TKN Telecommunication Networks Group

Technical University Berlin

Telecommunication Networks Group

Using the Remote Socket Architecture as NAT Replacement

Michael Eyrich, Tobias Poschwatta, Morten Schläger

{eyrich,posch,morten}@ee.tu-berlin.de

Berlin, March 2002

TKN Technical Report TKN-02-15

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

Abstract

The Internet is used everywhere today, at commercial sides as well as at private homes. Unfortunately, the number of available addresses of the IP protocol version 4, the core protocol of the Internet has become a scarce resource. Widely deployed solutions to stretch the number of available addresses are Network Address Translation (NAT) and Dynamic Host Configuration Protocol (DHCP). In spite of their wide-spread use these approaches have some drawbacks which limit their general deployment.

In this paper we present an alternative approach which is based on exporting the socket interface. The approach, named the Remote Socket Architecture (ReSoA), allows clients without a globally valid IP address or even without a TCP/IP stack at all to access the Internet without any application specific support, which is required in case of NAT. A ReSoA enabled client sees the IP protocol stack of the ReSoA-server as its own protocol stack and hence, uses the Internet Protocol (IP) address of the server. Thus, all ReSoA-clients share the IP address of the ReSoA-server.

Further appealing features of ReSoA are that technology optimized protocols can be used to improve performance, and that it might reduce protocol processing overhead for the endsystem, for instance for light weight wireless devices.

Contents

Chapter 1

Introduction

The Internet is the driving technology these days. Access to the Internet is required and found everywhere, in the industry as well as in schools and universities or at private homes. Unfortunately, there is an issue of addressing space. Every host which is connected to the Internet needs at least for one of its interfaces a globally valid IP address. As a consequence, IP addresses are becoming a scarce resource. This often implies that a user must pay for every single public Internet address in use. This is especially the case for private home users and Small Office Home Office (SOHO) users who have multiple end-systems which they want to connect to the Internet. For each internet address those users need a contract with an Internet Service Provider (ISP). To overcome the problem of paying for multiple contracts those users often apply the NAT approach and can therefore be satisfied by a single contract.

A second demanding concern is security. If an attacker knows the address of a host within a local network, he has the first key to start attacks. From a security point of view it would be much more convenient if all end-systems where hidden behind a single gateway and preferably not visible or accessible from the Internet at all. Further, if every user is allowed to install arbitrary services on his end-system, this would inherently introduce a serious security hole.

Classical remedies for the address shortness are NAT and DHCP. Both are widely deployed. DHCP is mostly found on commercial sites which have a number of IP addresses but not as many as end-systems, while NAT is mostly found for connecting private homes to a service provider or whenever a local network is to be hidden completely. In spite of the uncontradicted usefulness of both approaches, they both have some drawbacks which limit their transparent deployment. [1].

We propose a new architecture that overcomes a number of problems connected to the NAT approach. The architecture is named with its functional description: Remote Socket Architecture (ReSoA). In ReSoA function calls to the BSD socket interface are not executed on the local end-system but in an RPC-like fashion on a special network node. This means, that the socket interface on 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 the special network node, where the function calls are executed. Thus,

[1] IPv6 is not considered as an option here, because it requires the replacement of the infrastructure and it cannot be foreseen yet, whether and when it will be deployed

application utilize the service of the TCP/IP protocol stack on a remote host. The design of the ReSoA modules is such that neither the syntax nor the semantics of the socket interface is changed. This especially means that every application which was designed for the socket interface can operate on top of ReSoA without recognizing any differences. Consequently, any application on the ReSoA-client uses the address of the ReSoA-server just as any application on the server itself uses the address of the server system for its network access.

The motivation for the ReSoA approach is the observation that the Application Programming Interface (API) and not the protocol is what matters for the application. Applications are designed with regard to a specific interface semantics. In case of the BSD socket interface this means that the implementation of the socket layer can be changed as long as the syntax and semantics of the interface to the socket layer are maintained.

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 IP addressing.

Our ReSoA approach allows to hide an arbitrary number of systems behind a single system without the necessity of owning routeable addresses. In contrast to NAT our ReSoA-approach provides applications immediately with internet-wide valid addresses instead of modifying data streams to exchange addresses out of a private address range during server traversal with the server's address. Another key advantage of our proposal is the ability of applications to arbitrarily encrypt any sensible data, even in case that the local IP address is part of the application data. This is not possible in case NAT is applied!

The following chapters are structured as follows. In chapter 2 we discuss problems of the IPv4 addressing scheme that lead to the propagation of NAT in current networks. In the following two chapters (3.1 and 3.2) we discuss the two approaches DHCP and NAT. Then in chapter 4 we introduce our Remote Socket Architecture and we show how ReSoA can be used to solve the problem of address shortness. Finally in section 4.4 we compare the different approaches.

