linux operating system) is ready to create new sockets of a specific type after the corresponding socket family has been registered with the kernel. Normally AF_INET sockets are built in.

If the local socket interface is replaced by ReSoA, ReSoA-client and ReSoA-server must be configured. Both need to know which LHPs are used for TCP and UDP sockets. According to our LHP interface specification (see Appendix C.1) LHPs are specified by a provider identifier and a protocol number.

Next, the ReSoA-server must be started. It prepares the local LHP instances to accept incoming connections and then awaits ReSoA-clients to register. In RPC terminology the ReSoA-server implements a dormant server. According to the LHP interface specification an LHP informs a ReSoA-server about the

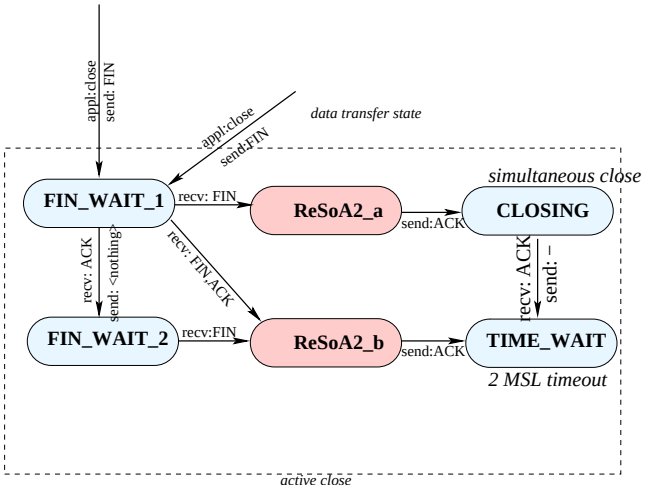


Figure 6.17.: Modification number 2 to TCP state transistion diagram

reception of a new connection request but does not accept the request before allowed by the ReSoA-server.

The ReSoA-client additionally needs the LHP address of the ReSoA-server. When a ReSoA-client is started the initial step is to establish the LHP connections with the ReSoA-server and to register itself at the ReSoA-server. After the registration is completed an application can access the TCP/IP protocol stack at the ReSoA-server without recognizing any difference to a local TCP/IP implementation with respect to the behavior of the socket interface.

The first EP message sent to the ReSoA-server is a registration message. This message is sent using a control PDU. In the current version the registration message only contains two fields which specify the Internet addressing mode and the ReSoA version installed at the ReSoA-client. The registration procedure has to be extended by security mechanisms in a later version. If the ReSoA-server receives a registration message it tests whether it has sufficient resources and whether it supports the selected addressing mode. If one of the tests fails it sends a control message indicating the error reason. If all tests succeed the ReSoA-server adds the ReSoA-client to its list of registered clients and sends a control message indicating the successful registration. The response message contains the IP address used for this ReSoA-client and the maximum number of sockets the client may have open at any time.

If the ReSoA-server has insufficient resources available it can inform the ReSoA-client about an alternative ReSoA-server in the negative reply.

If a ReSoA-client wants to finish its session with a ReSoA-server it should de-register with the ReSoA-server. The deregistration can be either graceful or abrupt. A graceful deregistration is only possible after all sockets of the client have been closed at the end system and the ReSoA-server. If the ReSoA-client tries to de-register from a server while there are open sockets, an error message is returned and the de-registration fails. An abrupt de-registration is always possible. In this case all resources at the ReSoA-client and ReSoA-server are released and a reset packet is sent to all open TCP connections.

6.5. Error Conditions

Due to the split implementation of the socket interface error situations can occur that are impossible with a local implementation. These error conditions are discussed in the following paragraphs.

6.5.1. Invalid Messages

We assume that the EP entities at the ReSoA-client and ReSoA-server work correctly. Nevertheless, invalid messages are possible if a ReSoA-client is not registered with the ReSoA-server, an invalid remote socket identifier is used (e.g. after a crash of ReSoA-server or ReSoA-client), or if a malicious attacker is involved. All messages from an unregistered ReSoA-client are simply discarded by the ReSoA-server. A ReSoA-client discards all messages that are not originated by the ReSoA-server. If the ReSoA-server or ReSoA-client receive a message with an invalid remote socket identifier they send a control message with the error type set to `INVALID_RSOCK_ID`.

6.5.2. Insufficient Resources

With ReSoA a new socket at the ReSoA-server is created if the socket is used for the first time and not when the socket function is called (see Section 6.2.1 on page 78). Thus it is possible that the ReSoA-server is unable to create a socket, which however is assumed to exist by the application. In this case the ReSoA-server must reply with a control message with the type control field set to `ERROR` and the error reason set to `ENOBUFS`.

6.5.3. Protection of Request Messages

As mentioned in Section 6.1.3, the ReSoA-client protects each request with a timer. If there are communication problems, or if the ReSoA-server takes longer to complete a request than expected by the ReSoA-client, this timer will strike. In this case the ReSoA-client cannot repeat the request, since the EP has no mechanism to detect duplicates. On the other hand simply dropping the request and reporting an error to the calling application would be too pessimistic. Instead the ReSoA-client asks the local LHP instance whether it has observed any communication problems (e.g. current number of timeouts, retransmissions). If the LHP reports communication problems the timer is restarted. If the communication link is good the EP entity at the ReSoA-client sends a control message to the ReSoA-server testing whether the ReSoA-server is still working on a request for the specified socket. The ReSoA-server must answer this query immediately. If the ReSoA-client receives the reply message it increases the timeout value and restarts the timer. If the timer has expired several times (currently 16), the ReSoA-client assumes serious problems and disconnects from the ReSoA-server. In this case it must release all sockets.

6.5.4. Link Failure

If the link between the ReSoA-client and the ReSoA-server is not available, no packets will be exchanged between ReSoA-client and ReSoA-server. If TCP would be located at the wireless end system, all packets would be either dropped at the access point or queued at a reliable link layer protocol. Nevertheless, the peer TCP instance would neither receive any data packets nor any acknowledgments. After some time the TCP instances at the peer and at the local host (whoever has unacknowledged data) would time-out and start retransmitting all unacknowledged data. After a number of timeouts the application would be informed about the connection failure and the connection would be closed. If the link-outage is only temporary the application will only recognize the link outage by means of a reduced throughput.

If ReSoA is used, this is different. Because TCP is located before the intercepted link, the TCP entity will continue to send (or retransmit) packets according to its policy, as long as unsent or unacknowledged data is available. All incoming packets are acknowledged. The peer TCP instance is not stopped until the corresponding receive buffer at the ReSoA-server is full. The receiver buffer fills up, since the ReSoA-server is not able to deliver the data to the ReSoA-client. If the link outage is only temporary the reliable LHP guarantees that the data is delivered to the ReSoA-client as soon as the wireless link is available again.

Any messages queued will be delivered, and the states at ReSoA-client and ReSoA-server, respectively, will catch up.

If the link outage is too long (with respect to the LHP retransmission count or internal timer), the LHP will have to signal the abrupt termination of the LHP connection to the service user (either ReSoA-client or ReSoA-server) using the `LHP_DISCONNECT_IND` service primitive. If the ReSoA-server receives this signal it must close all sockets and associated TCP connections. Two different possibilities exists for the termination of the TCP connection. First, the ReSoA-server can trigger TCP to send reset segments on all open connections. Second the ReSoA-server can only trigger those TCP connections to send a reset segment which have a full receive buffer. All other TCP connections are set to a mode in which they discard all subsequent packets. Thus the peer TCP recognizes the link failure as in the normal case due to expiration of the retransmission threshold. Currently we follow the first approach since this is easier to realize.

The problem here is that the ReSoA-server might have data which was acknowledged to the remote host but cannot be delivered to the receiving end system. Although this cannot happen if the TCP/IP protocol stack is located at the end system this does not violate the semantics expected by the application. As already pointed out, the TCP acknowledgment means that the data was received by the peer entity but not that it was read by the application. Hence it is possible that an end system or the receiving application crashes after the local TCP instance has acknowledged but before the application has read it. Furthermore ReSoA does not acknowledge the final TCP segment before all data was successfully passed to the ReSoA-client. This allows ReSoA to inform the remote end system about the abrupt termination of a connection. Thus, with ReSoA a link failure corresponds to a crash of an application in the case of a local socket implementation.

If the ReSoA-client receives the `LHP_DISCONNECT_IND` event it sets the error attribute of all open sockets to the connection reset by peer error code. A socket object is released after the next socket call by an application. Upon this call the stored error code is returned. Thus, for the ReSoA-client a permanent link outage looks like a local TCP connection that has reached its retransmission limit or received a reset segment from the peer.

Normally both ReSoA-client and ReSoA-server should get an `LHP_DISCONNECT_IND` signal in the case of a link outage, but not necessarily simultaneously. Hence both remove their state independently from each other. If only one of the two entities receives the `LHP_DISCONNECT_IND`, the states are not longer synchronized and not resynchronizeable. Since the LHP is connection oriented, this error will be detected by the LHP when it sends packets on a connection that was already closed by the peer LHP instance. In this case the LHP notifies the service user using the `LHP_DISCONNECT_IND` signal.

6.5.5. Crash of ReSoA-client

If a ReSoA-client crashes this does not directly effect its open sockets at the ReSoA-server. The ReSoA-server is left, holding state information that is no longer valid but cannot easily be withdrawn. Thus, TCP at the ReSoA-server continues sending, receiving, and especially acknowledging data until it has sent all data and received either all data or its receive buffer is full. This behavior is different from the classical end-to-end case in which all open sockets die with the host (fate-sharing principle). To synchronize the state of the ReSoA-server it must be informed about the crash.

If the ReSoA-client does not come up again, the LHP instance at the ReSoA-server will detect the lost connection and inform the ReSoA-server with the `LHP_DISCONNECT_IND` signal. Thus, for the ReSoA-server a ReSoA-client crash cannot be distinguished from a permanent link outage. Therefore the discussion in Section 6.5.4 is also valid for this case.

If the ReSoA-client comes up again before the LHP at the ReSoA-server has detected the crash, the

ReSoA-client will try to register with the ReSoA-server (using a new LHP connection). Since the ReSoA-client is still registered, the registration process will fail. The ReSoA-server interprets this registration attempt, as that the ReSoA-client has lost its state and therefore closes all sockets belonging to this client[11]. The ReSoA-client cannot register again before all sockets and attached protocol entities have been closed and all allocated resources have been freed.

6.5.6. Client Application is Killed/Crashes

Normally a crash of an application using ReSoA is not a problem. As with a local implementation of the socket interface, the systems ensures that all sockets are closed. The close requests are simply forwarded to the ReSoA-server.

The difference to the standard case is that it might take some time until the close requests are passed to the ReSoA-server. During this time the TCP entity at the ReSoA-server will still acknowledge data. However, this does not violate the semantics, since this just means that the crash virtually happens some time later and in any case the remote side does not know up to which byte the receiving application has read the received data.

6.5.7. Crash of ReSoA-server

In the case of an end-to-end approach a crash of an intermediate host has no effects as long as the host comes up again fast enough or an alternative route exists. In the case of ReSoA this is different, since with a crash of the ReSoA-server all TCP states are lost. However, this does not effect the semantics. From the remote user point of view this crash is identical to a crash of the end system it is communicating with.

To the local application this looks different. It can still pass data to the local send buffer and receive data as long as there is data available. It will not detect the problem until the LHP signals the failure to the ReSoA-client, or close or shutdown are called. In the first case the problems are signaled to the application with the next function call. In the second case the application is blocked until the error is detected.

6.6. Service of the LHP

ReSoA-client and ReSoA-server must be tightly coupled with the virtual links established by the lower layer, since they must be informed whether the LHP is unable to deliver a message, or if the link is unavailable. Therefore the LHP provides a connection oriented service. Besides an LHP, which provides a reliable service, subsequent optional LHPs might provide application specific services. In the case of UDP sockets a semi-reliable LHP can be used for send and receive socket calls.

The termination of a connection can either be graceful or abrupt. In the first case an LHP must ensure that both the local send and receive queues are empty before the communication endpoint is released. In the case of an abrupt connection termination the LHP must discard all queued messages.

The core functionality of the LHP is the reliable transport of messages between ReSoA-client and ReSoA-server as well as addressing. Reliability here means that every message passed to the LHP is delivered exactly once in the correct order to the service user at the peer node. The LHP is responsible for the detection of transmission errors. All messages delivered to the receiving service user should be error free (at least with a high probability). It is the responsibility of the LHP provider to choose protocol mechanisms for error control, flow control and so forth that are appropriate to offer this service in an efficient manner. The design of the LHP has a high impact on the performance of the whole system.

[11] Security mechanisms are required so that no malicious host can register under a false name.

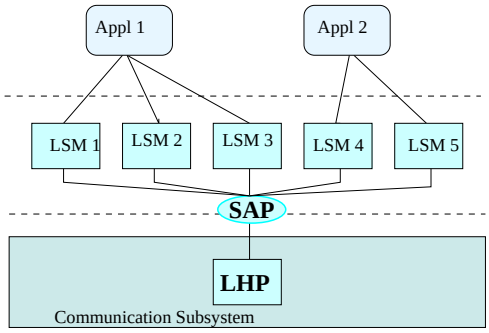


Figure 6.18.: One to one relation between LSM and CS instances

An LHP must preserve messages boundaries and must be able to accept messages of a length of up to 65600 bytes[12]. An LHP must support segmentation and reassembly in the case the deployed technology is not able to carry messages of such length.

The service user needs to protect some of the messages by timers to avoid deadlocks but is unable to measure the RTT between the ReSoA-client and ReSoA-server due to little bidirectional data traffic. Therefore an LHP is responsible for estimating the RTT of LHP-SDUs. The measured value must be delivered to the service user on request.

It is the LHP's task to signal if the peer(s) are no longer available/reachable. If the LHP detects an interruption in the connectivity of a certain length it has to inform the service user. Thus, the LHP must implement some kind of keep-alive functionality.

Finally, the LHP has to inform the service user about any kind of errors like that it was unable to deliver a message or that the peer entity has reset a connection.

[12]The size of data chunks passed via the socket interface is limited by the length parameter of the socket interface. This parameter allows for passing data chunks of up to 2^{32} bytes. However, most protocols limit this size to 65535 bytes. Therefore we decided to support only the latter size. Since the LHP-PDU must carry the data of the socket call as well as the header of the Export Protocol, the maximum size of a LHP-SDU must be larger than 65535.

Chapter 7.

Testing the Semantics of ReSoA against the BSD Socket Interface

7.1. Goal Definition

ReSoA is designed as a distributed realization of the Berkeley socket interface for sockets that belong to the Internet protocol family. In order to show that ReSoA is indeed a realization of the socket interface, three requirements need to be fulfilled:

1. The syntax of the socket API must not be changed.
2. ReSoA must be semantically equivalent to the BSD socket interface.
3. ReSoA must be operationally equivalent to the BSD socket interface.

Semantic equivalence means that ReSoA should respond to input signals exactly as a local implementation of the socket interface would do. If the application uses a certain socket function call, the responses of ReSoA must be identical to the response of the classic interface[1]. Furthermore, after a call to a `socket`-function has returned, the socket and the attached protocol should be in the same state as a local socket interface would be. For example, when the `connect`-call has returned with a positive status, TCP has to be in the established state.

This should not only be true for valid input messages but also for invalid (e.g. not expected) messages. Since semantic issues neglect the time component important for real systems, we added the third requirement. Operational equivalence means that the response time of ReSoA has to be comparable to the response time of the socket interface.

The third point is difficult to show and is best investigated using implementations of both systems (ReSoA vs. local socket interface). Remote procedure calls generally have a longer response time than an equivalent local function call due to the communication delay. This is especially true for wireless channels. In this case the response time depends on the current channel condition. In order to meet the third design goal, the ReSoA-client is designed in a way that most socket calls are handled locally and processed in the background.

The first property is met by design. The ReSoA application level interface is the socket interface, hence the syntax is not changed. This can easily be shown by running typical network applications on top of ReSoA. All applications that we tested were able to use ReSoA without any modification or recompilation. Among others we tested Netscape, Lynx, FTP, and Netperf. These tests are further discussed in Section 7.3.

The focus of this chapter is on the second point, namely the semantic equivalence. We show that ReSoA can be used instead of a local BSD socket object. Neither the local application nor the application at the far end can recognize that ReSoA is used instead of a local socket. Every call to the socket API has exactly

[1] This is not true for every socket function. For example, in the case of TCP and the `read`-function the amount of data returned depends on the available amount. This can be different for ReSoA and a local socket object.

the same semantics in ReSoA as for a local implementation. Thus, any application designed for the BSD socket interface can be executed on top of ReSoA without any change[2].

7.2. Methods

We decided to use two different approaches to show that ReSoA is semantically equivalent to the BSD socket interface. First we ran various tests with our ReSoA implementation using either existing network software like `Netscape` and `FTP`, or self written test programs in order to focus on specific characteristics of the socket interface (like the buffer management).

Second, we developed an SDL specification of ReSoA and the BSD socket interface. A formal specification has two advantages. It a clearly describes ReSoA and we can use the validation support of modern SDL software packages like SDT[1] to compare ReSoA against the BSD socket interface. Also we can show that ReSoA generally maintains the semantics instead of comparing it to one specific socket implementation for one specific operating system. A shortcoming of this approach is that there is no SDL specification of the BSD socket interface. Therefore both systems were specified by ourselves. In order to perform tests a TCP specification is also required. This inflates the state space. As for the BSD socket interface no official formal TCP specification is available.

7.3. Testing the Equivalence Based on an Implementation

If existing applications are able to run on top of ReSoA without any modification, this would be a first indication that we have met our design goals. Especially since most of the popular applications, for instance Netscape, use the socket interface in a more complex way than indicated by our example scenario in Figure 4.2 on page 44. However, showing that an application runs on top of ReSoA is not sufficient to show that ReSoA really operates in the expected way. In order to gain a deeper understanding of the behavior of the local socket interface and ReSoA the different function calls and their return values have to be observed and compared.

Linux provides the `strace` command functionality to observe system calls called by an application during runtime. Therefore it is possible to use any existing application for this test without modifying the source code or to extend the kernel to monitor system calls. This especially allows the observation of the behavior of applications for which the source code is not available.

All of our tests indicate that we have met our design goal. As an example we show a `netperf` extract using a local socket implementation and ReSoA in Appendix E. The entire trace file is too long and would hide the interesting details. However, the test can easily be reproduced.

7.4. Analyzing the Equivalence Using a Formal Specification

7.4.1. Idea

The basic idea to investigate equivalence is to perform a test where ReSoA, seen as a black box, should respond to input signals exactly as the socket interface would do. Generally we would say that the two realizations of an interface are *equivalent* if they generate the same sequence of output symbols when offered the same sequence of input symbols.

The general approach is illustrated in Figure 7.1. This figure shows the reference system with a local socket implementation on the left side, and the system under test with ReSoA replacing the local socket

[2] How ReSoA can be used instead of a local socket if both are available is an implementation issue.

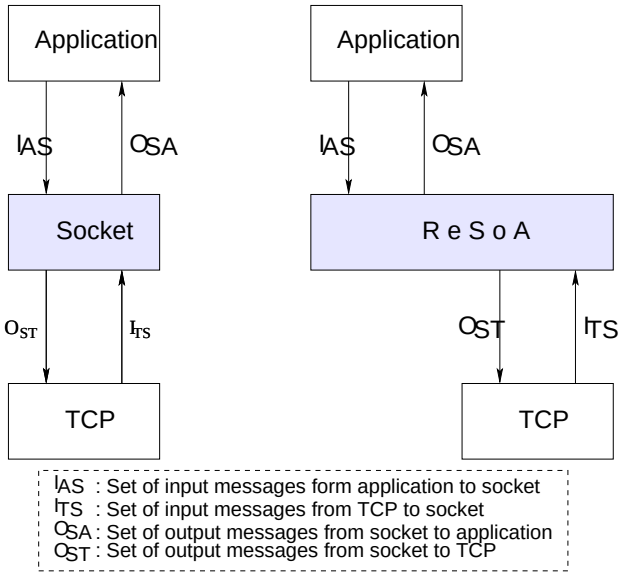


Figure 7.1.: Systems under study

implementation on the right side. In both cases the medium layer is of interest. In both cases it is surrounded by an application and the TCP protocol, which produce the input events and consume the output events of the layer being tested. The application is identical for both systems. The TCP instance is slightly modified in the ReSoA case, since ReSoA requires additional functionality (e.g. delaying the final acknowledgment). Therefore we use an intermediate step, as detailed below, to compare the two systems. In this intermediate step we show that the TCP modifications do not alter the behavior of our reference system.

Our approach to show the semantic equivalence is similar to *conformance testing*. A conformance test is used to verify that the external behavior of a given protocol implementation is equivalent to its formal specification (e.g. see [85]). The difference is that we neither have a reference specification nor an implementation. Instead of comparing an implementation against its formal specification, we intend to compare two systems using System Description Language (SDL) specifications.

As with conformance testing, our approach is to trigger the system under test with input events and observe and compare the output with that of the reference system. At first glance one possible approach to test the two specifications for semantic equivalence would be the following conformance testing algorithm:

1. For all possible combinations of state i and input signal j, perform the following three steps.

   a) Use a reset message to bring ReSoA to its initial state and then use a set message to transfer it to state i.

   b) Apply input signal j. Verify that any output received, including the null output, matches the output required by the socket interface.

   c) Use a status message to interrogate the ReSoA specification about its final state. Verify that this final state matches that of the socket interface.

Unfortunately this approach has two problems. First of all, the system under study has a high (infinite) number of states and input signals. Every variable allocation (e.g. number of bytes in the send buffer) is

a separate state, and every parameter modification of a function call is a different input signal. Second, the socket interface does not necessarily require that the output triggered by an input is deterministic. For example, if the `read`-call is used on a socket attached to a stream protocol (like TCP), the socket interface returns any amount of data between its low water mark and the amount of data demanded by the function call. Thus, if an output message generated by ReSoA is not identical to the output message of the BSD socket interface, this does not necessarily mean that ReSoA does not fulfill to the requirements.

Instead of running exhaustive tests we decided to select an interesting scenario based on our knowledge about the socket interface and to compare the behavior of the two systems for this test under different conditions. The first approach was to compare the Message Sequence Charts of the two systems which were expected to be identical. ReSoA should be equivalent to the local socket interface. Unfortunately, Message Sequence Charts alone are not sufficient for this purpose, as discussed in the following subsection.

Why MSCs do not work

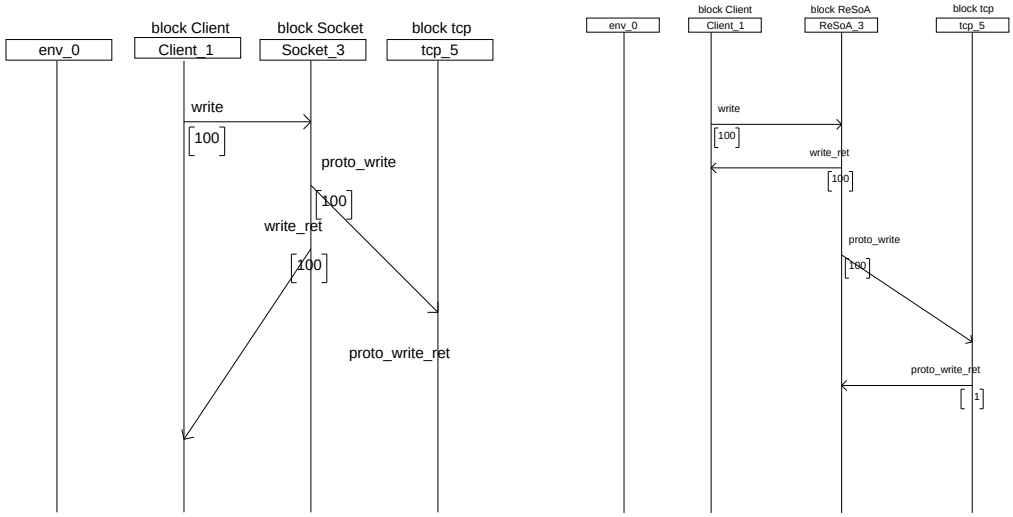
We already pointed out that the number of bytes returned by the `read`-function can be different for ReSoA and the classic socket interface. The reason for this is that ReSoA and TCP use different mechanisms to convey the data over the wireless link. Basically ReSoA is designed to be faster than TCP. The `read`-call returns the number of available bytes unless there is more data available than requested. Since ReSoA can be faster than TCP it is possible that more data is available which is returned upon this call. In this case a MSC verification fails, since the parameters of the read-signal differ. However, this is not a violation of the socket semantics for a stream-oriented protocol like TCP.

As shown in Figure 7.1, the system under test exchanges signals with the application and TCP. To permit the use of automated MSC validation the order of signals must not be changed. For example, when the application uses the `write`-function, two events are triggered. The write request is passed to TCP and the application is informed about the success or failure of the call. If the classic socket interface is used, the order of events is as described above. The status of the `write`-call depends on whether TCP (the reliable protocol) has accepted the data or not. In the case of ReSoA the status of the `write`-function is signaled to the application before the send request is passed to TCP. For performance reasons the write function has to return if the data fits into the send buffer of the local port of the distributed socket object. Later the reliable LHP makes sure that the data is passed to the TCP entity at the ReSoA-server. Thus, in the ReSoA case the order of these two signals is inverted, preventing the use of an automated MSC validation. This problem is illustrated in Figure 7.2.

However, this difference does not violate the socket semantics because the semantics of the `write`-call are that the data was accepted for transmission and not that it was sent or received by the peer. Data is accepted for transmission if it fits into the send-buffer, and this is exactly what ReSoA does.

The last two problems let a validation test fail although the semantics are not violated. However, the reverse case is also possible. An MSC can be verified but the semantics are violated. This happens for instance if the corresponding host uses the `linger` option. In this case the `close`-function should not return until all data has reached the client machine. To achieve this behavior, ReSoA requires a modified TCP entity which delays the acknowledgment of the `FIN`-segment until all data was passed to the ReSoA-client.

Let us now consider the following test: We collect an MSC trace using a client-server scenario using BSD sockets and the `linger` option turned on at the server. Then we try to validate our ReSoA specification against this Message Sequence Chart (MSC) . For the validation we use the unmodified TCP behavior which does not delay the acknowledgment for the `FIN`-segment. Thus, we would expect the validation to fail. The TCP entity acknowledges the `FIN`-packet immediately, so that the `close`-function at the server returns before all data has reached the client machine. Surprisingly, this validation tests succeeds. The reason is that we can only use block level traces because of the differences between both systems. ReSoA is designed to use more than one process, while the socket interface is modeled by a single process. The



(a) MSC trace of local socket interface

(b) MSC trace of ReSoA

Figure 7.2.: MSC violation due to inverse signal order

drawback of a block level trace is that the details of ReSoA are hidden. As indicated by Figure 7.1 ReSoA is viewed as a single module and not as communicating processes. With this level of abstraction it is not possible to distinguish whether the ReSoA-client or the ReSoA-server has received the data. Thus, the validation succeeds because all data was received by ReSoA. To test this scenario we need to know whether all data was received by the ReSoA-client.

Despite these shortcomings we use MSCs and automated MSC validation as a first test, and especially for manual comparison of the two systems. However, further mechanisms are needed to overcome the shortcomings described above.

Alternative Validation Tests

As automated MSC tests fail (manual comparison is possible), another method for automated validation tests is required. This method should not only allow the observation and comparison of signals at the edges of the black box, but also looking into the model and observing its state. Below we discuss four possible candidates. These approaches have in common that we try to observe some a-priori defined behavior. This means for example that we want to know when a specific variable exceeds a threshold or when a specific state is reached. The occurrence of such an event is signaled using different mechanisms.

Explicit Signals

The first idea is to define special signals that are sent to the SDL environment whenever certain conditions are met. For example, the distant host could send a signal to the environment after the `close`-function has returned, and the ReSoA-client could send a signal to the environment after it has received all data (end of file marker). To test whether the semantics of the `linger` option are preserved we must test whether

the signal sent by the ReSoA-client is received by the environment before it is received by the distant host. Since this approach is based on signals, it is possible to use it in combination with automated MSC validation. The levels of the MSC validation are the signals to the environment. Two different possibilities exist for the generation of MSCs. First, the reference system could be instrumented with the same signals. This would allow to generate the MSC automatically. Second, it is possible to specify the MSCs that should be produced by ReSoA manually.

User Defined Rules

User defined rules are one mechanism supported by the Validator of the SDT tool. A user defined rule provides the possibility to define predicates that describe properties of one particular system state. A rule is checked while the Validator walks through the state space. If a rule is satisfied, a report is generated. The used version of SDT only permits a single rule at a time. If multiple rules are needed they must be composed using boolean operators. One possible user defined rule that tests whether the linger semantics are maintained is the following:

```
define-rule sitype(signal(server:1))=close_ret AND
                                          LSM:1->rcv_no_more=true
```

This rule generates a report whenever the server consumes the close_signal and the variable *rcv_no_more* of the ReSoA-client process is true. This is the fact if the ReSoA-client has received a close request from the ReSoA-server. In other words, no report is generated if the `close`-function returns before the ReSoA-client has received all data including the `FIN`-segment.

Assertions

The second mechanism supported by the SDT tool is an assertion. The SDT Validator allows a user to define criteria for runtime errors. An assertion is a test which is performed at runtime, for example to verify that the value of a specific variable is within the expected range. As well as user defined rules, assertions are checked during state space exploration. If an assertion is true, then the Validator generates a report. The advantage of assertions opposed to using user defined rules is, that assertions are computed much more efficiently by the Validator than user defined rules.

Observer Processes

The purpose of an observer process is to make it possible to check more complex requirements on the SDL system than can be expressed using MSCs. The basic idea is to use SDL processes, which are called observer processes, to describe the requirements to be tested, and then include these processes in the SDL system.

To be useful the observer process must be able to inspect the SDL system without interfering with it and also to generate reports about the success or failure of whatever they are checking.

If observer processes are used the Validator executes the system in two steps. First, one transition of the SDL system without the observer process is executed, then all observer processes execute a transition and check the new system state. Observers can make use of the assertion mechanisms to generate reports. In order to be able to test the state and variables of other SDL processes the SDT package includes a special abstract data type, namely the access abstract data type.

Selection of Method

User defined rules provide the highest flexibility since they are defined at runtime after the SDL specification is completed. Therefore the test conditions can easily be modified without changing the SDL-system. On the other hand they are slow and complex since only a single user defined rule is allowed. Assertions are only tested at specific task symbols where they are included in the SDL-specification. Thus, we must carefully investigate whether all relevant locations are equipped with the correct assertions. Furthermore, it is difficult to check for a specific condition in multiple processes simultaneously. Explicit signals also have the problem that the SDL-system must be modified at locations that are considered interesting. Observer processes overcome these drawbacks but also slow the system down, since the number of transitions is increased. On the other hand, observer processes allow to observe the execution of the whole system similar as for user defined rules. The advantage is that different invariants can be observed by different observer processes.

We decided to use observer processes, although we had started with user defined rules.

7.4.2. Invariants

This section defines invariants for different scenarios. These invariants are expressed and integrated into the SDL-system using observer processes. Before we test an invariant with the ReSoA system, we will show that they hold for the specification of the socket interface.

We define that the semantics are met if the following rules hold:

- No input except `connect` is accepted until the socket is in the `CONNECTED`-State.
- A `write`-call only returns with a positive status if all data was accepted by the socket.
- There should never be more data in the socket send buffer than the high water mark allows.
- The `read`-function should return as much data as possible, but never more than requested.
- There should never be more outstanding data (unacknowledged data) than the socket receive buffer of the peer entity allows.
- The difference between the number of bytes sent by the sender and the number of bytes read by the other application is never larger than the size of the send buffer at the sender plus the receive buffer at the receiver.
- A `close`-call should not return until all data has been received by the destination host if the `LINGER` option is set
- After the socket is closed no additional messages are accepted.
- Invalid input signals trigger appropriate responses.

In the case of ReSoA the socket send and receive buffers are duplicated at the ReSoA-client and ReSoA-server. This duplication must not lead to a buffer size larger than configured by the user. For example, if data is transferred from the ReSoA-server to the ReSoA-client, it is removed from TCP's receive buffer. Without special precautions this would increase the Advertised Window, allowing the peer side to send more data. However, since the application has not yet read any data, the amount of outstanding data could be higher than allowed by the user. Especially the last rule is of importance, since with ReSoA, TCP acknowledgments are sent before data is passed to the end system.

7.4.3. SDL Specification Design

We decided to separate different layers (application, socket object or ReSoA, and the TCP service) into separate SDL blocks. Each block can consist of an arbitrary number of SDL processes or SDL sub-blocks.

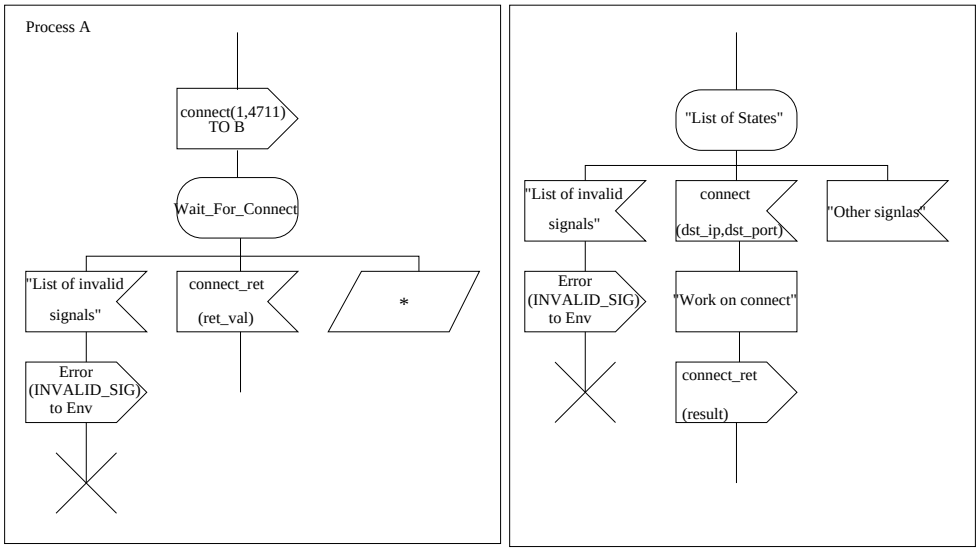


Figure 7.3.: Modelling a function call

This approach was chosen not only because SDL-systems usually have a hierarchical structure, but rather because this permits an easy replacement of a single block by a different one. Especially we intented to replace the socket block by the ReSoA block.

A different approach would have been to specify the entire system using procedures. This would have the advantage that the function call paradigm of the socket interface is inherently included in our model. Unfortunately this would not lead to a clear structure as our approach does. It would also have been more complicated to observe the behavior, because SDL is based on finite state machines and not on procedure calls.

The usage of multiple SDL processes introduces the problem that we must model the function call model of the socket interface. SDL only supports the exchange of signals between two processes. The exchange of signals is asynchronous in contrast to the synchronous behavior of function calls.

Figure 7.3 shows the concept we used to model function calls. The figure shows two processes. The modeled scenario is an application which calls the `connect`-function of the socket interface. The application is modeled by process A, and the socket object is modeled by process B. Every function call corresponds to a signal which has the same name as the function call (here: connect). After process A has sent the signal it changes to the *Wait_For_Connect*-State, since function calls are synchronous. In this state process A expects to receive the *connect_ret* signal, which models the return from a function call. A return from a function call is modeled as a signal with the same name as the function plus the suffix "_ret". Other signals are either not expected and lead to a system error (e.g. signals indicating a return from any other function) or must be handled later. The latter case is possible, for example, if the process still has signals pending in its input queue or if it receives signals from different processes. If an invalid signal is received, an error code is sent to the environment using the *Error*-signal and the analysis is stopped.

The modeling of a function always looks like shown in process B. When a signal indicating a function call is received, the called process performs some tasks followed by sending the corresponding return signal. If the return signal is omitted the system freezes.

