TKN Telecommunication Networks Group

Technical University Berlin

Telecommunication Networks Group

TWIST: A Scalable and Reconfigurable Wireless Sensor Network Testbed for Indoor Deployments

Vlado Handziski, Andreas Köpke, Andreas Willig, Adam Wolisz

{handzisk,koepke, willig, wolisz}@tkn.tu-berlin.de

Berlin, November, 2005

TKN Technical Report TKN-05-008

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

Abstract

We present TWIST, a scalable and flexible testbed architecture for indoor deployment of wireless sensor networks. The design of TWIST is based on an analysis of typical and desirable uses of sensor network testbeds. In addition to providing basic services like node configuration, network-wide programming, out-of-band extraction of debug data and gateWaying of application data, the architecture introduces several novel features, some of them exploiting the capabilities of the USB 2.0 standard. Firstly, TWIST supports different sensor node platforms. Secondly, it supports active power-control of the nodes. This enables easy transition between USB-powered and battery-powered experiments, dynamic selection of topologies as well as controlled injection of node-failures in the system. Thirdly, TWIST supports evaluation of both flat and hierarchical sensor networks. The "super nodes" that form the middle tier of the testbed infrastructure are dual-use devices that can also play a role as part of the sensor network application.

The self-configuration capability, the use of standardized hardware and open-source software make the TWIST architecture scalable, affordable, and easily replicable. To demonstrate its practical value, we present our experiences with building and using a specific realization of the TWIST architecture that spans three floors of our office building and supports over one hundred sensor nodes.

Contents

Chapter 1

Introduction

Wireless Sensor Networks (WSNs) are large-scale distributed embedded systems incorporating small, energy- and resource-constrained sensor nodes communicating over wireless media [10]. The design, implementation and evaluation of sensor network applications and communication protocols is a difficult task, considering their distributed nature. The first design steps can often be made with the help of simulations, which frequently force the designer to make artificial assumptions about traffic, failure patterns and topologies. The later steps of implementation and evaluation of application performance as well as error resilience and other nonfunctional properties, require the use of real hardware, realistic environments and realistic experimental setups.

The primary goal of any testbed is to support the design, implementation/test and evaluation of sensor network applications and protocols. We refer to these as SUE. To facilitate SUE design, it should be possible to implement different network architectures in the testbed, from which the designer can select the best one for his application. To support the implementation/test/debugging phase, functionalities like node (re-)programming as well as collection, processing and displaying of debug data are needed. For evaluation purposes, users should have means to precisely and reproducibly control network topologies, to inject node faults. Besides these development-oriented goals a WSN testbed should be scalable enough to support large-scale deployments with more than just a few dozen nodes while keeping the costs at a reasonable level.

In this paper we describe the design, architecture and deployment of the TKN Wireless Indoor Sensor network Testbed (TWIST). It evolved out of our experiences with the first WSN testbed [12] deployed at TKN (Telecommunication Networks group at Technical University Berlin) in the framework of the EU IST EYES project.

The architecture of TWIST is hierarchical and supports realization of different SUE architectures. The lowest tier is formed by *sensor nodes*, which can be of different types. The second tier is formed by *super nodes*, which are power-unconstrained and have much higher computational power. The sensor nodes are attached to the super nodes through a USB infrastructure. A third tier is formed by a dedicated *server* hosting, for instance, a database of the nodes installed in the testbed and their current locations. Finally, the *control station* provides the interface between the user and the testbed.

TWIST is based on cheap off-the-shelf hardware and uses open-source software. It is thus a cost-effective and open solution, which can be reproduced by other institutions. Furthermore,

the openness of the TWIST software gives full flexibility in the use of the testbed.

The rest of the paper is organized as follows: in the next Section 2 we give a high-level view on the design goals and design rationale of the TWIST architecture. The details of our architecture design are explained in Section 3, and the specific instance of TWIST in our building is described in Section 4. Related work is surveyed in Section 5 and in Section 6 we conclude the paper.

Chapter 2

The Design of TWIST: Goals and Rationale

In this section we explain the design goals and rationale for TWIST. The discussion is structured according to the different phases of sensor network application development: design, implementation/test and experimental evaluation. The management of the testbed itself is also examined.

2.1 Helping Designers: Matching SUE Architecture and TWIST Architecture

A number of different wireless sensor network architectures have emerged. In the *flat architecture* the sensor network is composed of homogeneous sensor nodes running the same application and protocol code. In a *segmented architecture* a number of flat networks are coupled by gateways. The different flat networks can use incompatible radio technologies. In a *multi-tier architecture* or *hierarchical architecture* a sensor network application is partitioned such that parts of it run on low-end sensor nodes, whereas other parts run on more capable high-end sensor nodes, which have no energy constraints and have better memory and computational resources. In addition to being connected to the low-end nodes, the high-end nodes can interact among themselves and with entities higher in the hierarchy via a backbone network.

These three scenarios can be easily realized with TWIST thanks to the flexible boundary between the SUE and testbed functionalities:

- Flat sensor networks: Under this scenario, the boundary between the SUE and the testbed is just the USB interface on the sensor nodes. The super node and the server / control station exclusively perform testbed functions. The testbed is used to program the sensor network and to extract debug data or application-related data from the WSN. The extracted data can be preprocessed, compressed, filtered or aggregated already in the super nodes in a distributed fashion. The debug data is transferred out-of-band and does not consume any wireless bandwidth. In cases when even the instrumentation of SUE code with debug code causes unwanted interactions, then some sensor nodes can

be programmed with the pure SUE code while others can be dedicated to snoop the network traffic.