Chapter 2

Problem description

2.1 IP address shortness

With the constantly growing Internet, globally valid addresses of the IP version 4 addressing scheme are on their way to get a scarce resource. Although the total number of available addresses is huge—nearly $2^{31} \approx 2.41 * 10^9$ minus the number of addresses needed to define networks and broadcasts within the network[1]—there are reasons for an increasing shortness of addresses.

IP addresses are categorized in classes of the types A, B, C and D. The last category is reserved for multicast addresses while the others define classes of addresses of three different sizes (A as the largest with ($2^{24} = 16 * 10.7^6$), C as the smallest one with $2^8 = 256$ addresses each). Each of these classes are assigned as a whole to companies. Given the assignment of complete classes and the routing connected with it, it is difficult to move parts of, for instance, a class A subnet out of their domain to another domain which is short of addresses. Therefore, the other domain will remain short of addresses although there are still globally valid addresses available. The problem of undersized domains is not avoidable because the size a specific network might grow to is not known in advance.

Another reason for the address shortness is found in the following dualism: On the one hand, IP addresses are used as an identifier for a specific system such that the system is uniquely identified. On the other hand, any system connected to more than one network has, and requires, more than one IP address and is therefore identified by multiple addresses. The number of such a routing-bound address "overhead" increases with the reduction of the subnet sizes. In case of SOHO configurations with a number of three or four hosts the overhead would be enormous.

2.2 Security and Service Control

Whenever computers are operational and connected to the world wide Internet there is the potential risk of being exposed to attacks. In general, there are three common types of

[1] For a classless network with four addresses such as 192.168.10.243/30 two of the four addresses are reserved—and therefore lost for real usage—for the domain and the network; in the given case .243 as network and .246 as broadcast address

attacks. (i) Denial of Service (DoS) attacks put a very high load on the access network or the system itself, making it eventually unusable and unable to react to service requests. (ii) With intrusion attacks, specific applications are attacked to give the intruder additional operational rights such as root access. Finally, (iii) backdoors may be opened by a malicious application, executed, for instance, from a Mail User Agent (MUA) without the knowledge of the user.

For SOHO and likewise installations, securing of the network should always be a concern. Potential solutions exist based on different approaches:

(i) Host/address based access control. Here, applications consult a local database, which contains a mapping between services and the hosts respectively the networks that are allowed to access this service. The access control can be realized either by a wrapper process (often called `tcpd`) or by a service providing application itself. This is the most basic approach which allows for basic access control but does not offer protection against attacks of—or by courtesy of—misbehaving applications.

(ii) Router based traffic control (firewalling) is the most powerful technique to protect a network. However, configuration of a well behaving firewall is difficult and requires a lot of experience to be successful.

(iii) Visibility reduction is an approach, which does not require much experience to be established. Protection is achieved by granting access to the Internet using a single system, e.g., an Access Router (AR). All requests to the Internet are routed via the AR. Within the Internet, only the AR is visible. For the reverse direction only a single destination address (that of the system itself) is valid and will be accepted. As a result, all local systems are shadowed by this AR. Because of the AR it is impossible to scan systems behind the AR, and systems behind the AR cannot unexpectedly offer services such as a File Transfer Protocol (FTP) service or a virus initiated backdoor to the Internet.

We will not cover the first two approaches in further detail. However, any kind of Access Router as presented in the following chapter provides for additional protection without the need of additional configuration work.

Chapter 3

Traditional approaches

In this chapter we discuss two traditional approaches to overcome the problem of IP version 4 address shortness. Of course, the introduction of IP version 6 will also solve these problems, however their introduction still seems to need more time, therefore we stick on IP version 4 solutions.

3.1 The DHCP approach

The Dynamic Host Configuration Protocol (DHCP) bases on the technique to acquire an IP address just for the time it is needed. By sending a DHCP-request a client asks for a free address. A DHCP-server (potentially one out of a pool of servers) answers the request with an available address, which the client then "leases". The lease is to be renewed at regular intervals. If it is not renewed and really unused (not configured on any interface in the network)[1], it is re-inserted into the pool of free addresses. This leads to a sharing of a number of addresses between machines.

The main field of application for DHCP are desktop computers which are only active as long as somebody is working with them. It found its justification in the observation that the working time of people differs such that only a subset is working at any point in time. This suggests to have only the number of addresses assigned to supply the systems of all concurrently working people with a valid address. Whenever a system is no longer in use, its IP address is released and ready for use by another system.

DHCP supports hiding of machines to some extent by changing a systems' IP address over time. However, this is not a real increase in security. Although DHCP is a nice approach, it limits the number of concurrently connected machines to the total number of available addresses. Network access is impossible for all other machines. This is often not acceptable and specifically a problem, if a number of machines is to be updated in unattended mode (for instance over-night). Additionally it does not allow systems to provide services within the network such as offering unused resources for distributed processing. It also does not alleviate the SOHO problem.