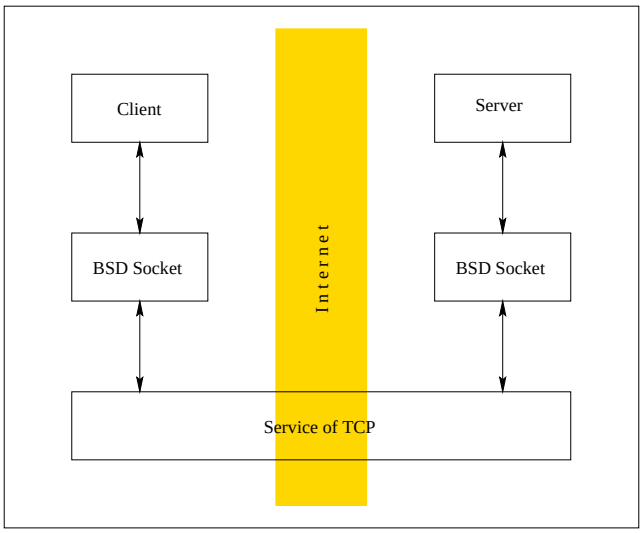


Figure 7.4.: BSD socket system

7.4.4. Test Cases

We decided to test a complete communication scenario. We chose a typical client server application. In a scenario like this the usage of ReSoA must be transparent to the application that uses ReSoA, as well as to the application at the other end that uses the classical BSD socket interface. The reference model is shown in Figure 7.4. Besides looking at the failure free case we also look at system failures like crashes or long outage periods of the wireless link. For all tests we assume the socket interface to be operating in blocking mode and that no UNIX signals like `SIGTERM` are used.

7.4.5. The Reference System

Figure 7.4 shows the SDL-system of the reference system. It is composed of five blocks. Two of these blocks specify the client and the server application, respectively. The next two blocks describe the socket interface. The socket block is identical for both sides of the reference system. The lowest (fifth) block specifies the TCP service. We decided to only specify the TCP service and not the protocol behavior, since a complete specification would enlarge the state space, disabling reasonable usage of the Validator, and would also require a separate validation of the specified protocol behavior.

Communication Scenario

In our communication scenario the client initiates the connection establishment phase. After the connection is established it sends a request and starts to wait for data. After the server has transmitted all data it closes the connection. Finally, after the client has read the last byte it also closes the connection.

Both, client and server use the socket interface to access the service of the underlying protocol stack. Later the socket interface at the client side is replaced by the ReSoA specification without any modification to client or server.

The client operates as follows. It is started after an initial start-up delay. We assume that the socket is already created and that the local port is automatically assigned by the system. The first function call of the

socket interface used by the client is the `connect`-call. The destination IP address and the destination port are specified as parameters. Like all function calls of the socket interface the `connect`-call is modeled as an SDL signal passed from the client process to a socket process. After sending the signal the client is blocked waiting for the return value of the `connect`-call. The return of a function call again is modeled by SDL signals. In the case of a negative return value the client sends a signal to the environment and exits. In the case of a positive return value the client sends a request using the `write`-function and changes the state to `RequestSent`. In this state the client again waits for the return of the function call. If the return value is different from the length of the request, a signal is sent to the environment and the client is terminated. Otherwise the client starts reading data. Between succeeding read calls the client processes data. This is modeled by a timer. The total number of received bytes is calculated. If one of the `read`-calls returns with zero bytes the client closes the connection using the `close`-function. If a read call returns with a negative value the client sends a signal to the environment and exits.

After the `close`-call has returned a signal is sent to the environment indicating either success or failure. Success is indicated by a positive parameter showing the number of received bytes.

The client supports several configuration parameters. It is possible to set the socket send and receive buffer sizes, the size of the request message, and the time interval between succeeding `read`-function calls.

The server sets the `LINGER` option, since this is the most interesting scenario. It then uses the `bind` function to assign a local address to the socket and calls the `listen` function. If these calls succeed, it waits for incoming connections using the `accept`-function. After a new connection is received (accept returns) it reads the request and starts sending the response. Again a timer is used to control the time between succeeding `write`-function calls. After the server has sent its data it closes the connection. Whenever the socket layer returns an error message, a signal is sent to the environment and the server exits. If the `close`-call succeeds the server reports the success to the environment.

The server can be configured with the amount of data it sends as response and the length of the idle time between successive `write`-function calls. It is also possible to set the socket send and receive buffer sizes.

Specification of TCP

As shown in Figure 7.1, the socket interface receives input from the application as well as from the attached protocol, in this case TCP. Since the behavior of a socket object depends on the attached protocol, it is necessary to specify the protocol at least to some extent as well. In our case it was sufficient to specify the service provided by TCP.

Our TCP model implements the state machine as defined in RFC793[142]. Each TCP instance is modeled by a separate TCP process. The two TCP processes use a medium process to communicate with each other. The medium process does nothing else than forward messages that are modeled by SDL signals between the two TCP processes. The medium is reliable (no messages are dropped, corrupted, or re-ordered) since TCP provides a reliable service.

The only configurable option is the receive buffer size, which is used by TCP as its Advertised Window. The send buffer is managed by the socket module. The socket module is responsible for not passing more data to TCP than the send buffer allows. A request to send data (`tcp_write` signal) only informs the TCP instance about the amount of new data to be added to the send buffer. No real data is exchanged. The TCP process transfers all unsent data to the peer TCP process up to the Advertised Window of the peer with one signal. The receiving TCP instance acknowledges the data immediately. If an acknowledgment signal is received TCP informs the socket process that it can free the corresponding amount of space in its send-buffer.

Contrary to the send buffer, the receive buffer is managed by the TCP process. Each time TCP receives new data it updates the receive buffer and signals this to the socket process. Since no real data is exchanged,

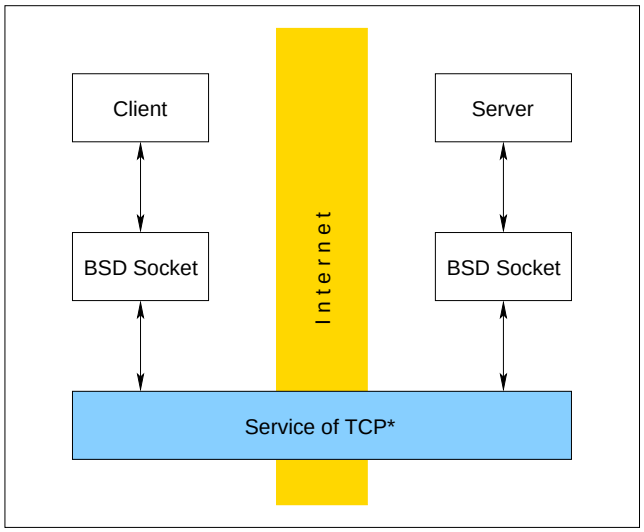


Figure 7.5.: BSD socket specification: Replacement of TCP by TCP*

the socket process is only informed about the amount of data received. The socket process informs TCP that the application has read data and TCP increases the receive buffer. If the receive buffer was zero an explicit window update message is sent.

Upon a connection request either from the application (active open) or from the network (passive open) our TCP model either handles the request or rejects it. Non-determinism is modeled by signals from the environment which either indicate that the whole machine has crashed or that the local application has crashed.

Specification of Socket Interface

The SDL-specification of the socket interface models a blocking socket that has already been created. The specification supports the send and receive low and high water marks, as well as the linger option. The send and receive buffer are managed in a shared fashion with the TCP process as explained in section 7.4.5.

From TCP to TCP*

As pointed out at the beginning of this chapter, it is necessary to change the interface between the socket and TCP in order to support ReSoA. Two required modifications are necessary. The socket layer (ReSoA) has to control the increase of TCP's Advertised Window and to decide when the `FIN` packet should be acknowledged.

The first modification does not require any modification of our SDL specification. The Advertised Window is increased whenever data is removed from the receive buffer, which is the case when the application consumes data. To model this behavior the TCP model includes a signal `update_read_queue`. The same signal is used in the case of ReSoA. The Remote Socket Module (RSM) module passes this signal to the TCP instance when it is informed that the application has read data by the corresponding Local Socket Module (LSM) module.

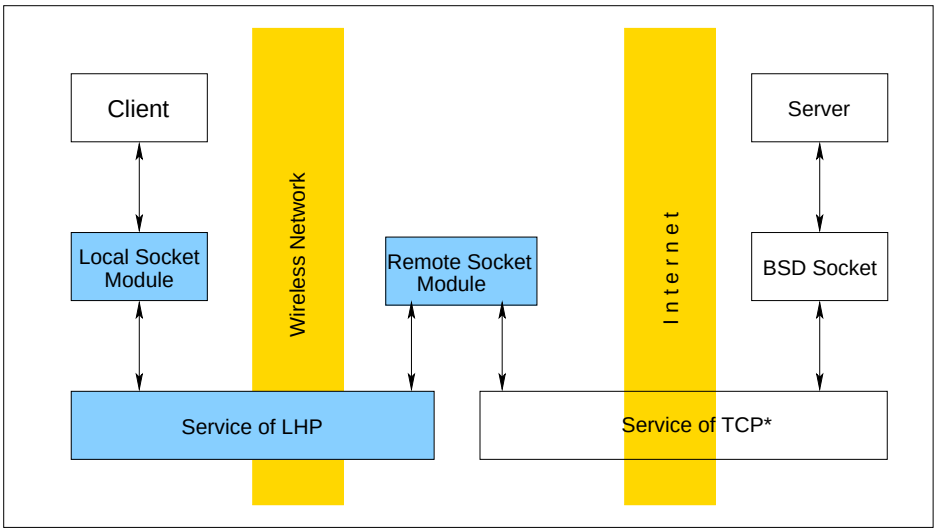


Figure 7.6.: ReSoA SDL system

The second modification requires some changes to the SDL specification. A new signal must be introduced. If TCP at the RSS has received a `FIN`-segment (a signal indicating the termination of a connection) it must not send an acknowledgment for this packet. Later, after the RSM has successfully delivered all packets to the corresponding Remote Socket Client (RSC) , the RSM triggers the transmission of the final acknowledgment. We named the modified TCP block TCP*.

To validate the usage of TCP* it must also be integrated into the reference system. This requires some modifications to the socket process. If the TCP process informs the socket that a connection was closed by the peer, the socket process immediately requests the TCP process to send the `FIN`-segment. The resulting reference system is illustrated in Figure 7.5.

7.4.6. System under Test

The system under test is identical to the system described so far, with the difference that the socket block at the client side is replaced by the ReSoA block.

ReSoA model

ReSoA is modeled by three new processes. Two of these processes model the LSM and RSM respectively. The third process models the service and interface of the LHP. The LSM and RSM processes use the service of the LHP to communicate with each other. Only the LHP service is modeled because ReSoA only specifies the service. The specification of LSM and RSM are similar to the specification of the socket module since both modules basically follow the same extended state machine. The Export Protocol is an internal part of the LSM and RSM processes.

7.4.7. Validation Results

We used the simulator of the SDT-tool to execute the SDL-specification of the reference system (local BSD Socket interface) and the system under test (ReSoA). The simulator not only permits the execution an entire SDL-system, but also to step through a system on different levels (e.g. process, transition). During the simulation of an SDL system it is possible to observe variables, states of different processes, input queues, and so on. It is also possible to set breakpoints to interrupt the execution if a specific state is reached, or upon signal exchange between source and destination process. Basically the SDT-simulator provides the functionality of a programming debugger. It is also possible to send signals from the environment to the SDL-specification. This feature was used to investigate error situations like a ReSoA-server crash. The execution of the system can either be watched by means of a graphical editor or by generating MSC-traces. The latter is especially interesting for a first comparison of the two systems.

Before running the automated tests based on our observer processes controlling the invariants defined in Section 7.4.2, we performed extensive tests using the features of the simulator. We primarily generated block level, as well as process level MSC-traces. The process level traces were used to observe the behavior of our system. The block level MSC-traces were used to manually compare the reference system with the system under test. The discovery that MSC traces cannot be used for automated tests (see Section 7.4.1) is one result of these tests. In fact we started with automated MSC validation. We used the reference system to generate MSCs for different scenarios and then used the SDT-Validator to test our ReSoA specification against these generated MSCs. Since some of these validation tests failed, we thoroughly analyzed these MSCs and finally came to the conclusion that MSC validation is not a supported option. The manual comparison of the MSCs generated for the reference system and the system under test showed that both systems are identical at the block level. A second good example for the potential of these tests is a configuration where the ReSoA system uses TCP instead of TCP*. The major difference between TCP and TCP* is that TCP* delays the acknowledgment for the last packet while TCP does not (see Section 7.4.5). If ReSoA uses TCP, tests with the `LINGER` option fail. In this case the final acknowledgment is sent to the server before the client has received all data. An example block level MSC is shown in Appendix B.

Following the manual tests we ran automated tests using the SDL-Validator and our observer processes. The observer processes send a signal to the environment whenever one of the invariants (see Section 7.4.2) fails. Hence we generated MSCs that show the flow of such an error signal to the environment and tested the reference system as well as the system under test against these MSCs. A test is successful if the MSC tests fails, since this means that the error signal was not generated by the observer process. For all performed tests our invariants held.

Although the ReSoA specification passed all tests the validity is limited. First of all, both the reference system as well as the system under test were specified by ourselves. Second, it was impossible to apply the algorithm described in Section 7.4.1 due to the large state space of our system. Unfortunately, automatic MSC validation is not possible. Therefore we could only investigate some interesting cases like the usage of the `LINGER`-option by means of manual MSC comparison. The observer process could only be used to observe the high level behavior of our system. However, the combination of thorough discussion and design of the socket calls in ReSoA (as discussed in Section 6.2), the tests with our ReSoA implementation, and the tests performed with our formal ReSoA specification show that ReSoA can be used to replace a local socket implementation without violating the socket interface semantics.

Part III.

Performance Evaluation

Chapter 8.

Performance Evaluation Preconsiderations

In this part of the thesis we compare the performance of ReSoA against TCP. See Section 8.1 for the discussion why TCP was chosen as reference. The goal of the performance evaluation is twofold. First we need to verify our claim that ReSoA is able to generally provide a better performance than TCP. Second we want to determine some rules of thumb under which conditions the usage of ReSoA is in fact beneficial. The performance evaluation is organized in three chapters. In this chapter the performance evaluation environment is defined. We start with the discussion about an appropriate reference. Then we define the metric used to compare both approaches and introduce the scenario under study. Next we identify the parameters of interest as well as interesting values. Finally we discuss whether simulation or measurements should be used for the performance evaluation. The next two chapters present the results of the performance evaluation. In Chapter 9 we present measurement results and in Chapter 10 we discuss performance results obtained by simulation.

8.1. Selection of a Reference

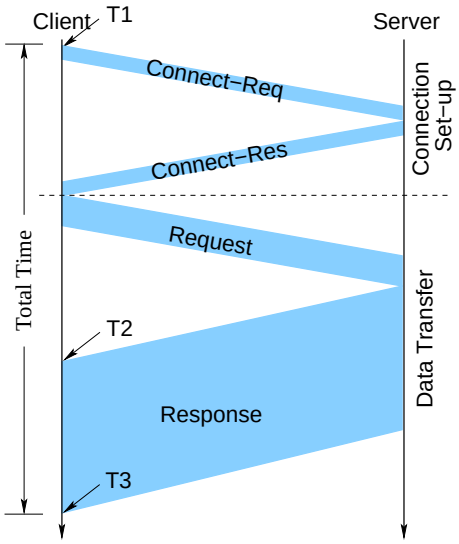
Performance figures of ReSoA like normalized throughput (achieved throughput over bit-rate) alone are not of much interest. The question is rather how ReSoA's performance compares to other approaches. Possible candidates for such a comparison were discussed in Section 3.3 on page 25. A comparison of ReSoA against all of these approaches is beyond the scope of this thesis. Instead a distinguished approach was selected.

We decided to use TCP as reference. Since TCP is unable to deal with the peculiarities of wireless networks we used a reliable link layer protocol to hide the error characteristics of the radio link from TCP. The reasoning behind choosing TCP as reference is threefold. First, different research results advocate the use of TCP in combination with a persistent reliable link layer protocol over error prone links (e.g. [116, 60, 61]). Second, RFC 3135, which deals with PEPs, recommends that a PEP should only be used if it provides a significant improvement over an end-to-end approach. Thus, to justify the deployment of ReSoA we need a corresponding performance comparison. Third, since other solutions are normally compared to TCP, we can indirectly compare ReSoA to other approaches.

8.2. Performance Metric

We are interested in the performance perceived by the user for typical communication scenarios found in the Internet today. This means that we are rather interested in the download time of a block of data (e.g. WWW page) than in the steady state throughput.

The predominant application of the Internet is WWW[156, 104]. As many other applications, WWW belongs to the class of client-server applications. A client opens a TCP connection to a server and then sends a request. The server replies with the requested data. The request normally has a smaller number of



$$\text{Throughput} = \frac{\text{RequestSize} + \text{ResponseSize}}{\text{TotalTime}} \tag{8.1}$$

Figure 8.1.: Communication scenario and definition of the metric used for the performance evaluation

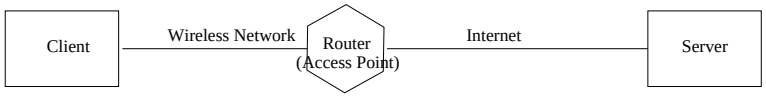
bytes than the response. Thus, the major flow of data is the one from server to client. This is also true for other applications like file transfer (ftp) or remote login.

Figure 8.2 shows a typical client-server communication scenario including the evaluation points. Many other performance evaluations neglect connection establishment and the request transmission, considering only the bulk data flow from server to client. Our scenario considers the transmission of the request from client to server, as well as the time required for connection establishment. We decided to include these factors, since we expect the overhead to play a significant role, especially for short response sizes. In addition to that, if we did not consider an entire session one could argue that ReSoA is more efficient for the data download but has a higher overhead during the connection establishment due to the exported interface. The performance metric is the throughput seen by the client (application), resulting in identical evaluation points for ReSoA and the end-to-end case (TCP). The throughput is defined as the total number of application level bytes transfered, divided by the total time as described in equation (8.1). The time required to close the connection is not included in the throughput definition (and therefore also not shown in the figure), since it is not visible to the client application. The client only has to know when the response is complete and this is usually indicated by the last data segment (`FIN`-bit is set).

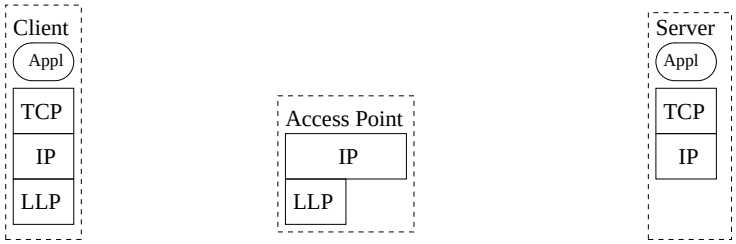
Besides assessing the application level throughput we also collected the entire network traffic at the wireless end system, both access point interfaces, and the fixed host (server). This makes it possible to investigate the resource utilization of TCP and ReSoA on the wireless link, and to investigate how the TCP sender on the fixed host behaves (e.g., any retransmissions).

8.3. Investigated Scenario

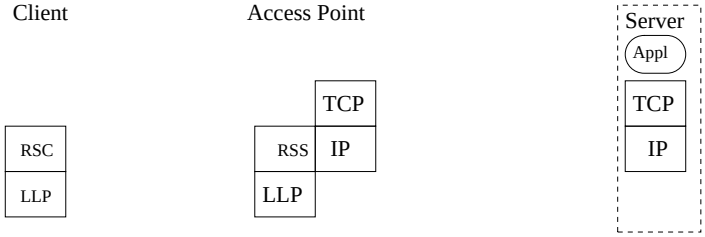
We decided to use a single base scenario for our performance evaluation and to only vary parameters of this scenario. The scenario under study is a client which downloads data from a server. The client is located at a wireless host, while the server is located anywhere in the Internet. The communication between client and server is degraded by bit errors on the wireless link and background traffic on its path through the Internet. An abstraction of the basic performance evaluation environment is shown in Figure 8.2(a). The



(a) Basic Simulation Setup



(b) Protocol Stacks of the reference case (end-to-end TCP)



(c) Protocol Stacks for ReSoA

Figure 8.2.: Basic performance evaluation set-up and protocol stacks of the two systems under study.

illustrated set-up used for all experiments consists of two end systems and a single access router. The client is connected to a wireless network (left side of Figure 8.2(a)) and the server is connected to the Internet. The access router connects the wireless network and the Internet.

Figure 8.2(b) shows the protocol stacks of the three different components for the reference configuration. Figure 8.2(c) depicts the protocol stacks for the ReSoA configuration. As can be seen from the figures, the server protocol stack is identical for both approaches. Further on both approaches have in common that they use a reliable link layer protocol for the communication between client and access router (if not stated otherwise). In the reference scenario the client also has a TCP/IP stack and the access router only supports the IP protocol. For ReSoA the TCP/IP stack is located at the access router and the ReSoA protocols are used for the communication between client and access router.

8.3.1. Application

In Section 8.2 we decided to use a client-server application for our performance investigations. However, instead of using an existing application (like `ftp` or `netperf`) we decided to implement our own simple client-server-application where the client sends a request to the server and the server replies with the amount of data requested. The size of the request and the size of the response are parameters of the client application. The advantages of our test application over existing applications are that we can set the exact size of the request and the response and that we can calculate the application level throughput as defined by equation (8.1). For the latter the client records the time when it initiated the connection establishment (`connect`-function of the socket interface) and the time when the response is complete (when it has read the amount of data requested).

8.4. Measurement vs. Simulation

We decided to use both measurements as well as simulations for the performance evaluation. Measurements are used to show the achievable throughput in a real environment (see Section 9) and to validate our simulation model. However for the systematic performance evaluation we decided to use simulation for several (classical) reasons:

- Measurements make it difficult to investigate a larger parameter space.
- In the case of measurements not all parameters of an experiment can be controlled (e.g.: error rate).
- Measurements are difficult to reproduce, as the results are influenced by non-observable side-effects[1].
- In the case of measurements the obtained performance results not only depend on the concepts but rather on the implementation expertise and time. The TCP/IP implementation in most current operating systems is quite sophisticated. They are tuned for the operating system and hardware architecture. A new protocol implementation, such as ReSoA, will incur a performance penalty due to a sub-optimal implementation.

8.5. Experiment Design

The next step before we can start with the performance comparison is to specify the different experiments in more detail. This means that various parameters of our configuration under study need to be identified and we have to determine which of these parameters should have a constant setting over all experiments and which ones should be varied. The latter are referred to as factors (see [94], chap. 2.2, page 24). Next the settings for the parameters and levels of the factors need to be determined. Finally we have to decide which configurations should be investigated by simulation and which ones by measurements. We do not describe how the performance evaluation is conducted as the method is different for simulation and measurements.

8.5.1. Parameters

Figure 8.2(a) provides an overview of the different components of our set-up. Each of these components can be configured using different parameters.

The client-server application has two parameters: request size and response size.

[1] In fact, we repeated our measurements with different computers at the same positions and observed different results. However, the results differ in the absolute numbers, but the difference between TCP and ReSoA was comparable to the results presented in Chapter 9.

TCP has many facets which influence the performance of a system. First, we have to determine whether TCP-Reno, TCP-NewReno, or TCP-SACK should be used. Even more TCP versions exist as discussed in Section 3.3. Those are not considered here since they are not widely deployed. Second, different options like the timestamp option can be enabled. Third, different parameters like the maximum segment size, the initial Congestion Window, send/receive buffer size, or timer granularity can be configured.

The ReSoA modules only allow to set the size of the send and receive buffer of ReSoA-client and ReSoA-server. The usage of send and receive buffers in ReSoA is discussed in Section 6.2.4. Similar to TCP's buffers, the send buffer determines the maximum number of unacknowledged bytes and the receive buffer determines the maximum number of bytes the ReSoA-client can receive before the application has to consume data.

The IP layer allows to configure the size of the outgoing queue. This determines the maximum number of packets that can be stored at a router waiting for service before packets are dropped.

The reliable link layer protocol allows to set the size of the ARQ window and the maximum number of retransmissions. The networks (wireless access network and Internet) have the following parameters: bitrate, propagation delay, MTU-size, error model, and background traffic.

8.5.2. Selection of Factors

Unfortunately, our performance evaluation scenario has numerous parameters and possible settings. Therefore we have to answer three questions:

1. What are the parameters and what are the factors?
2. Which values should be used for the parameters?
3. Which levels should be used for the factors?

In order to answer the first question we used a 2^k factorial test and an analytical approach. Both are discussed in the following sections. The values of the parameters and levels of the factors are determined in Section 8.5.3.

2^k Factorial Tests

A standard approach to determine important parameters for experiments with a large number of factors is a 2^k factorial test (see [107, 94] for a detailed description). In this test two different levels are assigned to each factor. Then a simulation (experiment) is conducted for each possible combination. Altogether 2^k runs are required[2]. For each run the response is recorded. Then the effect of the different factors and combination of factors is measured by the proportion of the total variation in the response that is explained by this factor. The total variation or *Sum of Squares Total (SST)* is defined as follows:

$$\mathrm{SST} = \sum_{i=1}^{n} (y_i - \bar{y})^2 \tag{8.2}$$

We applied the 2^k factorial test to our scenario using the parameter settings as shown in Table 8.1. The 2^k factorial test was performed via simulation. The question we try to answer is which of the parameters have a high impact on the performance difference between ReSoA and the end-to-end case. Hence, the input for the 2^k factorial test is not the result of a single simulation but the difference of two simulations. For each possible configuration we performed a simulation for the reference case and for the ReSoA case. Then the difference of the achieved throughput was calculated.

[2] In the case of stochastic input a $2^k r$ test should be used instead.

Parameter	x1	x2	Impact	Parameter	x1	x2	Impact
ErrorModel	None			Background T.	None		
$\mathrm{QueueSize}_{ij}$	60			ConnectMode	0		
RequestSize	100 Bytes			PacketSize	100	1000	3.54%
WindowSize	10	60	2.67%	AccessLinkSpeed	9.6kb	11Mb	6.09%
Internet Speed	2Mb	100Mb	0.73%	FileSize	10^4	10^6	0.87%
AccessLinkDelay	1ms	200ms	0.21%	Internet Delay	10ms	200ms	3.34%

Table 8.1.: Parameter and result of 2^k factorial test

As can be seen from Table 8.1, a configuration without packet drops in the wireless network and without background traffic was investigated. The result of the 2^k factorial test is shown in the column labeled 'Impact' of Table 8.1. Table 8.1 omits factor combinations. These have an impact in the range around 1%. However, due to the high number of factors, it is difficult to determine the key factors. We repeated the 2^k factorial test with different factors and levels but could not find a trend. All performed 2^k factorial tests showed that all factors have a small impact on the performance. They also indicate that there is a high dependency between the factors. This high dependency probably is the reason why we could not determine particularly important factors. In the next section we use an analytical approach in order to gain a deeper understanding about the role of the different factors.

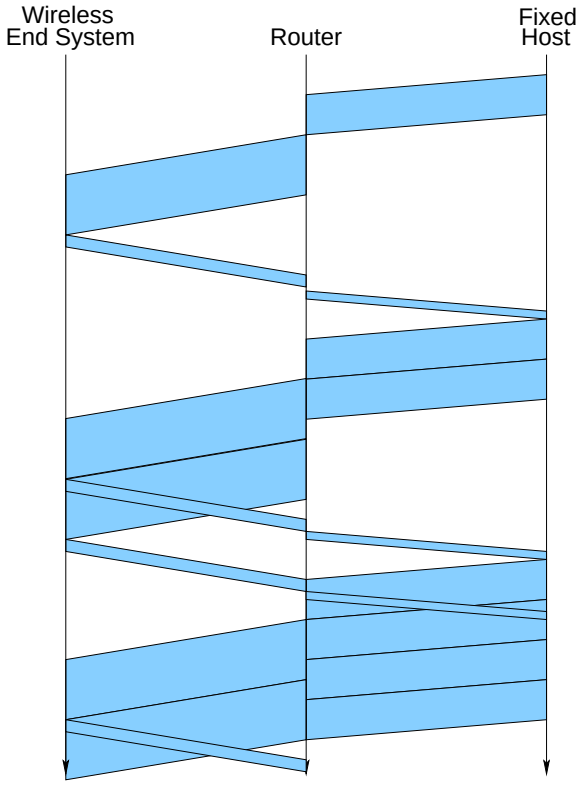
Analytical Approach

The performance of TCP is relatively well understood. Padhye et al. have developed Formula (8.3) [130] to describe TCP's steady state throughput.

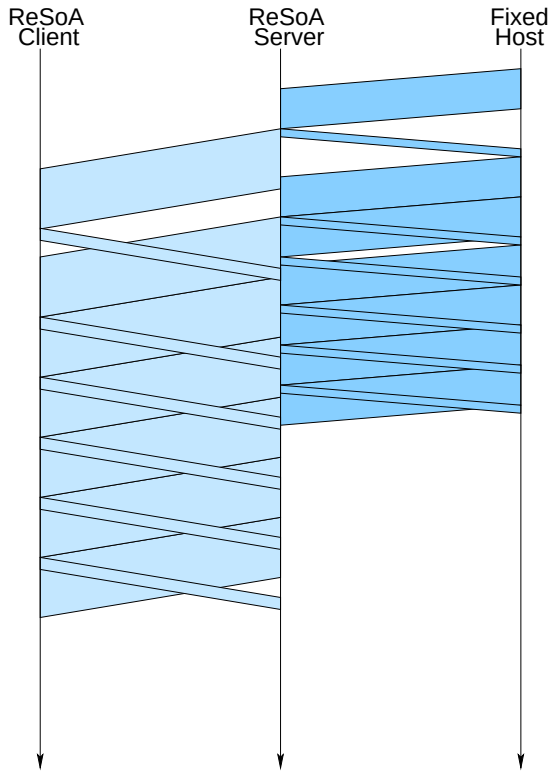
$$B(p) \approx \min\left(\frac{W_{max}}{\mathrm{RTT}}, \frac{1}{\mathrm{RTT}\sqrt{\frac{2bp}{3}} + T_o \min(1, 3\sqrt{\frac{3bp}{8}})p(1+32p^2)}\right) \tag{8.3}$$

This formula shows that the TCP throughput depends on TCP's Advertised Window, RTT, and error rate. The latter in turn depends on the propagation delay, queuing delay as well as the bitrate. Although, this formula indicates which parameters influence TCP's performance, it cannot directly be applied to our scenario. In Section 8.2 we decided to investigate the performance for different file sizes. If small files are transferred the steady state often is not reached. Therefore in the following we investigate the Slow Start phase in more detail and develop equations for this case.

In order to find equations for the throughput during the Slow Start phase for both the ReSoA case and the end-to-end case we start with an example communication scenario as illustrated in Figure 8.3(a) and Figure 8.3(b). These two figures show a typical time chart for the packet exchange between the involved instances for TCP and ReSoA, respectively. In this example ReSoA performs better than TCP as can be seen from the figures. At the end of the time chart the wireless host has received seven packets with ReSoA and only five packets with TCP. The reason why ReSoA performs better than TCP can be derived from these figures. In the end-to-end case (Figure 8.3(a)) TCP is unable to fully utilize the bottleneck link due to its Slow Start mechanism. If ReSoA is used the TCP sender sees a shorter RTT and hence opens its window faster. In the case of ReSoA the bottleneck link is completely utilized after the second packet was received by the access point.



(a) TCP packet time chart



(b) ReSoA packet time chart

Figure 8.3.: Example communication scenario for ReSoA and TCP

For the analysis we assume that there is no background traffic, no queue overflow, and no packet drops occur due to bit errors. We also omitted the LLP for the sake of simplicity and we only consider the data transfer from the fixed host to the wireless host.

We assume that a file of size $M = N * L$ bytes is transferred, where M is the file size in bytes, N is the number of packets and L is the payload of each TCP segment. W is the Advertised Window, τ_1 is the propagation delay of the link between the fixed host and the router, B_1 is its bitrate. The corresponding parameters of the second link are τ_2 and B_2, respectively. The time TCP needs to transfer the file is calculated as shown in Equation (8.4).

$$T_{\mathrm{TCP}}(N) = \begin{cases} \lfloor \log_2 N \rfloor \mathrm{RTT} + (N - 2^{\lfloor \log_2 N \rfloor} + 1)G_2 + G_1 + \tau_2 + \tau_1 & \text{if } B_2 < B_1 \wedge N \leq 2^b, \\ T_{\mathrm{TCP}}(2^b) + (N - 2^b)G_2 & \text{if } B_2 < B_1 \wedge N > 2^b, \\ \lfloor \log_2 N \rfloor \mathrm{RTT} + (N - 2^{\lfloor \log_2 N \rfloor} + 1)G_1 + G_2 + \tau_1 + \tau_2 & \text{if } B_2 \geq B_1 \wedge N \leq 2^b, \\ T_{\mathrm{TCP}}(2^b) + (N - 2^b)G_1 & \text{if } B_2 \geq B_1 \wedge N > 2^b, \end{cases}$$

where

$$b := \lceil \log_2 W_{\text{eff}} \rceil,$$

$$W_{eff} = \min(W, \left\lceil \frac{\mathrm{RTT}}{G_i} \right\rceil),$$

$$\mathrm{RTT} := G_1 + G_2 + A_1 + A_2 + 2(\tau_1 + \tau_2),$$

$$G_i = \frac{\mathrm{PacketLength} * 8}{B_i}, A_i = \frac{\mathrm{ACKLen} * 8}{B_i}.$$

(8.4)

Without bit errors or packet drops due to queue overflow the achievable throughput should be close to the bottleneck bit rate. However, due to TCP's Slow Start algorithm it takes some time until TCP can fully utilize the bottleneck link. This time, in turn, depends on the bandwidth-delay product of the network.

If ReSoA is used the time to transmit the file is computed as shown in Equation (8.5). The main difference compared to equation (8.4) is that now Slow Start only sees the first link, while ReSoA is used over the second link. Thus, the RTT seen by the TCP sender is smaller. The performance over the second link depends on the LHP. In the calculation we consider the optimal case where a packet can be transmitted to the ReSoA-client as soon as it arrives at the ReSoA-server.

$$T_{\mathrm{ReSoA}}(N) = \begin{cases} \lfloor \log_2 N \rfloor \mathrm{RTT}_1 + (N - 2^{\lfloor log_2 N \rfloor} + 1)G_2 + G_1 + \tau_2 + \tau_1 & \text{if } B_2 < B_1 \wedge N \leq 2^c, \\ T_{\mathrm{ReSoA}}(2^c) + (N - 2^c)G_2 & \text{if } B_2 < B_1 \wedge N > 2^c, \\ \lfloor \log_2 N \rfloor \mathrm{RTT}_1 + (N - 2^{\lfloor \log_2 N \rfloor} + 1)G_1 + G_2 + \tau_1 + \tau_2 & \text{if } B_2 \geq B_1 \wedge N \leq 2^b, \\ T_{\mathrm{ReSoA}}(2^b) + (N - 2^b)G_1 & \text{if } B_2 \geq B_1 \wedge N > 2^b, \end{cases}$$

where

$$\mathrm{RTT}_1 := G_1 + A_1 + 2\tau_1,$$

$$W_{\text{eff}} := \min(W, \left\lceil \frac{\mathrm{RTT}_1}{G_1} \right\rceil),$$

$$b := \lceil log_2 W_{\text{eff}} \rceil,$$

$$c := \lceil log_2 \frac{\mathrm{RTT}_1}{G_2} \rceil.$$

(8.5)

Our analysis makes a clear statement about when the usage of ReSoA is beneficial. It especially shows that ReSoA can be beneficial even if no packet loss occurs. ReSoA should (at least) be used for large access network delays and small response sizes. On the other hand this statement is based on deterministic assumptions and neglects all effects of real networks. This raises the question whether these findings still hold if packets are dropped due to transmission errors or congestion, or for long outage periods.

In order to take the effects of real networks into account we decided to use measurements in order to investigate ReSoA's performance under real conditions, and simulations to study the effects of isolated network characteristics.

8.5.3. Determination of Experiments

In order to take the effects of real networks into account we decided to perform measurements using IEEE 802.11b as access network.

According to our analysis, in this set-up ReSoA does not promise a performance gain. Wireless LANs have a small propagation delay and hence the achieved RTT difference between TCP and ReSoA is small. However, the assumptions we have made for our analysis are not valid here. First, the error rate is not zero. Although an almost error-free link can be accomplished by a large number of MAC level retransmissions, the residual error rate should not be neglected. There are bad reception positions where an error free link cannot be guaranteed. Additionally, for end-to-end TCP the number of MAC-level retransmission often must be set to a small number because not all applications benefit from a reliable link layer protocol (e.g. delay sensitive application). Third, 802.11 only provides a half-duplex channel, whereas our analysis assumes a full duplex channel.