- Hierarchical sensor networks: Here, the super nodes can play a role both in the SUE and in the testbed. They do this in two ways. One possibility is to let some of the super nodes act as high-level nodes in the sensor network application and others as testbed nodes. The other possibility is to execute both roles at the same time on a single super node. Hence, the super nodes are dual-use devices. Again, debug data originating in sensor nodes is not transmitted over the wireless channel. But in the time-sharing case the super nodes have to split their computational resources and Ethernet bandwidth between SUE and testbed-related functionalities.

- Segmented sensor networks: To implement this scenario, super nodes can be used as gateways between different flat segments.

The communication between the super nodes and the server / control station is carried out using TCP/IP, making it easy to export the testbed services to (authorized) remote users.

The issue of scalability is not only a general requirement for sensor networks, but the testbed infrastructure itself also requires scalability in order to support non-trivial numbers of sensor nodes. As an example, let us consider the continuous generation of debug data or the observation of application-related data. Data from all sensor nodes (or at least from large groups) are required to obtain insight into the operation of the network. For a large sensor network the sheer volume of debug data can be overwhelming, potentially congesting the backbone network or overloading server and control stations. The super nodes of TWIST can be used to filter, aggregate, or compress the generated data, thus pushing the "congestion barrier" towards higher numbers of nodes. The server and control station can be used to store the aggregated data and present online- and offline evaluations to the user.

2.2 Helping Implementers/Testers: Programming and Time Synchronization

The implementation and debugging of sensor network applications requires frequent reprogramming of the nodes with a new (hopefully less buggy) software. There exist approaches (and protocols) to distribute new application code *within* the sensor network and over the wireless interface [9, 19], but it takes resources on the node, uses valuable wireless bandwidth and may take a long time. TWIST allows reprogramming of sensor nodes using the testbed infrastructure ("out-of-band programming"), freeing the sensor network programmer from implementing code distribution protocols and allowing him to concentrate on other aspects. The reprogramming facility of TWIST is based on the ability of certain microcontrollers to be reprogrammed over the serial line, exported over the USB interface.

Fixing bugs on single nodes can be a tedious task, it gets even more complicated in a distributed system like a WSN, where individual nodes only contribute small parts to the global state and race conditions can cause serious headache to the programmer. Determining the current state of the system and how it was entered, is crucial for distributed debugging. However, writing a distributed debugger that allows a complete "happened-before" ordering

of the events simply exceeds the computational capacities of the sensor nodes. Using the testbed, a simpler but less precise approach can be taken. When the application dumps debug data, this data is time stamped by the testbed. The precision of the time stamping using Network Time Protocol (NTP) [13] can be in the range between hundreds of microseconds and a few milliseconds [3]. The WSN can only send packets every few milliseconds, hence the precision of the time stamping is often sufficient for debugging and mapping of "WSN-time" to wall-clock time.

2.3 Helping Experimenters: Power Control

A main purpose of a testbed is research. This means that it should enable controlled and reproducible experiments.

Sensor networks differ from other types of networks by the fact that energy aspects are of crucial importance. Energy consumption is one of the major performance metrics of sensor network protocols and applications. Furthermore, sensor nodes often have a finite energy budgets and sensor networks have highly time-variable topologies due to death of nodes and deployment of new nodes. This means that applications and protocols must be robust against node failures or addition of new nodes. A WSN testbed should offer support for testing this robustness under realistic circumstances.

TWIST offers a key facility to realize this: binary power-control. As explained in Section 3, by switching off the power of sensor nodes, the death of nodes can be emulated. Conversely, by switching them on, the deployment of new nodes is mimicked. Hence, it is possible to create topologies and inject faults in a controlled fashion. When sensor nodes are both battery-powered and USB-powered, another option becomes feasible: The nodes are programmed and configured while USB power is on. The USB power is switched off for all nodes at the same time, and a common, well-defined starting point is created for measuring the lifetime achievable with a certain sensor network application / protocol.

Another requirement in the controlled setup of experiments is to have control over the times when certain actions like the configuration or start of sensor nodes have to be performed. For example, for investigating the influence of interference on the transmission of data between two nodes, the transmitting node and the interfering node should be started at the same time. Here, the structure of TWIST, where time-synchronized super nodes can tightly control the behavior of sensor nodes over the USB interface is helpful.

2.4 Helping Testbed Owners: Management

In WSN applications, the unique identifier of a node is usually not overly important. However, this does not apply to testbed management. To keep the testbed operational, it is important to identify nodes correctly, for instance to make a clear decision on which of the deployed nodes is malfunctioning and needs replacement. For the same reason, the exact position of nodes is also helpful.