[1] Unused addresses can be discovered for instance by `ping`ing the address

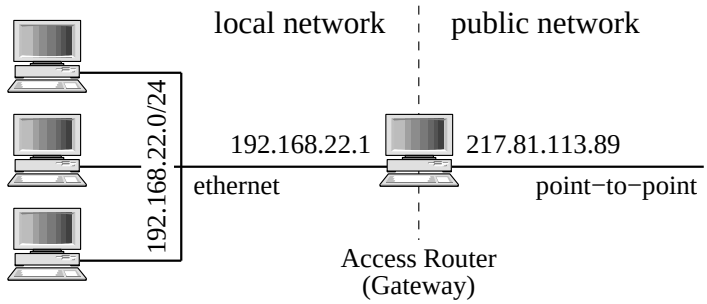

Figure 3.1: Internet access by applying NAT

3.2 The NAT approach

NAT [1, 8] is used to hide a group of end-systems behind a single globally valid IP address provided by an Access Router (AR). The operational mode of a NAT device is the following: For a Protocol Data Unit (PDU) passing a NAT gateway any source address within a preconfigured range of source addresses is substituted with the external interface address of the NAT gateway. The respective mapping is stored in a table for later use. For PDUs travelling into the private network and with no application listening on the destination port, the mapping table is consulted and the destination address and port is substituted with the original source address and port found in the table.

NAT is mostly used to offer multiple end-systems without an own, globally valid IP address access to the service of the Internet. A typical field of application is a home user who has multiple end-systems but only a single contract with his ISP (SOHO).

Figure 3.1 illustrates such a typical configuration. In this scenario four PCs form a private Internet. One of these PCs operates as an AR. Here, the AR is defined as a system with two or more interfaces, where at least one of them offers a globally valid IP address. For instance, the connection to an ISP might be provided as a Point-to-Point Protocol (PPP) connection. The other three computers form an intranet using addresses out of the pool of private addresses. They are configured to use the AR as their default gateway. Because of the lack of globally valid IP addresses they cannot get direct access to the Internet. To offer these machines access to the Internet, the AR must perform NAT as explained in Section 3.2.1. In general, end-systems behind a NAT server only use the services (e.g., WWW) of the Internet but do not offer services (e.g.: host a WWW server) to the public Internet. The end-system is not aware of the address translation process, it might only know about having a private IP address.

3.2.1 NAT types

In general, there are two kinds of NAT. *Basic NAT* maps a set of local addresses to an equal-sized set of globally valid addresses. Access for local systems is granted as long as there are not more local systems than addresses in the global set of addresses. Otherwise, simultaneous access of all local systems is not possible. Access for individual systems can be granted by establishing a static NAT. Static NAT also allows for systems to be accessible from

external hosts. *Network Address Port Translation (NAPT)*, also known as dynamic NAT is an extension to the NAT service. In contrast to basic NAT a NAPT scheme requires only a single external address to serve quite a large number of private systems[2]. The translation is achieved by mapping the source address of packets to the externally valid address of the NAPT-host, while potentially also substituting the port with an unused one of the NAPT-host. For returning packets these port numbers allow to map the IP-addresses back to the correct system. If not otherwise mentioned, NAT is used as a generalization of NAT and NAPT.

3.2.2 NAT and Application Level Addressing

Although NAT is widely deployed it has many well-known drawbacks. First of all, some applications encapsulate the source IP address in their payload (e.g. FTP). In order to support such applications, an Application Level Gateway (ALG) is needed. An ALG is specific to the application layer data stream and therefore aware of application structures within the data stream. ALGs are necessary for applications such as FTP, H.323, COPS, multi-party document sharing[3], network based games, etc.[6, 2]. The ALG scans the payload and replaces private IP addresses (and ports in case of NAPT) of the NAT client with the address of the NAT server. If the application encrypts its payload the ALG either has to know the encryption key or NAT cannot be used for address-aware streams. This is therefore specifically true for Secure Socket Layer (SSL)-encrypted FTP traffic, which contains (and requires) the originating IP-address. However, for SSL-encrypted http-traffic, for instance, this is not a concern as the encoded address is not necessary to route and transmit data back to the request originator.

Particularly for flows which need an ALG, address and port translation may get expensive. In an FTP flow for instance, ports and IP addresses are encoded in ASCII format. Therefore, the packet length is subject of change depending on the number of places required to code the value (port 257 coded as “1,1” in contrast to port 32890 coded as “128,122”, which requires four additional places). This in turn requires continued recalculation of sequence and acknowledge numbers for the remains of the flow.

3.2.3 End-to-End Semantics

In case of dynamic NAT, the NAT server must establish a context for any active flow, which specifies the (above mentioned) mapping. Depending on the type of flow the necessity to create a context and the lifetime of the context is a matter of guess. For TCP flows the SYN and FIN segments respectively give a detailed hint[4], but in case of the connection-less UDP this is more complicated and, handling results in a trade-off between available resources and the number of broken connections due to an unavailable mapping. RFC 793[3] proposes a time-out of $2 * MSL$ at earliest, or 4 minutes.