In order to limit the number of uncontrollable side-effects, we decided to investigate a set-up where a wireless end system communicates with a server within the same local area network. The server is attached to the access point using a cross-connect cable. The detailed measurement set-up is described in Section 9.

8.5.4. Determination of Simulations

The goal of the simulations is to understand the effects of isolated network properties or combinations of specific properties on the performance of ReSoA. Therefore we decided to investigate the following four cases via simulation:

1. Packet loss in the access network (due to transmission errors).
2. Internet background traffic, leading to variable delay and packet drops.
3. Combination of the previous two effects.
4. Outage period on the last (wireless) hop.

We want to show that the results of the analysis are valid even for specific network conditions. We especially assume that the performance gain due to ReSoA will increase if these network characteristics are taken into account. Therefore we chose bitrate and latency of the access network and the Internet as factors. Additionally the network property under study (e.g. error-rate) was chosen as factor. In Table 8.2 the parameter defaults used for all simulations are summarized. The factor levels depend on the four investigated cases and are discussed below.

Simulation 1: Packet Loss in the Access Network (due to Transmission Errors)

Table 8.3 summarizes the factors and levels of our first simulation. The bitrates for the wireless network were chosen to represent GSM, UMTS and wireless LAN speeds. The Internet speed (the bitrate of the

Parameter	Setting
Request Size	100 Bytes
Maximum Segment Size (MSS)	1448 Bytes
TCP's Advertised Window	Function of Bandwidth-Delay-Product
TCP timestamp option	Enabled
Other TCP options	Disabled
Internet bottleneck queue	60 packets
All other queues	Infinite
LLP retries	255
LLP ARQ window	32 packets

Table 8.2.: Parameters used for all simulations

Factor	Setting	
Response Size	1 KByte - 1 MByte	
	Wireless Network	**Internet**
Bitrate	$9600\frac{\text{bits}}{\text{s}}$	$2\frac{\text{Mbits}}{\text{s}}$
	$384\frac{\text{kbits}}{\text{s}}$,$11\frac{\text{Mbits}}{\text{s}}$	0.5, 1, 2 times wireless network speed
Latency	1ms - 250ms	10ms-100ms
ErrorRate	0% - 60%	0%
Background Traffic	none	0% -100%

Table 8.3.: Factors and their levels used for simulation scenarios one, two, and three

slowest link of the path between server and access point) is chosen as a function of the wireless network speed. Three different cases can be distinguished. The Internet bitrate is either lower, identical, or higher than the bitrate of the access network. However, not all possible combinations of wireless network and Internet bitrate are worth considering. If the bitrate of the wireless link is 9600 bit/s (GSM case) we only consider the case where the Internet offers a higher bitrate than the wireless network.

For the propagation delay of both networks we decided to use arbitrary values (not derived from a specific technology) in order to show the dependence of performance gain on the RTT.

We used two different error models to simulate packet drops on the wireless link (see Section 10.2.4). The mean error rate is varied from 1% to 60%. The results of these simulations are discussed in Section 10.3.1.

Simulation 2: Internet Background Traffic, leading to Delay Variation and Packet Drops

The second simulation uses the same factor levels as the first simulation. The only differences are that the packet drop rate of the wireless link is set to zero and that background traffic is introduced. The load caused by background traffic is set to values from 10% to 100% of the Internet capacity(see Section 10.2.5). The results of these simulations are discussed in Section 10.3.2.

Factor	Setting	
Response Size	5000, 50000, 500000 Bytes	
	Wireless Network	**Internet**
Bitrate	384 $\frac{\text{kbits}}{\text{s}}$	2 Mbits/s
Latency	50ms - 200ms	10ms - 100ms
Error Rate	none	
Background Traffic	none	
Outage-Period length	0.01s,. . . , 5s	none
Outage-Period start	Start, middle and end of response	none

Table 8.4.: Factors and their levels used for simulation scenario four

Simulation 3: Combination of Packet Loss in the Access Network and Internet Background Traffic

Since the third simulation is a combination of the previous two simulations, we again used the factor levels shown in Table 8.3. The results of these simulations are discussed in Section 10.3.3.

Simulation 4: Outage Period on the last (wireless) Hop

Table 8.4 summarizes the factors and levels used for simulation number four. Here we investigated the effects of (long) outage periods on the performance of ReSoA and TCP. In order to reduce the number of simulations we decided to only investigate a single wireless network bitrate and to reduce the number of response sizes. Since we expect outage periods to be more likely in cellular systems than in wireless LANs, we selected the UMTS-case where the wireless network offers a bitrate of 384 kbit/s. Packet error rate and background traffic are set to zero. The length of an outage period is varied from 0.01s up to 5s. The effects of an outage period as a function of the start point of an outage were also investigated. The start of an outage is set to coincide with the beginning of the response, in the middle of the response, or near the end of the response. The results of these simulations are discussed in Section 10.3.4.

Chapter 9.

Performance Measurements

This chapter presents a comparison of TCP (our reference system) and ReSoA in a wireless LAN IEEE 802.11b environment using measurements. We investigated ReSoA under real conditions and show that the deployment of ReSoA can also be beneficial for access networks with a low latency. We start with the definition of the measurement set-up. Then we describe our measurement methods and finally present the measurement results.

9.1. Set-up

In Section 8.3 we defined that we use the same basic set-up for all performance investigations. This basic set-up is illustrated in Figure 8.2(a). Figure 9.1 describes a real world scenario corresponding to Figure 8.2(a).

According to our basic set-up, the measurement setup consists of three nodes as shown in Figure 9.1: a wireless end system, an access point operating either as router or ReSoA-server, and a fixed host. The access point and the fixed host are connected by a 100 Mbit/s Ethernet using a crossconnect cable. Wireless host and access point communicate over an 802.11b WLAN. The wireless host and the access point are equipped with PCMCIA ZoomAir cards (Model 4100, Rev. 7C5). The wireless LAN is configured to use automatic rate adaption and the number of MAC level retransmissions was set to 255[1] if not stated otherwise. All three computers use Linux 2.4.18 as operating system. The access point uses the HostAP[117] software to implement a software 802.11b access point.

We chose this simple set-up for the measurements because simulation is used to investigate the effects of various parameters on the performance of ReSoA. Especially, we did not use any emulators to investigate specific delay, error, or background traffic patterns. The Internet box of our basic set-up is realized by a single cross connect cable. This means that we neglect effects of background traffic (e.g. no congestion) in this experiment and investigate the performance assuming a rather small RTT.

The application used for the measurements corresponds to the application described in Section 8.3.1. We used a fixed request size of 100 bytes and varied the response size for the experiments.

9.2. Methods

We performed measurements for different response sizes ranging from approximately one kilobyte up to 100 kilobytes. The sequence shown in Equation (9.1) defines the investigated response sizes. We investigated the performance for 70 different response sizes. As step size we chose 1448 bytes, since 1448 is the maximum transfer unit and we only wanted to send full size packets.

$$(r_n)_{n\in\{1,\ldots,70\}} = 1448n. \tag{9.1}$$

[1] Maximum number currently supported.

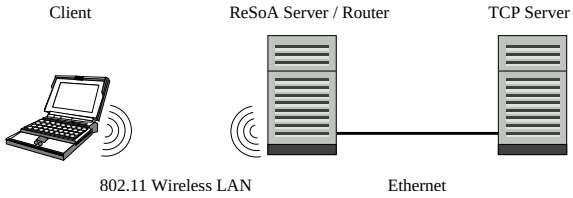


Figure 9.1.: Setup used for the measurements

Measurements in the wireless context are complicated due to the time varying nature of the wireless channel. Measurements in an 802.11b environment are even more complicated due to the fact that the license-free ISM-band is used. Although we can control the channel allocation of our own wireless network, and hence are able to use a dedicated channel for our measurements, we cannot control applications using the same frequency in the vicinity. During all measurements we ensured that neither the wireless LAN channel used for the measurements nor the adjacent channels were used by other wireless LAN users.

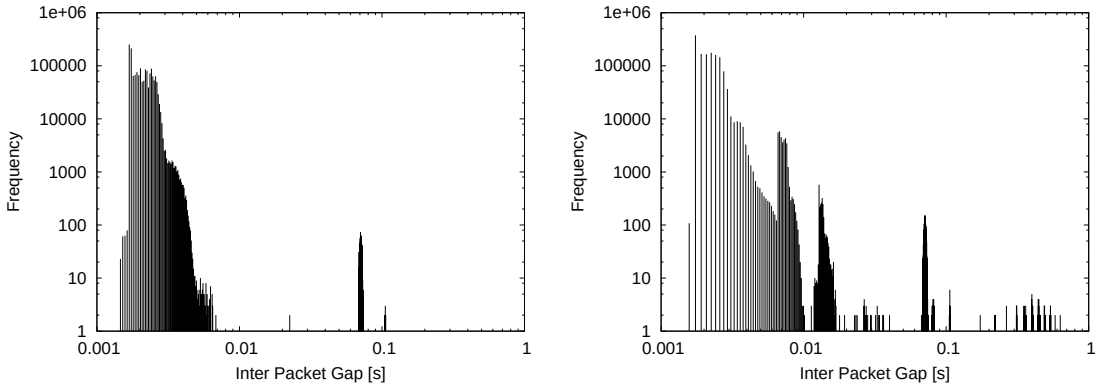
In order to achieve a fair comparison between ReSoA and the reference system they have to be investigated under comparable conditions. Since network conditions may vary with time and are not fully controllable, we spread the repetitions for a single measurement point in time. Additionally we organized the sequence of measurements in such a way that a measurement of the reference system is always paired with a ReSoA measurement using identical parameter settings. The resulting sequence of measurements is defined by the series shown in Equation (9.2). In this equation $T_{(a_n)}$ means a TCP measurement using response size (a_n), $R_{(a_n)}$ means a ReSoA measurement using response size (a_n), and (E_i) is a random gap between 1 and 10 seconds between two measurements with different response sizes. The random gap E_i was introduced in order to prevent cyclic effects. The idea of using random gaps is taken from RFC2330[135] which deals with Internet measurements. However, RFC2330 advocates the use of exponential distributed gaps in order to avoid predictability of the next measurements.

$$M = (T_{r_1}, R_{r_1}, E_1, \ldots, T_{r_{70}}, R_{r_{70}}, E_{70}). \tag{9.2}$$

After having defined our measurement series we decided where to place the wireless end system. The position of the wireless end system is an important parameter of our experiment as it influences the channel quality. However, in order to limit the number of measurements we decided to only use two different positions instead of performing a time consuming systematic measurement. The two positions were chosen in such a way that one position, named *Good-Position*, provided a high channel quality and the other position, named *Bad-Position*, provided a low channel quality resulting in a high error rate or low bit-rate, respectively. In [152] we used a similar classification of positions.

In order to find appropriate positions we observed the inter-packet gap between adjacent packets of a constant packet rate UDP flow. In the case of a good channel quality (no packets are lost) the inter-packet gap between successive packets should be nearly equidistant (some variation is introduced by the stochastic MAC). On the other hand, if the channel quality is poor the inter-packet gap will vary due to packet losses and automatic adjustment of the transmission rate[2]. Hence the variation of the inter-packet gap seems to be an appropriate metric to measure the quality of a position. We did not use the packet loss rate observed at the receiver as metric for to two reasons. First, this metric would neglect the adaption of the bitrate by the MAC which is the first indication of a bad channel state. Second, this would require the

[2]We did not use a fixed bitrate because bitrate adjustment is one of the features used to handle poor channel state. Therefore we decided to keep it enabled for our measurements.



(a) Good-Position

(b) Bad-Position

Figure 9.2.: Rating of Good- and Bad-Position using the inter-packet gap of a UDP flow

implementation of a traffic generator that guarantees that no packets are lost in the sending end system due to buffer overflow.

In order to find appropriate positions we ran a number of test measurements using UDP. UDP was chosen because it allows sending a constant packet stream without retransmitting packets or waiting for timeouts. The `netperf`-tool[96] was used as application. `netperf` was configured to transmit UDP packets with a payload of 1448 bytes continuously. The traffic source was located at the access point while the data sink was located at the wireless host. The network traffic at the wireless end system was collected with the tcpdump-tool[3]. Figure 9.2 shows the distribution of the inter-packet gap for our two positions. The Good-Position is close to the access point. The Bad-Position is at the edge of the coverage area of our access point.

The measurement series M was repeated at least 50 times at each position. For each T_{ri} and R_{ri} we calculated the mean throughput, the median, and the width of the 99% confidence interval. The median was included in the evaluation since we observed strong outliers in some experiments which (as we assume) were caused by lost interrupts at our router. In addition, we counted the total number of bytes sent over the wireless link for each measurement point.

The entire measurement set was repeated several times on different days (weeks) in order to confirm that we did not measure a sporadic situation.

9.3. Performance Results at 'Good-Position'

Figure 9.3 shows the measurement results obtained at the Good-Position. The figure shows three curves. Two for the ReSoA case and one for the reference case. The configurations for the two ReSoA curves differ in the packet size used for the wireless network. For the upper curve the maximum packet size allowed by IEEE 802.11b was used, while the lower ReSoA curve was obtained with the packet size set to 1500 bytes. TCP cannot utilize the full packet size allowed by IEEE 802.11 because this would require fragmentation

[3]We did not measure the inter-packet gap at the application layer because in this case it would include scheduling delay.

and the fixed host being able to discover the larger MTU size. Fragmentation would be required since the server (fixed host) is connected to the access point via Ethernet, which has a maximum frame size of 1500 bytes. Since fragmentation is usually disabled in the Internet, we only used 1500 byte packets for TCP. The configuration used for the TCP measurements and the lower ReSoA curve are identical. The same amount of payload per packet is used for both approaches, although ReSoA would be able to encapsulate more data bytes into a 1500 byte frame than TCP (due to smaller header sizes). We decided to use the same payload size for both approaches because we wanted to investigate the benefits from using ReSoA instead of TCP. We did not want to investigate the positive effects of a few additional bytes per packet.

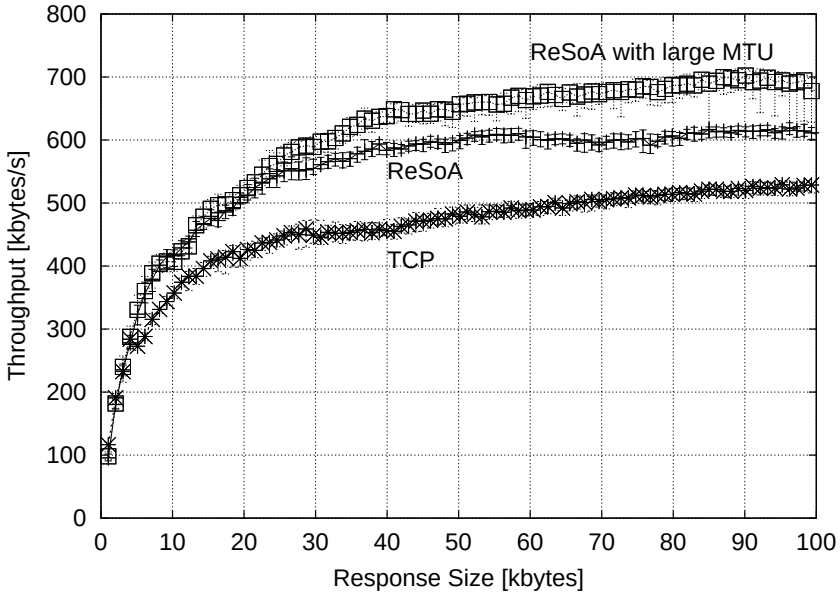


Figure 9.3.: ReSoA versus TCP at the Good-Position. The figure shows the median throughput achieved for different response sizes. The errorbars show the range between the lower and upper quartiles.

As can be seen in Figure 9.3, ReSoA clearly outperforms TCP even if ReSoA uses the same packet size as TCP. The performance of TCP and ReSoA is identical only in the case of very small response sizes[4]. As soon as the response size becomes larger than 5 KBytes, ReSoA offers the higher throughput. For instance, in the case of a response size of 50 Kbytes the throughput achieved with ReSoA (small MTU) is about 25% higher than the throughput achieved with TCP. A comparable performance improvement can be observed for all response sizes larger than 30 Kbytes. Furthermore, not only the median throughput achieved with ReSoA is higher than the median throughput of TCP, but also the quartiles show a genuine performance improvement due to ReSoA. Even when we compare the first quartile (25%-quantile) of ReSoA with the third quartile (75%-quantile) of TCP, ReSoA shows a better performance. The 99% confidence intervals for all response sizes are narrow for both ReSoA and end-to-end TCP. For every measurement point the width of the confidence interval is below 30 KBytes/s. The confidence intervals for TCP and ReSoA do

[4]We believe that this is due to implementation issues. We expect that a more optimized ReSoA implementation would outperform TCP even for small response sizes. However, we cannot prove this yet.

not overlap for any response size larger than 5 KByte, showing that ReSoA is statistically significantly better than TCP.

The usage of the full IEEE 802.11 frame size is a good example for how ReSoA is able to exploit technology-specific features. In this case the achieved performance is additionally increased. With our 50 KByte response size example the larger MTU size increases the ReSoA throughput by 9%. Compared to TCP the throughput is increased by nearly 37%. A similar performance improvement can be found for all response sizes larger than 40 KBytes. However, two points attract attention. First, in the case of response sizes below 30 KBytes larger MTUs do not improve the performance of ReSoA. Second, the first quartile for different response sizes is nearly identical to the performance achieved with small MTUs. We believe that the first aspect is due to our ReSoA implementation while the second aspect could partially be caused by the wireless channel, since retransmissions of larger frames are more expensive. Our ReSoA implementation currently suffers from the fact that it tries to send full sized packets. A small packet is only sent when the connection is terminated. Since we are only interested in showing the potential of larger MTUs, we did not try to further optimize our ReSoA implementation in this aspect. However, a possible approach would be to send small packets if the network driver signals that it is idle.

In order to understand the performance results we analyzed the packet flow between the Internet-server (fixed host) and the access point, as well as the packet flow between the access point and the wireless host. The first was taken into consideration since we need to know whether the TCP sender has retransmitted any segment. The latter was analyzed to compare the overhead introduced by ReSoA and the reference case. For the sake of simplicity, the following discussion concentrates on the same MTU size for TCP and ReSoA. For the analysis of TCP's packet flow we used the `tcptrace`-tool by Shawn Ostermann http://www.tcptrace.org/ while we implemented our own scripts to analyze the ReSoA packet flow.

The analysis of the trace files did not show a single TCP retransmission (as it could be expected at the Good-Position). Thus, TCP retransmissions are not the reason for the performance improvement by ReSoA. ReSoA's performance gain is the result of a better utilization of the wireless channel as can be seen from Figure 9.4. It shows the mean overhead in percent over the response size. For all response sizes ReSoA transfers less bytes over the wireless network than TCP. For response sizes larger than 10 KBytes ReSoA is able to reduce the network traffic by around 6%.

There are two reasons for this. First, ReSoA has a smaller protocol overhead than TCP. Second, ReSoA is able to exploit the knowledge that it is operating on top of a half-duplex network with random access providing a nearly reliable service. Based on this knowledge, the LHP is optimized to send acknowledgments seldom. This not only saves wireless resources because less data is transmitted, but might even reduce the collision rate, since competing traffic in the upstream direction is reduced. Although it would be possible to reduce the header size of TCP using header compression, the second optimization (reduced number of acknowledgments) is not possible for TCP, as it needs the acknowledgments for its ACK Clock.

9.4. Performance Results at 'Bad-Position'

Figure 9.5 shows the performance results obtained at the Bad-Position. Again ReSoA outperforms the reference (end-to-end TCP) in the case of response sizes larger than 10 KBytes. For response sizes smaller than 10 KBytes the confidence intervals are overlapping. Hence, statistically both approaches show an identical throughout. If we compare the upper quartile, ReSoA provides the better performance in most cases even for response sizes smaller than 10 KBytes.

If we compare the measurements results of ReSoA at the Good- and the Bad-Position we see that ReSoA nearly achieves identical throughput over response size at both positions. There is no response size where the throughput at the Bad-Position is more than 60 KBytes lower than at the Good-Position. This is different For TCP. TCP only achieves about 60% of the Good-Position throughput at the Bad-Position. Thus, the performance gain by ReSoA is higher at the Bad-Position. For instance for a 50 KByte response

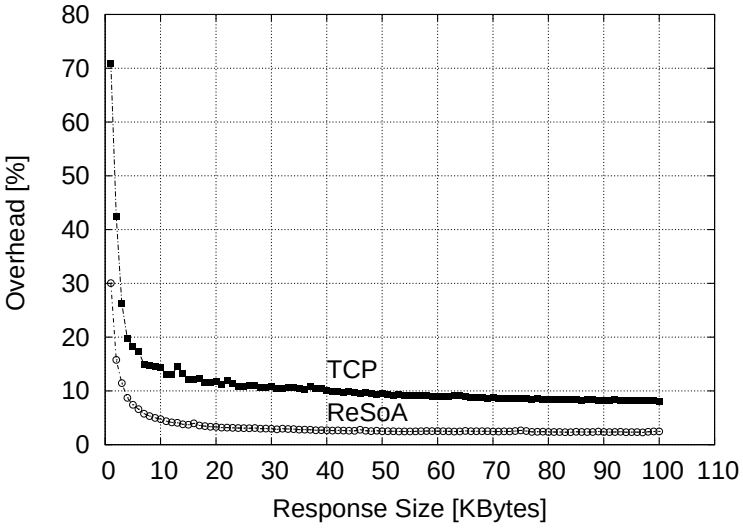


Figure 9.4.: ReSoA vs. TCP at the Good-Position. The figure shows the mean overhead of ReSoA and TCP as a function of the response size.

size ReSoA achieves nearly twice the throughput of TCP.

We also performed measurements with larger MTUs at the Bad-Position. However, the results were rather disappointing. Although some measurement points exist where the performance was improved, at other measurement points the performance of ReSoA was reduced. We observed a much more varying throughput for large MTUs at the Bad-Position. Since large MTUs improve the throughput at a good position, but might reduce the throughput at a bad position, they should not be used as default configuration. Unfortunately the wireless end system does not know a priori whether it is located at a good or at bad position and it is also possible that a position changes from good to bad and vice versa. One possible remedy would be an LHP that is able to adapt its current MTU size to the channel state. However, we did not further investigate this issue.

To interpret the results (for the presented case) we analyzed the network traffic at the three nodes of our measurement set-up as we did for the Good-Position. The first assumption is that TCP retransmissions (albeit spurious) are the cause for the bad performance of TCP. Hence we analyzed the network traffic trace file of the wireless end system and of the fixed host in order to count the number of retransmissions as a function of the response size.

The analysis showed that only a small fraction of the file transfers suffered from TCP retransmissions. At the Bad-Positions we performed 160 measurements for each response size. There was no response size where we observed more than ten experiments with retransmissions. In most cases only two or three file transfers suffered from retransmissions. Additionally whenever retransmissions occurred only one or two packets were retransmitted. Therefore TCP retransmissions are not the reason for TCP's bad performance. The 802.11 MAC level mechanisms (retransmission and rate adaption) are able to hide the bad channel from the TCP sender. Neither packets are lost nor does the TCP retransmission timer expire spuriously.

Next we considered the mean RTT as a function of the response size at the Good- and the Bad-Position. Figure 9.6 shows that the mean RTT develops differently at the Good- and at the Bad-Position. At the Good-Position the RTT increases with increasing response size. This is caused by queuing at the access

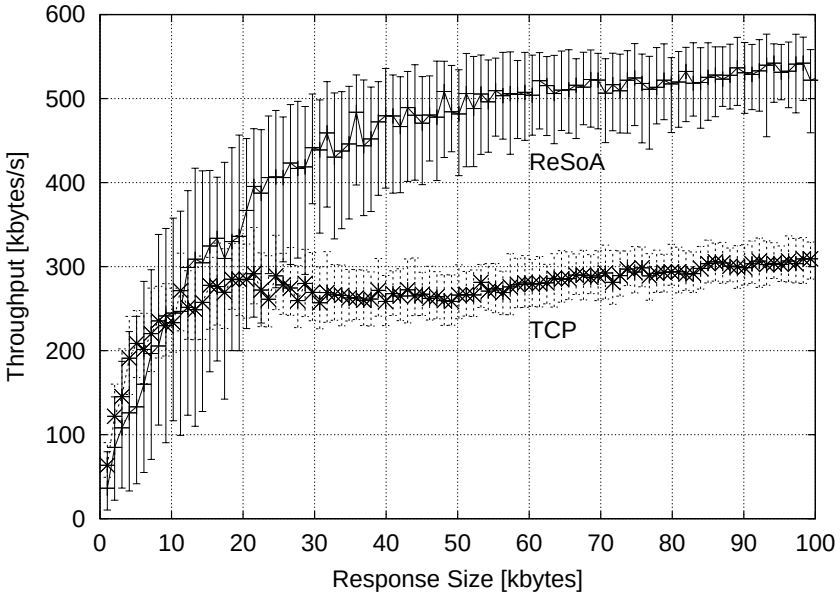


Figure 9.5.: Measurement results for ReSoA versus end-to-end TCP at the Bad-Position. The figure shows the median throughput achieved for different response sizes. The errorbars show the range between the lower and upper quartiles.

point, since the TCP sender has not reached steady state yet. This effect cannot be observed at the Bad-Position. At the Bad-Position the mean RTT increases fast for small response sizes and stays constant for response sizes larger than 30 KBytes. Also the mean RTT at the Bad-Position is larger than the mean RTT at the Good-Position. The higher RTT at the Bad-Position indicates that the performance is reduced either by MAC-level retransmissions or by automatic adaption (reduction) of the bit-rate of 802.11.

We assume that only a few packets are buffered in the outgoing queue of the access point waiting for transmission to the wireless host. If packets had queued-up at the access point the RTT would increase with increasing response size as it is the case for the Good-Position. One interpretation of this observation is that the server is not allowed to transmit packets at full speed due to a slow incoming Acknowledgment Clock (e.g. acknowledgments are delayed on the wireless link due to MAC-level retransmissions and a low bitrate). Apart from this it is also possible that the wireless channel is idle during good channel state. If the channel state is bad, the acknowledgments of the TCP instance at the wireless end-system are queued by the local MAC for transmission. If the channel state changes from bad to good it takes some time until the acknowledgments are received by the TCP instance at the fixed host. If no packets are queued at the access point the wireless link will be idle until the first acknowledgment triggers a transmission of a data segment at the server. For ReSoA this is different, since ReSoA decouples the communication on the backbone (Ethernet) from the communication over the access network (wireless LAN) by the size of the receive buffer. In order to obtain a deeper understanding and to investigate this assumption we analyzed the trace files on the connection level.

For the analysis we investigated many trace files in detail. Since it is not possible to discuss the analysis of multiple trace files, we only present the analysis of a single connection as example. However, all analyzed connections lead to the same conclusion. In order to verify that the TCP instance at the server is not

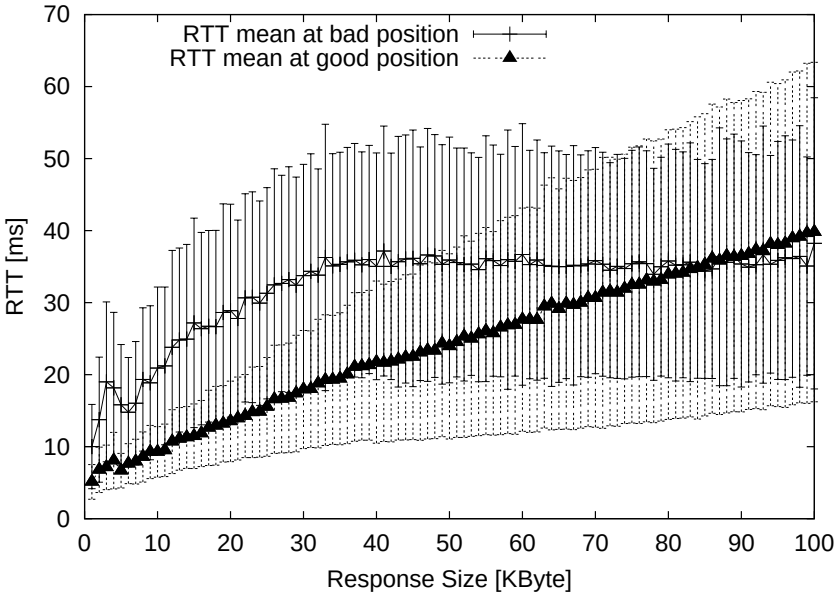


Figure 9.6.: Mean RTT and standard deviation at Good- and Bad-Position for end-to-end TCP.

allowed to send at full speed we looked at the number of outstanding[5] bytes. This provides us with an approximation of the Congestion Window, the development of the RTT, and the packet arrival and departure time at the three nodes of our measurement set-up.

Figure 9.7 shows the number of outstanding bytes for our example connection and for a connection with the same response size at the Good-Position. The latter is included in the figure to show how the number of outstanding bytes should develop. In the case of the Good-Position the number of outstanding bytes increases exponentially, since TCP opens the Congestion Window exponentially during Slow Start. In the case of the Bad-Position the number of outstanding bytes does not increase between 0.02 and 0.07 seconds, indicating communication problems on the wireless channel.

Figure 9.8 shows the RTT samples for these two connections. This figure again shows that the RTT at the Bad-Position is larger than the RTT at the Good-Position.

In order to substantiate the observation that TCP is unable to fully utilize the wireless channel during good channel states we show time sequence plots of the data (response) transfer phase in Figure 9.9. Figure 9.9(a) shows data packets leaving the fixed host and acknowledgments reaching the fixed host. Figure 9.9(b) shows the data flow at the two interfaces of the access point. It shows the point of time at which data packets arrive at the access point and when they are forwarded to the wireless client, as well as the flow of acknowledgments in the reverse direction. Finally, Figure 9.9(c) shows the points of time of arriving data packets and transmitted acknowledgments at the wireless end system. Since the clocks at the three machines are not synchronized we do not show data from different hosts in the same graph. For all figures we used the timestamp of the initial SYN-segment as reference point.

Figure 9.9(b), which shows the packet flow at the access point, is best suited to explain the reason for TCP's poor performance. It shows when TCP data segments are sent form the access point to the

[5]Outstanding: Number of bytes sent but not acknowledged yet. Assuming that the sender always has data to send, the number of outstanding bytes is equal to the Congestion Window.

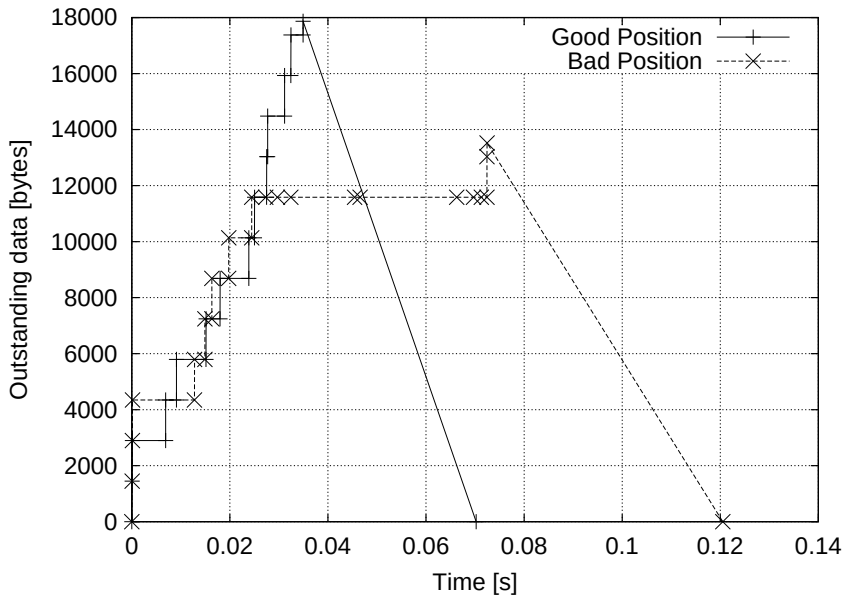


Figure 9.7.: Outstanding data at Good- and Bad-Position for end-to-end TCP

wireless end system and when the corresponding acknowledgments are received by the access point. This figure shows that the acknowledgments are often delayed, which in turns leads to idle periods of the data stream. An example for the delaying of acknowledgments can already be found at the beginning of the connection. The access point tries to deliver three packets to the wireless end-system. Figure 9.9(c) shows that these three packets are received in a single burst, but Figure 9.9(b) shows that the corresponding acknowledgments are not received in a burst. The third acknowledgment is delayed longer than the other two acknowledgments.

9.5. Summary

The measurements show that wireless Internet access using IEEE 802.11b can benefit from the deployment of ReSoA. The performance gain depends on the channel quality (position of wireless end system). At good positions ReSoA improves the performance by about 25%. This improvement could be further improved by using the full packet size allowed by IEEE 802.11. For bad positions ReSoA is able to nearly double the throughput for larger response sizes. Larger MTU's did not lead to a performance improvement at the bad position. Hence, an adaptive link layer protocol would be an interesting candidate for further investigations.

The main reasons for the performance improvement are that ReSoA can exploit the knowledge that it is operating on top of a half-duplex channel, and on the other hand the decoupling of the two hops of our network. Since ReSoA has information about the service of the underlying network it can reduce the number of acknowledgments in the upstream direction, leaving more capacity for the transmission of data packets in the downstream direction. The decoupling is especially important at the bad position. Here the decoupling guarantees that the access point has data to send when the channel changes from a bad state to a good state.

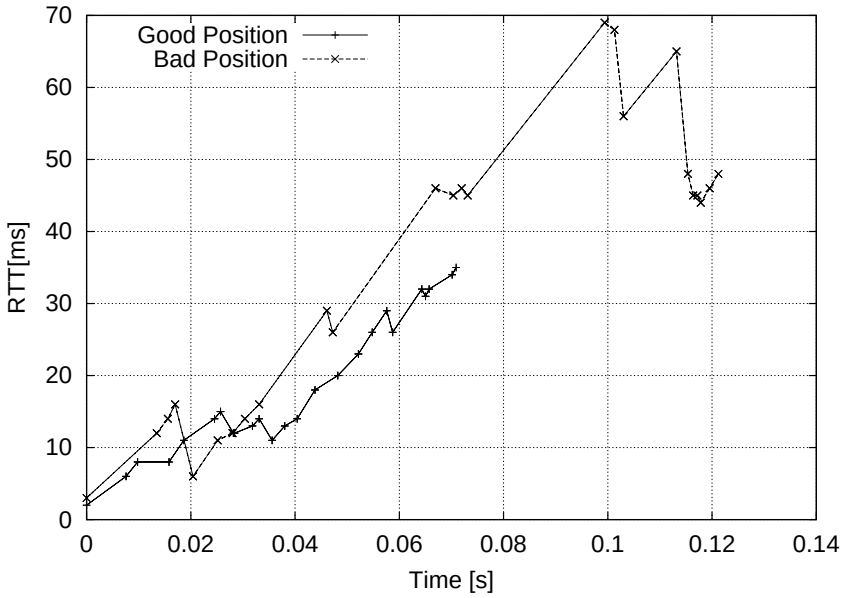
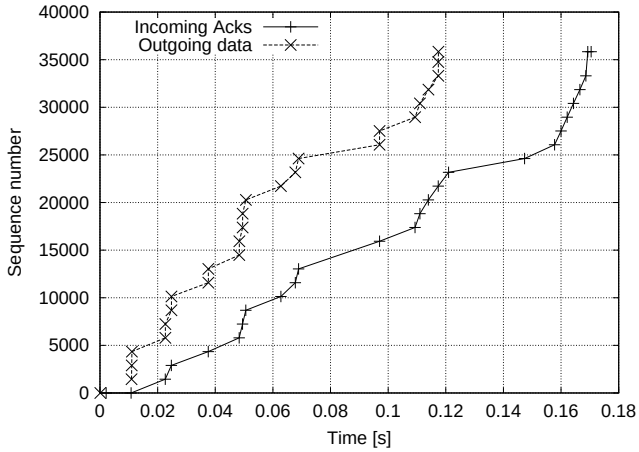
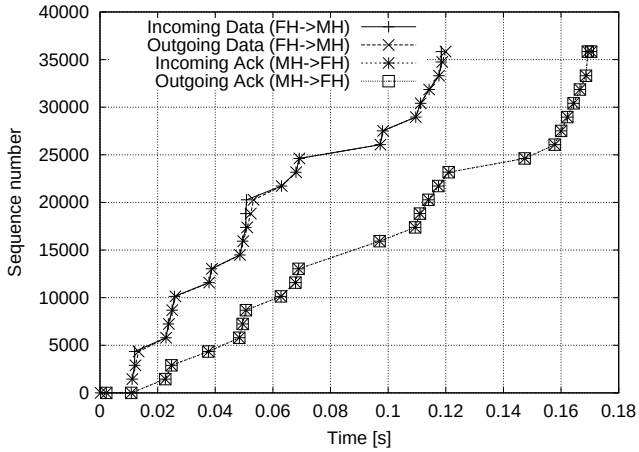


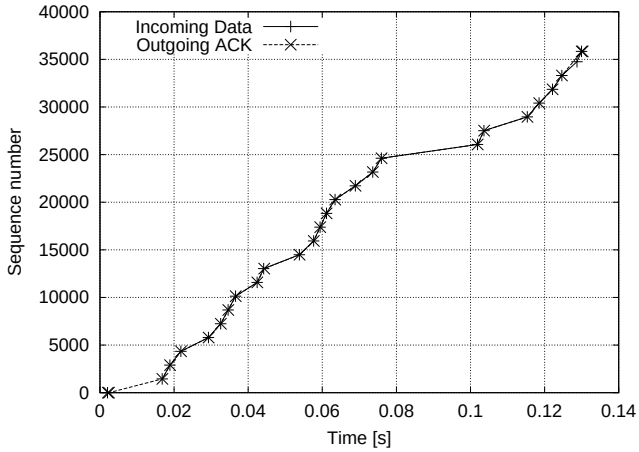
Figure 9.8.: RTT over time at Good- and Bad-Position for end-to-end TCP



(a) Fixed host



(b) Access point



(c) Wireless host

Figure 9.9.: Data and acknowledgment flow from fixed host to wireless host.