The exchange of nodes is a frequent event in a testbed. Therefore, TWIST separates node identifiers from the locations where nodes are attached to the testbed. In fact, in TWIST the sensor nodes are plugged into fixed USB *sockets*, which have unique identifiers and a

priori known the geographic locations. The USB interface on the super nodes detects when a node is plugged into a socket. As a result, a software event is triggered. On receiving such an event, the super node extracts the manufacturer serial number from the event data and determines the unique node identification (nodeID). Then it registers the binding between the node and the socket identifier in a database on the server. This database also keeps the association between the socket identifiers and their geographical positions. This way it is possible to put nodes into arbitrary sockets and to automatically keep the database in a consistent state. Furthermore, it is an easy task to figure out the precise position of a sensor node given its identification, to determine all sensor node identifications pertaining to a given geographical area and so forth. The ability to identify nodes allows to track nodes with, for example, faulty sensors. It also permits storing per-node configuration and calibration data in the database.

As a secondary benefit, the localization information can be made available to the SUE. The SUE implementer hence does not need to implement localization algorithms on his own and can concentrate on the problem at hand. When, by chance, the problem happens to be the implementation and test of a localization algorithm, the known socket positions provide a ground truth against which the new algorithm can be compared. Similar arguments hold true for the time synchronization capabilities of TWIST.

In the next section we provide a more detailed discussion of the TWIST architecture.

Chapter 3

TWIST Architecture

Figure 3.1 depicts the hardware architecture of TWIST. In the following we describe the most important characteristics of the testbed entities, starting from the lowest layer, comprised by the sensor nodes, moving up to the testbed backbone, with the attached server and control station.

3.1 Sensor Nodes

As main hardware units of the SUE the sensor nodes need a set of hardware capabilities facilitating their seamless integration with the rest of the testbed infrastructure. According to the identified core functionalities in Section 2 they have to expose suitable hardware interfaces that support external powering, reprogramming, as well as out-of-band exchange of configuration, debug and application data. Traditionally, these functions have been served by dedicated interfaces: power supply bus, JTAG for programming and debugging, RS-232 for serial communication, etc. This was significantly complicating the hardware and software integration between the SUE and the tester, making customized hardware solutions necessary, and driving the costs for even simple testbed configurations prohibitively high.

Recently, several WSN node platforms have emerged using Universal Serial Bus (USB) as standardized hardware interface. Thanks to the features of the USB specification and the flexibility of the current UART-to-USB chips, this hardware interface is able to replace almost all of the custom interfaces, simplifying integration of the sensor nodes with external systems. This migration to a single standardized interface drastically lowers the costs for the testbed due to the reduced prices of the components as a result of the economies of scale.

The overall architecture of the TWIST is crucially centered on the use of the USB interface. By relying on this open standard, we are able to support a heterogeneous mixture of WSN platforms as long as they export the above listed capabilities (power-supply, programming and communication) via a standard-compliant USB interface. At the moment, both the eyesIFX [7] and the Telos [15] mote families satisfy the above conditions and have been successfully interfaced with TWIST.

On the software side, the operating system running on the sensor nodes has to satisfy several basic requirements. First, it has to provide a suitable execution environment for the application logic of the SUE. Secondly, it should support node configuration, instrumentation