[2]The number of concurrently supported hosts is mainly restricted by the number of simultaneous connections (the more connections originate from a host, the less hosts can be supported totally). A theoretical maximum are 65534 connections per public interface

[3]control and data are different sessions, which might get different addresses

[4]which, of course, requires protocol analysis of the flow

The loss of a context (either by unavailability of the system or given by a time-out) essentially breaks the idea of an end-to-end semantics for flows: Data transmission should be independent from the routes packets are traveling along; the Internet had conceptually been developed as a connectionless system, where two consecutive packets to the same destination are not required to take the same route. To achieve a routing independent behavior, flow specific states should be kept in end systems only. In contrast, for flows over a NAT device, state information is kept at three places: at both of the end systems as well as at the NAT device. Once a flow has been established, all packets have to pass the NAT device, which in turn presents a single point of failure that cannot be bypassed. For static NAT devices the impact of a temporary unavailability is less strong, as the mapping can be reconstructed much more easily.

Header Modification

Since the NAT server modifies the IP header, IP and TCP/UDP checksum need to be re-calculated. This is even true, if the ports are not modified since the IP addresses are part of a pseudo-header which the Internet transport protocols use to calculate their checksum. However, such a checksum recalculation is not resource-intensive, since only the checksum for the changed part of the header has to be re-calculated[7].

IP options handling, such as Record Route, Strict Source Route and Loose Source Route also require recalculation of checksums. These options may expose private addresses and network structures. However, application of those options does not lead to a misbehavior since each registered address is valid for the next hop only (couple of hops for Loose Source Routing).

If fragmentation is used, a NAT server cannot guarantee that the identifier of the IP header is unique (whenever fragments of different NAT clients are directed to the same destination). This is in general not a problem with TCP, since TCP's path Maximum Transfer Unit (MTU) discovery allows to divide the byte stream in appropriate sized segments that do not require additional fragmentation. UDP on the other hand might require the fragmentation mechanisms of the IP protocol stack. To solve the problem of non-unique identifiers for fragmented packages, the NAT server either has to reassemble the original packet or also has to map the IP identifier. The latter introduces a major security problem: A malicious sender might introduce fragments with spoofed address information into a packet such that answers are sent back to a different target.

When a NAT server receives fragmented packets from the Internet it should reassemble them in order to decide to which client they belong. Following the address translation the IP packet again has to be fragmented in order to be deliverable to a client. A different approach is to recognize the identifier of fragments, which complicates the design of the NAT server. Therefore, NAT servers are typically configured to re-assemble all packets before starting a NAT/NAPT translation.

Security Associations

Major implications in the end-to-end semantics can be found when security associations are established and an end-to-end connection is required. Such a connection is possible only for

data streams that do not need ALGs. For IPSec, a secure connection is only feasable based on securing the connection in a hop-by-hop manner. This, of course, presupposes that the NAT gateway is a member of the clients trusted domain.

Chapter 4

The ReSoA approach

The approaches discussed so far all have in common that the TCP/IP stack is located in the end-system. In opposite to this, our approach implements a remote access to the service of the TCP/IP stack.

Our idea is motivated by the observation, that applications are designed for a certain interface (syntax and semantics) but are independent from the implementation of the interface or the protocols used to provide the service. Following this idea, it is 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 exporting the protocol stack's interface to the end-systems (e.g., following an RPC-like approach). The export of the interface must be in a way that neither syntax nor semantics of the interface are changed. When our approach is applied, then arbitrary protocols, which provide a certain service, can be used for the communication between end-system and ReSoA server. Therefore, end-system and access network can be IP free.

We applied our idea of an interface export to the socket interface, because the Berkeley socket interface is the most widely used interface for unix-like OS architectures. Sockets are a general abstraction for intra- and inter-process communication and are protocol and addressing format independent. Since our interest concentrates on the Internet domain, the export of the socket interface only deals with sockets belonging to the Internet family. We named our architecture *Remote Socket Architecture (ReSoA)*. Details about ReSoA as well as a performance comparison, showing that ReSoA can significantly improve the performance over wireless TCP, can be found in [5].

Figure 4.1 on the next page gives a general overview of the components of ReSoA 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. In principle ReSoA can be viewed as a client located on the end-system, and a server which is located on the network node. 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. For the communication between the ReSoA client and the ReSoA server a two layered architecture was chosen. The idea behind the two layers is, that the upper layer which is responsible for the message exchange between ReSoA client and ReSoA server is technology independent, while the lower layer, which is responsible for the provision of the required communication service