Chapter 10.

Performance Simulation

In contrast to the previous chapter, which deals with the performance of ReSoA in a wireless LAN context using measurements, this chapter investigates the performance of ReSoA for a larger parameter space as described in Section 8.5.

We selected the Network Simulator (NS) [124, 39, 66] as simulation tool. This decision is based on the widespread usage of NS for Internet protocol performance evaluation, especially TCP. Many TCP related performance evaluations are based on NS (e.g. [15, 19, 18, 65]).

Although NS already supports many different protocols it was necessary to extend the package by our own protocols. We used standard modules whenever possible, but had to add support for ReSoA and a reliable link layer protocol. These extensions can be downloaded from http://www-tkn.ee.tu-berlin.de.

10.1. Methods

In simulations all parameters are controllable. Hence, the methods used to perform the simulations are different from the methods used to run the measurements. The simulation methods are described in the following sections.

10.1.1. Running Simulations

Pawlikowski et al. found that many simulation based studies in the field of communication protocols neglect the stochastic nature of simulations[133]. In the same paper the authors give advice on how a thorough simulation study should be performed.

Our simulation experiment belongs to the class of terminating simulations. The simulated time only depends on the file size we actually investigate (we terminate the simulation if the file could not be delivered within a certain interval). Thus, it is very unlikely that a single run reaches a steady state. Since we are interested in the throughput as a function of the file size, we do not need to remove the transient phase. The simulation output of a single run is the throughput seen by the client as discussed in Section 8.2. In order to obtain meaningful results we repeated every simulation with a different seed for the random number generator. The number of repetitions depends on the variance of the simulation response as well as the number of unique and good seeds provided by NS. For each factor allocation we calculated the 95% confidence level and tried to limit the confidence interval to $\pm 2\%$ of the mean by repeating each experiment as necessary. The minimum number of repetitions was set to 30 and the maximum number of repetitions was set to 63, since NS provides 63 predefined seeds. Each of these seeds guarantees 33 million non-overlapping random numbers which should be sufficient for our short file transfers. Unfortunately these seeds are not available via the NS interface language. Therefore we had to add the required support by ourselves. The random number generator used was the standard random number generator of the NS simulator.

The start of the file transfer of the observed connection does not coincide with the simulation start. We delayed the start of the file transfer in order to avoid sending our data into an empty network. The initial state of the error model was randomly chosen using the ratio of the sojourn times of the two states.

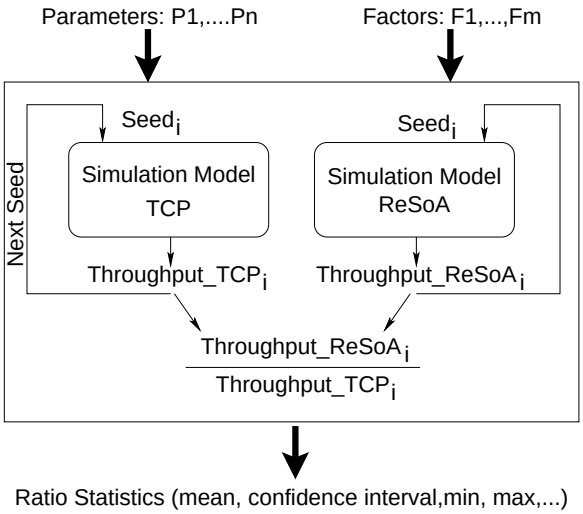


Figure 10.1.: Paired comparison of two approaches

Simulation Output

The result of each simulation is the achieved throughput as defined in Section 8.2. Since we are interested in the performance gain due to ReSoA, and our simulations have a one-to-one correspondence, we used the method of paired observations as illustrated in Figure 10.1(see [94], Section 13.4.1). Instead of showing only the throughput achieved with either TCP or ReSoA we also show the ratio of ReSoA to TCP as well as the 95% confidence interval. The confidence intervals are included in all plots as error bars. ReSoA is statistically significant better than TCP if the lower bound of the confidence interval is larger than one. If the upper bound is smaller than one, TCP is better. If the confidence interval includes one, the systems are not significantly different.

10.2. Simulation Model Design

In Chapter 8 we discuss the scenario under investigation including the type of application and the evaluation points. In this section we describe how the different components are modeled and which additional parameters are introduced by the modeling process.

10.2.1. TCP/IP Simulation Model

TCP is one of the central components of our investigations and must therefore be modeled in detail. The modeling of TCP is difficult because of the wide range of parameters, environments, and implementations available. In [6] hints about how to select a TCP model are given. In the following we discuss requirements on our TCP model.

A TCP connection can be divided into the three phases: connection establishment, data transfer, and connection termination (see Chapter 2.2). As discussed in Section 8.2, our model has to include connection establishment and the data transfer phase. Connection termination is not required since it has no effect on

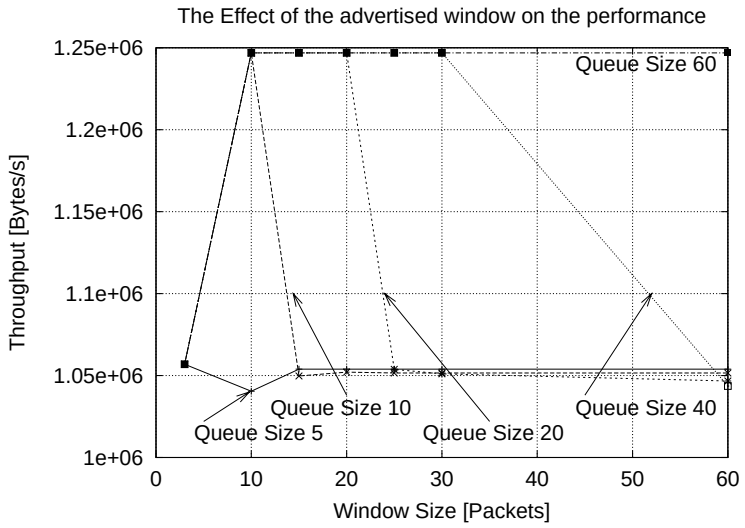


Figure 10.2.: Effects of TCP's Advertised Window on throughput

the throughput as defined in Equation (8.1)[1].

The data transfer model must include all the mechanisms of a standard TCP implementation as described in Section 2.2. It has to support bidirectional traffic because our source model (see section 8.2) generates traffic in both directions.

Different research work has shown that TCP performance depends on many parameters like the initial window, Advertised Window, MSS and so forth. In our simulations the throughput is network limited as long as the network capacity does not exceed the size of TCP's window field (we do not use the window scale option). Contrary to sender limited, network limited means that TCP's Advertised Window is large enough to fill the pipe[2]. Furthermore we assume full size Ethernet frames resulting in an MSS of 1448 bytes. This is a typical packet size for burst transfers. The initial window of TCP is set to one MSS. The different parameters are summarized in Table 10.1 on the next page.

The IP protocol is simply used to route packets through the network. The routes are static. Therefore there is no routing overhead. The IP protocol does not support any additional mechanisms like fragmentation.

10.2.2. ReSoA Simulation Model

Like TCP, ReSoA is a core component and is therefore modeled in detail. Unfortunately NS does not support the socket interface, but uses a much simpler interface between the application modules and the protocol modules. Thus, it is not possible to realize a one to one mapping of the ReSoA specification to the simulation model in NS. Therefore a certain degree of abstraction is needed. This is not a problem since

[1] For instance, when a user sees the requested web page he is satisfied and does not care how long it takes TCP to close the connection.

[2] We could assume that the Advertised Window should always be set to its maximum value to obtain the highest performance results. However, if the Advertised Window is larger than the network capacity, packets will be queued and dropped which can reduce performance. The congestion control mechanisms of TCP need some time to reach steady state. This is illustrated in Figure 10.2. Hints for automatic buffer tuning are given in [153].

Parameter	Setting	Comment
Advertised Window	$\mathrm{RTT} * \mathrm{Bitrate}$	We assume that throughput is network limited.
MSS	1448 bytes	Burst transfers usually use large packets
Timestamp option		
Initial window	1 packet	Slow Start is started with a single segment.
TCP ticks	100ms	Granularity of RTT estimation and calculation of retransmission timeout value

Table 10.1.: TCP model parameters

the purpose of the simulation is not to verify the semantics of ReSoA but to investigate its performance. The goal of the modeling is to include the behavior of the Export Protocol (EP) which is essential for the performance evaluation.

The NS interface between application and protocol neither has an explicit create-connection-end-point function (like the socket call) nor a function to open a connection either actively or passively. Most applications do not use consecutive, separate write calls to pass data to a protocol. Instead they tell the protocol how much data to transfer with a single function call. It is the protocols task to divide the data into packets according to its maximum packet size. From ReSoA's point of view this means that no socket send-buffer exists. Normally ReSoA uses the socket buffers for flow control purposes. Messages containing application data are only delivered if the socket buffer is not full. Therefore our ReSoA implementation for NS needs to support the socket buffer concept.

In NS a new connection is opened when an application passes data to the corresponding TCP agent for the first time. The passive side must be configured during simulation set-up.

The mapping of the NS interface to ReSoA is discussed in the following two sections, dealing with active connection establishment and data transfer. We do not discuss the passive connection establishment as it is not used in our simulations[3]. To integrate ReSoA into NS two new agents named `ReSoAClient` and `ReSoAServer` and the EP were added. The ReSoA model does not include the registration process. We assume that the client is already registered with a ReSoA-server.

Active Connection Establishment

The active connection establishment functionality is part of the `ReSoAClient`. When the application passes the data to the ReSoA-client for the first time, the `ReSoAClient` initiates the connection establishment process. The implementation supports three different modes.

In default mode the data is buffered[4] and the client sends a connection establishment request to the ReSoA-server. When the ReSoA-server receives this request it triggers the local TCP instance to create the TCP connection. After the TCP connection is established the ReSoA-server responds to the request and the ReSoA-client starts to pass the application data to the ReSoA-server.

The second mode assumes that the connection establishment will be successful after the request has reached the ReSoA-server. Thus, the ReSoA-server responds to the connection set-up request immediately without waiting for the TCP connection establishment to complete. After receiving this response the ReSoA-client starts sending data to the ReSoA-server. The data is buffered at the ReSoA-server until the

[3] Of course, our implementation supports passive connection establishment as well.

[4] The simulation model does not transport real data. Only a counter is incremented.

TCP connection has been successfully established.

The third mode is even more optimistic. In this mode the ReSoA-client immediately starts transmitting data to the ReSoA-server without waiting for any response messages.

The EP, which is used to exchange messages between the ReSoA-client and ReSoA-server, is implemented in detail using exactly the same messages and behavior as described in Section 6.1.3.

Data Transmission

As mentioned above the client side application requests the transmission of large data chunks with a single function call (it is even possible to request infinite transfer). It would be an unrealistic approach if the ReSoA-client would pass such a function call to the ReSoA-server using a single EP-SDU. This would lead to an underestimation of the overhead introduced by ReSoA. To implement the behavior of ReSoA the ReSoA-client has a parameter which determines the size of a write request per EP-PDU. Thus, the application request is split into multiple EP packets, each (virtually) carrying the maximum or less data bytes until all data is delivered. In our simulation we set the payload size of an EP packet to the payload size of a TCP segment.

The ReSoA-client supports a send buffer and guarantees that the ReSoA-server never has more data outstanding (for the peer TCP to acknowledge) than its send buffer allows. Data is removed from the send buffer whenever the ReSoA-server receives a TCP acknowledgment. Control messages are used to inform the ReSoA-client about the buffer update. The size of the buffer is a parameter of the simulation model.

Data Reception

When the application at the remote host (located anywhere in the Internet) sends data to our ReSoA-client, the data is received and processed by the TCP entity at the ReSoA-server. This especially means that the ReSoA-server sends an acknowledgment for the received data before the data is delivered to the ReSoA-client. Corresponding to the ReSoA specification the data is passed to the ReSoA-client without waiting for an explicit request. Again, we used a maximum packet size to avoid huge packets (to simulate the overhead due to ReSoA). Furthermore a receive buffer is introduced to implement the flow control behavior of ReSoA. Upon reception of data the ReSoA-client immediately passes the data to the application (as every TCP agent does) and sends a receive buffer update message to the ReSoA-server [5].

Although the received data is immediately acknowledged by TCP and forwarded to the ReSoA-client, TCP's Advertised Window must not be increased before the application at the ReSoA-client has consumed the data. Thus, the end-to-end flow control between the two applications via the send and receive socket buffers is preserved. Unfortunately, the NS TCP models do not support an Advertised Window mechanism. Every TCP agent has a fixed window. The window is decreased each time a segment is sent, and the window is increased when the corresponding acknowledgment is received. Such an approach is not feasible for ReSoA since it assumes that the received data is consumed at once after reception. This assumption is not true for ReSoA since the data still has to be delivered to the wireless host.

To implement a suitable model for the Advertised Window it was necessary to derive a new class from the FullTCP class. For the sake of simplicity this class explicitly sends packets for window updates instead of piggybacking the updates on TCP acknowledgments. A window update packet is sent whenever the ReSoA-server receives an update receive buffer message from the ReSoA-client. Each update packet has

[5] Sending a receive buffer update for every data notification introduces unnecessary overhead. Normally these updates should be sent following a more sophisticated strategy. In the case of our simulation environment the large number of update messages is not a concern since we have a full duplex link and there is no data flow in the reverse direction.

Parameter	Setting	Comment
Send Buffer	60000 bytes	Size of the socket send buffer
Receive Buffer		Same size as TCP's Advertised Window
Packet Size	1448 bytes	Maximum number of payload bytes
Connect mode	0	Default mode: Data packets are not transmitted before TCP connection is complete.

Table 10.2.: Parameters of the ReSoA model.

40 bytes (TCP header + IP header)[6]. Except for the usage of a real Advertised Window the behavior of the new class is identical to the base class.

Table 10.2 list the parameter of the ReSoA model.

10.2.3. LLP Simulation Model

As can be seen from Figure 8.2(b) and Figure 8.2(c), both approaches utilize a reliable link layer protocol. Reliability here means that every link layer packet is delivered to the service user exactly once and in the correct order. In both cases the LLP should therefore provide a persistent retransmission mechanism and in-sequence delivery of packets. The Link Layer Protocol (LLP) can be expected to have a significant impact on the observed performance. In order to conduct a fair comparison between ReSoA and TCP the link layer protocol of both approaches must offer a comparable performance gain. We could not find any argument against using the same LLP for end-to-end TCP and ReSoA. Therefore we decided to use the same protocol for both approaches.

The simulated LLP only implements the data transfer phase, although a connection oriented service is assumed. The connection establishment was neglected since we assume that an LLP connection is not established per TCP connection but per session (e.g. when the wireless end systems registers itself with an access point/ReSoA-server). Hence the LLP connection is already established when the first TCP/ReSoA packet is received.

We decided to use a link layer protocol that belongs to the class of continuous ARQ protocols. We chose a continuous ARQ protocol since it offers a higher efficiency than an idle ARQ protocol (e.g. Alternating Bit Protocol) for a bandwidth-delay product larger than the packet size. The retransmission strategy follows a hybrid approach. If the sender knows which packets have reached the receiver it uses selective repeat to retransmit packets. Otherwise Go-Back-N is used. Selective Repeat is used whenever possible as it is generally more efficient than Go-Back-N. Go-Back-N is used as a fallback, since in this case we assume that all packets of a window were lost. An alternative approach would have been to send short probe packets until the sender receives a control packet from the receiver containing information about the state of the receiver. We did not follow this approach because it requires at least one additional RTT to retransmit packets. Our LLP only enters Go-Back-N mode if either all packets or all acknowledgments were lost. If all packets are lost they have to be retransmitted in any case. Hence we use Go-Back-N only if all acknowledgments are lost, and this is rather an unlikely case.

[6] In a real environment a more sophisticated algorithm should be used to reduce overhead. For our simulation environment the additional overhead is not a problem, since the reverse direction provides sufficient bandwidth.

Parameter	Setting	Comment
Packet size	1500 bytes	Maximum payload per packet. If the service user request delivery of a larger packet, the packet is fragmented.
Window size	32 packets	The window allows for utilizing the link
Header Size	20 bytes	Overhead per packet

Table 10.3.: Link Layer Protocol parameters

The sender detects packet losses either by the reception of a negative acknowledgment or when a retransmission timer expires. The receiver sends an acknowledgment for every in-sequence packet[7]. Acknowledgments are cumulative, tolerating a few lost acknowledgments. If the receiver gets an out of order packet it sends a negative acknowledgment. Each negative acknowledgment informs the sender about the entire state of the receive buffer. It includes information about the last in-sequence packet as well as information about which packets were received out of order. For the latter we use a bit mask. This approach allows the sender to use selective repeat even if a few negative acknowledgments are lost.

We use a single retransmission timer for all outstanding data. This retransmission timer is restarted when a new packet is sent. The idea is that the retransmission timer should only expire if we cannot get information about the receiver's state. Since the transmission of each new packet opens the possibility that the sender either receives an acknowledgment or a negative acknowledgment, the timer can be restarted. If the retransmission timer expires we enter Go-Back-N mode. The retransmission timer value is determined by exploiting information about the bitrate and propagation delay of the link and not by RTT measurements. The retransmission timer is set to expire shortly after the next acknowledgment is expected. This allows us to react fast to packet losses. The retransmission timer value is doubled for the first four consecutive expirations of the retransmission timer. It is reset after a valid acknowledgment was received.

Flow control is realized using a window based approach. Permits are included in every acknowledgment and therefore sent after the reception of every packet. Since every in-sequence packet is immediately passed to the service user, transmission of acknowledgments and permits coincide. The maximum window size is limited to 32 since the bit field is only 32 bits wide.

The LLP segments packets exceeding the maximum packet size. The LLP introduces a fixed overhead because of its header. The header size is a parameter of the model. The parameters of the LLP are summarized in Table 10.3. The integration of our LLP into NS is described in Appendix D.1.1.

10.2.4. Wireless Network Simulation Model

In our scenario the client uses a wireless network to access the Internet. In order to reduce the complexity of our simulation model and with the intention of formulating general statements about the performance of ReSoA, we decided not to model a specific wireless network (e.g. GSM). Instead we used an abstraction which only includes the parameters of interest. The wireless network abstraction is shown in Figure 10.3.

Our wireless network is described by its bitrate, propagation delay, and error rate and distribution. Other parameters like delay variations and congestion losses due to a stochastic access protocol and cross traffic are neglected. The basic model is one queuing network per direction. This queuing network consists of a single server with a queue and a server farm. The single server models the bitrate. Its service time only

[7] A possible extension is to use piggybacking and to only send acknowledgments after a certain number of packets were received or a timer has expired. We did not include these extensions since we have a full-duplex link, and there is no traffic in the reverse direction.

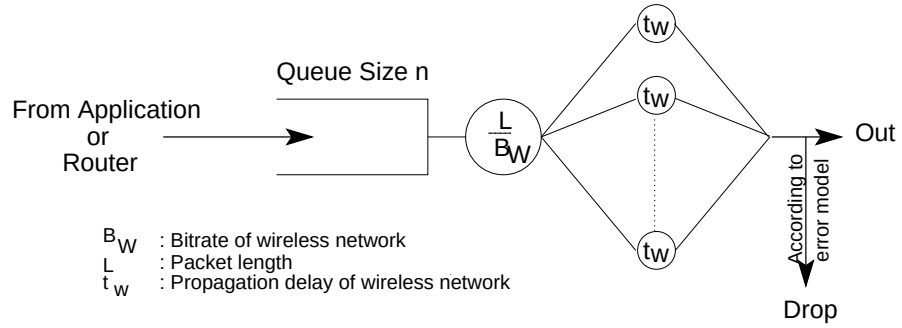


Figure 10.3.: Abstraction of the wireless network

depends on the packet size. The server farm models the transmission delay. The queuing network is fed by the client application or the access router, respectively. A packet can be dropped after it has left the server farm according to our error model[8]. The wireless network model is implemented using the bidirectional link object of NS.

Either uniform distributed errors, or a two state model with a good and a bad state are used as error model. As the error model is a crucial aspect of the simulation study, we discuss it in more detail in the following chapter. Details about the implementation of the error models are given in Appendix D.1.

Error Model

Radio channels have a higher error rate than wired networks and especially a non-stationary behavior. For instance, measurements in the 2.4 GHz ISM band showed error rates as bad as $\approx 10^{-2\ldots-3}$[33, 56, 59].

Techniques for modeling channel conditions play an essential role in protocol performance evaluation. Selecting an appropriate error model is a difficult part of simulating a wireless link. Many researchers have dealt with this question and found different formulas. Generally we can distinguish between radio propagation modeling using a stochastic process to model bit or packet errors, or to use a trace based error model.

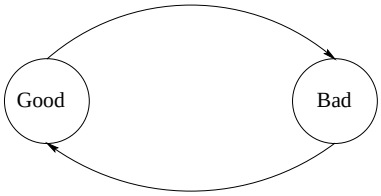
Radio propagation modeling offers the highest level of detail. It accounts for parameters like propagation distance, statistical approximation of shadowing, signal attenuation by large obstacles, line-of-sight communication, indirect communication, antenna geometries, and ray tracing, to name a few.

In [83] it is stated that detailed models are especially useful for indoor environments because of the strongly varying radio propagation. However, the authors admit that accurate models become very expensive computationally and require much more detail about the environment than is typically available.

The usage of a stochastic process to model the error characteristics of a wireless link requires a lower level of detail. The main difficulty is to find an appropriate stochastic distribution.

Due to the non-stationary behavior of a radio channel IID random variates are generally not appropriate to model the channel behavior. A simple and often used model is the Gilbert-Elliot model[62, 77, 161]. It models the radio channel as a two state Markov Chain as illustrated in Figure 10.2.4. The two states are named *Good* and *Bad*, respectively. Each state has a bit error probability, whereas the bit error rate of the bad state is much higher than the bit error rate of the good state. For each state independent bit errors are assumed. The bit error rates as well as the state transition probabilities depend on physical parameters such

[8] A possible extension of the model is to introduce delay variations by using a random variable for the service time of the single server.



$$\bar{p}_{error} = \frac{\bar{T}_good}{\bar{T}_{good} + \bar{T}_bad} p_{good} + \frac{\bar{T}_{bad}}{\bar{T}_{good} + \bar{T}_{bad}} p_{bad} \tag{10.1}$$

Figure 10.4.: Gilbert Elliot error model

as the frequency and coding scheme, as well as the environment. Normally either geometrically sojourn times or exponentially distributed sojourn times are assumed. A Gilbert-Elliot model, for example, is used in the following papers regarding the performance of TCP over lossy links [172, 3, 2, 106].

In [161] the authors develop a more general model and derive parameters for it regarding physical channel characteristics. They consider the special case of biphase shift keying (BPSK) coding over a Rayleigh fading channel[9]. The authors assume that the time variations in the received Signal to Noise Ratio (SNR) are a result of mobility (Doppler effect). Based on these assumptions the authors construct a homogeneous discrete Time Markov Chain where the transitions can occur only following a channel symbol. The states correspond to different channel qualities modeled by a specific bit error rate for each state. The bit-error rates are derived from the received SNR. The model was validated against a ray-tracing based simulation for a given set of parameters. No advice is given for the determination of the number of states.

One of the shortcomings of models based on Markov processes is that they require error statistics to remain constant over time, whereas many radio channels experience time varying effects. To model this behavior Konrad et al. in [103] propose a Markov-based Trace Analysis algorithm for the design of channel models. The approach is to divide a non-stationary trace into stationary sub-traces.

A different approach is to use either trace-driven simulations or to derive models and their parameterization from measurements. This approach is for example pursued in [128] and in [165]. In [128] the wireless channel is modeled using a two state Markov-chain and the accuracy of this model is compared against measurements and an empirical model. The paper states that the Markovian model's accuracy is limited, since neither the error nor the error-free distributions are geometric. In [165] detailed detailed measurements in an industrial environment are presented and different error models are derived from these measurements.

For our purposes uniformly distributed errors and a two state error model are sufficient. The justification for selecting these two models is on one hand that we are interested in the residual error rate seen by the data link layer and not the channel error rate. For instance, in the case of wireless LANs, the MAC level retransmissions are able to hide most of the packet losses from the higher layer protocols. On the other hand we do not investigate a specific technology but aim to derive general statements about the influence of different parameters on the performance gain due to ReSoA. Since more complex models (e.g. radio propagation models) or their parameterization generally depend on a specific technology they were no option. Furthermore the more complex a model the harder the derivation of parameters. To find good parameter settings, measurements must be carried out, leading to a parameter set valid for one specific environment.

With the selection of uniformly distributed errors and a two state error model we investigate two interesting scenarios. One shows the impact of sporadic packet loss on performance of end-to-end TCP and ReSoA while the other shows the influence of burst errors on the performance.

[9] A Rayleigh fading process is used to model a scenario in which the transmitter and receiver only move with a moderate speed and there exists a high number of signal paths between transmitter and receiver with approximately the same signal strength.

10.2.5. Internet Simulation Model

In Figure 8.2(a), which shows our basic performance evaluation setup, we pictured the Internet as a single link. If the Internet is modeled as a bidirectional link with fixed bitrate and propagation delay, important parameters such as delay variations (which can lead to a high TCP retransmission timeout value) and packet drops due to congestion are neglected. On the other hand such a simple model can be used in order to limit the number of parameters. For instance, if the influence of packets drops in the wireless network is to be studied as an isolated effect, such a model would be appropriate.

An intuitive example illustrates that a bidirectional link is not a sufficient Internet model for our simulation goal for all scenarios. In the case of multiple packet losses within a single window it is likely that TCP detects these losses by a timeout event. If the delay variation of the Internet is high, TCP's retransmission timer value will also be high since the RTT variation is used to calculate this value (see Section 2.2, Equation (2.7)). Hence, the time until these losses are detected is also high, leading to a long idle period and low throughput. In [158] it is shown that the influence of background traffic on TCP performance in a wireless Internet access context is significant. Thus, our Internet model should include delay variation and packet drops.

Basically the following questions should be considered modeling the Internet:

- **Topology:** How are networks and computers connected?
- **Traffic Generation:** What does a typical traffic mixture of the Internet look like?
- **Stability:**How often do link failures occur or how often is a route through the network changed?
- **Future:** How will the Internet develop?

Finding an answer to each of these questions is a challenging task and might even be impossible. The difficulties of Internet traffic modeling have been addressed by quite a number of researchers (e.g. [97, 68, 67, 166, 75, 139, 138, 24]).

The main reason for the complexity of the problem lies in the heterogeneity of the Internet. As pointed out in section 2.1, the Internet connects different sub-networks. Each of these sub-networks as well as the links connecting these networks can have quite different properties regarding bitrate, delay, or error probability. The routers have different queue sizes and can use different queuing disciplines to queue packets. The different sub-networks are engineered by various companies, some of which are not willing to provide information about the topology. Even if the topology was known, the paths taken by packets can change. Paxson has shown that routes through the Internet can change on time scales of seconds to days[137].

The heterogeneity argument is not only valid for the topology but also for the Internet's traffic mixture. Many different applications contribute to the Internet traffic. The generated traffic not only depends on the application but rather on the user running it. The characteristics of the generated traffic flow is additionally altered by different versions of the same protocol, interactions between different protocols, as well as different network characteristics. The traffic mixture is completed by periodically generated traffic such as routing updates.

Last but not least, even if we would know the topology of the Internet as well as the traffic mixture as it is today, this information would soon become invalid because of the dynamic nature of the Internet[10].

In spite of this heterogeneity, the Internet has some invariants as pointed out by Paxson and Floyd[139]. An invariant is defined as a facet of the Internet behavior that has been empirically shown to hold for a very wide range of environments. The following list shows some of the invariants:

[10]Current statistics about the Internet traffic can be found at http://www.internettrafficreport.com and http:www.mids.org/weather.

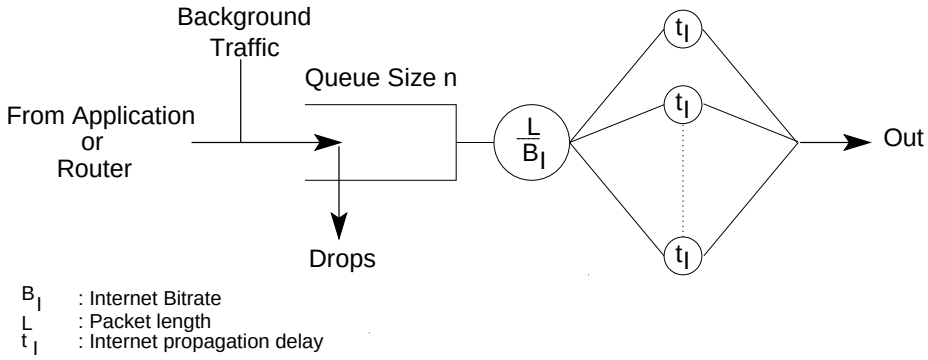


Figure 10.5.: Abstraction of Internet model

- Longer-term correlations in packet arrivals seen in aggregated Internet traffic are well described in terms of self-similar (fractal) processes. Unfortunately accurate synthesis of self-similar traffic remains an open problem, especially to determine when exactly an individual packet arrives.
- Network user sessions can be modeled by a Poisson process[138].
- The sizes or duration of connections can be modeled by a log-normal distribution instead of an empirical one. However, due to the high variability of connection characteristics, the fit was not good in many cases for both models[136].
- When determining distributions associated with the network activity expect to find heavy tails.

Background Traffic Model

As it is impossible to simulate the Internet, we decided to concentrate on different isolated effects like delay variations and losses caused by congestion. For our comparison it is rather important to control these effects than having an accurate model of the Internet. Another advantage of a simpler model is a reduced simulation time.

The basic abstraction used for our Internet model is shown in figure 10.5. It is similar to the wireless network model. The only difference is that the loss model is replaced by a background traffic process and that we use limited queues. We assume that only one link (called the bottleneck link) causes significant queuing delays, the delays caused by the other links being constant.

To model the background traffic different approaches are possible. One solution that appears at first hand is to use traces of real traffic to model the traffic injection. This approach oversees that the dominant protocol (TCP) of the Internet uses an adaptive congestion control mechanism. This means that any trace reflects the conditions of the network at the time the trace was collected. It is therefore impossible to alter some of the parameters (e.g. link speed or router queue size) since the traces cannot adapt to these changes and are hence invalid. The failure of using traces to model the Internet for example is shown in [97].

A possible alternative is to collect application level traces instead of packet level traces. Application level traces describe the traffic produced by the applications instead of traffic shaped by various protocols, links, and routers. Although the network characteristics have an influence on the user behavior, this influence is much smaller than the effects of traffic shaping. Application level behavior is investigated in [51, 136, 25]. In addition, topology generators can be used (e.g. [44]).

Another simple approach is to use an error model to discard packets. Although this approach can be used to get a first understanding about the effects of losses in the Internet, it is not appropriate as it neglects the

delay variation.

Although the modeling of user behavior and traffic generated by an application seems to be a promising approach we decided to use a simpler model. Our aim is to investigate the effects of different levels of backbone traffic on the performance gain achieved by ReSoA. Hence, for us it is more important to control these parameters than to model user behavior.

Our approach to model the Internet is shown in Figure 10.5. The background traffic is generated by a number of TCP sources that are all connected to the bottleneck router. According to the invariants discussed above, each background source has a Poisson distributed session interarrival time and the length of each session is determined by a log-normal distribution. The parameters of the distributions as well as the number of sources are model parameters. These parameters determine the degree of congestion at the bottleneck link.

10.2.6. Validation

The more complex a simulation model becomes, the harder it is to determine whether the model is an accurate representation of the actual system being studied. Since our model needs to include many components in detail, our model is already quite complex. To verify that our model works as we intend, we followed the guidelines given in [107, 94].

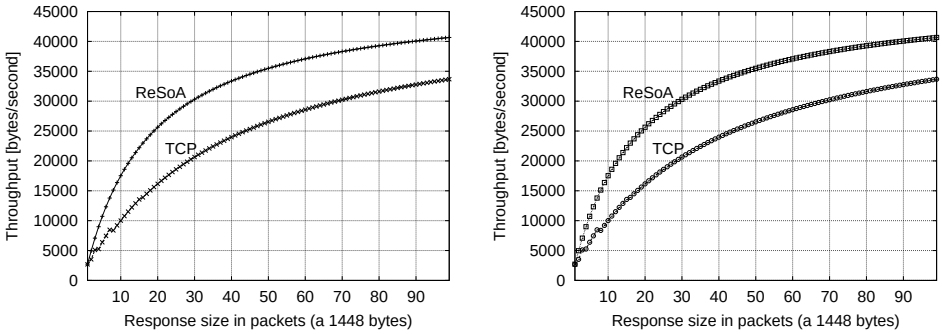
The validation process consists of the steps debugging, tracing, and thorough analysis of the trace files. In addition we compare or simulation model to our analytical model as well as our measurements. For the former we ran simulations based on the same assumptions as used for our analytical approach in Section 8.5.2. Furthermore the implementation of the ReSoA model is equipped with assertions which were enabled for all simulations. If an assertion test fails the simulation is terminated and an error is reported.

Tracing was performed on two levels. On a detailed level, trace points are inserted into the code to observe the behavior of our implementation. This level of tracing was especially important for the software modules we implemented. On a less detailed level of tracing we used the NS facilities to observe the packet flow through the network. The NS link object collects information about the exact time a packet is queued, dequeued, and when it leaves a link. This data was used to check the overall system behavior.

Comparison of Simulation Results and Analytical Results

If the simulation scenario contains neither random processes nor packet drops it is possible to calculate the arrival time of each packet at the end systems as well as the achievable throughput. For such a configuration the simulation results should be identical to the analytical results.

In Section 8.5.2 we develop formulas to calculate the achievable throughput. Since these formulas only include the data transmission from the server to the client, they cannot be used directly to compare simulation results with analytical results. Therefore we add the connection set-up time and the request transmission to these formulas in Equation (10.2) for the end-to-end case, and Equation (10.3) for ReSoA. Using these formulas we implemented scripts which calculate the throughput as a function of various parameters. These scripts allow us to compare analytical results with simulation results for a wide range of parameters. Besides looking at the final result (namely the throughput) we also calculated the arrival time of each packet at the client, and automatically compare the calculated arrival times with the trace data of the simulation log file. All performed tests have succeeded.