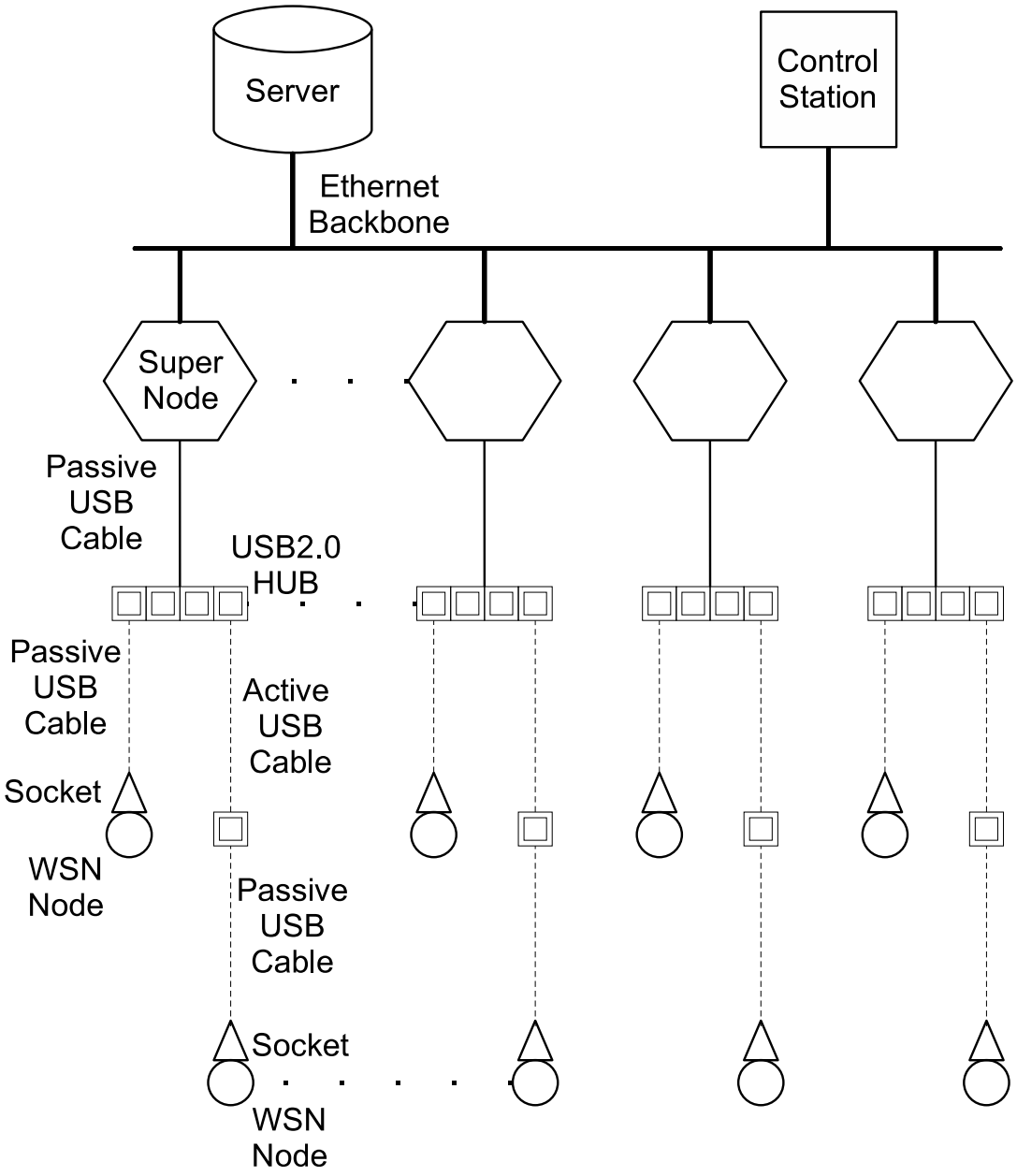


Figure 3.1: Hardware architecture of the TWIST testbed

of the application code and allow for out-of-band communication with the super nodes over the USB infrastructure.

TinyOS [8, 4] satisfies these requirements and can be run both on the Telos and the eyesIFX platforms. It provides a generic and lightweight execution platform for sensor network applications. TinyOS is already shipped with components that support communications over the serial interface using framed protocol similar to PPP/HDLC. On top of this protocol, TinyOS components have been developed that enable `printf`-like logging as well as bridging debug and application messages. Using tools like Pytos [17], even more powerful, RPC-like interactions can be supported.

3.2 Testbed Sockets and USB Cabling

Seen as plain hardware, a testbed *socket* is nothing more than the point where the USB interface of the sensor node attaches to the USB infrastructure of the testbed. As explained in Section 2.4, the logical significance of this point for the TWIST architecture is, however, much greater.

The sockets are connected to the rest of the testbed using a combination of passive and active USB cables, depending on the distance between the location of the socket and the next element of the infrastructure – the USB hubs. Using passive cables a maximum distance of 5 m can be bridged. For greater distances, "active USB cables" can be used (single port USB hubs with fixed cable), or several USB hubs can be daisy-chained together.

(a) (b)

Figure 3.2: (a) Telos mote in a testbed socket; (b) Super node and USB-hub arrangement

3.3 USB Hubs

The hubs are the central element of the TWIST USB infrastructure and provide support for some of the most important features of the TWIST architecture.

At the most basic level, the USB hub is a multiplexing device that enables us to break the one-to-one correspondence between the sensor nodes and the second-level testbed devices,

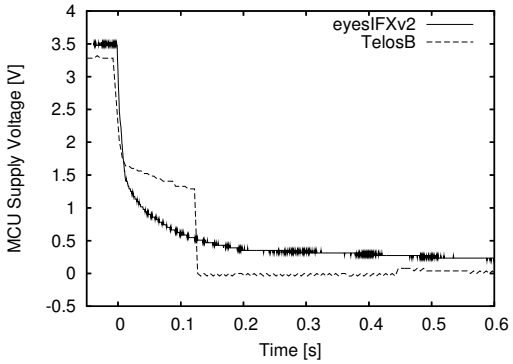
typical for many of the existing WSN testbeds. This enables significant cost savings without compromising any of the testbed functionality. Even more, the USB hubs give TWIST one of its most powerful capability: the binary power-control over the sensor nodes in the testbed.

The USB Hub Specification 2.0 requires that self-powered hubs support port power switching. By sending a suitable USB control message, the software can control the power state of a given port on the hub, effectively enabling/disabling the power supply for any downstream device(s) attached. In the case of TWIST, these downstream devices are the sensor nodes attached to the testbed sockets. This means that we are able to individually control the power supply of any sensor node in the testbed by simply issuing a suitable USB control message to that particular hub to which the corresponding testbed socket is connected.

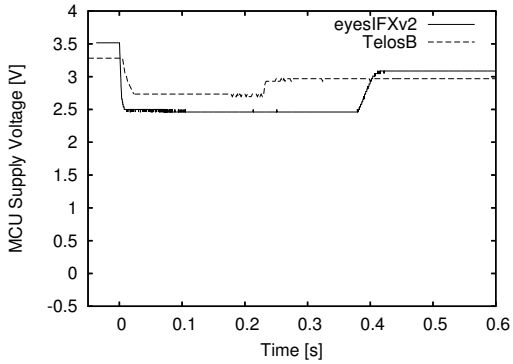
Depending on whether the sensor node attached to the socket has a battery or not, this enables us to perform four different transitions:

- from "USB-powered" state into "off" state
- from "off" state into "USB-powered" state
- from "USB-powered"state into "battery-powered" state
- from "battery-powered" state into "USB-powered" state

Figure 3.3 shows the time response of the microcontroller supply voltage on an eyesIFXv2 and on a TelosB node during two such transitions, captured using a high-speed digitizer. The nodes are connected to a Linksys USB2HUB4 hub (shown in Figure 3.2b).