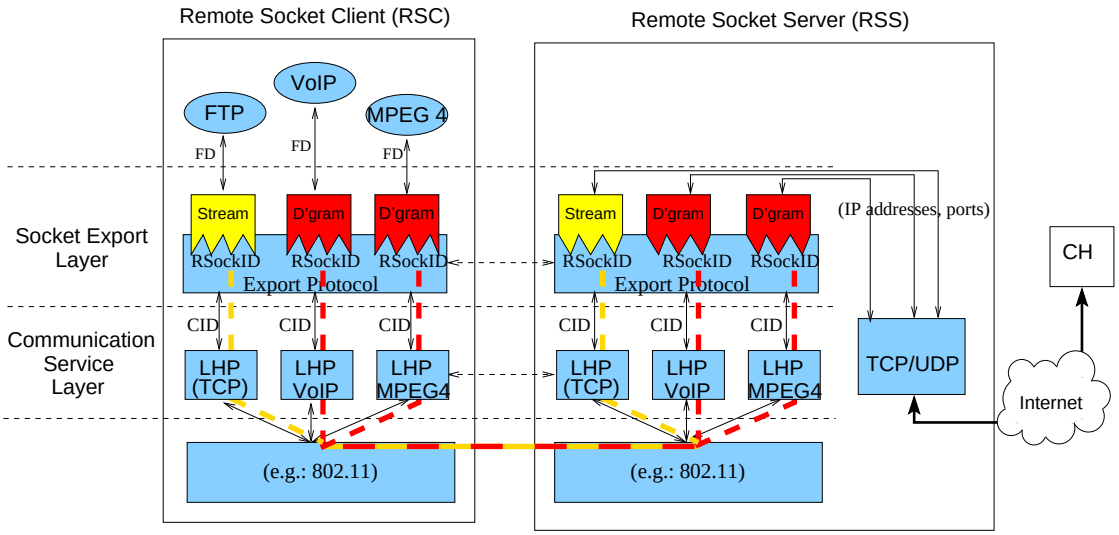

Figure 4.1: Basic components of ReSoA

is technology dependent. The protocol of the upper layer is called Export Protocol (EP). The EP is a simple connection-less request-response protocol. The protocols of the lower layer are called Last Hop Protocol (LHP)s. The service provided by an LHP depends on socket type and application, and may be adapted to the properties of the applied technology. In case of a TCP socket an LHP must provide a fully reliable service. In case of a UDP socket an LHP may provide a semi-reliable service. In the latter case the LHP may behave application specific. Further, we have decided that all LHPs are connection oriented. For further details about the service of the Communication Service Layer (CSL) and its interface to the Socket Export Layer (SEL) see [4].

From the application point of view the existence of ReSoA is transparent. Every application designed for the socket interface can operate on top of ReSoA without any modification.

4.1 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 four tuple consisting of the source and destination IP addresses and the source and destination port to find the correct transport protocol entity.

When ReSoA is used, two additional addressing questions must be solved. On the one hand an association of the two parts of a socket, which are loaded on the ReSoA-server and the ReSoA-client respectively, has to be established and on the other hand the source address to be used for the transport protocol entity has to be determined. We refer to the former as *Access Network Addressing* and to the latter as *Internet Addressing*.

4.1.1 Access Network Addressing

Access Network Addressing defines how the two parts which 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 on which ReSoA runs. On the upper layer the Export Protocol (EP) is responsible for delivering incoming packets to the correct socket object.

An LHP must provide a communication channel between the ReSoA client and ReSoA server. The channel is created when ReSoA is started (see 4.3 on the following page). 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 covers different technologies, technology independent address formats should 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, then the LHP must 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 messages. 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 on which LHP connection the packet was received. Since the LHP provides a connection oriented service, the ReSoA client and ReSoA maintain a connection identifier for each created LHP connection. This connection identifier is included in every indication from an LHP 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 in which the transport protocol ports are also not sufficient to identify a communication end-point. Instead the transport protocols use the IP address in addition to their ports. The difference here is, that IP is connection-less and hence the complete source and destination addresses must be used, while the LHP utilizes the addressing format of the underlying technology and uses only a connection identifier for multiplexing[2].

The addressing formats of both levels are independent from 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.

ReSoA uses an explicit addressing between ReSoA clients and ReSoA server. An advantage of the explicit addressing is that the ReSoA-server can be placed anywhere in the domain, it does not need to be placed on the common path of a flow.

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

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

Details on the assignment of socket identifiers are given in section 4.3, where we go through an example ReSoA session.

4.1.2 Internet Addressing

ReSoA provides two different IP address assignment schemes which are described in the following. An assignment scheme describes how the ReSoA server maps globally valid IP addresses to its clients. In the first IP addressing scheme the ReSoA server provides a separate globally valid IP address for every registered client. We therefore call this mode *1-to-1 mode*. In the second scheme the ReSoA server has only a limited number of IP addresses (e.g.: only a single IP address). These IP addresses are shared between all registered clients. This mode is called *n-to-1 mode*. Furthermore, a combination of the two addressing schemes is possible. This means that a client can have its own public IP address registered with the server, while other clients share the server's IP address. It is up to the administrator of the ReSoA-server which addressing schemes will be available.

Please note that the address mapping mode does not necessarily determines how the ReSoA server obtains its IP addresses. In 1-to-1 mode it is possible that the client registers its IP address with the ReSoA server or that the ReSoA may use DHCP on behalf of the client to get a new IP address, every time a new client is registered[3]. 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.