(a) Simulation Results

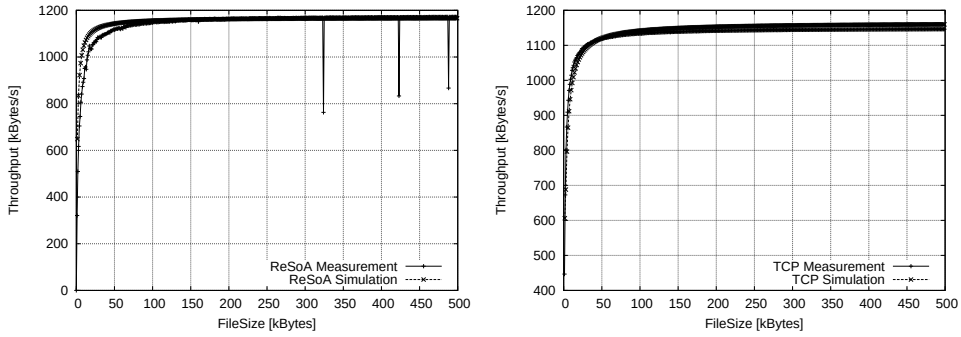
(b) Analytical Results

Figure 10.6.: Comparison of analytical results and simulation results. The set-up consists of a two hop network. The client and the router (ReSoA-server) are connected by a full-duplex link with 384Kb and 100ms latency one way. The server is connected to the router (ReSoA-server) by a second full-duplex link with 100Mb and 25ms latency one way. No packets were lost. We show the simulation results and analytical results in different figures since the curves are congruent.

$$
\begin{aligned}
T_{\mathrm{TCP}_{\mathrm{ConnEstab.}}} &= 2(A_{\mathrm{Wireless}} + A_{\mathrm{Internet}}) + 2(\tau_{\mathrm{Wireless}} + \tau_{\mathrm{Internet}}) \\
T_{\mathrm{TCP}_{\mathrm{Req}}} &= \begin{cases} 2A_{\mathrm{W}} + G_{\mathrm{Req}_{\mathrm{W}}} + G_{\mathrm{Req}_{\mathrm{I}}} + A_{\mathrm{I}} + \tau_{\mathrm{W}} + \tau_{\mathrm{I}} & \text{if } B_{\mathrm{W}} \leq B_{\mathrm{I}} \\ A_{\mathrm{W}} + 2A_{\mathrm{I}} + G_{\mathrm{Req}_{\mathrm{I}}} + \tau_{\mathrm{W}} + \tau_{\mathrm{I}} & \text{else} \end{cases} \\
T_{\mathrm{Total}_{\mathrm{TCP}}} &= T_{\mathrm{TCP}_{\mathrm{ConnEstab.}}} + T_{\mathrm{TCP}_{\mathrm{Req}}} + T_{\mathrm{TCP}}(N) \\
\mathrm{Throughput} &= \frac{\mathrm{RequestSize} + \mathrm{ResponseSize}}{T_{\mathrm{Total}_{\mathrm{TCP}}}}
\end{aligned}
\tag{10.2}
$$

$$
\begin{aligned}
T_{\mathrm{ReSoA}_{\mathrm{ConnEstab.}}} &= G_{\mathrm{ReSoA\_req}} + 2A_{\mathrm{I}} + G_{\mathrm{ReSoA\_res}} + 2(\tau_{\mathrm{W}} + \tau_{\mathrm{I}}) \\
T_{\mathrm{ReSoA}_{\mathrm{Req}}} &= G_{\mathrm{ReSoA\_inv}} + G_{\mathrm{Req}_{\mathrm{I}}} + A_{\mathrm{I}} + \tau_{\mathrm{W}} + \tau_{\mathrm{I}} \\
T_{\mathrm{Total}_{\mathrm{ReSoA}}} &= T_{\mathrm{ReSoA}_{\mathrm{ConnEstab.}}} + T_{\mathrm{ReSoA}_{\mathrm{Req}}} + T_{\mathrm{ReSoA}}(N) \\
\mathrm{Throughput} &= \frac{\mathrm{RequestSize} + \mathrm{ResponseSize}}{T_{\mathrm{Total}_{\mathrm{ReSoA}}}}
\end{aligned}
\tag{10.3}
$$

Figure 10.6 shows the simulated and calculated throughput for TCP and ReSoA for a single parameter setting as example for the performed tests. Since the curves of simulation and analysis are congruent we use two figures to show both curves. Figure 10.6(a) shows the simulation results and Figure 10.6(b) shows the analytical results.



(a) ReSoA (b) TCP

Figure 10.7.: Comparison of simulation and measurement

Comparison between Implementation and Simulation

A second approach to show the correctness of our simulation model is to compare simulation results with measurements. Contrary to the comparison in Section 10.2.6 we should not expect the results obtained by simulation and measurement to be identical. The results will differ to some degree since not all side effects of measurements are observable, and the simulation model abstracts from some details (e.g. processing time). However, to show the correctness of the simulation model the throughput curves obtained by either approach should be similar.

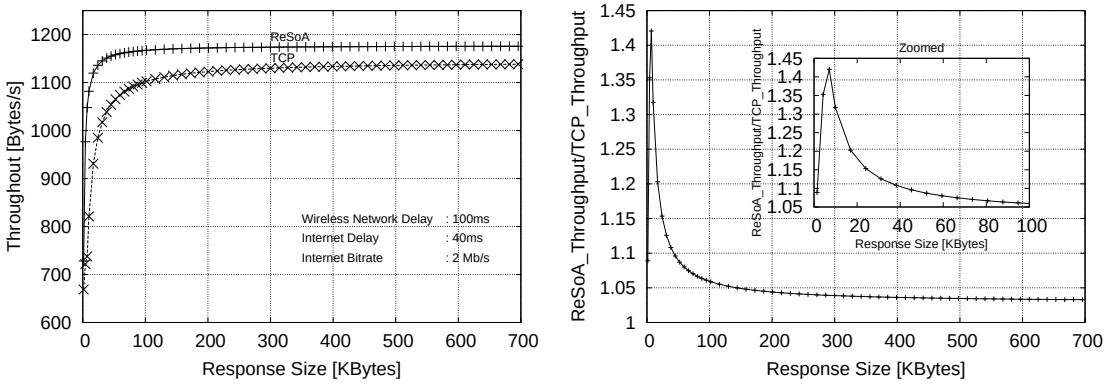
In order to reduce the number of side-effects inherent to measurements we decided to use a cross-connect cable instead of a wireless LAN to connect the client and the access router. A second advantage of a cross-connect cable is that it provides a bidirectional link as we assume in our simulations. In order to model the characteristics of the cross-connect cable, we set the error rate of our link to zero, the bitrate to 10 Mbit/s and the delay to 0.09 ms. The delay was obtained by using the `ping`-tool. The rest of the environment was identical to our measurement set-up in Section 9.

Figure 10.7(a) and Figure 10.7(b) show the results of the comparison for ReSoA and TCP, respectively. The TCP curves for simulation and measurement match well. For ReSoA there is a small difference for small response sizes. The simulation shows a slightly higher throughput than the measurement. In Section 9.3 we saw that our ReSoA implementation is not optimal for small response sizes (e.g. overhead due to scheduling delay). Therefore we believe that the difference is caused by our implementation. The curves for TCP show no difference because the TCP implementation is optimized. Furthermore, the measurement shows three outliers which cannot be found in the simulation curve. The shape of both curves is identical. Because the curves nearly match (and we expected some differences between measurement and simulation) we consider the test as successful.

10.3. Simulation Results

10.3.1. Results 1: Packet Loss in the Access Network

As discussed in Section 8.5.4, the goal of the simulations discussed in this section is to investigate whether packet loss in the wireless network increases the performance gain by ReSoA. The analytical approach



(a) Throughput of ReSoA and TCP

(b) Ratio

Figure 10.8.: TCP vs. ReSoA over response size. Parameters: No errors, wireless link delay: 100ms, Internet delay: 40ms, wireless bitrate 9600b/s, Internet bitrate: 2Mb/s

in Section 8.5.2 shows that the performance improvement depends on the wireless network delay and the response size. Therefore we used these two factors and investigated how packet loss influences this finding.

Wireless Link Bitrate: 9600 bit/s

In the case of a wireless network bitrate of 9600 bit/s the throughput is determined by the time required to send a packet on the wireless link (generation time). For instance, it takes 1.25 seconds to send a full sized packet (1500 bytes) which obviously has the highest impact on the RTT. The analysis of the simulation results showed that the throughput is similar for all investigated combinations of wireless network delay and Internet delay (see Section 8.5.4). Therefore the discussion below is based on a limited number of levels.

Figure 10.8(a) shows the throughput of ReSoA and end-to-end TCP as a function of the response size for the error free case, while Figure 10.8(b) shows the ratio of the throughput achieved by ReSoA to the throughput achieved by TCP. We show these two figures in order to give an overview of the achievable throughput. A ratio smaller than one means that TCP is better, a ratio higher than one means that ReSoA is better, and a ratio equal to one means that both approaches provide the same throughput. In the pictured scenario the wireless network delay was set to 100 ms, the Internet bitrate was set to 2 Mb/s and the Internet delay was set to 40 ms.

From Figure 10.8 we can see that ReSoA offers a higher throughput than TCP for all investigated response sizes. However, the performance improvement is only significant for response sizes smaller than 40 KBytes. For a response size of 40 KBytes the performance improvement is around 10%. If the response size is larger than 40 KBytes the improvement will be reduced to around 5%.

ReSoA is able to improve the throughput for two reasons. First, ReSoA is able to fully utilize the wireless network as soon as the first packet of the response is received by the access point. Since the TCP acknowledgment is sent before the packet is forwarded to the wireless end system, the wireless link would only be idle if the RTT between fixed host and access point is longer than 1.25s (time to send the packet on the radio link). In the case of end-to-end TCP this is different. The TCP-acknowledgment is sent by the wireless end system. Hence the wireless link is idle while the acknowledgments travel from the wireless

end system to the fixed host. In the case of TCP the wireless link is fully utilized if the TCP sender is allowed to send two packets. Hence, TCP fully utilizes the wireless link one RTT later than ReSoA.

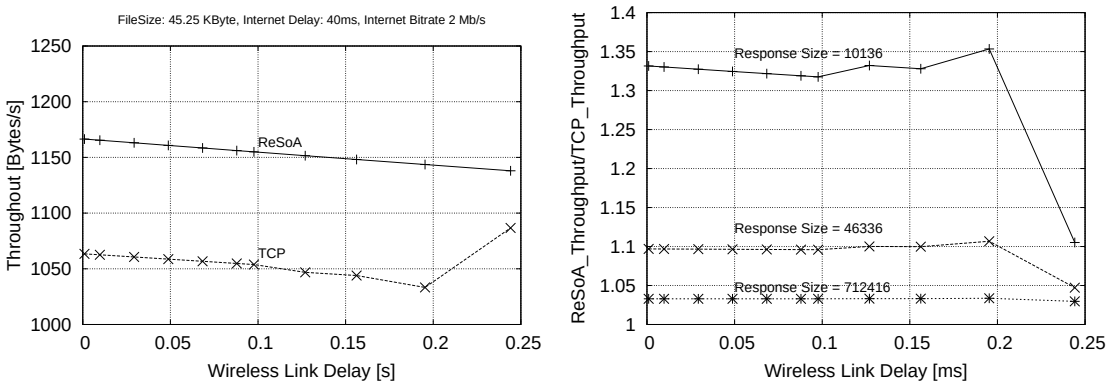
Second, TCP suffers from a spurious retransmission of the first packet of the response while ReSoA does not. For TCP the estimation of the RTT for the first packet of the response is wrong because the packet length has a high impact on the RTT. Unfortunately, the first packet (SYN-ACK) that is sent from the fixed host to the wireless end system is shorter than the data packets. When the acknowledgment for the first packet is received TCP uses the RTT measurement to calculate the retransmission timer for the next packet as shown in Equation (2.4). The resulting retransmission timer value is usually smaller than the RTT of the first full sized packet as shown in Equation (10.4). In Equation (10.4) we use the aforementioned parameter values as an example. For these parameter settings the RTT of the SYN/ACK-segment is 0.3664s, resulting in a retransmission timer value of 1.0092s. Unfortunately, the RTT of the next packet is 1.5792s.

The throughput of both approaches is smaller for small response sizes since our throughput calculation includes the connection establishment phase as well as the transmission of the request. The performance gain by ReSoA is larger for small files because the start-up problems of TCP play a less important role for larger response sizes.

$$
\begin{aligned}
T_{\mathrm{Internet}_{\mathrm{SYN/ACK}}} &= 0.0002\mathrm{s} \\
T_{\mathrm{Wireless}_{\mathrm{SYN/ACK}}} &= 0.043\mathrm{s} \\
\mathrm{RTT}_{\mathrm{SYN/ACK}} &= 2 * (T_{\mathrm{Internet}_{\mathrm{SYN/ACK}}} + T_{\mathrm{Wireless}_{\mathrm{SYN/ACK}}} + \tau_{\mathrm{Internet}} + \tau_{\mathrm{Wireless}}) \\
&= 0.3664ms \\
\mathrm{RTO} &= 0.3664 + 0.7328 = 1.0992 \\
T_{\mathrm{Internet}_{1500}} &= 0.006\mathrm{s} \\
T_{\mathrm{Wireless}_{1500}} &= 1.25\mathrm{s} \\
\mathrm{RTT}_{1500} &= T_{\mathrm{Internet}_{1500}} + T_{\mathrm{Wireless}_{1500}} + T_{\mathrm{Internet}_{\mathrm{ACK}}} + \\
&\quad T_{\mathrm{Wireless}_{\mathrm{ACK}}} + 2 * (\tau_{\mathrm{Internet}} + \tau_{\mathrm{Wireless}}) \\
&= 1.5792
\end{aligned}
\tag{10.4}
$$

Figure 10.9 shows the throughput of both approaches as well as the improvements by ReSoA as a function of the wireless network delay. As we already mentioned, the performance gain by ReSoA is nearly independent from the wireless network delay. However, an interesting point of Figure 10.9 is that the throughput of TCP increases if the latency exceeds 0.2 seconds, while the throughput of ReSoA is nearly constant. The reason for this performance improvement is that the retransmission timer does not expire too soon if the access network delay is large enough. In this case the RTT of the SYN/ACK is 0.6664, resulting in an RTO value of 1.999s. This RTO value is larger than the RTT of the first full sized packet. Since TCP's throughput increases for a wireless network delay of more than 250ms and ReSoA's throughput remains constant, the performance gain by ReSoA is smaller for high wireless network delays. Figure 10.9(b) shows that the influence of the wireless network delay on the performance gain becomes even smaller for larger response sizes. For a response size of 712416 bytes the performance gain by ReSoA is constant for all investigated wireless network delays. Especially the performance penalty due to the spurious retransmission has no effects for large response sizes.

Figure 10.10(a) shows the throughput for uniformly distributed errors as a function of the response size, while in Figure 10.10(b) the wireless network delay is used as x-axis. Both figures show the throughput of the error free case as reference and the throughput achieved if the packet loss rate is set to 0.2. In the latter case the figure also shows the 95%-confidence intervals (for large response sizes the confidence intervals are too small to be visible). Although we also investigated configurations with lower error rates, we do not



(a) Throughput of ReSoA and TCP

(b) Ratio

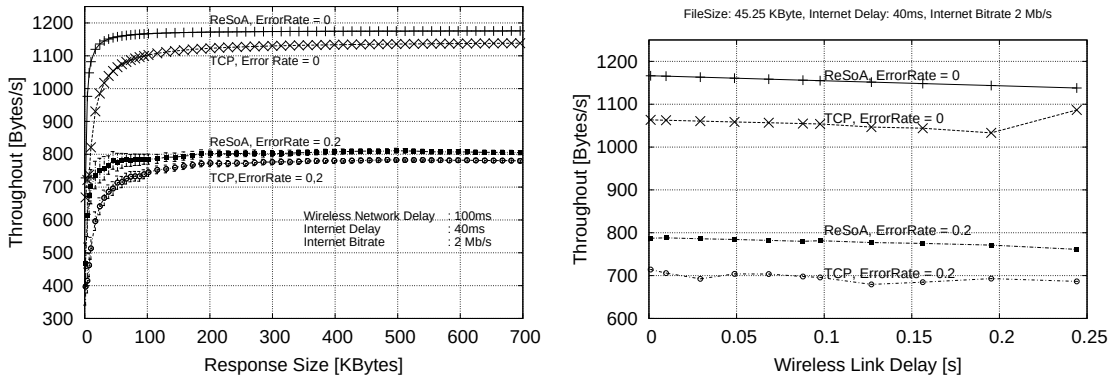
Figure 10.9.: TCP vs. ReSoA as a function of the wireless network delay. Parameter: No erros, wireless bitrate 9600 bits/s, Internet delay: 40 ms, Internet bitrate: 2Mb/s

show the results here because the resulting curves are similar to the one presented here. The only difference is that the throughput is higher if the error rate is lower. We did not investigate error rates higher than 0.2 for uniformly distributed errors since averagely losing every fifth packet is already a high error rate.

Figure 10.10(a) shows that the observations of the error free case are still valid if the error rate is set to 0.2. ReSoA achieves a higher throughout than TCP, and with increasing response size the performance difference between ReSoA and TCP decreases. Further it can be seen that the performance gain due to ReSoA is marginally smaller than in the error free cases. This is caused by link layer retransmissions. As time is needed to retransmit lost frames for both approaches, there are less resources that can be better utilized by ReSoA.

A result of these simulations is that a reliable link layer protocol is well suited to hide uniformly distributed errors from a TCP sender. Therefore ReSoA is unable to increase the performance gain for this configuration.

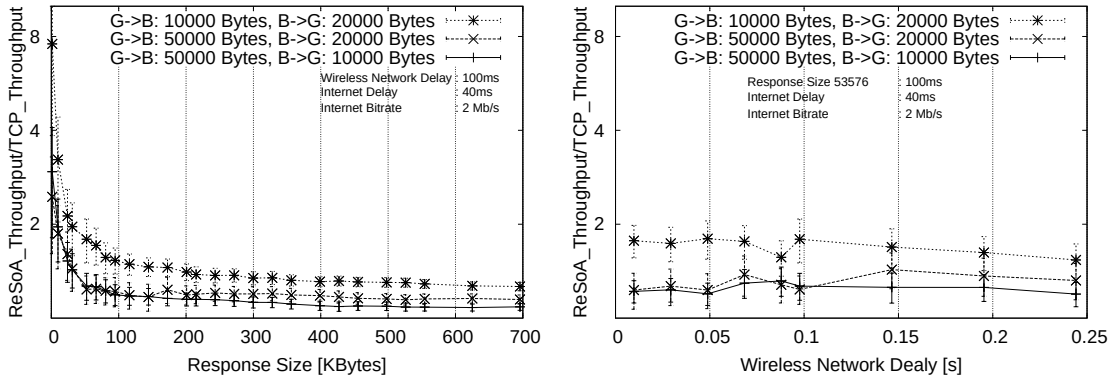
Figure 10.11 shows performance results for packet loss occurring in the wireless network according to our two state error model. This time we only show the ratio of ReSoA's performance to TCP's performance in order to be able to show the performance results of different two state error model factor levels in a single figure without overloading it. Further we chose a logarithmic representation of the y-axis. The general shape of Figure 10.11 is identical to the previously shown figures. The performance gain by ReSoA decreases with increasing response size and is independent of the wireless network delay. However, the performance gain by ReSoA is increased for a lousy channel. If the sojourn times of the good and bad states are set to the time needed to send 10000 Bytes and 20000 Bytes, respectively, ReSoA nearly doubles the throughput of TCP for response sizes smaller than 100 KBytes and still improves the performance by about 30% for larger response sizes. Although the link layer protocol is able to retransmit all lost frames the TCP sender cannot handle the RTT variation if the TCP receiver is located in the wireless network, resulting in spurious retransmissions.



(a) Throughput over response size

(b) Throughput over wireless network delay

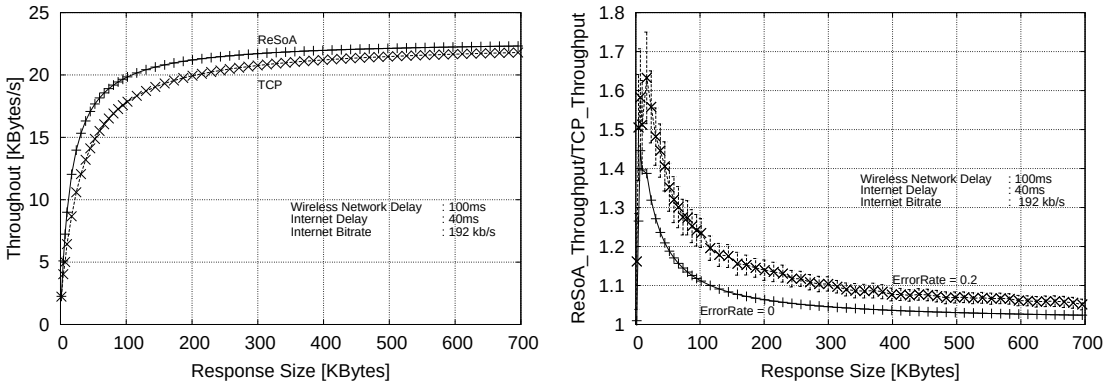
Figure 10.10.: TCP's and ReSoA's throughput in the case of uniformly distributed errors and a wireless network bitrate of 9600 b/s



(a) As function of response size

(b) As function of wireless network delay

Figure 10.11.: Comparison of ReSoA and TCP using a two state error model



(a) Throughput of ReSoA and TCP

(b) Ratio of ReSoA to TCP

Figure 10.12.: ReSoA vs. TCP for a wireless network bitrate of 384kb/s as a function of the response size.

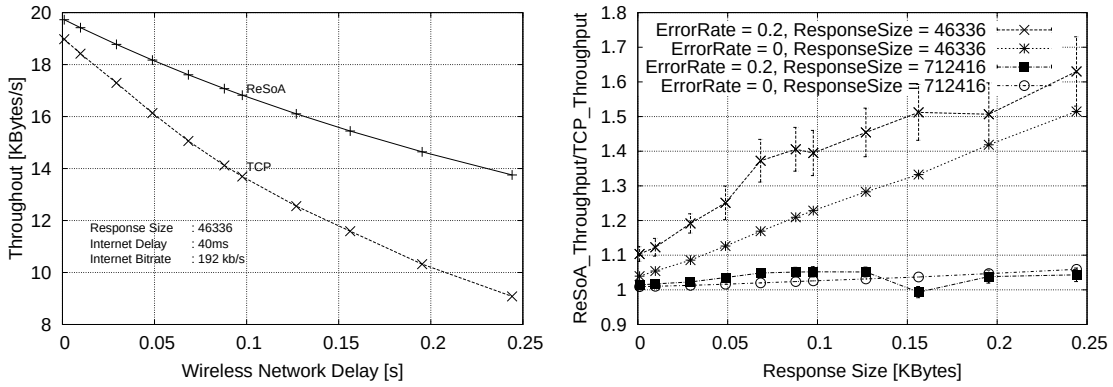
Wireless Link Bitrate: 384kb/s

In this section we discuss simulation results for the same configurations as in the previous section but for a wireless network bitrate of 384kb/s. We added configurations where the Internet offers a smaller bitrate than the wireless network. Similar to the previous section we observed that the Internet delay only has a marginal impact on the performance gain by ReSoA. Therefore we use the same value of 40 ms for all figures in the following discussion.

Figure 10.12 shows the simulation results if the Internet includes the bottleneck link. Figure 10.12(a) shows the throughput of TCP and ReSoA as a function of the response size for the error free case. Figure 10.12(b) shows the ratio of ReSoA to TCP for the error free case and for an error rate of 0.2. Since the Internet link includes the bottleneck link, it limits the achievable throughput. In the error free case ReSoA and TCP achieve nearly the same throughput for response sizes larger than 400 KBytes. For response sizes smaller than 400 KBytes ReSoA can improve the throughput significantly. An improvement of around 40% is achieved for small response sizes (e.g. 10136 Bytes). Although the performance gain decreases with increasing response size the performance improvement is still approximately 10% for response sizes of 100 KByte.

Packet loss in the wireless network increases the performance gain by ReSoA as can be seen from Figure 10.12(b). In this Figure the vertical lines represent the 95% confidence interval. We do not show a confidence interval for the lower curve (ErrorRate = 0) as its width is zero. If the communication over the wireless links suffers from a packet loss rate of 0.2, the wireless link becomes the bottleneck. Every link layer retransmission increases the RTT seen by TCP (end-to-end case). If the RTT is longer, TCP's Slow Start mechanisms needs more time to utilize the available resources. Therefore the performance gain by ReSoA is larger for small response sizes. For instance, for a response size of 50 KBytes, ReSoA is able to improve the performance of TCP by approximately 35%, whereas for a response size of 100 KBytes the improvement is around 23%. The reason for the performance improvement is solely the increased delay of the wireless network due to link layer retransmissions. The link layer protocol is able to fully hide the losses from the TCP sender and the TCP sender does not suffer from spurious retransmissions.

Figure 10.13 shows the simulation results as a function of the wireless network delay. Figure 10.13(a)



(a) Throughput of ReSoA and TCP

(b) Ratio of ReSoA to TCP

Figure 10.13.: ReSoA vs. TCP for a wireless network bitrate of 384kb/s as a function of the wireless network delay

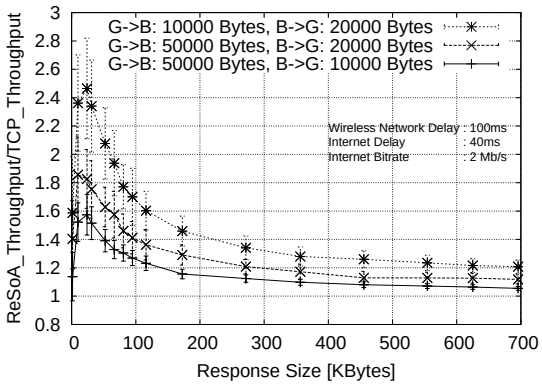
shows the throughput of ReSoA and TCP while Figure 10.13(b) shows the ratio of ReSoA to TCP. As expected, the performance gain increases with increasing delay while the throughput decreases. However, this is only true if the response size is not too large. For example, if the response size is 700 KBytes, TCP and ReSoA nearly offer the same throughput, even in the case of errors. For large response sizes TCP is able to open its window wide enough to utilize the available resources. Since the error rate of the wireless network is hidden from the TCP sender, TCP is able to catch up with ReSoA's performance.

Figure 10.14 pictures the performance of ReSoA and TCP for the two state packet loss error model. To show the effects of this error model on the performance gain by ReSoA we selected three error model configurations. The sojourn times in the case of the lower curve of Figure 10.14 were set to 50 KBytes in the good state and 10 KBytes in the bad state[11]. This configuration corresponds to a mean packet error rate of 0.16. For the middle curve we used sojourn times of 50 KBytes and 20 KBytes (mean packet error rate 0.28), respectively. For the upper curve the sojourn times were set to 10 KBytes and 20 KBytes resulting in a mean packet error rate of 0.66. The performance gain by ReSoA decreases with increasing sojourn time in the good state and increases with increasing sojourn time in the bad state. The general observation, namely that the performance gain decreases with increasing response size and increases with increasing wireless network delay, is not changed by the two state error model. As in the case of uniformly distributed errors, TCP (end-to-end case) only sees an increased average RTT. The performance increase by ReSoA is solely due to the delay introduced by link layer retransmissions and not due to spurious retransmissions.

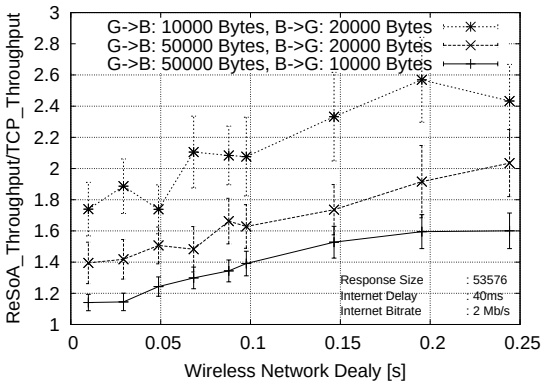
Figures 10.15 to 10.17 show the simulation results where the wireless network is the bottleneck. We could not find a genuine difference between the configuration where the bitrate of the Internet is identical to the bitrate of the wireless network and the configuration where the Internet offers a higher bitrate. Therefore we limit the discussion to the latter case.

Figure 10.15(a) shows the throughput of ReSoA and TCP. The throughput of TCP and ReSoA is higher than in the previous case because the bitrate of the bottleneck is 384kb/s in contrary to 192 kb/s in the previous case. However, ReSoA only benefits from the higher bottleneck bitrate as long as the error rate

[11] We use bytes to describe the sojourn times although this is no measure for time. However bytes are more descriptive than seconds. A sojourn time of n Bytes means the time to transfer these n Bytes.



(a) Ratio of ReSoA to TCP as a function of the response size

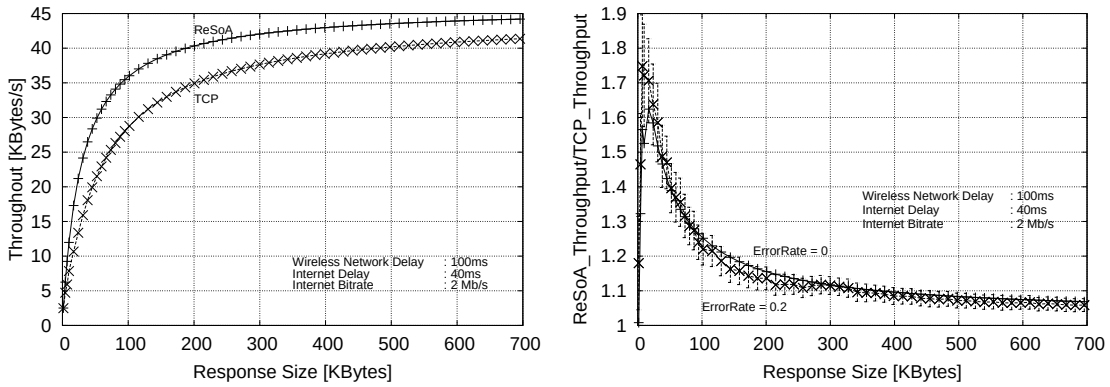


(b) Ratio of ReSoA to TCP as a function of the wireless network delay

Figure 10.14.: ReSoA vs. TCP in the case of burst errors and a wireless network bitrate of 384kb/s

is not too high as can be seen in Figure 10.15(b). For an error rate of 0.2 the performance improvement is identical to the previous configuration where the Internet includes the bottleneck link. If the packet loss rate is high the wireless network becomes the bottleneck independent of the Internet bitrate. Hence the gain is identical in both cases.

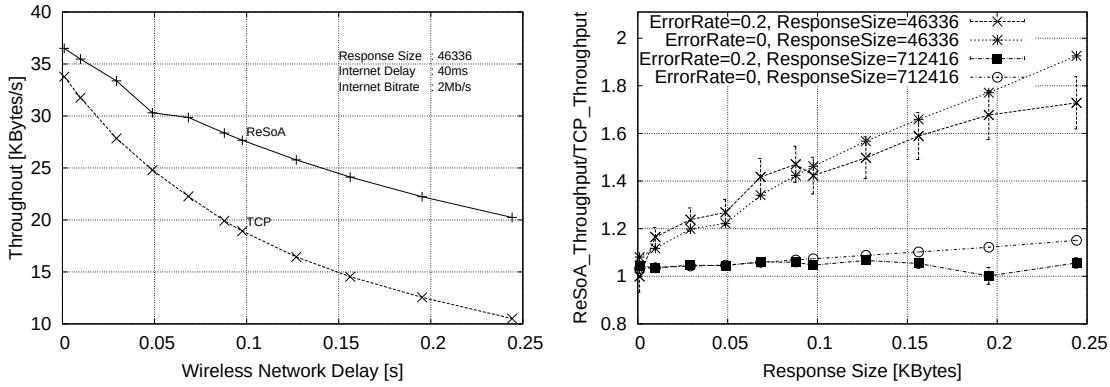
The curves shown in Figure 10.16 and in Figure 10.17 have similar shapes as the curves shown in Figure 10.13 and Figure 10.14. The delay introduced by the link level retransmission dominates the performance gain while bitrate difference between wireless network and Internet does not.



(a) Throughput of ReSoA and TCP

(b) Ratio of ReSoA to TCP

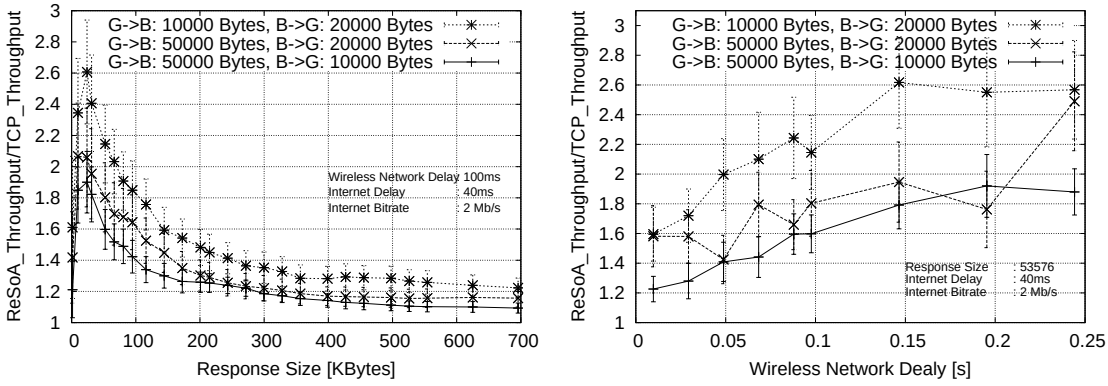
Figure 10.15.: ReSoA vs. TCP for a wireless network bitrate of 384kb/s as a function of the response size.



(a) Throughput of ReSoA and TCP

(b) Ratio of ReSoA to TCP

Figure 10.16.: ReSoA vs. TCP as a function of the wireless network delay. The wireless network bitrate was set to 384kb/s.



(a) Ratio of ReSoA to TCP as a function of the response size

(b) Ratio of ReSoA to TCP as a function of the wireless network delay

Figure 10.17.: ReSoA vs. TCP in the case of burst errors and a wireless network bitrate of 384kb/s

Wireless Link Bitrate: 11Mb/s

Figure 10.18 to Figure 10.23 show the effects on the performance gain if we further increase the bitrate of the wireless network. These figures show simulation results of a wireless bitrate of 11 Mb/s and Internet bitrates of 5.5 Mb/s and 22 Mb/s.

Figure 10.18 shows the results as a function of the response size. In contrary to the previous configurations, the Internet latency was set to 1 ms because 100 ms is quite a long time for 11 Mb/s. Figure 10.18(a) shows the throughput of ReSoA and TCP if the Internet includes the bottleneck link, while Figure 10.18(b) shows the ratio of ReSoA to TCP. The former only shows the throughput obtained in the error free case while the latter shows the ratio for the error free case and for an error rate of 0.2. For error rates less than 0.2 the performance gain lies between the two curves shown. Since the wireless link only has a small impact on the total RTT, both ReSoA and TCP nearly achieve the same throughput in the error free case. For all investigated response sizes the improvement is below 3%. Although the throughput curves are nearly congruent the ratio curve has an interesting shape for small response sizes. For a single packet response (response size 1448 bytes) the ratio is one since the performance is dominated by the time needed for the connection establishment and transmission of the request. If the response size is smaller than 100 KBytes ReSoA will slightly benefit from a shorter RTT seen by TCP. This advantage vanishes as the capacity of the bottleneck link is reached.