(a) From "USB-powered" to "off"



(b) From "USB-powered" to "battery-powered"

Figure 3.3: Time response of the sensor node MCU supply voltage after the respective USB hub has received a port power-control command from the testbed

The results show that the transitions are tightly time bounded, with the MCU voltage dropping below the brown-out protection point of 1.8 V within 20 ms of the hub receiving the power-off command. The same tight bounds are observed for powering-on as well as for the transitions between USB and battery-powered states.

The figure shows small differences in the response between the two platforms, due to the different designs of the respective power-supply subsystems. We have also evaluated how the

time response depends on the length of the USB cabling between the node and the hub. The results (not presented here for lack of space) show that for the lengths we have in our local TWIST instance, i.e. for up to 10 m of USB cabling, the effects are negligible.

The quick time response, as demonstrated by the above results, makes this power-control capability one of the most useful features of the TWIST architecture.

It is important to note that in our testing we have detected several hubs on the market that claim to fully support the USB 2.0 standard, but fail to support the port power-control feature. This function is seldom used by normal "home" users, and it requires additional power-control circuitry in the hub, so we suspect that many vendors have decided to silently drop the support in order to drive the manufacturing costs for the hubs down. From our extensive testing of different hubs, we can conclude that the problem is present in almost all hubs having a controller chip from Genesys Logic. The ones that fully support the standard tend to have a controller chip produced by NEC. In particular, we can confirm the proper functioning of the DUB-H4 rev.A1 hub from D-Link (now out of production, the later revisions B1, B2 and B3 silently ignore the port power-control commands) and the USB2HUB4 from Linksys which is used at our local TWIST instance.

3.4 Super Nodes

If TWIST only relied on the USB infrastructure, it would have been limited to 127 USB devices (both hubs and sensor nodes) with a maximum distance of 30 m between the control station and the sensor nodes (achieved by daisy-chaining of up to 5 USB hubs). While suitable for small to medium size testbeds [16], these limitations are not compatible with our goals for scalability of the architecture and support for geographically extensive deployments.

To genuinely tackle the scalability problem, a distributed solution is needed that will spread the testbed functionality among multiple entities. These *super nodes* have to be able to interface with the previously described USB infrastructure. In addition, they have to support a secondary communication technology that does not have the size and cable length limits of the USB standard, and forms the *testbed backbone* to which the server and control stations can be attached. Finally, adequate computational, memory and energy resources are needed.

Optimally, a device is needed that can satisfy the requirements, while keeping the expenses for a medium to large-scale testbed to a reasonable level. In this sense, the class of 32-bit embedded devices used for attaching networked storage seems as a very promising alternative since they offer a very attractive cost/performance ratio. At the same time, these devices have similar capabilities as the so-called "high-end wireless sensor nodes" or "microservers" [6], enabling dual use of the super nodes as parts of the testbed and as parts of the SUE.

We are using the Network Storage Link for USB2.0 (NSLU2) device from Linksys (depicted on Figure 3.2b) as super nodes. The NSLU2 costs about 80 €. It has two USB2.0 ports, uses a IXP420 processor from Intel's XScale family (clocked at 133 MHz), has 32 MB of SDRAM and 8 MB of flash as persistent storage. One particular feature of the IXP4xx family are the two integrated "Network Processor Engines (NPE)" that implement, among else, two full Ethernet MAC and physical layer units with the related packet-processing functionality.

For testbed usage, we replace the Linksys-supplied operating system support for the

NSLU2 with a customized OpenSlug [14] distribution of Linux. OpenSlug is a variant of the OpenEmbedded [11] source distribution of Linux that is specially adapted for use on embedded devices.

Our customization is motivated by the need to change the default "self-healing" behavior of the Linux kernel when dealing with errors in the communication between the USB host controller and attached USB hubs. We rely on the user-space *libusb* library for injecting control messages that trigger the explained port power-control functions. This library, however, bypasses the USB hub drivers in the Linux kernel. When there are hubs downstream from the powered-off port (i.e. we are controlling the power of a sensor node that is connected with an active cable), this will cause loss of the keep-alive messages between the host controller and the USB hub, which in turn causes the kernel to reset and subsequently re-power the whole USB subsystem, including the sensor node that we wanted to turn-off.

The effective management of testbeds with large number of super nodes requires self-configuration capabilities. For the most basic system parameters, like the super node IP address and the address of the DNS server, we rely on the DHCP protocol. Time synchronization is achieved using NTP.

Because of the limited flash memory on the super nodes, the root and the swap file systems have to be provided over the network using NFS. To avoid maintenance of as many separate file system images as there are super nodes in the network, the UnionFS concept [21] is used to separate between a common file system part (shared by all super nodes and exported in read-only mode) and a private per-super-node part (exported in read-write mode to each super node separately).

3.5 Server

The server and the control stations must interact with the super nodes using the testbed backbone, so they have to support the same communication technology. Due to the critical role of the server (it contains the testbed database, provides persistent storage for debug and application data from the SUE, runs the daemons that support the system services in the network, etc.) its hardware resources should be adequately dimensioned to guarantee high levels of availability. For example, we are currently using an Intel Pentium 4 machine with 3.2 GHz CPU, 2 GB of RAM as well as 4x200 GB of HD space organized as RAID5.