In order to combat the IP address shortness problem it is required that ReSoA operates in n-to-1 mode. By running in n-to-1 mode the ReSoA-server supplies all connected clients with its own address and therefore provides—with regard to the number of required IP addresses—the same functionality as NAPT.

4.2 Operation modes

ReSoA can be operated either in *dual protocol stack* mode or in *pure ReSoA* mode. In dual protocol stack mode an end-system supports both a complete local TCP/IP stack and a ReSoA protocol stack (remote access to the TCP/IP stack of the ReSoA server). The application must chose by some mechanism (like `LD_PRELOAD`), which protocol stack it intends to use. In pure ReSoA mode the end-system has no TCP/IP protocol stack. Every application automatically uses the exported interface.

The dual protocol stack mode can be used to allow direct local communication (between ReSoA clients) and for automatic configuration purposes.

4.3 Operation of ReSoA

The purpose of this chapter is twofold. On the one hand we want to provide a better understanding of the operation of ReSoA and on the other hand we intend to illustrate why

[3]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

ReSoA is a more natural approach, to share a single IP address between multiple hosts. Therefore, we go through the setup process of a ReSoA system and continue with an example session of an FTP connection.

For this discussion and for the rest of the paper 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. Dual protocol stack mode is chosen to enable local clients to communicate with each other without using a ReSoA-server [4].

However, the IP address of the clients belongs to the class of locally valid addresses. This configuration was chosen, since it is from a service point of view functionally equivalent to a NAT environment.

At 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. The ReSoA server prepares the local LHP instances to accept incoming connections and is now awaiting ReSoA clients to register.

A ReSoA client needs additionally the LHP address of the ReSoA server. In case of a 802.x environment a ReSoA client gets 6 bytes as the LHP address, which represent the MAC address of the ReSoA server. The initial step when a ReSoA client started is to establish the LHP connections with the ReSoA server and to register[5] itself at the server. The registration process concludes with providing the source address used for future connection over 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 on the ReSoA server without recognizing any difference to a local TCP/IP implementation[6].

For the further discussion let us consider an FTP session as an example as shown in Figure 4.2 on page 17. In this session the FTP client runs on 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 listing Transmission Control Protocol (TCP) end-point.

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 function call `socket`. Due to performance reasons, this call is not 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, assignes an `rsockID` to it, and returns the control back to the application.

Next, the application initiates the establishment of the transport protocol connection, using the socket call `connect`. The ReSoA client sees from the state of the socket that the corresponding part of the socket on the ReSoA-server was not yet created. Therefore it sends a request message to the server (using the previously opened LHP connection) which includes the socket function call as well as the connect function call. This request message includes

[4]The IP address of the local protocol stack belongs to the class private IP addresses. Please note that even local communication without IP addresses is possible using ReSoA.

[5]The registration process is out of the scope of this paper and therefore not further discussed.

[6]Well, in some environments an improved performance is possible.

[7]To limit the probability that the ReSoA server has not sufficient resources for the already allocated socket, each ReSoA client is only allowed to have a certain number of parallel open sockets.

the `rsockID` as explained in section 4.1.1 on page 13. The calling application is blocked until the ReSoA server sends the corresponding return value. The response of the ReSoA server does not only contain the result of the `connect` call but also the port of the local connection endpoint. This accelerates the fetching of address information (using `getsockname`).

The ReSoA server knows on the basis of the LHP connection to which ReSoA client the request belongs. 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. For the TCP connection the local IP address of the ReSoA server is used as source address. Thus, every application—independent from the end-system it is running on—uses the IP address of the server for its connections. When the TCP connection is established, the ReSoA-server sends a response message to the ReSoA client. Again, this response message 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. Exactly this is one of the advantages of ReSoA compared to NAT. In NAT the `getsockname` call would return the local IP address of the client, which is then encoded into the application request and would require translation by the NAT device. [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 Remote Procedure Call (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.

When the TCP instance on the ReSoA server receives data (from the Internet), it uses the four tuple (IP addresses and ports) to map the data to a socket as usual using a callback. The socket is equipped with information about 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, which is used by the ReSoA client to map the received data to the correct socket receive buffer.

In principal, the behavior of the TCP/IP stack on the ReSoA server is identical to the behavior of a TCP/IP stack of a classical end-system. However, two modifications are required in order to maintain the semantics of the socket interface. First of all, TCP must not acknowledge the FIN-segment before all data are delivered to the ReSoA-client. Second, TCP's advertised window should not be increased when data is successfully transfered to the

[8] Normally, we had to create a second socket, export it and put it into listening mode for the data connection of an FTP transfer, which we left out of the message sequence chart in Fig 4.2 on the following page for better readability