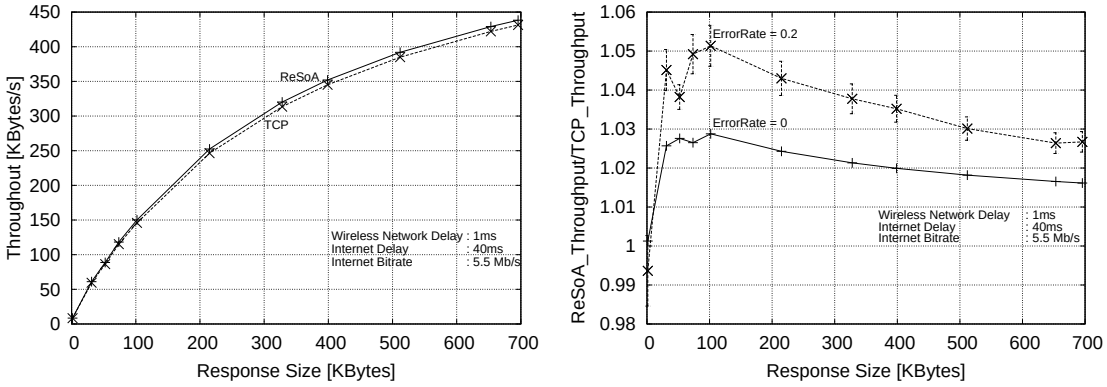
In the case of errors (upper curve of Figure 10.18(b)) the average RTT seen by TCP is increased while it is constant for ReSoA. Hence the performance gain by ReSoA is increased. This is especially an advantage for small response sizes because the TCP sender at the fixed host needs RTTs before it can fully utilize the bottleneck link. During this time the link layer protocol has sufficient time to retransmit lost packets before a new packet arrives.

Figure 10.19 shows the simulation results as a function of the wireless network delay for two different response sizes. The curves substantiate the interpretation of Figure 10.19(a). In the case of a long response the performance gain by ReSoA is negligible. This is true independent of the wireless network delay or the error rate. For short response sizes (e.g. 31856 Bytes) the performance gain increases with increasing wireless network delay and error rate. If the wireless network delay is higher the RTT seen by end-to-end TCP will increase while the RTT seen by TCP will stay constant in the case of ReSoA. Hence the performance gain by ReSoA increases. Please note that the absolute performance improvement is higher for larger response sizes. As can be seen from Figure 10.19(a), ReSoA improves the throughput by up to 50KBytes/s if the response length is 712416 Bytes, while it improves the throughput by less than 20 KBytes/s for a 31856 Byte response. However, the throughput achieved by both TCP and ReSoA is higher in the former case, and therefore the ratio of ReSoA to TCP is smaller than in the case of a short response.

Figure 10.20 shows the ratio of ReSoA to TCP in the case of burst errors. As in the previous configurations, the performance gain increases with increasing error rate. The reason for the performance improvement again is the increased delay of the wireless link due to link layer retransmissions.

Figure 10.21 to Figure 10.23 show simulation results for an identical configuration but with an Internet bitrate of 22Mb/s instead of 5.5Mb/s. The shapes of these Figures are similar to Figure 10.18 through Figure 10.20 albeit the achieved throughput is higher for both TCP and ReSoA. Contrary to configurations with a wireless bitrate of 9600b/s or 384kb/s the performance in this configuration is dominated by the delays between fixed host and access point, as well as the between access point and wireless end system. As these delays are independent of the wireless network bitrate, the performance gain is independent from the Internet bitrate, unless the generation time is in the order of the Internet delay.

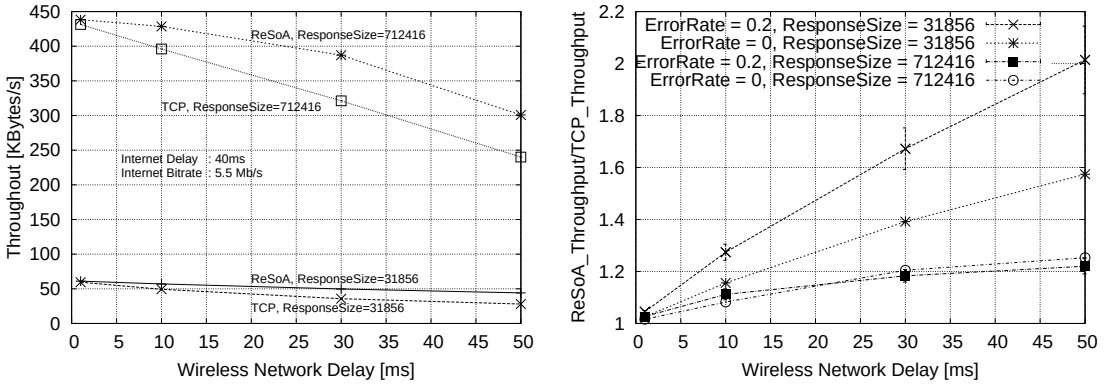
Our first simulation set showed that the performance gain by ReSoA in fact depends on the time the wireless network adds to the total RTT. With increasing delay of the wireless network the performance gain by ReSoA increases. The RTT share of the wireless network in turn depends on its propagation delay, bitrate, and error probability and distribution. This is especially true for small response sizes. If the response size is large enough that also TCP is able to fully utilize the bottleneck link, the performance



(a) Throughput of ReSoA and TCP

(b) Ratio of ReSoA to TCP

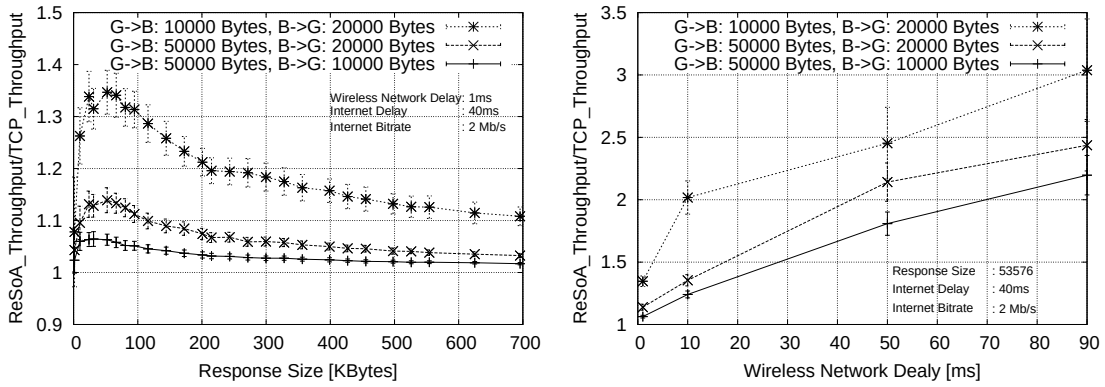
Figure 10.18.: ReSoA vs. TCP for a wireless network bitrate of 11Mb/s as a function of the response size



(a) Throughput of ReSoA and TCP

(b) Ratio of ReSoA to TCP

Figure 10.19.: ReSoA vs. TCP for a wireless network bitrate of 11Mb/s as a function of the wireless network delay



(a) Ratio of ReSoA to TCP as a function of the response size

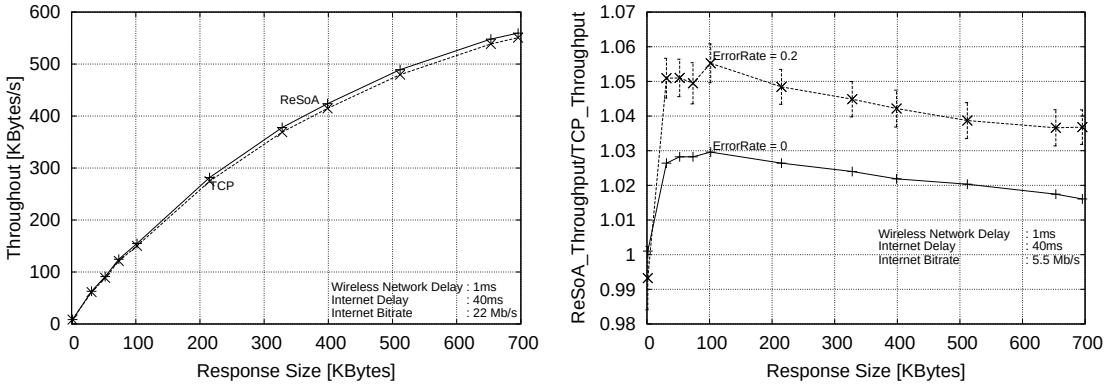
(b) Ratio of ReSoA to TCO as a function of the wireless network delay

Figure 10.20.: ReSoA vs. TCP in the case of burst errors and a wireless network bitrate of 11 Mb/s

improvement will disappear with increasing response size.

The performance increase for almost all investigated configurations is solely based on ReSoA's ability to utilize the bottleneck link quicker than TCP. We observed spurious retransmissions only if the two state error model was used with a long mean sojourn time for the bad state.

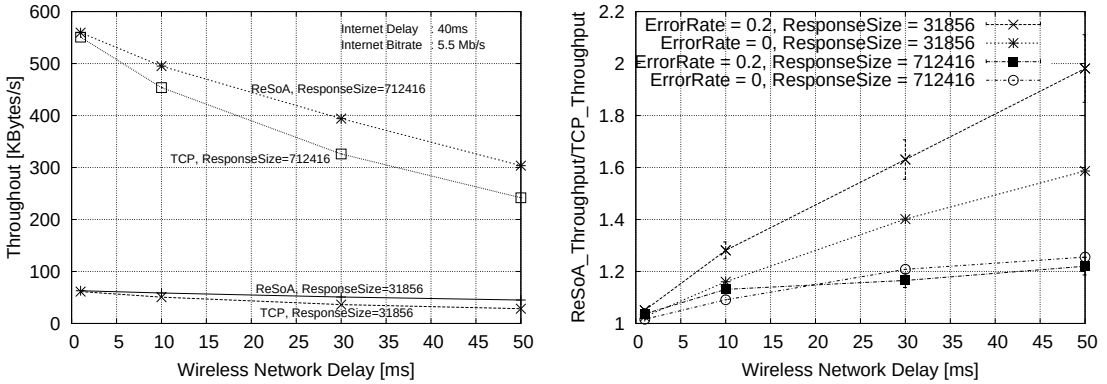
Thus, we can conclude that an error prone link improves the performance gain by ReSoA. This is true, even if a reliable link layer protocol is used to hide the losses from the TCP sender. However, the performance gain is not caused by wrong TCP actions (interpreting losses as congestion or spurious retransmission) but by an increased RTT. If local retransmissions are required, TCP's performance will suffer because the ACK Clock has a lower frequency and the Slow Start algorithm takes longer to reach network capacity. Since both mechanisms are crucial for the operation of the Internet, ReSoA has an inherent advantage over end-to-end TCP. The performance gain is additionally increased in the case of spurious retransmissions. However, spurious retransmissions rarely occur.



(a) Throughput if ReSoA and TCP

(b) Ratio of ReSoA to TCP

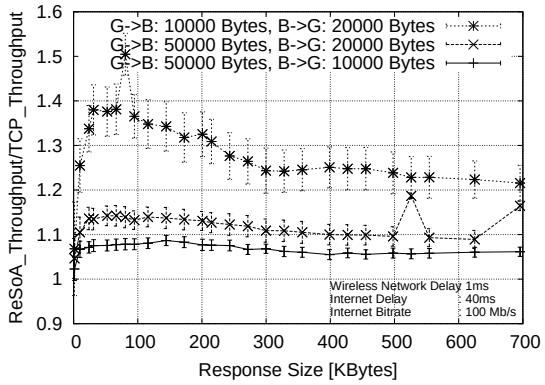
Figure 10.21.: ReSoA vs. TCP for a wireless network bitrate of 11Mb/s as a function of the response size



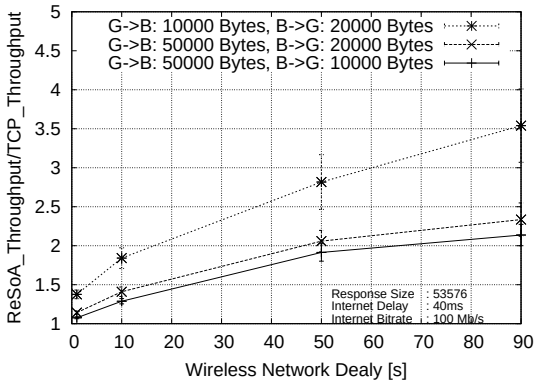
(a) Throughput of ReSoA and TCP

(b) Ratio of ReSoA to TCP

Figure 10.22.: ReSoA vs. TCP for a wireless network bitrate of 11Mb/s as a function of the wireless network delay

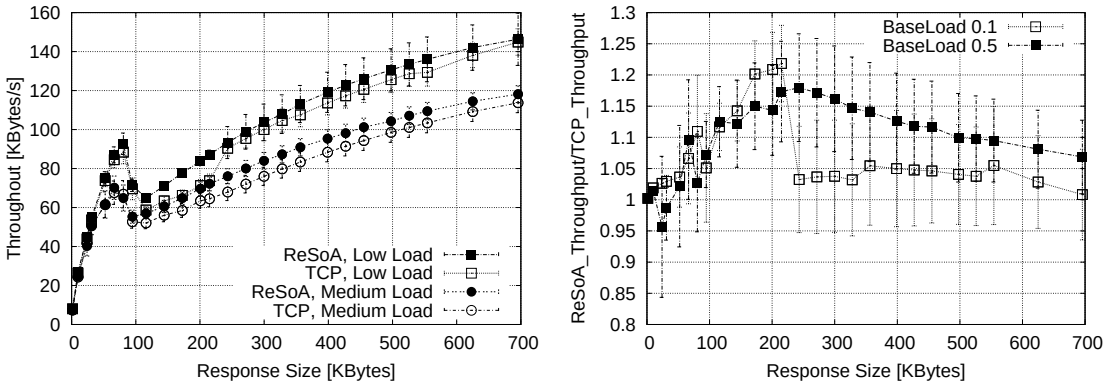


(a) Ratio of ReSoA to TCP as a function of the response size



(b) Ratio of ReSoA to TCP as a function of the wireless network delay

Figure 10.23.: ReSoA vs.TCP in the case of burst errors and a wireless network bitrate of 11Mb/s



(a) Throughout of ReSoA and TCP

(b) Ratio of ReSoA to TCP

Figure 10.24.: Effects of background traffic on the performance of ReSoA and TCP as a function of the response size. Wireless network bitrate 11Mb/s, wireless network delay 1ms, Internet bitrate 2Mb/s, Internet delay 40ms

10.3.2. Results 2: Internet Background Traffic

Figure 10.24 through Figure 10.27 show the effects of Internet background traffic on the performance of ReSoA and TCP. We present results for two different wireless network configurations and at least two different background traffic levels. Figure 10.24 and Figure 10.25 show the simulation results for a wireless network with a bitrate of 11Mbit/s and a delay of 1ms. Figure 10.26 to Figure 10.27 present results for a wireless network with a bitrate of 384kb/s and a delay of 50ms. For each configuration we show the simulation results as a function of the response size and wireless network delay.

For a wireless network bitrate of 11Mbit/s the RTT is nearly independent of the time needed to send a packet on the wireless network. If the delay of the wireless network is also small the RTT will be dominated by the backbone network. Hence the RTT seen by the TCP sender is nearly identical for end-to-end TCP and ReSoA. This is illustrated by Figure 10.24(a). Independent of the background traffic level ReSoA and TCP achieve a comparable throughput. Although the ReSoA curve is slightly better the confidence intervals are overlapping. Therefore statistically both approaches are identical. However, the curves have a quite interesting shape that requires further explanation. In the case of small response sizes the transmission of the response is not disturbed by losses caused by background traffic. This is a consequence of our background traffic model and the load levels we used. With increasing response size the total load also increases until the first losses occur at a response size of 94 KBytes. This leads to the local throughput maximum at 80 KBytes for the low load scenario and at 66 KBytes for the medium load. As the first losses occur, end-to-end TCP has more difficulty to adapt to the network capacity due to its slightly larger RTT. Therefore the performance difference between ReSoA and TCP is larger for response sizes between 97 Kbytes and 195 KBytes. This can be observed for both load levels but is more distinct for the lower levels. With increasing response size end-to-end TCP is able to adapt to the network state as well and both approaches achieve a comparable throughput.

Figure 10.25 shows the simulation results as a function of the wireless network delay. As already shown in section 10.3.1 the performance gain increases with increasing delay. Internet background traffic does

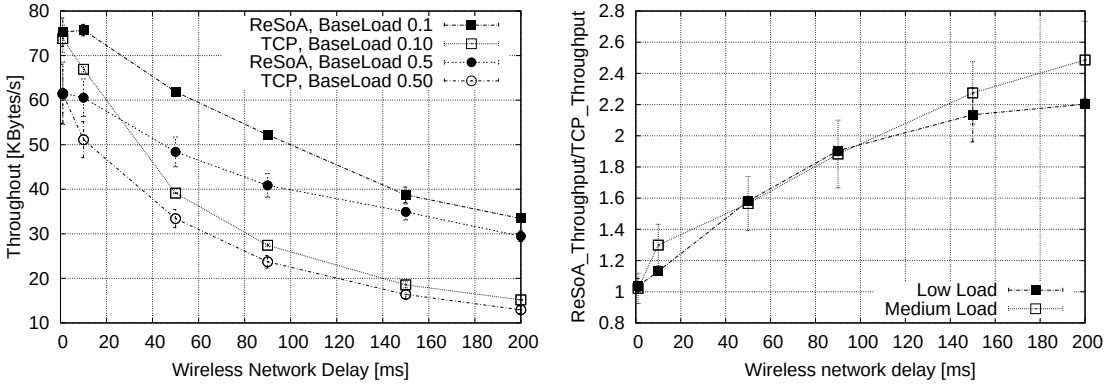
not change this observation. This is illustrated in Figure 10.25(b) for a response size of 53576 bytes. The performance gain shown in this figure is an effect of the increased wireless network delay and not caused by the background traffic, since no packets are lost due to background traffic. If the response size is increased, packets will be lost in the Internet. In this case the performance gain by ReSoA increases faster with increasing wireless network delay than for small response sizes. The performance gain by ReSoA is higher for an increased background traffic load. This is illustrated for a response size of 306976 Bytes in Figure 10.25(b). Here ReSoA is able to improve the performance significantly because the RTT seen by end-to-end TCP increases with increasing wireless network delay. Hence end-to-end TCP takes more time than ReSoA to detect losses and longer to reach the network capacity after a loss has occurred.

If the wireless network bitrate is set to 384kb/s the time needed to transmit a packet will significantly influence the RTT. Hence, end-to-end TCP takes longer than ReSoA to seize the network and especially the RTO value is higher for end-to-end TCP. This means that end-to-end TCP takes longer than ReSoA to detect a loss event. The effects on the performance are illustrated in Figure 10.26 and Figure 10.27.

Figure 10.26(a) shows the throughput of ReSoA and TCP for two load levels as a function of the response size. Figure 10.26(b) shows the corresponding performance gain by ReSoA. In addition this figure shows the performance improvement as the background load is increased. In the case of a low wireless bitrate ReSoA is able to improve the throughput for all response sizes. For low background traffic the performance gain is more than 10% if the response size is smaller than 100 KBytes and between 5% and 10% otherwise. The ratio increases if the response size becomes larger than 500 KBytes since no losses occur for smaller response sizes. ReSoA is able to achieve the same throughput if the background traffic load is increased, while TCP's performance is significantly reduced. In the case of this load level we start observing retransmissions as soon as the response size reaches 100 KBytes. The performance improvement by ReSoA is more than 20% for all response sizes, although it decreases with increasing response size. For a response size of 100 KBytes the performance gain is 40%. For smaller response sizes the performance gain is determined by the wireless network delay and not by the background traffic. Therefore the gain is identical for low and medium loads. In the case of a high load ReSoA is able to improve the performance by around 80%.

Figure 10.27 shows the results as a function of the wireless network delay, whereas a response size of 53576 Bytes is used. The Figure shows that ReSoA is even able to improve the performance if the wireless network delay is smaller than 50ms.

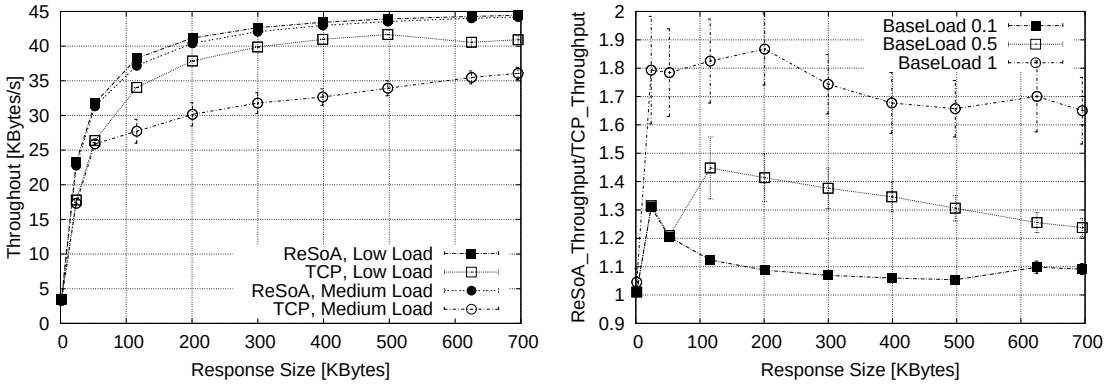
The results presented in this section are rather qualitative than quantitative because they depend on the configuration as well as on our background traffic model. It is difficult to show the effects of background traffic on ReSoA since it is difficult to model the Internet. However, the simulation gives a good qualitative impression on how the performance gain by ReSoA increases with background traffic. Background traffic has two effects on the connection under study. First, the RTT and especially its variance increases. Second, packets might get lost due to queue overflow. The increased RTT effects the throughput of both TCP and ReSoA because the RTT share of the Internet is increased. Packet losses are more harmful to end-to-end TCP, especially if the delay of the access network is also high. The RTT and consequently the retransmission timer value of end-to-end TCP depend on the sum of the wireless network delay and the Internet delay, whereas for ReSoA both values only depend on the Internet delay. Hence, if packets are lost, end-to-end TCP takes longer to recover.



(a) Throughout of ReSoA and TCP

(b) Ratio of ReSoA to TCP

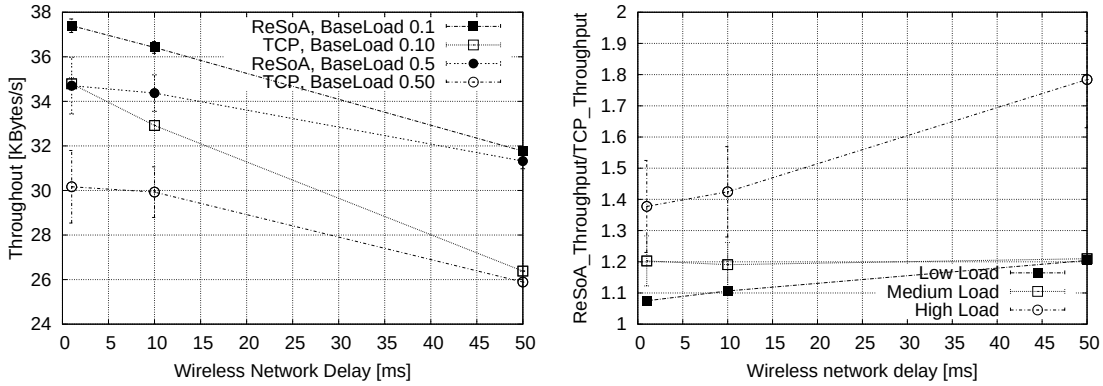
Figure 10.25.: Effects of background traffic on the performance of ReSoA and TCP as a function of wireless network delay. Wireless network bitrate 11Mb/s, Internet bitrate 2Mb, Internet delay 40ms



(a) Throughout of ReSoA and TCP

(b) Ratio of ReSoA to TCP

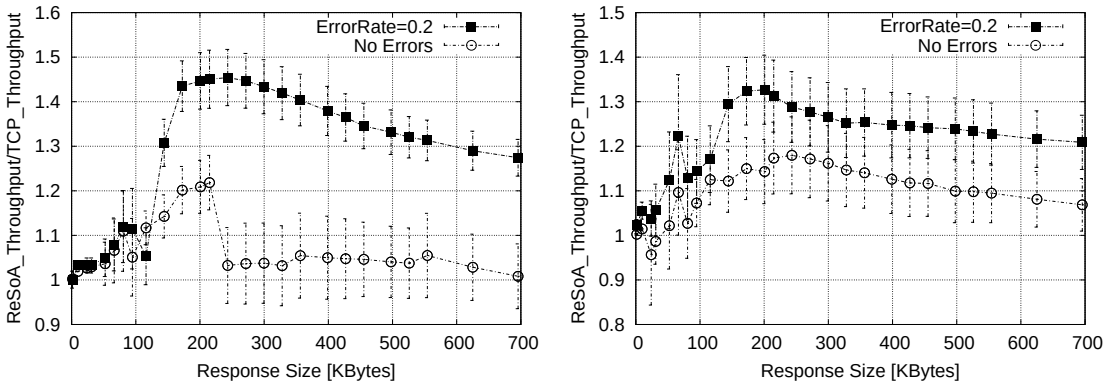
Figure 10.26.: Effects of background traffic on the performance of ReSoA and TCP as a function of the response size. Wireless network bitrate 384 kb/s, Wireless network delay 50ms, Internet bitrate 2 Mb, Internet delay 40 ms



(a) Throughout of ReSoA and TCP

(b) Ratio of ReSoA to TCP

Figure 10.27.: Effects of background traffic on the performance of ReSoA and TCP as a function of wireless network delay. Wireless network bitrate 384kb/s, Internet bitrate 2Mb, Internet delay 40ms



(a) Ratio in the case of low background traffic

(b) Ratio in the case of medium background traffic

Figure 10.28.: ReSoA vs. TCP in a configuration with background traffic and uniformly distributed errors

10.3.3. Results 3: Combining the previous two Scenarios

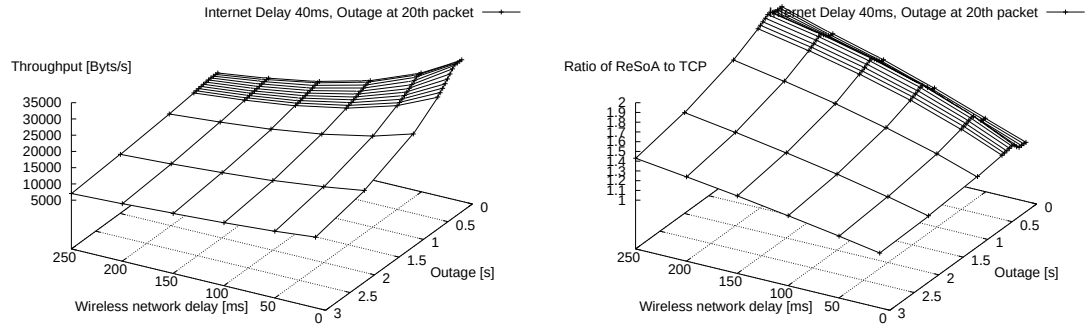
Basically the effects of this scenario can be deduced from the previous discussion. Therefore we limit the discussion to a single configuration. In order to show the effects of packet loss in the wireless network in combination with background traffic, we use the same configuration as for Figure 10.24, but set the packet error rate of the wireless network to 0.2. Figure 10.28 compares the ratio of ReSoA to TCP for the error free case with the ratio for an error rate of 0.2. The curves of the error free case are taken from Figure 10.24. Figure 10.28(a) shows the comparison for low background traffic, while Figure 10.28(a) is based on medium background traffic.

As can be seen from the figures, the performance gain by ReSoA is higher in the case of errors in the wireless network. However, as long as the transmission of the response does not suffer from packet drops caused by congestion, the performance gain is identical for both configurations. In section 10.3.1 we already saw that the link layer protocol is able to hide all losses completely from the TCP sender. If losses occur within the Internet, the error rate of the wireless network will play a role for end-to-end TCP. Losses in the wireless network are translated into delay by local retransmissions. Thus, end-to-end TCP sees an increased RTT. Hence, end-to-end TCP takes longer than ReSoA to detect a loss within the Internet and to recover from it.

10.3.4. Results 4: Outages on the Wireless Hop

In this section we discuss the simulation results for configurations where we stopped the communication over the wireless link for a certain time (called outage) at a certain segment number. Since no packets are lost this scenario in principle corresponds to our two state error model scenario. The differences are that we set the exact length (in packets) and the start of the outage and that we simulate a single outage per file transfer.

The simulation results show that the performance gain by ReSoA is nearly independent from the start point of the outage. Therefore we limit the discussion in this section to configurations where the outage



(a) TCP's throughput

(b) Ratio of ReSoA to TCP

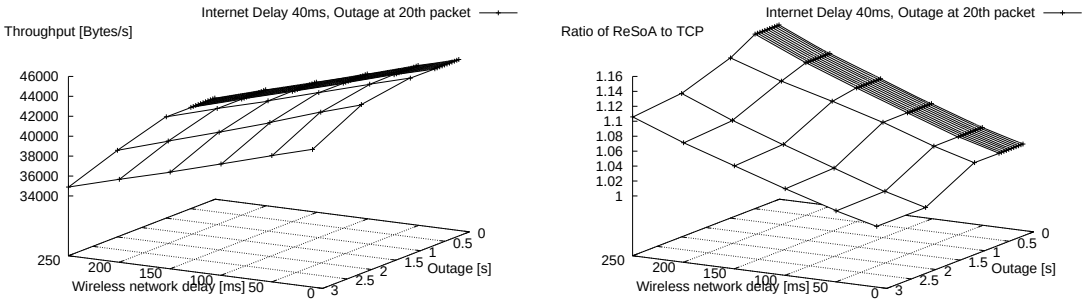
Figure 10.29.: ReSoA vs. TCP in the case of outages and a response size of 50680.

starts in the middle of the transmission of the response. In addition we only look at two different response sizes, namely 50680 Bytes and 709520 Bytes.

Figure 10.29(a) and Figure 10.30(a) show the performance of TCP as a function of the wireless network delay and the outage length for response size 50680 and 709520 respectively. Figure 10.29(b) and Figure 10.30 show the ratio of ReSoA's throughput to TCP's throughput.

As expected, the performance of TCP decreases with increasing outage length and increasing wireless network delay. The ratio increases with increasing wireless network delay (as we already know from section 10.3.4) but decreases with increasing outage length. ReSoA is able to improve the performance here because the outage increases the delay of the wireless network, and TCP suffers from spurious retransmissions with increasing outage length. The performance gain by ReSoA decreases with increasing outage length since the wireless network is idle during the outage for both approaches.

The performance gain is larger for small response sizes since, in the case of a long response, TCP has more time to catch-up with ReSoA at the beginning of a connection or after a timeout.



(a) TCP's throughput

(b) Ratio of ReSoA to TCP

Figure 10.30.: ReSoA vs. TCP in the case of outages and a response size of 709520 bytes.

10.4. Discussion

Our performance evaluation shows that in almost all cases a reliable link layer protocol is able to hide the error-prone nature of a radio link from a TCP sender. Spurious retransmissions occur rarely, despite the fact that packet losses in the wireless network are translated into delays by link layer retransmissions. In our simulations we could only observe spurious retransmissions if we simulated long error bursts or long outage periods. The conservative retransmission timer of TCP normally tolerates the delay variations caused by link layer retransmissions. Therefore the claim that TCP's performance degradation over wireless networks is caused by the reduction of the transmission rate caused by packet losses does not hold. A reliable link layer protocol is an appropriate solution for TCP over wireless networks. In this respeect our performance study is in line with current research results recommending the use of a reliable link layer to remedy TCP's performance problems over error prone links. However, our performance evaluation clearly shows that ReSoA is able to outperform the combination of TCP and a reliable link layer. It especially deserves mentioning that the performance increase by ReSoA does not depend on certain wireless network configurations. In fact the performance evaluation showed an increase for almost all investigated configurations. The reasons why ReSoA is able to additionally improve the performance gain already achieved by a reliable link layer are:

1. small protocol headers,
2. the LHP, which determines the performance, can be optimized for the underlying technology, and
3. the communication over the Internet is decoupled from the communication over the wireless network as far as possible without violating socket interface semantics.

Small protocol headers are only an issue for slow access networks. It also needs to be mentioned that the TCP/IP header may also be reduced to a few bytes by header compression. Therefore the last two reasons are mainly responsible for ReSoA's performance improvement.

The measurements presented in Chapter 9 are a good example for the second point. They document how ReSoA exploits the knowledge of the underlying communication technology. Since, in this case, the MAC layer already provides an error free channel, ReSoA minimizes the number of messages that are sent from the wireless end system to the access point. The performance at the 'Good-Position' is additionally improved by using the full frame size supported by IEEE 802.11b.

Our simulations show the potential of the third point because here we did not take advantage of an optimized protocol. We used the same link layer protocol for both ReSoA and end-to-end TCP. The simulation results show that decoupling of networks leads to a significant performance improvement. The analytical approach in Section 8.5.2 shows that ReSoA's performance gain depends on the response size and the bandwidth delay product of the wireless network. ReSoA improves the performance even for the error free case. If ReSoA is used the TCP sender at the fixed host only sees the Internet share of the RTT. Hence TCP is able to Slow Start faster and full utilization of the bottleneck link is achieved faster than in the end-to-end case. Generally the performance gain decreases with increasing response size and increases with increasing bandwidth delay product. The response size at which the performance gain starts to decrease depends on the bandwidth delay product of the wireless network. If the response size is large enough for TCP to reach a steady state (enabling TCP to fully utilize the bottleneck link), the end-to-end performance of TCP approaches that of ReSoA.

Packet losses in the wireless network (causing additional and varying delays due to link layer retransmissions) or Internet background traffic additionally increase the performance gain by ReSoA. Although TCP does not directly interact with the link layer error control, throughput is negatively effected by the additional delay. Link layer retransmissions increase the RTT seen by the TCP sender in the end-to-end case but not for ReSoA. This results in two disadvantages for end-to-end TCP. The first disadvantage is

that TCP's ACK Clock triggers the transmission of new packets at a reduced rate, causing slow start to seize the network capacity at a slower rate. The second disadvantage concerns packet losses in the Internet. For end-to-end TCP duplicate acknowledgments take longer to arrive at the TCP sender. They additionally have to travel across the wireless link. The value of the retransmission timer value in addition to the increased delay also includes the variation of the RTT across the wireless link. The deployment of ReSoA leads to a more timely retransmission of lost packets.

The comparison to TCP is only one evaluation aspect. An additional interesting question is how ReSoA's performance compares to the other approaches discussed in Section 3.3. This discussion is kept on a qualitative level for two reasons. First of all, most approaches are compared to TCP without hiding the error prone link by a reliable link layer protocol. Obviously this leads to a considerably increased performance gain compared to our investigation. The second reason is that a unified performance evaluation environment does not exist. Hence, a quantitative performance comparison is not possible, even though all approaches use the same reference. Since the list of alternative solutions is long we selected two distinguished approaches for the discussion, namely Snoop and I-TCP. Approaches that do not exploit local error control and are solely based on modifications of TCP are not considered. Although interesting for specific environments (e.g. small total RTT) they are not suited for a wide range of scenarios (e.g. high error rates).

Snoop is a link layer approach, albeit a smart one. Its main feature compared to classic link layer protocols is that Snoop does not use separate protocol messages and headers to implement the link layer protocol. Instead it uses TCP segments. Although this is optimal regarding protocol overhead, it limits the design options. For example, it is not possible to implement link layer fragmentation or packet combining. Snoop has shown good performance results in all performance comparisons (e.g. [20]). However, its potential is as limited, as that of our reference (TCP plus a separate link layer protocol). In the end, TCP determines the performance of any link layer approach. Snoop can neither utilize protocols optimized for a specific technology (e.g. reduction of control packets in the case of a half-duplex channel) nor does it hide the wireless network RTT share from the TCP sender. We would expect Snoop to offer a higher throughput than our reference case. However, the fundamental observations are not changed by Snoop with respect to the performance evaluation presented in this section. Snoop was not chosen as reference because it needs a TCP-aware protocol booster at the access point.

I-TCP is a split connection approach. It has all performance advantages inherent to split connection approaches that decouple the communication over wireless network and the Internet.

I-TCP and ReSoA provide the same throughput if both approaches use the same protocol mechanisms for the communication over the wireless link. In general the performance of approaches based on decoupling wireless network and Internet is determined by the protocols used for communication over the wireless network, and the protocol overhead needed to synchronize the two halves. There might be more overhead for ReSoA than for I-TCP because ReSoA must synchronize the two socket halves in order to maintain the semantics of the socket interface.