The operating software support on the server is also based on Linux, at the moment a vanilla Fedora Core 3 server installation. For the management of the super node network, the server runs the DHCP, DNS, NTP, and NFS daemons, as well as the UnionFS kernel module.

At the heart of the server is the PostgreSQL database that stores a number of tables including configuration data like the registered nodes (identified by the NodeIDs), the sockets and their geographical positions (identified by the SocketIDs) as well as the dynamic bindings between the SocketIDs and NodeIDs. The database is also used for recording debug and application data from the SUE. An important reason for choosing PostgreSQL is the availability of the PostGIS extension that enables us to represent the locations of the sockets in a natural 3D coordinate system and provides support for spatial queries and experimentation with location-based services.

3.6 Control Station

Any normal workstation-class machine that is attached to the testbed backbone can serve as a control station. The required systems software is fairly standard, any Linux distribution can be used. The interesting aspect, however, is how the control station acts in controlling experiments, in configuring nodes and network topologies etc.

Our current solution consists of two parts. On the one hand, we have developed a number of Python scripts which run locally on the super nodes and provide functionalities like sensor node programming, executing power control, collecting debug and application data, etc. On the other hand, the actual invocation of these scripts is done by the control station using ssh remote command execution. Without further optimization, the control station would have to log onto the super nodes serially and invoke the Python scripts. Clearly, this would require a lot of time when activities involving all the nodes (like reprogramming) have to be executed.

To speed up such tasks, we use a hierarchical threading approach to exploit parallelism: the control station first creates a separate thread of control for each of the super nodes. Every such thread starts the Python scripts on the super node via the ssh remote command execution. Each of these Python scripts, in turn, create separate threads (on the super node) for each of the attached sensor nodes.

In this way, by utilizing the natural parallelism in the system, we are able to execute network wide tasks in approximately the same amount of time as it would have taken on a single sensor node. For example, programming a single sensor node with the TinyOS SurgeTelos application takes about 16 s on the TelosB nodes. For a 30 node sensor network, a linear execution would require 8 minutes. Using the parallel approach we are able to program the whole network in under 25 s, making rapid edit/compile/run cycles possible.

Chapter 4

The TWIST Instance at the TKN building

In this section we present our experiences deploying a large scale instance of TWIST in our office building. We also give one concrete example how the testbed can support a building automation related experiment.

4.1 Deployment

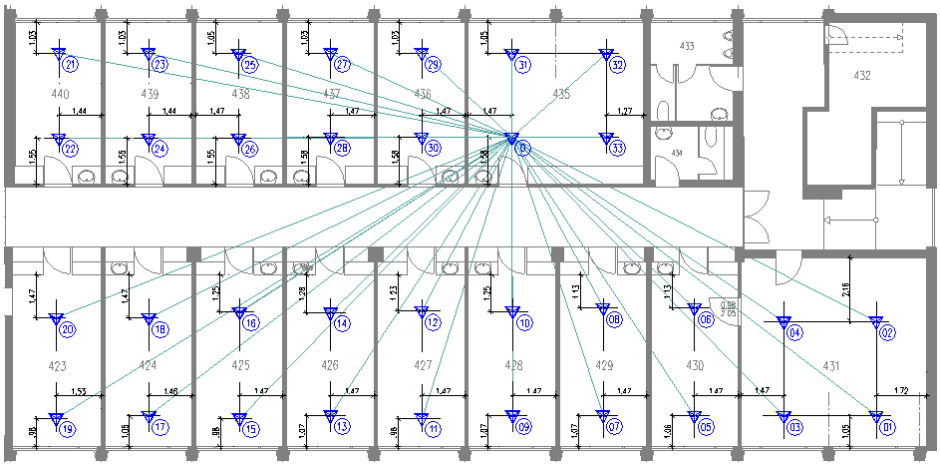


Figure 4.1: Node placement on the 4th floor

The local instance of TWIST spans three floors of our office building. Currently, we have 90 fixed locations for nodes with known positions and in addition to them 90 free slots on the USB hubs. We used 37 NSLU2s, 53 USB hubs and about 600 m of USB cables. The NSLU2s communicate over Ethernet, here we benefited from the fact that our Ethernet cabling was over-dimensioned with two Ethernet cables per room. This is a very comfortable situation, but not available everywhere. An alternative is a USB-to-WLAN adapter that can be attached to the free port of the NSLU2 and establish a backbone network using WLAN.