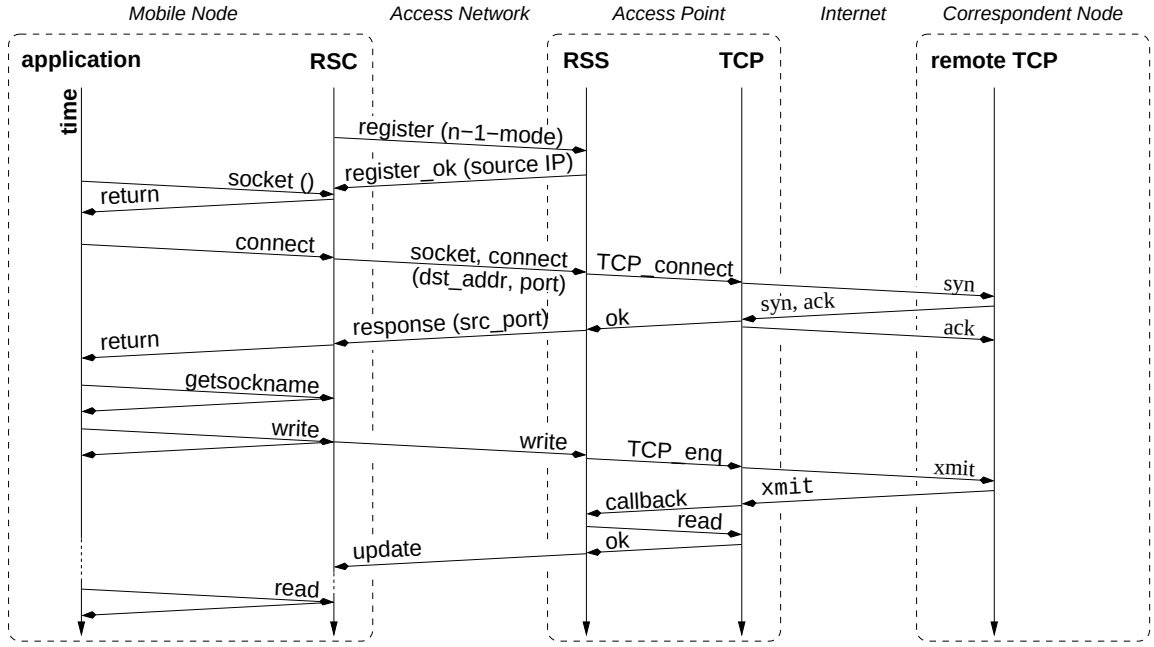

Figure 4.2: Example message exchange resulting from an application communicating using a ReSoA-client and a ReSoA-server configuration.

ReSoA-client but when the application has read the data.

4.4 Comparison with NAT

In ReSoA different end-systems share the TCP/IP protocol stack of a single ReSoA server, just as different processes share the TCP/IP protocol stack of a local system. Just like processes sharing the same IP address, in ReSoA (when operating in n-to-1 mode) all applications share the IP address of the server. The difference to classical end-systems is that the application can be located on different end-systems (as applications can belong to different users). Since different end-systems share the IP address of the server, ReSoA reduces the IP address usage in the same way as NAT does.

The service offered by ReSoA in dual protocol stack and n-to-1 mode, is in functionality equivalent to the service offered by NAT. The clients can communicate with each other using their private addresses. All packets destined for the Internet get the IP address of the border device (either NAT device or ReSoA-server as source address). Thus, all end-systems are hidden behind a single, globally valid IP address.

However, there is a significant advantage from the application point of view. With NAT the local application is only in knowledge of its private IP address. In Section 3.2.1 we pointed out, that some applications encode their local IP address into their payload. With ReSoA the application is in knowledge of the IP address of the ReSoA server as its local IP address, which is just now the “official” address. If the application encodes its IP address, then the

Feature	ReSoA	NAT
End-to-End IPsec	no	no
Arbitrary Positioning	yes	no
Knowledge of IP address seen by correspondent system	yes	no
End-to-End Application Layer Encryption	yes	Depends on application
Hiding of clients	yes	yes
Intermediate state	yes	yes
Single Point of Failure	yes	yes
Protocol Processing at intermediate System	yes	yes

Table 4.1: Comparision of ReSoA and NAT

address of the ReSoA server—and therefore the real, globally valid end-address—is encoded. Thus, no translation process, including the correction of sequence numbers and checksums is necessary. This especially implies, that application level encryption can be used. Further, with ReSoA the communicating applications see the same connection (same 4-tuple) when they query the socket, while they see different 4-tuples in case of NAT. Although the ReSoA approach does not allow for applying IPsec end-to-end (just as the NAT/NAPT approach) it nevertheless allows end-to-end encryption at application layer. Therefore, the ReSoA-server does not necessarily need to be part of the trust-domain of the ReSoA-client.

Furthermore, with NAT two communicating TCP instances have a different view on the connection. Beside the different addresses of the connection end-points, the receiving TCP does not always receive the segments sent by the other TCP. The received segments can differ in checksum, sequence number, acknowledgment number, ports and payload. This obviously violates TCP's end-to-end semantics.

In case of ReSoA the communication end-point is on the ReSoA-server. Therefore, it is not necessary to alter any TCP segment. Although TCP's acknowledgments are sent by the ReSoA-server, this does not violate the semantics seen by the application, since the meaning of an acknowledgment is that the data has reached the peer TCP instance but not the application. Even with a local TCP/IP implementation, it is possible that an application never receives data already acknowledged by TCP.