These findings are based on the metric we chose for our performance evaluation. However, it would be possible to choose a different metric for the performance evaluation that would favor I-TCP. The metric is based on a client (application) oriented view of the performance issue. If this were changed to a server oriented view, an interesting metric would be the time during which resources are allocated to a connection. Since ReSoA delays the acknowledgment of the last message until all data has reached the wireless end system, this time would definitely be longer for ReSoA than for I-TCP. This additional time is the price for preserving the semantics of the socket interface.

The performance evaluation not only indicates that ReSoA should be used for wireless or error-prone networks. ReSoA is also is a promising candidate for other access networks. For instance, Internet access via POTS or DSL could benefit from the fact that ReSoA does not suffer from Slow Start on the dedicated line of a local loop. TCP's Slow Start mechanism prevents the utilization of the available bandwidth at the

beginning of a connection. The characteristics of such access networks are similar to the assumptions that our analysis is based on (e.g. bidirectional links). Therefore a significant performance gain, especially in the case of small files (e.g. web pages), can be expected. However, this is is beyond the scope of this thesis.

Chapter 11.

Conclusions

Everything needs to be changed, therewith everything can stay as it is. The Internet has evolved from a few users network to a must for everyone, being an important economical factor. Although it is still the Internet with the core protocols as they were designed about two decades ago, many aspects had to be changed to support the increasing popularity of the Internet. Almost every new application or technology challenges the principles of the Internet. This is true for multimedia applications requiring quality of service support, as well as for applications that can operate on top of a best effort network like the World Wide Web. Although the design of the Internet protocols has been driven by the end-to-end argument, this principle is weakened in many ways today. The popularity of the World Wide Web increases the load, especially in the case of important events, on parts of the network or a single server providing the demanded information so that the Internet as it has been designed would collapse. As remedy proxies are used. If a user requests a web page from a certain web server, the domain name system query, which is used to translate the server address to an IP address, might answer the request differently in succeeding queries (e.g. for users at different geographical locations). Therefore the user never knows from which web server or proxy he is reading data.

Another example is the integration of NAT boxes into the Internet. A NAT gateway is used to hide a private network behind a single IP address and to counteract the shortage of IP addresses. For this purpose the IP addresses of a packet are modified, while the packet passes the NAT gateway. Often the modification of the IP address alone is not sufficient, as some applications encode their source address within the payload. Hence, so called application level gateways are used to parse and, if necessary, modify the payload of a packet.

The introduction of wireless Internet access networks created a new research field. It turned out that TCP is unable to deal satisfactorily with the error-prone nature of wireless links. The reason lies in TCP's basic throttling mechanism. TCP must reduce its transmission rate upon detection of packet losses in order to protect the Internet from a congestion collapse. Many different researchers have contributed to this question by either introducing new architectures for wireless Internet access, analyzing the effect of different error models on TCP performance, or by rethinking conjectures of the very beginning of this research field. One of these conjectures was that a reliable link layer protocol would adversely interact with TCP's error control mechanisms, leading to spurious retransmissions. Although this is true for specific configurations, it does not hold in general. Today the IETF recommends the use of a reliable link layer protocol to protect the communication of error prone links.

An alternative remedy for wireless Internet access network problems is the deployment of performance enhancing proxies. On one hand, proxies break the end-to-end principle of the Internet. On the other hand, proxies that could not be found in the Internet a couple of years ago are already almost omnipresent, albeit on higher network layers. Nevertheless the question whether performance enhancing proxies should be deployed in the Internet is still an open research area. The IETF's current opinion on this questions is that PEPs are evil but should be used if they provide a significant improvement.

This thesis contributes to the discussion about the suitability of proxies for Internet access and as a result advocates the deployment of our performance enhancing proxy ReSoA. Although local error control as

well as new technologies like OFDM are well suited to solve TCP's performance problems over wireless links, we show that ReSoA is able to improve performance beyond local error control.

The idea behind most PEPs is to split the end-to-end transport protocol connection into two halves. A tailored protocol is used for the communication over the wireless network, while TCP is used for the communication over the Internet. Although the decoupling of access network and Internet (backbone network) is also the core idea behind ReSoA, we started from the application point of view.

Network applications access the service of the Internet protocol stack through a defined API. The protocol providing the service is unimportant. The interface is what matters. The interface implementation may be changed as long as its syntax and semantics are preserved. The BSD socket interface is widely deployed. ReSoA uses a split implementation of this interface. Socket calls are encapsulated at the end system and conveyed to the ReSoA-server where they are executed on behalf of the client. We designed a two-layered protocol stack for the communication between ReSoA-client and ReSoA-server. The upper layer is responsible for the encapsulation of socket calls, as well as for maintaining the semantics, while the lower layer is responsible for providing a reliable communication service. The upper layer is technology independent, while the lower layer is not. ReSoA only specifies the service it expects from the lower layer but not the protocol providing this service. The lower layer protocol is not part of the ReSoA specification since different technologies might require different protocol mechanisms in order to achieve a good performance.

For the design of ReSoA we analyzed every socket call thoroughly. In order to show that ReSoA is equivalent to a local implementation of the socket interface, we tested existing network applications on top of ReSoA and developed an SDL-specification of ReSoA and the BSD socket interface. This specification allowed use to investigate interesting cases like connection termination, independent of a specific implementation. All performed tests showed that ReSoA can transparently replace the local socket interface implementation. Whatsoever, ReSoA breaks with the fate sharing principle of the Internet. With ReSoA, TCP connections survive a crash of the end system until the ReSoA-server detects the crash. Although during this time the ReSoA-server acknowledges incoming TCP segments on behalf of the crashed client, this does not violate the interface semantics. In the case of a local protocol stack the client could crash after the local TCP instance has acknowledged data but before the application has consumed this data. Hence, even in this case transport layer acknowledgments are not sufficient to implement a reliable end-to-end service. Next, ReSoA does not acknowledge the reception of the last packet until it has delivered all data to the ReSoA-client successfully. Hence, the remote end system will be informed under any circumstances if the ReSoA-client crashes. Another problem is that the ReSoA-server can crash, resulting in a lost state of all TCP connections while the application survives. To combat this problem, redundancy should be added to the system. This is currently investigated in an ongoing diploma thesis.

The performance evaluation presented in this work is based on measurements as well as simulations. The measurements have shown that ReSoA is able to improve the performance in a wireless LAN context substantially. The performance improvement at a good position is increased by more than 20% if the response size is larger than 5 KBytes. For of a bad position ReSoA can double the throughput for response sizes larger than 30 Kbytes. It should be mentioned that end-to-end TCP does not suffer from (spurious) retransmissions under these circumstances.

The systematic performance evaluation has shown that ReSoA is able to improve Internet access performance for different configurations. We will not present quantitative figures about the performance gain here, as those are very configuration specific. However, qualitative results can be deduced. The simulations have shown that the performance gain due to ReSoA mainly depends on the latency of the access network. If the delay of the access network as well as the capacity of the bottleneck link are high, ReSoA improves the performance substantially. Increased delay or delay variance improve the performance gain achieved by ReSoA. For instance, although a reliable link layer protocol is able to protect the TCP sender from the error prone nature of the wireless link, and even if no spurious retransmission occurs, the additional

delay of the wireless link adversely effects TCP's performance. This is especially true if packets are lost in the Internet due to congestion. In this case the RTT estimation of end-to-end TCP includes the additional delay, and hence end-to-end TCP takes longer to recover from the losses than ReSoA. The performance improvement achieved by ReSoA is particularly significant for small to medium response sizes. Since web pages usually belong to this group, ReSoA should be able to improve WWW access performance significantly.

Since the performance improvement achieved by ReSoA manly depends on the delay of the access network, we expect that ReSoA is not only suitable for wireless Internet access but also for other access technologies like telephone lines (POTS) or DSL. For dedicated lines TCP's Slow Start mechanism is not needed in the access network.

Besides performance improvement ReSoA has additional appealing features. The end system and the access network can be operated without IP. Hence, the end-user does not need to worry about TCP/IP protocol stack configuration (e.g. enable ECN, window scaling, ...). Instead he only uses the LHP of the current Internet provider. The Internet provider is responsible for providing a fine tuned TCP/IP protocol stack. Additionally new TCP mechanisms or enhancements may be distributed faster, as they would not have to be installed on the end system. ReSoA is also able to overcome the IP address shortage without some of the problems inherent to NAT boxes. With ReSoA the TCP/IP protocol stack is located at the ReSoA-server. The application at the ReSoA-client uses this protocol stack as if it were local. Hence it sees the IP address used for communication with the distant host. Therefore this address is used by the application layer protocol to encode its source address into the payload.

11.1. Outlook

Unfortunately this work cannot cover all aspects that should be included in a proxy based Internet access architecture. The performance evaluation of ReSoA presented in this thesis has shown that the deployment of ReSoA is beneficial for many different access network configurations. Therefore we provide an overview about which aspects should be explored in the near future.

Our work focuses on TCP, since TCP is the dominating protocol of the Internet and offers a more complex service than UDP. However, UDP support should be added to ReSoA in order to facilitate an IP-free end system. The integration of UDP into ReSoA is rather an implementation aspect than a design issue. The most interesting point about the UDP case is that many UDP applications would not benefit from our fully reliable last hop protocol because they often are delay sensitive. Hence, if UDP support should be added to ReSoA, it will have to support different classes of LHPs.

Additionally we could distinguish between legacy applications and ReSoA-aware applications. ReSoA-aware applications could benefit from a modified socket interface which, for instance, could allow the application to pass QoS information to ReSoA (e.g. select an LHP). Security also is a crucial aspect that needs to be added to ReSoA. This includes encryption of the last hop protocol connection as well as authentication and authorization. Redundancy is needed to reduce the probability of an abrupt connection termination due to a crashed ReSoA-server.

We only looked at wireless communication without considering mobility. Mobility, however, is an important issue that definitely needs to be supported by ReSoA. Basically there are two possibilities to manage mobility. The first possibility would be to locate the ReSoA-server close to the wireless end system (e.g. at the access point). This would require the transfer of the context associated with each client (e.g. protocol control block) to the next ReSoA-server if the client moves. One ReSoA-server would only have to handle a small number of clients. The second possibility would be to locate the ReSoA-server at an edge router deeper in the network. In this case a single ReSoA-server could manage multiple access points (cells) and it would not be necessary to transfer client contents. The downside of this variation is the possibility of scalability problems, since such an ReSoA-server would be responsible for a large number of end

systems. In addition, the design of the LHP would become more complex, as it would have to protect the communication across multiple links instead of a single link. Therefore the integration of mobility support into ReSoA requires a thorough investigation.

Besides the aforementioned aspects that are crucial for ReSoA to become a real alternative for Internet access, there are additional interesting aspects to be considered. Currently ReSoA sends an acknowledgment as soon as it has received a TCP segment. Although this is important at the beginning of a new connection for performance reasons, this could lead to exhaustive buffer allocation at the ReSoA-server if backbone bitrate and access network bitrate differ too much. In order to optimize buffer allocation at the ReSoA-server, acknowledgment pacing algorithms should be investigated. If the ReSoA-server is already buffering several packets for an ReSoA-client, it should gradually start delaying acknowledgments in order to adapt the sending rate of the fixed host to the bitrate of the access network (without triggering retransmissions).

Different design goals should also be investigated. Currently ReSoA acknowledges every segment except for the last one before they are forwarded to the end system. This is necessary in order to improve performance. If however performance is not the issue, acknowledgments can be delayed until the client has acknowledged the reception of the data. This configuration is imaginable if ReSoA is used as a NAT-replacement. Also ReSoA could be altered to support asynchronous operation. In this case the application sends a request and then enters power saving mode. The ReSoA-server collects response data on behalf of the client without immediately forwarding the data to the client (as it is currently the case). At some point the client wakes up and fetches all data in a single burst.

Appendix A.

Acronyms

API	Application Programming Interface
ARP	Address Resolution Protocol
ARQ	Automatic Repeat Request
BPSK	biphase shift keying
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
ECN	Explicit Congestion Notification
ELN	Explicit Loss Notification
EP	Export Protocol
FEC	Forward Error Correction
FTP	File Transfer Protocol
GSM	Global System for Mobility
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISM	Industrial Scientific and Medical band
LHP	Last Hop Protocol
LLP	Link Layer Protocol
LSM	Local Socket Module
MAC	Medium Access Control
MSC	Message Sequence Chart
MSR	Mobile Support Router
MSS	Maximum Segment Size
MTU	Maximum Transfer Unit
NAT	Network Address Translation
NIS	Network Information Service
NS	Network Simulator
PDU	Protocol Data Unit
PEP	Performance Enhancing Proxy
PILC	Performance Implications of Link Characteristics
ReSoA	Remote Socket Architecture
RPC	Remote Procedure Call
RSC	Remote Socket Client
RSM	Remote Socket Module
RSS	Remote Socket Server
RTO	Retransmission Timeout
RTT	Round Trip Time
SDL	System Description Language
SNR	Signal to Noise Ratio
SRTT	Smoothed Round Trip Time
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Appendix B.

Example MSC

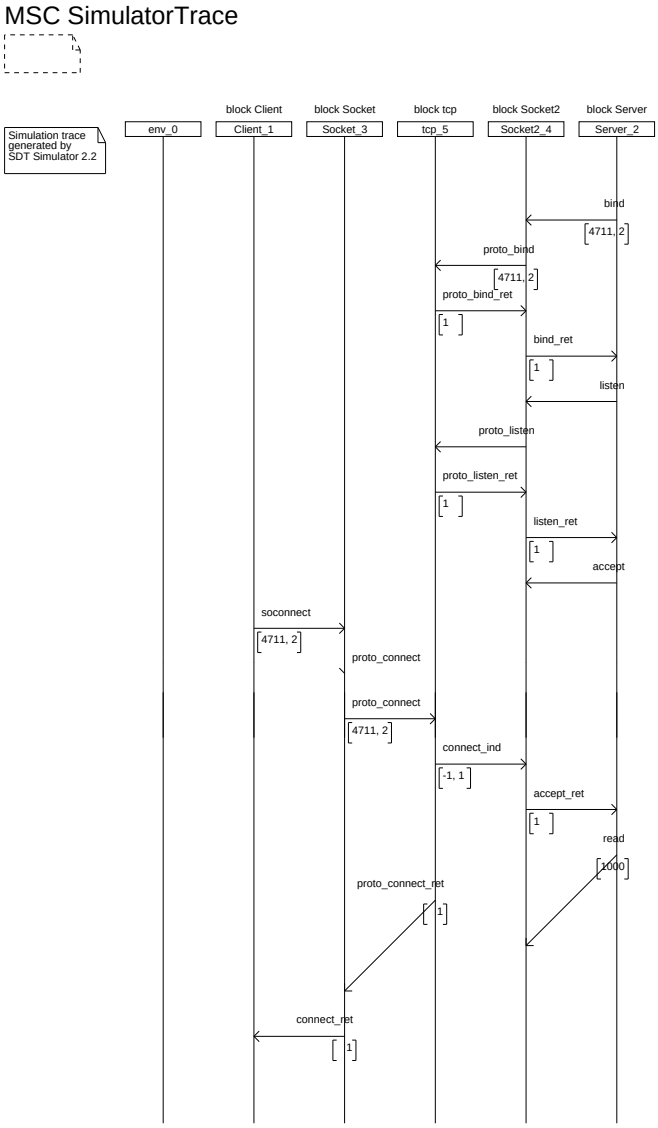


Figure B.1.: MSC: Connection establishment using BSD socket

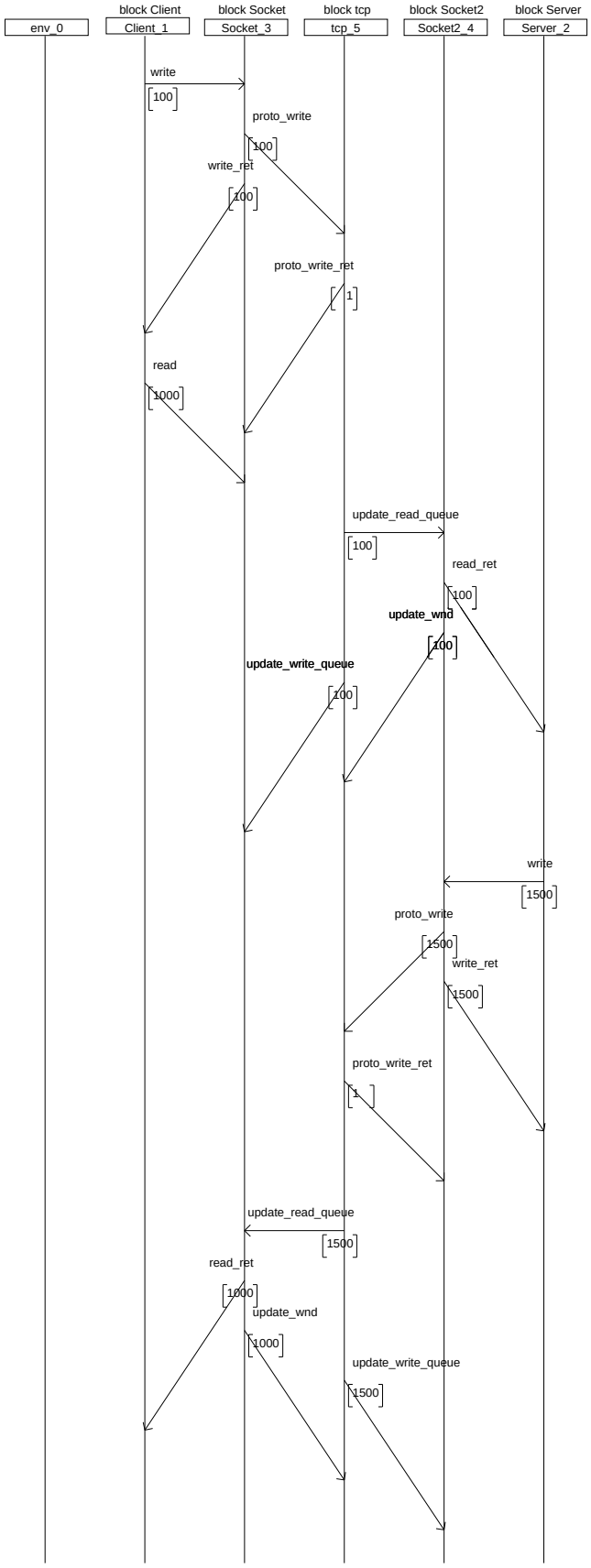


Figure B.2.: MSC: Data transfer (I/II)

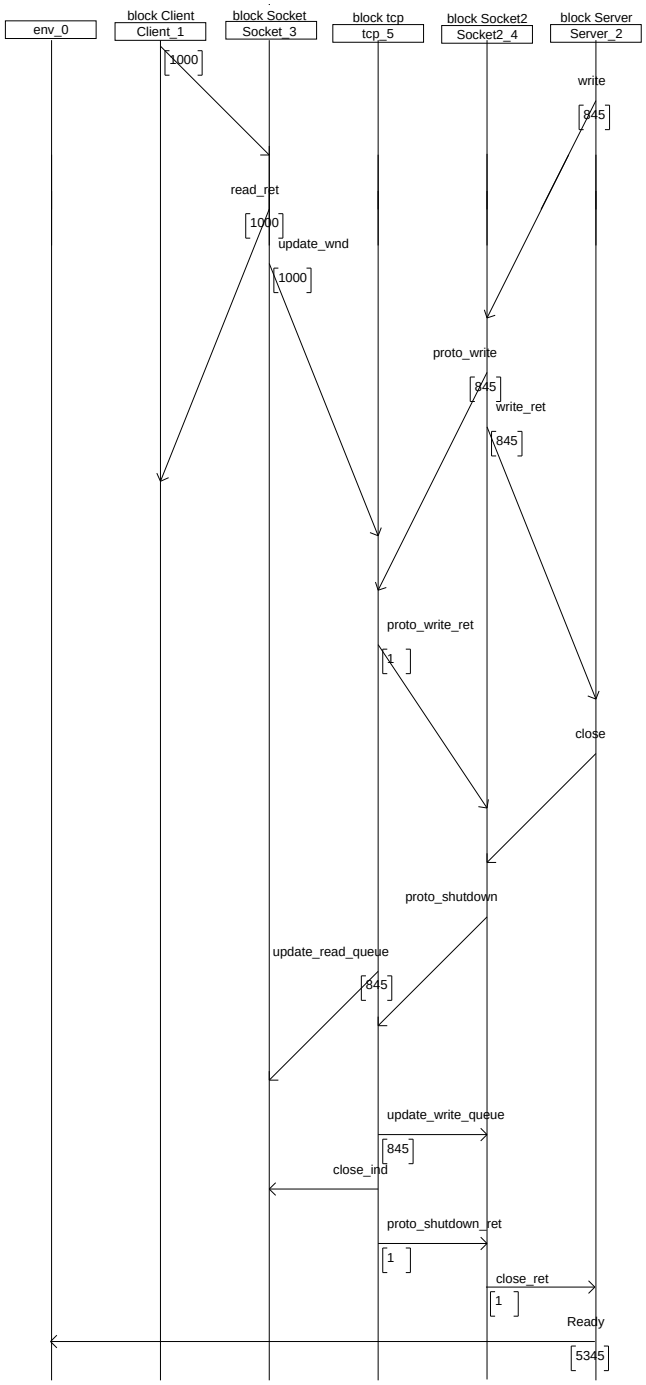


Figure B.3.: MSC: Data transfer (II/II)

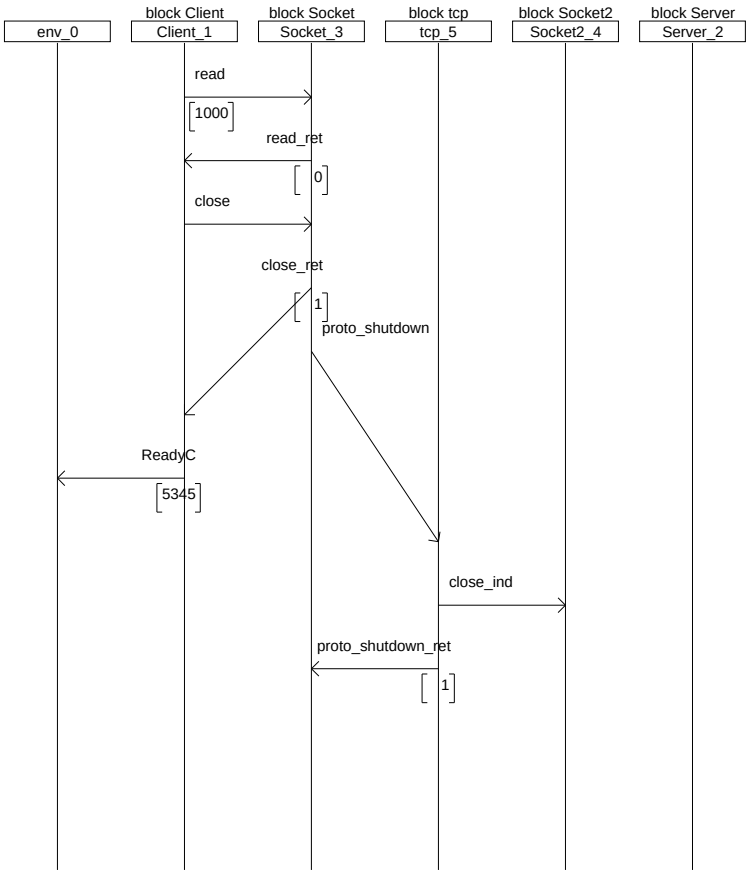


Figure B.4.: MSC: Connection termination

Appendix C.

Interface between EP and LHP

C.1. Interface between EP and LHP

Figure C.1 through Figure C.4 show the available service primitives. Besides the classical primitives for connection management and data communication some special primitives are defined to collect statistics or to implement an interface flow control. An important property of the interface is that no message may get lost, since the service user does not implement an own error control. To achieve this, an interface flow control is needed.

Some requests need a confirmation as indicated in Figure C.1 through Figure C.4. This is described in more detail below.

The different primitives can be classified into four categories, namely communication endpoint management, connection-management, data transfer and miscellaneous.

Registration

LHP_REGISTER

The `LHP_REGISTER` primitive is used to create a new local communication endpoint by a service user. This primitive should not trigger any data exchange between the peer hosts. If the primitive succeeds, a connection identifier is returned using the `LHP_REGISTER_OK`-primitive. This identifier must be used for all later requests belonging to this communication endpoint. If the communication endpoint could not be created, an error reason is returned using the `LHP_REGISTER_FAILED`. With the `LHP_REGISTER` primitive the service user must specify which service (e.g. reliable,semi-, unreliable) it requests and which provider is asked to provide the service[1]. If either the service or the provider are unknown, the creation of the new endpoint fails.

[1] Currently, only a reliable service is supported.

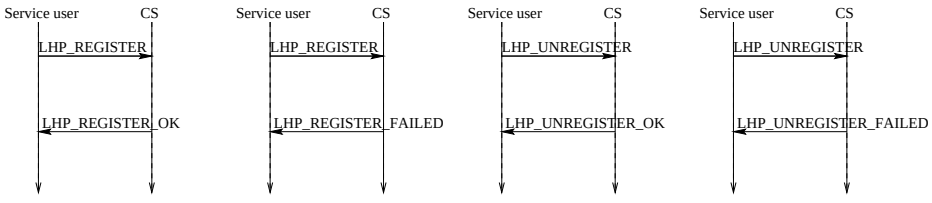


Figure C.1.: LHP service primitives for registering

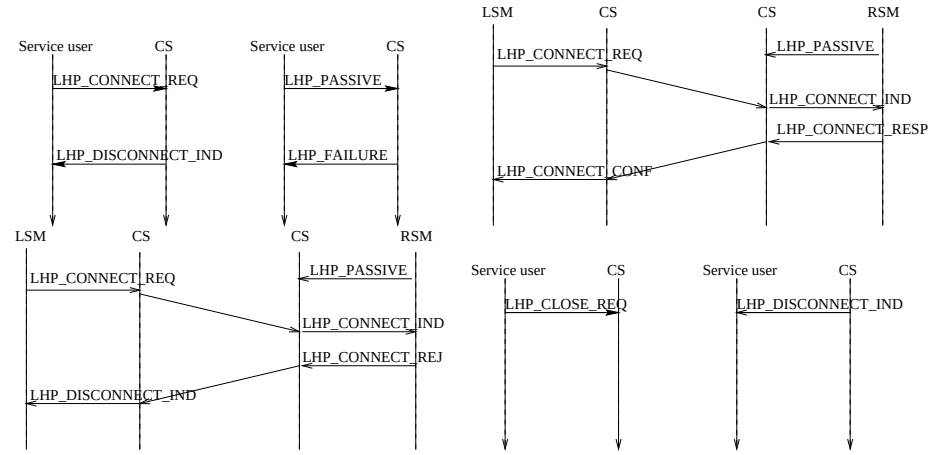


Figure C.2.: LHP service primitives for connection management

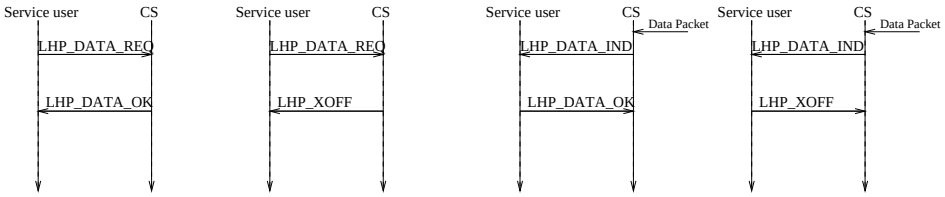


Figure C.3.: LHP service primitives for data exchange

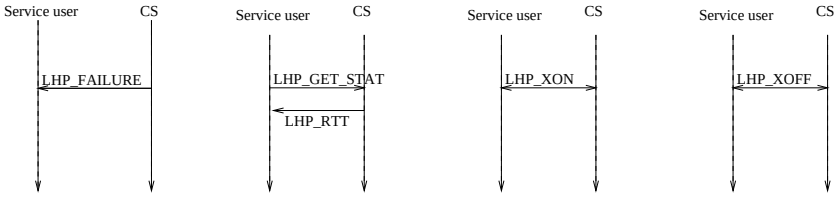


Figure C.4.: LHP service primitives - miscellaneous

LHP_UNREGISTER

The LHP_UNREGISTER primitive is used to release a communication endpoint. If the communication endpoint is valid, the service provider confirms the request using the LHP_UNREGISTER_OK primitive, otherwise the service user answers with the LHP_UNREGISTER_FAILED primitive.

Connection Management

Connection Establishment

A connection is either actively or passively established. The active end initiates the establishment of a new connection using the LHP_CONNECT_REQ-primitive while the passive end (the ReSoA-server is waiting for incoming connections) of a connection uses the LHP_PASSIVE-primitive to inform the service provider that it should accept incoming connection requests.

When an LHP-entity receives a connection request (initial packet of a new connection) and if it is set to accept incoming connections, it informs its service user about the new connection using the LHP_CONNECT_IND-primitive. If the LHP was not configured to accept new connections, it must deal with the request locally, without notifying the service user (e.g. send a reset packet).

After the passive end of an LHP connection has received a connection request from its peer entity, it must create a new communication endpoint for this new connection. The old communication endpoint must remain ready to process further incoming connection requests.

The service user (at the passive end) can either accept a new connection using the LHP_CONNECT_RESP primitive or reject it using the LHP_CONNECT_REJ primitive. In the former case the LHP sends a connection confirmation to its peer entity. In the latter case it sends a disconnect indication. The initiator of an LHP connection is informed with LHP_CONNECT_CONF-primitive when the connection establishment is completed. If the LHP could not fulfill a connect request, it informs its service user using the LHP_DISCONNECT_IND-primitive. This can either mean that the peer LHP entity has rejected the connection (the local LHP has received something like a reset packet) or that the initiating LHP has received no response at all after a number of retransmissions.

Connection Release

To release a connection, three primitives are available. The LHP_CLOSE_REQ-primitive is used to initiate a connection release, while the LHP_DISCONNECT_IND primitive is used to inform the service user about the fact that an LHP connection was terminated. Normally only the ReSoA-client should use this primitive.

The LHP_RESET-primitive should be used by the service user to trigger an abrupt connection release. Upon this request the LHP should send a reset packet rather than starting a normal connection termination sequence.

Data Transfer

LHP_DATA_REQUEST

The LHP_DATA_REQUEST primitive is used to request the transparent delivery of a LHP_SDU. The SDU can have an arbitrary length of up to 65600 octets.

The LHP responds to a data request with one of the following primitives

- LHP_DATA_OK: This response indicates success. A positive result however neither means that the message was delivered to the peer nor that the request was consumed by the peer service user but

only that the request was accepted. The service user is not informed if the message was successfully delivered.

- LHP_XOFF: This response tells the service user that the LHP currently is unable to accept further requests. However, this is only temporary. The LHP informs the service user if it is able to process further requests using the LHP_XON primitive.

LHP_DATA_INDICATION

The LHP_DATA_INDICATION primitive is used to deliver messages to the service user.

All messages for which the service user on the peer host has requested reliable delivery must be delivered in the same order as they were sent. It is the task of the LHP to preserve the message boundaries. A message must not be removed from the receive buffer before it was successfully delivered to the service user. Thus no messages may be lost at the interface.

The LHP_DATA_INDICATION primitive is acknowledged with one of the following primitives.

- LHP_DATA_OK to indicate success.
- LHP_XOFF to signal that the service user is currently unable to accept the message. In this case the service user will later trigger the service provider to deliver all queued messages.

Miscellaneous

LHP_GET_STATS

With this primitive the service user can request some of the statistics of the underlying communication system, like the RTT. Such information might be required to set private timers or to learn about the state of the lhp. Currently only a single return value is specified, namely the measured RTT of a SDU.

LHP_XON

This service primitive can be used by both the service user and the service provider. It signals that originator of this signal is ready to accept further data.

LHP_XOFF

This service primitive is used to stop the data exchange via the local interface until new resources become available.

LHP_FAILURE_IND

If the communication subsystem detects an error situation, it must inform its service users. Triggering errors are for example that a message could not be delivered to the peer (after a certain number of attempts), that the keep-alive function signals communication problems, or if the peer resets a connection.

The indication should inform the service user about the error reason and the connection on which the error occurred. This is especially important to the ReSoA-server if the communication to a single ReSoA-client is impossible but all other ReSoA-clients are still reachable.

Appendix D.

Implementation Details of Simulation Model

D.1. Implementation of the Error Model

As described in section 10.2.4 on page 150, we decided to use two different kinds of error models. To simulate uniformly distributed errors we used an error model provided by the NS environment. The two state error model was implemented by ourself, since the NS multi state error model has some bugs as described by Abouzeid et al. in [3] and the error model contributed by Abouzeid does not permit bit error rates other than zero and one for the good and the bad state respectively. Aspects of implementing simulation models for the different approaches are discussed in [58].

Both error models consider a packet erronemous using equation (D.1). In this equation X is a uniformly distributed random variable between zero and one, p_e is the bit error rate and l is the total packet length in bits. In the case of the two state error model p_e is chosen according to the bit error probability of the current state.

$$X < (1 - (1 - p_e)^l) \rightarrow \text{Packetloss} \tag{D.1}$$

The sojourn time distribution of the two state error model is a parameter of our two state error model. If not stated otherwise, we used a exponential distributed random variable. The sojourn time mean is expressed in bytes instead of time. The advantage of this metric is that the parameters of the error model are independent of the simulated bitrate.

In order to simulate an error-prone link in NS, an error model must be attached to the NS link object. This operation is independent from the error model as long as the error model implements the correct interface. However, a common pitfall is that the error model is attached at the wrong place. NS provides functionality to attach the error either at the head of the object actually simulating the link (bitrate and delay) or after this object. If the first position is chosen, the packets are dropped before the actually consumed resources. This means that you can deliver 2 Mbit/s over a 2 Mbit/s link, although 50% of the packets are dropped by the error model. Thus, our simulation uses the second position (see Figure D.2).

To test the uniform error model provided by NS as well as our two state error model, we performed simulation using a two node scenario. In this scenario one node sends a constant bit rate packet stream exactly utilizing the available bandwidth to the second node. The link, which connects the two nodes, was equipped with one of the two error models. We observed the total number of packets injected into the link and the number of dropped packets. This scenario was repeated for different packet sizes and bitrates. In all simulations the observed error model behavior corresponds to the expected behavior. Figure D.1 shows an example plot for uniformly distributed errors and table D.1 summarizes some of the results of the two state error model.

D.1.1. Integration of our Link Layer Protocol into NS

Figure D.2 shows how the LLP was integrated into NS. Normally protocol implementations in NS are subclasses of the `Agent` class and are attached to a node. However, in the case of the LLP we decided to

Simulation Input		**Simulation Output**		
$\overline{T}_{\text{Good}}$	$\overline{T}_{\text{Bad}}$	$\overline{T}_{\text{Good}}$	$\overline{T}_{\text{Bad}}$	P_{PktErr}
100	100	[100.0,100.2]	[99.9,100.1]	[1.00,1.00]
100	1000	[100.0,100.4]	[999.6,1003.1]	[1.00,1.00]
100	10000	[99.7,101.1]	[9948.5,10130.7]	[1.00,1.00]
100	100000	[98.3,101.6]	[97195.4,101243.2]	[1.00,1.00]
100	1000000	[95.7,115.6]	[927812.9,1119092.7]	[1.00,1.00]
1000	100	[997.0,1002.1]	[99.8,100.5]	[0.66,0.67]
1000	1000	[996.6,1003.4]	[997.0,1004.6]	[0.81,0.82]
1000	10000	[1001.1,1016.3]	[9936.2,10073.4]	[0.97,0.97]
1000	100000	[974.5,1024.9]	[97850.6,101243.2]	[1.00,1.00]
1000	1000000	[941.4,1079.4]	[882825.2,1021267.4]	[1.00,1.00]
10000	100	[9909.0,10078.6]	[99.3,100.6]	[0.10,0.11]
10000	1000	[9920.6,10083.4]	[991.5,1004.2]	[0.18,0.18]
10000	10000	[9955.9,10151.8]	[9928.4,10118.0]	[0.54,0.55]
10000	100000	[9912.3,10532.4]	[97887.4,105384.8]	[0.91,0.92]
10000	1000000	[9568.6,11267.5]	[980477.3,1123484.2]	[0.99,0.99]
100000	100	[98542.0,102541.8]	[95.8,100.4]	[0.01,0.01]
100000	1000	[97851.1,102541.7]	[955.6,999.1]	[0.02,0.02]
100000	10000	[95668.7,101295.2]	[9746.5,10085.1]	[0.10,0.10]
100000	100000	[97191.6,105330.6]	[95852.7,103479.1]	[0.49,0.51]
100000	1000000	[90309.5,106860.1]	[993399.4,1113562.1]	[0.91,0.92]
1000000	100	[968003.9,1059987.0]	[92.5,109.5]	[0.00,0.00]
1000000	1000	[936402.2,1094342.0]	[926.6,1059.8]	[0.00,0.00]
1000000	10000	[961300.9,1146307.0]	[8896.6,10250.2]	[0.01,0.01]
1000000	100000	[984242.7,1094871.0]	[90897.6,105108.8]	[0.08,0.10]
1000000	1000000	[982476.3,1146061.5]	[893848.1,1024831.2]	[0.44,0.49]

Table D.1.: Test of two state error model. The bitrate was 1 Mb/s and the packet size was 1000 Bytes. The bit error probabilities in good and bad state were 0 and 1 respectively. The column $\overline{P}_{\text{PacketError}}$ shows the expected packet loss probability. For the simulation output we show the 95% confidence interval.