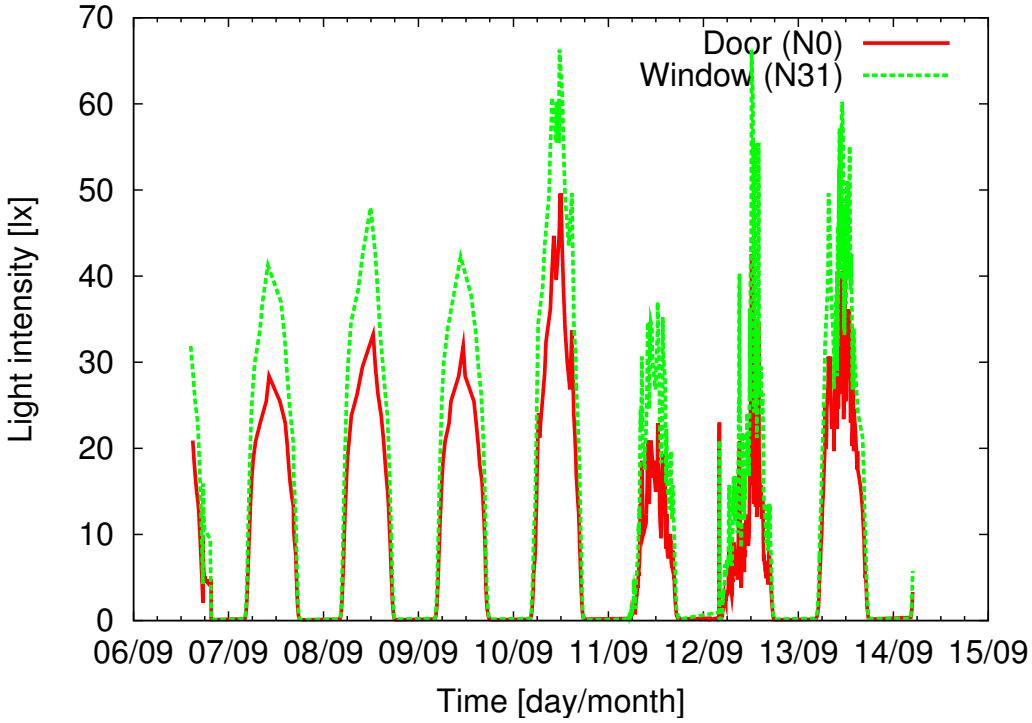


Figure 4.2: Illumination differences measured in the same room

Considering the placement, we decided to install two sockets in each room. This provides two fixed and known locations for nodes per room. Having at least two nodes per room is enough to give some insight into the spatial distribution of values like light (see Figure 4.2) and temperature, also allowing some cross-comparison of sensors of the same type on the two nodes. The node placement is shown in Figure 4.1. This figure also shows the exact location of the sockets, measured after the installation. The sockets are placed 1 m and 4 m from the window, and 1.5 m from each wall. For the large rooms, the story is a bit different. In order to keep the distances comparable, we installed four sockets in the large rooms. Any sparser, more random placement of the nodes can be emulated using the power control feature of the testbed.

The testbed is installed in our normal office building, and we did not want to let people trip over dangling wires. We hence placed the wires neatly into cable channels that are mounted at the ceiling. To connect a node to a socket, we simply connect it to the USB cable and fix it at the cable channel as shown in Figure 3.2a. This also means that the node is about 4 cm below the ceiling.

4.2 Usage example

As a usage example, we present a distributed experiment to measure the illumination rate of change of using the testbed equipped with eyesIFX nodes. Figure 4.3 shows how often the

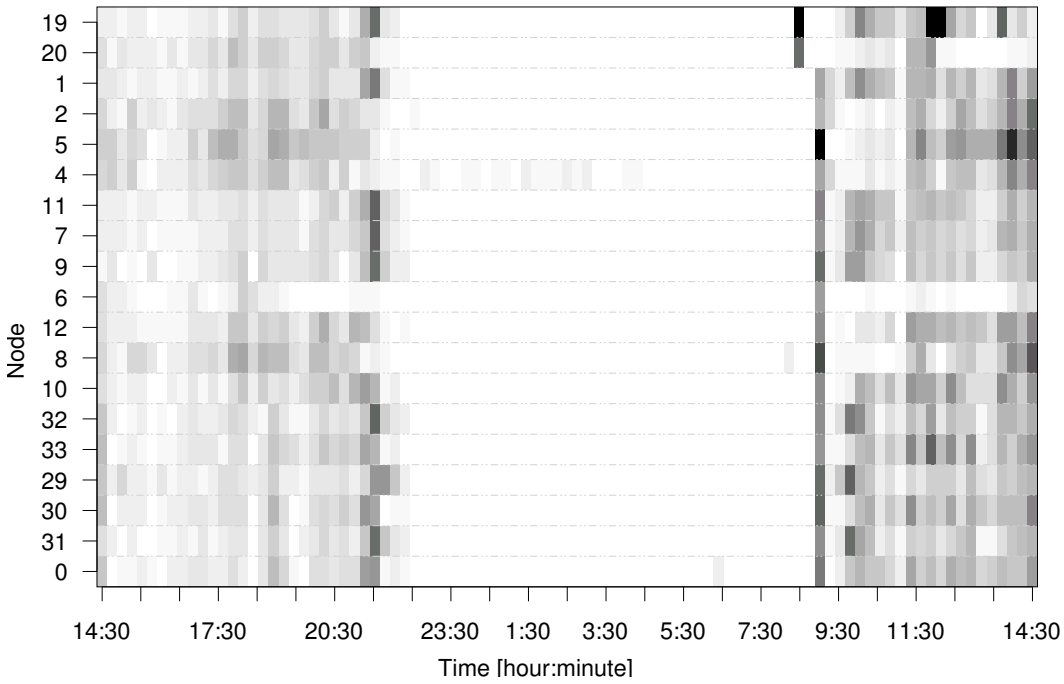


Figure 4.3: Light traffic intensity matrix, September 12th, 2005

light sensors noticed "significant" illumination changes during a single day. The measurement used a subset of the testbed on the 4th floor. The x-axis denotes the time in 15 min intervals, the y-axis uses the Euclidean distance (shown in Figure 4.1) to embed the two-dimensional placement of the nodes into one-dimension. The gray scaling indicates how many changes were observed by the light sensors, white denotes zero changes and black denotes more than 32 changes within a 15 min interval. During the night there is no change in the illumination, but that is not surprising. The plot makes the correlations in space and time of illumination changes visible. On this day, we had a thunderstorm and one flash at about 9 am was picked up by all light sensors, except those on nodes 19 and 20, which picked up an earlier flash. We did not want to loose data, hence we first stored it into the local flash and subsequently dumped it over the testbed.