As with NAT the ReSoA-server is a single point of failure. When the ReSoA-server crashes, all connections are lost and traffic exchange with the Internet is not longer possible. The difference between ReSoA and NAT is, that in case of ReSoA the communication end-point dies, while in case of NAT an intermediate system dies. Since this intermediate system holds state information which are crucial for the connection, it makes no difference, that the TCP end-points survive in case of NAT.

In case of NAT an end-system crash comes along with an abrupt termination of all connections. With ReSoA this is not the case. The TCP instance on the ReSoA-server continues with its normal operation (e.g.: sends acknowledgments) until it is stopped by the ReSoA-server. This does not violate the semantics seen by the application, since a TCP acknowledgment only means that the data were received by TCP but not that they were consumed by an application. The remote application cannot wrongly assume a correct data transfer, as the final packet is not acknowledged before all data are passed to the ReSoA-client. In addition, in case of long transfers, the advertised window will fill-up and stop the

other end.

A further important difference is that NAT devices need to be in the common path of the flow while a ReSoA-server needs not. This path must not be changed (e.g.: due to multi-homed networks and routing changes connected with it). The direct addressing of the ReSoA server allows for placing the ReSoA-server anywhere in the access network addressing domain. This also allows for load sharing between several ReSoA-servers.

The differences and similarities of both approaches are summarized in table 4.1 on the page before.

Chapter 5

Conclusion

In its current way of application the IPv4 addressing scheme often encounters a regional shortness in the number of available addresses. Various solutions exist such as NAT/NAPT which lead to problems with regard to application layer encryption and on-the-fly data modification necessary when traversing such devices.

We have introduced the ReSoA framework as an alternative approach to the IPv4 addressing problem. In the final outcome ReSoA provides a service similar to NAPT, but with a number of pretty enhancements: (i) by exporting the socket interface, any application using ReSoA has knowledge of its regular IP address to immediately code it in its application data stream where necessary; (ii) by allowing applications to immediately code addresses into its data stream, no data stream translation is necessary. (iii) the same feature allows application to encrypt its data without requiring an intermediate system to be able to decode (and re-encode) the data or to encode data on behalf of the NAPT-client. Thus, ReSoA allows an application to operate even in an untrusted environment, since it is possible to encrypt data with arbitrary encryption algorithms; (iv) the operation of a ReSoA-server is completely independent from the applications served via the ReSoA-server (as opposed to NAPT, where a specific ALG is required for every application data stream that contains address or port information); and finally, (v) ReSoA even allows for devices without an TCP/IP protocol stack to communicate with corresponding applications in the Internet using the remote TCP/IP stack.

In combination with appropriate packet classifiers and adaptable link layer retransmissions controlled by the Communication Service Layer ReSoA also allows to significantly improve the performance of wireless Internet access.

Nomenclature

ALG	Application Level Gateway
API	Application Programming Interface
AR	Access Router
CSL	Communication Service Layer
DHCP	Dynamic Host Configuration Protocol
DoS	Denial of Service
EP	Export Protocol
FTP	File Transfer Protocol
IP	Internet Protocol
ISP	Internet Service Provider
LHP	Last Hop Protocol
MAC	Medium Access Control
MTU	Maximum Transfer Unit
MUA	Mail User Agent
NAPT	Network Address Port Translation
NAT	Network Address Translation
PDU	Protocol Data Unit
PPP	Point-to-Point Protocol
ReSoA	Remote Socket Architecture
RPC	Remote Procedure Call
SEL	Socket Export Layer
SOHO	Small Office Home Office
SSL	Secure Socket Layer
TCP	Transmission Control Protocol

Bibliography

[1] K. Egevang and P. Francis. RFC 1631: The IP network address translator (NAT), May 1994. Status: INFORMATIONAL.

[2] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. RFC 1928: SOCKS protocol version 5, April 1996. Status: PROPOSED STANDARD.

[3] J. Postel. Transmission control protocol. Request for Comments 793, Internet Engineering Task Force, September 1981.

[4] M. Schlaeger and T. Poschwatta. Remote Socket Architecture Service and Interface of the Last Hop Protocol for TCP. Technical Report TKN-01-016, Telecommunication Networks Group, Technische Universität Berlin, October 2001.

[5] 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.

[6] Shiuh-Pyng Shieh, Fu-Shen Ho, Yu-Lun Huang, and Jia-Ning Luo. Network Address Translators: Effects on Security Protocols and Applications in the TCP/IP Stack. IEEE Internet Computing, pages 42–49. IEEE, Nov-Dec 2000.

[7] P. Srisuresh and K. Egevang. RFC 3022: Traditional IP Network Address Translator (Traditional NAT), January 2001. Status: INFORMATIONAL.

[8] P. Srisuresh and M. Holdrege. RFC 2663: NAT Terminology and Considerations, August 1999. Status: INFORMATIONAL.