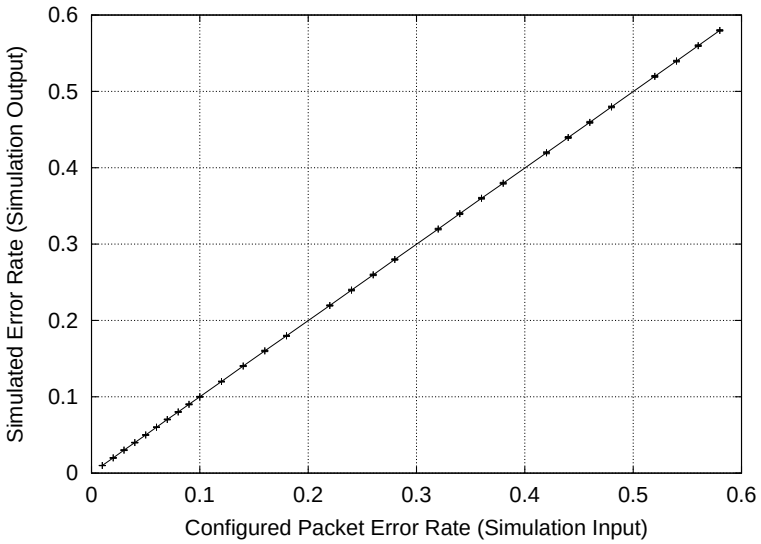


Figure D.1.: Test of the uniform error model. The bitrate was set to 1 Mb/s and the packet size was 1000 Bytes. The simulation time was 500 seconds. The plot shows the mean observed packet error rate and a 95% confidence level.

follow a different approach. In NS the IP level (outgoing) queue is part of the link connecting two nodes. If we had implemented the link layer protocol as a standard NS agent, the LLP would be a head of the IP queue.

To overcome this problem, the LLP was integrated into the NS link object. The LLP object is derived from the `Connector` class.

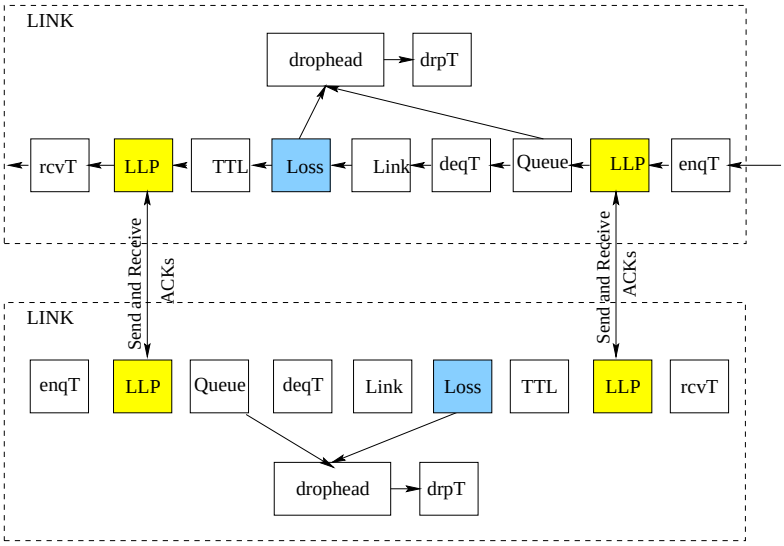


Figure D.2.: LLP integration into NS

Appendix E.

Output of `strace`

E.1. Using TCP

```
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 4
connect(4, {sin_family=AF_INET, sin_port=htons(12865), sin_addr=inet_addr("10.12.12.2")}}, 16) = 0
send(4, "..."..., 256, 0) = 256
select(1024, [4], NULL, NULL, {60, 0})  = 1 (in [4], left {59, 990000})
recv(4, "..."..., 256, 0) = 256
...
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
write(1, "TCP STREAM TEST to 10.12.12.2\n", 30) = 30
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 5
getsockopt(5, SOL_SOCKET, SO_SNDBUF, [16384], [4]) = 0
getsockopt(5, SOL_SOCKET, SO_RCVBUF, [87380], [4]) = 0
...
send(4, "\0\0\0\n\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\10\0\0\0"..., 256, 0) = 256
select(1024, [4], NULL, NULL, {60, 0})  = 1 (in [4], left {60, 0})
recv(4, "\0\0\0\v\0\0\0\0\0\1UT\0\1UT\0\0\0\0\0\0\0\0\0\0\0\0"..., 256, 0) = 256
connect(5, {sin_family=AF_INET, sin_port=htons(32820), sin_addr=inet_addr("10.12.12.2")}}, 16) = 0
gettimeofday({1053693857, 467237}, {4294967176, 0}) = 0
send(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4000, 0) = 4000
send(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4000, 0) = 4000
send(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4000, 0) = 4000
shutdown(5, 1 /* send */)               = 0
recv(5, "", 4000, 0)                    = 0
gettimeofday({1053693857, 484676}, {4294967176, 0}) = 0
close(5)                                = 0
select(1024, [4], NULL, NULL, {60, 0})  = 1 (in [4], left {59, 980000})
recv(4, "\v\0\0\0\0\0\0\0@\307p\0\0\0\0\0\0\0\t<\21153\0\0\0\0"..., 256, 0) = 256
write(1, "Recv   Send    Send             "..., 138) = 138
write(1, "bytes  bytes   bytes    secs.    "..., 49) = 49
write(1, " 87380  16384   4000    0.02     "..., 44) = 44
shutdown(4, 1 /* send */)               = 0
select(1024, [4], NULL, NULL, {60, 0})  = 1 (in [4], left {60, 0})
recv(4, "", 256, 0)                     = 0
_exit(0)                                = ?
```

E.2. Using ReSoA

```
socket(0x1b /* PF_??? */, SOCK_STREAM, 0) = 4
connect(4, {sin_family=AF_INET, sin_port=htons(12865), sin_addr=inet_addr("10.12.12.2")}}, 16) = 0
send(4, "\0\0\0\2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 256, 0) = 256
select(1024, [4], NULL, NULL, {60, 0})  = 1 (in [4], left {59, 990000})
recv(4, "\0\0\0\3\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 256, 0) = 256
...
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
write(1, "TCP STREAM TEST to 10.12.12.2\n", 30) = 30
```

```
socket(0x1b /* PF_??? */, SOCK_STREAM, 0) = 5
getsockopt(5, SOL_SOCKET, SO_SNDBUF, [65535], [4]) = 0
getsockopt(5, SOL_SOCKET, SO_RCVBUF, [65535], [4]) = 0
...
send(4, "\0\0\0\n\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\10\0\0\0"..., 256, 0) = 256
select(1024, [4], NULL, NULL, {60, 0})  = 1 (in [4], left {60, 0})
recv(4, "\0\0\0\v\0\0\0\0\0\1UT\0\1UT\0\0\0\0\0\0\0\0\0\0\0\0"..., 256, 0) = 256
connect(5, {sin_family=AF_INET, sin_port=htons(32821), sin_addr=inet_addr("10.12.12.2")}}, 16) = 0
gettimeofday({1053693877, 878024}, {4294967176, 0}) = 0
send(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4000, 0) = 4000
send(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4000, 0) = 4000
send(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4000, 0) = 4000
shutdown(5, 1 /* send */)               = 0
recv(5, "", 4000, 0)                    = 0
gettimeofday({1053693877, 894823}, {4294967176, 0}) = 0
close(5)                                = 0
select(1024, [4], NULL, NULL, {60, 0})  = 1 (in [4], left {59, 990000})
recv(4, "\v\0\0\0\0\0\0\0@\307p\0\0\0\0\0\0\0\t<\204}=\0\0\0\0"..., 256, 0) = 256
write(1, "Recv   Send    Send             "..., 138) = 138
write(1, "bytes  bytes   bytes    secs.    "..., 49) = 49
write(1, " 87380  65535   4000    0.02     "..., 44) = 44
shutdown(4, 1 /* send */)               = 0
select(1024, [4], NULL, NULL, {60, 0})  = 1 (in [4], left {60, 0})
recv(4, "", 256, 0)                     = 0
_exit(0)                                = ?
```

Bibliography

[1] Sdl design tool (sdt) v3.2. http://www.telelogic.se.

[2] A.A. Abouzeid, M Azizoglu, and S. Roy. Stochastic modeling of a single tcp/ip session over a random loss channel. In *Proc. of DIMACS Workshop on Mobile Networking and Computing*, 1999.

[3] Alhussein A. Abouzeid, Sumit Roy, and Murat Azizoglu. Stochastic modeling of TCP over lossy links. In *INFOCOM (3)*, pages 1724–1733, 2000.

[4] T. Alanko, M. Kojo, K. Laamanen, K. Raatikainen, and M. Tienari. Mobile computing based on gsm: The mowgli approach. In *Proc. IFIP World Conf. Mobile Communication, Canberra, Australia*, pages 151–158, September 1996.

[5] M. Allman, H. Balakrishnan, and S. Floyd. RFC 3042: Enhancing tcp's loss recovery using limited transmit, 2001. January.

[6] M. Allman and A. Falk. On the effective evaluation of tcp. *ACM Computer Communication Review*, 5(29), October 1999.

[7] M. Allman, S. Floyd, and C. Partridge. RFC 2414: Increasing TCP's initial window, September 1998. Status: EXPERIMENTAL.

[8] M. Allman and V Paxson. On estimating end-to-end network path properties. In *SIGCOMM 99*, August 1999.

[9] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control, April 1999. Obsoletes RFC 2001.

[10] A.L. Ananda, B.H. Tay, and E.K. Koh. A survey of asynchronous remote procedure calls. *sigops*, 26(2):92–109, April 1992.

[11] E Ayanoglu, Paul. S., T.F. LaPorta, Sabnani K.K., and R.D. Gitlin. Airmail: A link-layer protocol for wireless networks. *Wireless Networks*, 1:47–60, 1995.

[12] B. Badrinath, A. Acharya, and T. Imielinski. Impact of mobility on distributed computations. *ACM Operating Systems Review*, 27(2), April 1993.

[13] Ajay Bakre and B. R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *15th International Conference on Distributed Computing Systems*, May 1995.

[14] Ajay V. Bakre. *Design and Implementation of Indirect Protocols for Mobile Wireless Environments*. PhD thesis, Graduate School—New Brunswick Rutgers, The State University of New Jersey, 1996.

[15] B. Bakshi, P. Krishna, D.K. Pradhan, and N.H. Vaidya. Improving performance of tcp over wireless networks. In *17th Intl. Conf. on Distributed Computing Systems (ICDCS)*, May 1997. Baltimore.

[16] Bikram S. Bakshi, P. Krishna, Nitin H. Vaidya, and Dhiraj K. Pradhan. Improving performance of TCP over wireless networks. In *International Conference on Distributed Computing Systems*, 1997.

[17] H. Balakrishnan and R. Katz. Explicit loss notification and wireless web performance. *Proc. IEEE Globecom Internet MiniConference, Sydney, Australia*, November 1998.

[18] Hari Balakrishnan, Venkata Padmanabhan, Srini Seshan, Mark Stemm, and Randy H. Katz. TCP behavior of a busy internet server: Analysis and improvements. *Proc. IEEE Infocom*, 1998.

[19] Hari Balakrishnan, Venkata N. Padmanabhan, and Randy H. Katz. The effects of asymmetry on TCP performance. In *Third ACM/IEEE Mobicom conference, Budapest, Hungary*, September 1997.

[20] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, 1997.

[21] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving tcp/ip performance over wireless networks, November 1995.

[22] Hari Balakrishnan, Srinivasan Seshan, and Randy H. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks*, 1(4), 1995.

[23] R.K. Balan, B.P. Lee, K.R.R. Kumar, W.K.G. Jacob, L.and Seah, and A.L. Ananda. Tcp hack: Tcp header checksum option to improve performance over lossy links. In *INFOCOM 2001. Proceedings. IEEE*, volume 1, pages 309 –318, 2001.

[24] C. Barakat. Tcp/ip modeling and validation. *IEEE Network*, pages 38–47, May/June 2001.

[25] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[26] E.R. Berlekamp, R.E. Peile, and S.P. Pope. The application of error control to communications. *IEEE Communication Magazin*, 25(4):44–57, April 1987.

[27] P. Bhagwat, C. Perkins, and S Tripathi. Network layer mobility: an architecture and survey. *IEEE Personal Communications*, 3(3):54–64, June 1996.

[28] Pravin Bhagwat, P. Bhattacharya, A. Krishna, and Satish K. Tripathi. Enhancing throughput over wireless lans using channel state dependent packet scheduling. In *INFOCOM*, 1996.

[29] S. Biaz, M. Mehta, S. West, and N. Vaidya. Tcp over wireless networks using multiple acknowledgements. Technical Report Tech-Rep. 97-001, Computer Science Dept. Texas A&M University, January 1997.

[30] S. Biaz and N. Vaidya. Discriminating congestion losses from wireless losses using inter-arrival times at the receiver. *IEEE Symposium ASSET'99, Richardson, TX, USA*, 1999.

[31] S. Biaz and N. H. Vaidya. Distinguishing congestion losses from wireless transmission losses: A negative result. *Seventh International Conference on Computer Communications and Networks (IC3N), New Orleans*, October 1998.

[32] A. D. Birell and B. Nelson. Implementing remote procedure calls. *ACM Transaction on Computer Systems*, 2(1):1–14, 1985. Not downloaded, not printed, not read, yet.

[33] K.L. Blackard, Rappaport T.S., and Bostian C.W. Measurements and models of radio frequency impulsive noise for indoor wireless communication. *Journal on Selected Areas in Communication*, 11(7):991–1001, September 1993.

[34] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. RFC 3135: Performance enhancing proxies intended to mitigate link-related degradations. not complete yet, August 2001.

[35] L Bos and S Leroy. Toward an all-ip-based umts system architecture. *IEEE Network*, pages 36–45, January/February 2001.

[36] R. T. Braden. RFC 1122: Requirements for Internet hosts — communication layers, October 1989.

[37] Lawrence Brakmo and Larry Peterson. TCP vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communication*, 13(8), Oktober 1995.

[38] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[39] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000. Expanded version available as USC TR 99-702b.

[40] K. Brown and S Singh. M-udp: Udp for mobile networks. *ACM Computer Communication Review*, 26(5):60–78, October 1996.

[41] K. Brown and S. Singh. A network architecture for mobile computing. In *Proc. of IEEE INFOCOM, S.F., CA*, pages 1388–1396, 1996.

[42] Kevin Brown and Suresh Singh. M-TCP: TCP for mobile cellular networks. *ACM Computer Communication Review*, 27(5), 1997.

[43] Ramon Cáceres and Liviu Iftode. The effects of mobility on reliable transport protocols. In *International Conference on Distributed Computing Systems*, pages 12–20, 1994.

[44] K.L Calvert, M.B Doar, and E.W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[45] B. Carpenter. RFC 1958: Architectural principles of the Internet, June 1996. Status: INFORMATIONAL.

[46] M.C. Chan and R. Ramjee. Tcp/ip performance over 3g wireless links with rate and delay variation. In *Proceedings of the eighth annual international conference on Mobile computing and networking*, pages 71 – 82, Atlanta, Georgia, USA, 2002. ACM Press New York, NY, USA.

[47] W.P. Chen, Y.C. Hsiao, J.C. Hou, Y. Ge, and M.P. Fitz. Syndrome: a light-weight approach to improve tcp performance in mobile wireless networks. *Wireless Communication & Mobile Computing, Special Issue: Reliable Transport Protocols for Mobile Computing*, 2(1):37–58, February 2002.

[48] D. Clark and M. Blumenthal. Rethinking the design of the internet: The end to end arguments vs. the brave new world, 2000.

[49] David D. Clark. The design philosophy of the DARPA internet protocols. In *SIGCOMM*, pages 106–114, 1988.

[50] J. Cobb and P. Agrawal. Congestion or corruption? a strategy for efficient wireless tcp sessions, 1995.

[51] P.B. Danzig, S. Jamin, R. Caceres, D. Mitzel, and D Estrin. An empirical workload model for driving wide-area tcp/ip network simulations. *Internetworking: Research and Experience*, 3(1):1–26, March 1992.

[52] S. Dawkins, G. Montenegro, M Kojo, V. Magret, and N. Vaidya. RFC 3155: End-to-end performance implications of links with errors, August 2001.

[53] M. Degermark, M. Engan, B. Nordgren, and S. Pink. Low-loss TCP/IP header compression for wireless networks. *ACM/Baltzer Journal on Wireless Networks*, 3(5):375–387, 1996.

[54] M. Degermark, B. Nordgren, and S. Pink. RFC 2507: IP header compression, February 1999. Status: PROPOSED STANDARD.

[55] A. DeSimone, M. Chuah, and O. Yue. Throughput performance of transport-layer protocols over wireless lans. *Proceedings of Globecom '93*, December 1993.

[56] D. Duchamp and N.F. Reynolds. Measured performance of wireless lan. In *Proc. of 17th Conf. on Local Computer Networks*, Minneapolis, 1992.

[57] D. Dutta and Y. Zhang. An active proxy based architecture for tcp in heterogeneous variable bandwidth networks. In *Proc. of IEEE GLOBECOM*, 2001.

[58] J.-P. Ebert and A. Willig. A gilbert-elliot bit error model and the efficient use in packet level simulation. Technical Report TKN-99-002, Telecommunication Networks Group, Technische Universität Berlin, March 1999.

[59] D. Eckhard and P Steenkiste. Measurement and analysis of the error characteristics of an in-building wireless network. In *Proc. of ACM SIGCOMM*, pages 243–254, Stanford University, California, August 1996.

[60] D. Eckhardt and P. Steenkiste. Improving wireless lan performance via adaptive local error control. *Int. Conf. Network Protocols*, pages 327–338, 1998.

[61] D.A. Eckhardt and Steenkiste P. An internet-style approach to wireless link errors. *Wireless Communication & Mobile Computing, Special Issue: Reliable Transport Protocols for Mobile Computing*, 2(1):21–36, February 2002.

[62] E.O. Elliot. Estimates of error rates for codecs on burst-noise channels. *Bell Syst. Tech. J.*, 42:1977–1997, September 1963.

[63] M. Eyrich, M. Schläger, and A. Wolisz. Using the remote socket architecture as nat replacement. In *Proc. of 5th European Personal Mobile Communications Conference (EPMCC)*, Glasgow, Scotland, April 2003.

[64] G. Fairhurst and L. Wood. RFC 3366: Advice to link designers on link automatic repeat request (arq), August 2002.

[65] K Fall and S Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *Computer Communication Review*, 26(3):5–21, 1996.

[66] K. Fall, K. Varadhan, and the VINT project. The ns manusl. http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf.

[67] A. Feldmann, A. Gilbert, W. Willinger, and T. Kurtz. The changing nature of network traffic: Scaling phenomena. *ACM SIGCOMM Computer Communication Review*, 28(2), April 1998.

[68] Anja Feldmann, Anna C. Gilbert, Polly Huang, and Walter Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *SIGCOMM*, pages 301–313, 1999.

[69] D.C. Feldmeier, A.J. McAuley, J.M. Smith, D.S Bakin, W.S. Marcus, and T.M Raleigh. Protocol booster. *IEE Journal on Selected Areas in Communication*, 16(3):437–444, April 1998.

[70] A: Festag. *Mobility Support in IP Cellular Networks - A Multicast-Based Approach. Dissertation*. PhD thesis, Technische Universität Berlin, June 2003.

[71] A. Festag, H. Karl, and G. Schaefer. Current developments and trends in handover design for all-ip wireless networks. Technical Report TKN-00-007, Telecommunication Networks Group, Technische Universität Berlin, August 2000.

[72] S. Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24:10–24, October 1994.

[73] S. Floyd and T. Henderson. RFC 2582: The newreno modification to tcp's fast recovery algorithm, April 1999. State:Experimental.

[74] S. Floyd, J. Mathis, J. Mahdavi, M.Podolsky, J. Heidmann, J. Touch, and T. Henderson and. RFC 2883: An extension to the selective acknowledgment (sack) option for tcp, July 2000.

[75] S. Floyd and V. Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking*, 9(4):392–402, August 2001.

[76] Brahim Ghribi and Luigi Logrippo. Understanding GPRS: the GSM packet radio service. *Computer Networks (Amsterdam, Netherlands: 1999)*, 34(5):763–779, 2000.

[77] E.N. Gilbert. Capacity of a burst-noise channel. *Bell. Syst. Tech. J.*, 39:1253–1265, September 1960.

[78] Tom Goff, James Moronski, and D.S. Phatak. Freeze-tcp: A true end-to-end tcp enhancement mechanism for mobile environments. In *Proc. of Infocom 2000, Israel*, pages 1537–1545, 2000.

[79] A. Gurtov. Making tcp robust against delay spikes. Technical report, University of Helsinki, Department of Computer Science, Series of Publications C, November 201. http://www.cs.helsinki.fi/u/gurtov/papers/report01.html.

[80] Z. Haas and P. Agrawal. Mobile-TCP: An asymmetric transport protocol design for mobile systems. *ICC'97, Montreal, Canada*, June 1997.

[81] Fred Halsall. *Data Communications, Computer Networks and Open Systems*, chapter 4.3.4, page 202. Addison-Wesley Publishing Company, fourth edition edition, 1996.

[82] M. Handley, J. Padhye, and S. Floyd. RFC 2861: Tcp congestion window validation, June 2000. Status:Experimental.

[83] John Heidemann, Nirupama Bulusu, Jeremy Elson, Chalermek Intanagonwiwat, Kun chan Lan, Ya Xu, Wei Ye, Deborah Estrin, Ramesh Govindan, and John Heidemann. Effects of detail in wireless network simulation. submitted to SCS Communication Networks and Distributed Systems Modeling and Simulation Conference, September 2000.

[84] F. Hillebrand, editor. *GSM and UMTS - The Creation of Global Mobile Communication*. John Wiley & Sons, Ltd, 1 edition, 2002. inkl. CD-ROM.

[85] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series. Prentice Hall, 1991.

[86] E. Hossain, D.I. Kim, and V.K. Bhargava. Tcp performance under dynamic link adaptation in cellular multi-rate wcdma networks. In *Proc. of the IEEE ICC'2002*, New York, USA, April 2002.

[87] IEEE, editor. *Information Technology—Portable Opersting System Interface (POSIX)— Part xxx: Protocol Independent Interfaces (P2II)*. Number P1003.1g,D6.6 in POSIX. Institute of Electrical and Electronics Engineers, Piscataway, N.J., March 1997.

[88] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[89] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low-speed serial links, February 1990. Status: PROPOSED STANDARD.

[90] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP extensions for high performance, May 1992. Obsoletes RFC1072, RFC1185 [91, 92]. Status: PROPOSED STANDARD.

[91] V. Jacobson and R. T. Braden. RFC 1072: TCP extensions for long-delay paths, October 1988. Obsoleted by RFC1323 [90]. Status: UNKNOWN.

[92] V. Jacobson, R. T. Braden, and L. Zhang. RFC 1185: TCP extension for high-speed paths, October 1990. Obsoleted by RFC1323 [90]. Status: EXPERIMENTAL.

[93] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM Computer Communication Review*, 19:56–71, 1989.

[94] R. Jain. *The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation, and Modeling*. WILEY Professional Computing, 1991.

[95] P. Johansson, M. Kazantzidis, R. Kapoor, and M. Gerla. Bluetooth: An enabler for personal area networking. *IEEE Network*, 15(5):28–37, September/October 2001.

[96] Rick Jones. The public netperf homepage. http://www.netperf.org/.

[97] Y. Joo, V. Ribeiro, A. Feldmann, A.C. Gilbert, and W. Willinger. Tcp/ip traffic dynamics and network performance: A lesson in workload modeling, flow control, and trace-driven simulation. *ACM SIGCOMM Computer Communication Review*, 31(2):25–37, April 2001.

[98] P. Karn and C Partridge. Improving round-trip time estimates in reliable transport protocols. In *SIGCOMM*, 1987.

[99] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, 1991.

[100] S. Keshav and S. Morgan. Smart retransmission: Performance with overload and random losses. In *Proc. of Infocom*, 1997.

[101] M. Kojo, K. Raatikainen, and T. Alanko. Connecting Mobile Workstations to the Internet over a Digital Cellular Telephone Network. In *Mobidata Workshop on Mobile and Wireless Information Systems*, November 1994.

[102] M. Kojo, K. Raatikainen, M. Liljeberg, J. Kiiskinen, and T. Alanko. An efficient tranport service for slow wireless telephone links. *IEEE Journal on Selected Areas in Communication*, 15(7):1337–1348, September 1997.

[103] A. Konrad, B.Y. Zhao, A.D. Joseph, and R. Ludwig. A markov-based channel model algorithm for wireless networks. In *Proc. of Fourth ACM Inter. Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2001.

[104] D Kotz and Essien. Analysis of a campus-wide wireless network. In *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking*, pages 107–118. ACM Press, September 2002.

[105] R. Krishnan, M. Allman, C. Partridge, and J. P.G. Sterbenz. Explicit transport error notification (eten) for error-prone wireless and satellite networks – summary. In *Earth, Science Technology Conference, Pasadena, CA, USA*, June 2002.

[106] Anurag Kumar. Comparative performance analysis of versions of TCP in a local network with a lossy link. *IEEE/ACM Transactions on Networking*, 6(4):485–498, 1998.

[107] A.M Law and W. Kelton. *Simulation, Modeling and Analysis*. MCGraw-Hill, second edition edition, 1991.

[108] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. RFC 1928: SOCKS protocol version 5, April 1996. Status: PROPOSED STANDARD.

[109] M. Liljeberg, H. Helin, M. Kojo, and K. Raatikainen. Mowgli www software: Improved usability of www in mobile wan environments. In *Proc. IEEE Global Internet 1996, London, England*, pages 33–37, November 1996.

[110] S. Lin, D.J. Costello, and M.J. Miller. Automatic repeat request for error control schemes. *IEEE Communication Magazine*, 22(12):5–16, December 1984.

[111] Steven Low, Larry Peterson, and Limin Wang. Understanding TCP Vegas: A Duality Model. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 226–235, June 2001.

[112] R. Ludwig, A. Konrad, and A. Joseph. Optimizing the end-to-end performance of reliable flows over wireless links. In *Fifth Annual International Conference on Mobile Computing and Networks (MobiCOM '99)*, 1999.

[113] R. Ludwig, B. Rathonyi, A. Konrad, K. Oden, and A.D. Joseph. Multi-layer tracing of TCP over a reliable wireless link. In *Measurement and Modeling of Computer Systems*, pages 144–154, 1999.

[114] R. E. Ludwig. *Eliminating Inefficient Cross-Layer Interactions in Wireless Networking*. PhD thesis, Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen, 2001.

[115] Reiner Ludwig and Randy H. Katz. The eifel algorithm: Making TCP robust against spurious retransmissions. *ACM Computer Communication Review*, 30(1):30–37, 2000.

[116] Reiner Ludwig and Bela Rathonyi. Link layer enhancements for TCP/IP over GSM. *IEEE INFOCOM*, 1999.

[117] Jouni Malinen. Host ap driver for intersil prism2/2.5/3. http://hostap.epitest.fi/.

[118] D. Maltz and P. Bhagwat. Msocks: An architecture for transport layer mobility. In *Proceedings of the IEEE INFOCOM*, 1998.

[119] D.A Maltz and P. Bhagwat. Tcp splicing for application layer proxy performance. *Journal of High Speed Networks*, 1999.

[120] S. Mascolo, C. Casetti, M. Gerla, Sanadidi M., and R. Wang. Tcp westwood: End-to-end bandwidth estimation for efficient tranpsport over wired and wireless networks. In *Proc. of MOBICOM,Rome, Italy*, July 2001.

[121] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP selective acknowledgment options, October 1996. Status: PROPOSED STANDARD.

[122] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: Refining TCP congestion control. In *SIGCOMM*, pages 281–291, 1996.

[123] S. McCanne and S. Floyd. ns network simulator. http://www.isi.edu/nsnam/ns/.

[124] S. McCanne and S. Floyd. ns network simulator. http://www.isi.edu/nsnam/ns/.

[125] Miten Mehta and N. H. Vaidya. Delayed duplicate acknowledgements: A proposal to improve performance of TCP on wireless links. *Tecnical Report (Texas A&M University)*, 1998.

[126] G Montenegro, M Kojo, and V Magret. RFC 3150: End-to-end performance implications of slow links, July 2001.

[127] Tim Moors Moors. A critical review of "end-to-end arguments in system design".

[128] G. Nguyen, R. H. Katz, B. Noble, and M. Satyanarayanan. A trace-based approach for modeling wireless channel behavior. In *Proc. Winter Simulation Conf., Coronado, CA, December*, 1996.

[129] Bob O""Hara and Al Petrick. *IEEE 802.11 Handbook*. IEEE, 1999.

[130] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe. Modeling TCP throughput: A simple model and its empirical validation. *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 303–314, 1998.

[131] C. Parsa and J. Garcia-Luna-Aceves. Tulip: A link-level protocol for improving tcp over wireless links. In *Proc. IEEE WCNC*, pages 1253–1257, 199.

[132] C. Parsa and J. J. Garcia-Luna-Aceves. Improving TCP congestion control over internets with heterogeneous transmission media. *Proc. IEEE ICNP 99: 7th International Conference on Network Protocols*, 1999.

[133] K. Pawlikowski, Jeong H.-D.J., and J.-S.R. Lee. On credibility of simulation studies of telecommunication networks. *IEEE Communication*, 40(1):132–139, 2002.

[134] V. Paxson and M. Allman. RFC 2988: Computing tcp's retransmission timer, November 2000.

[135] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. RFC 2330: Framework for IP performance metrics, May 1998. Status: INFORMATIONAL.

[136] Vern Paxson. Empirically derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, 1994.

[137] Vern Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, 1997.

[138] Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[139] Vern Paxson and Sally Floyd. Why we don't know how to simulate the internet. In *Winter Simulation Conference*, pages 1037–1044, 1997.

[140] J. Postel. RFC 768: User datagram protocol, August 1980.

[141] J. Postel. RFC 791: Internet Protocol, September 1981.

[142] J. Postel. RFC 793: Transmission control protocol, September 1981.

[143] K Ramakrishnan, S. Floyd, and D. Black. RFC 3168 the addition of explicit congestion notification (ecn) to ip, September 2001. Obsoletes: 2481, Updates: 2474, 2401, 793.

[144] K. Ratnam and I. Matta. WTCP: An efficient transmission control protocol for networks with wireless links. Technical Report NU-CCS-97-11, Dept. of Electrical and Computer Engineering, Northeastern University, July 1997.

[145] K. Ratnam and I. Matta. WTCP: An efficient transmission control protocol for networks with wireless links. *Proc. Third IEEE Symposium on Computers and Communications (ISCC '98), Athens, Greece*, June 1998.

[146] S.M. Redl, M.K. Weber, and M.W. Oliphant. *An Introduction to GSM*. Artech House, 1995.

[147] D. Reed. The end of the end-to-end argument, 2000.

[148] A.K. Salkintzis, C Fors, and R Pazhyannur. Wlan-gprs integration for next-generation mobile data networks. *IEEE Wireless Communication*, pages 112–124, October 2002.

[149] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[150] N.K.G. Samaraweera and G. Fairhurst. Reinforcement of TCP error recovery for wireless communication. *Computer Communication Review*, 28(2), April 1998.

[151] R Sànchez, J. Martinez, J. Romero, and R. Järvelä. Tcp/ip performance over egprs network. In *IEEE Vehicular Technology Conference*, pages 1120–1124. IEEE, 2002.

[152] M. Schlaeger, B. Rathke, S. Bodenstein, and A. Wolisz. Advocating a remote socket architecture for internet access using wireless LANs. *Mobile Networks and Applications (Special Issue on Wireless Internet and Intranet Access)*, 6(1):23–42, January 2001.

[153] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic TCP buffer tuning. In *SIGCOMM*, pages 315–323, 1998.

[154] W.R. Stevens. *Unix Network Programming*, volume 1. Prentice Hall, Upper Saddle River NJ, 2 edition, 1998.

[155] A.S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 4 edition, 2003.

[156] K. Thompson, G Miller, and R Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, Nov/Dec 1997.

[157] V. Tsaoussidis and H. Badr. Tcp-probing: Towards an error control schema with energy and throughput performance gains. *Proceedings of the 8th IEEE Conference on Network Protocols, Japan*, November 2000.

[158] V. Tsaoussidis and I. Matta. Open issues on tcp for mobile computing. *Wireless Communication & Mobile Computing*, 2(1):3–20, February 2002.

[159] N. Vaidya, M. Mehta, C. Perkins, and G. Montenegro. Delayed duplicate acknowledgements: A TCP-unaware approach to improve performance of TCP over wireless. *Wireless Communication & Mobile Computing, Special Issue: Reliable Transport Protocols for Mobile Computing*, 2(1):37–58, February 2002.

[160] B. Walke. *Mobilfunknetze und ihre Protokolle*, volume Band 1. B.G. Teubner, Stuttgart, 1 edition, 1998. Grundlagen, GSM, UMTS und andere zellulare Mobilfunknetze.

[161] H.S. Wang and N. Moayeri. Finite-state markov channel: A useful model for radio communication channels. *IEEE Trans. on Veh. Tech*, 44(1):163–171, February 1995.

[162] K. Wang and S. Tripathi. Mobile-end transport protocol: An alternative to tcp/ip over wireless links. In *IEEE Infocom*, pages 1046–1053, March 1998.

[163] Z. Wang and J. Crowcroft. A new congestion control scheme: Slow start and search (tri-s). *ACM Computer Communication Review*, 21:32–43, 1991.

[164] D. Wenqing and A Jamalipour. A new explicit loss notification with acknowledgment for wireless tcp. In *12th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, volume 1, pages 65–69, September 2001.

[165] A. Willig, M. Kubisch, and A. Wolisz. Measurements and stochastic modeling of a wireless link in an industrial environment. Technical Report TKN-01-001, Telecommunication Networks Group, Technische Universität Berlin, March 2001.

[166] W. Willinger, V. Paxson, and M. S. Taqqu. Self-similarity and Heavy Tails: Structural Modeling of Network Traffic. *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*, 1998.

[167] A. Wolisz, C. Hoene, B.Rathke, and M.Schlaeger. Proxies, active networks, re-configurable terminals: The cornerstones of future wireless internet. In *Proc. IST Mobile Communications Summit*, pages 795–803, Galway, Ireland, October 2000.

[168] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated Volume 2 - The Implementation*. Addison-Wesley, Reading, Massachusetts, 1995.

[169] R. Yavatkar and N. Bhagawat. Improving end-to-end performance of tcp over mobile internetworks, 1994.

[170] M. Yavuz and F. Khafizov. Tcp over wireless links with variable bandwidth. In *Proc. of IEEE Vehicular Technology Conference*, pages 1322–1327. IEEE, September 2002.

[171] X. Zhao, C. Castelluccia, and M. Baker. Flexible network support for mobility. In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, Dallas, Texas, October 1998.

[172] M. Zorzi and R.R Rao. The effect of correlated errors on the performance of tcp. *IEEE Communication Letters*, 1(5):127–129, 1997.