Chapter 5

Related Work

MoteLab [20] is a very popular WSN testbed solution. It provides Ethernet back-channel to each sensor node in the network by attaching a dedicated Stargate board [2]. The interaction between the users and the testbed is batch-oriented and is controlled via a dynamic web interface supported by a back-end database. Like MoteLab, the Kansei testbed [18] also uses Stargate boards, but it allows richer interaction with the SUE. It uses the EmStar development system for Linux-based WSNs [5] and, similarly to TWIST, allows evaluation of both flat and hierarchical WSNs with different communication technologies. In contrast to TWIST, these testbeds use one-to-one mapping between the sensor nodes and the "concentrators". This and the custom nature of the "concentrators" make the above solutions much more expensive than TWIST.

The Omega architecture [16] resembles the lower tier of TWIST. Both testbeds rely on the standardized USB interface available on some of the newer mote platforms and use USB-hubs to bridge the 5 m length barrier. Nevertheless, the Omega is not utilizing the features of the USB 2.0 standard and is not exporting a power-control function to its users. This is one of the main features of TWIST and is providing high flexibility. The Omega design is also less scalable. The USB-hub daisy-chaining approach taken in Omega can only scale up to 127 USB devices. With the super node tier, TWIST is not suffering from such restrictions.

The TWIST architecture does not impose any policy for handling concurrent use of the testbed resources by different users. When simple "cooperative" scheduling is not sufficient, more sophisticated solutions like Mirage [1] can be employed.

Chapter 6

Conclusion

The design of TWIST presented in this paper addresses many of the needs we have identified for wireless sensor network testbeds: support for different application network architectures, control over the network topology, fast reprogramming, distributed debugging and a high degree of scalability. We have also demonstrated the feasibility of the TWIST architecture by building a large-scale testbed in our building.

TWIST is based on affordable hardware components and uses open-source software components. This makes TWIST a valuable template that other researchers can use to create their own testbeds.

In this paper we have concentrated mainly on the hardware aspects of TWIST. We currently focus on extending the software support. Our existing script-based approach is effective but lacks reusability. To attack this problem we are working on a solution that is based on an object-oriented representation of the important testbed entities and their capabilities. To hide the distributed nature of these objects we consider using a middleware that provides remote method invocation, an object repository, object persistence and other services.

Bibliography

[1] Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, and Amin Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors*, May 2005.

[2] Crossbow. Stargate. `http://www.xbow.com/Products/XScale.htm`.

[3] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proc. Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.

[4] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[5] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of USENIX General Track 2004*, Boston, MA, USA, June 2004.

[6] Lewis Girod, Thanos Stathopoulos, Nithya Ramanathan, Jeremy Elson, Deborah Estrin, Eric Osterweil, and Tom Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 201–213, New York, NY, USA, 2004. ACM Press.

[7] V. Handziski, J. Polastre, J. H. Hauer, and C. Sharp. Flexible hardware abstraction of the ti msp430 microcontroller in tinyos. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 277–278, Baltimore, MD, USA, November 2004. ACM Press.

[8] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[9] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[10] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, Chichester, 2005.

[11] Michael Lauer. Building linux distributions with bitbake and openembedded. In *Proceedings of the Free and Open Source Developers' European Meeting (FOSDEM 2005)*, Brussels, Belgium, February 2005.

[12] Suet Fei Li, Vlado Handziski, Adreas Köpke, Martin Kubisch, and Adam Wolisz. A wireless sensor network testbed supporting controlled in-building experiments. In *Proceedings of the 12th International Conference Sensor 2005*, volume 1, Nuremberg, Germany, May 2005.

[13] David L. Mills. Network time protocol (version 3) specification, implementation and analysis. RFC 1305, 1992.

[14] The openslug linux distribution. `http://www.nslu2-linux.org/wiki/OpenSlug/HomePage`.

[15] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS*, 2005.

[16] Omega testbed at uc berkeley, telos (revision b) 802.15.4 wireless sensor network. `http://omega.cs.berkeley.edu/`.

[17] Pytos development environment. `http://nest.cs.berkeley.edu/nestfe/index.php/Pytos_Development_Environment`.

[18] Ohio State University. Kansei: Sensor testbed for at-scale experiments. In *Poster, 2nd International TinyOS Technology Exchange*, February 2005.

[19] Chieh-Yih Wan, Andrew T. Campbell, and Lakshman Krishnamurthy. Psfq: A reliable transport protocol for wireless sensor networks. In *Proc. First ACM Intl. Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, 2002.

[20] Geoffrey Werner-Allen, Pat Swieskowski, and Matt Welsh. Motelab: A wireless sensor network testbed. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, 2005.

[21] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 1(4), November 2005.