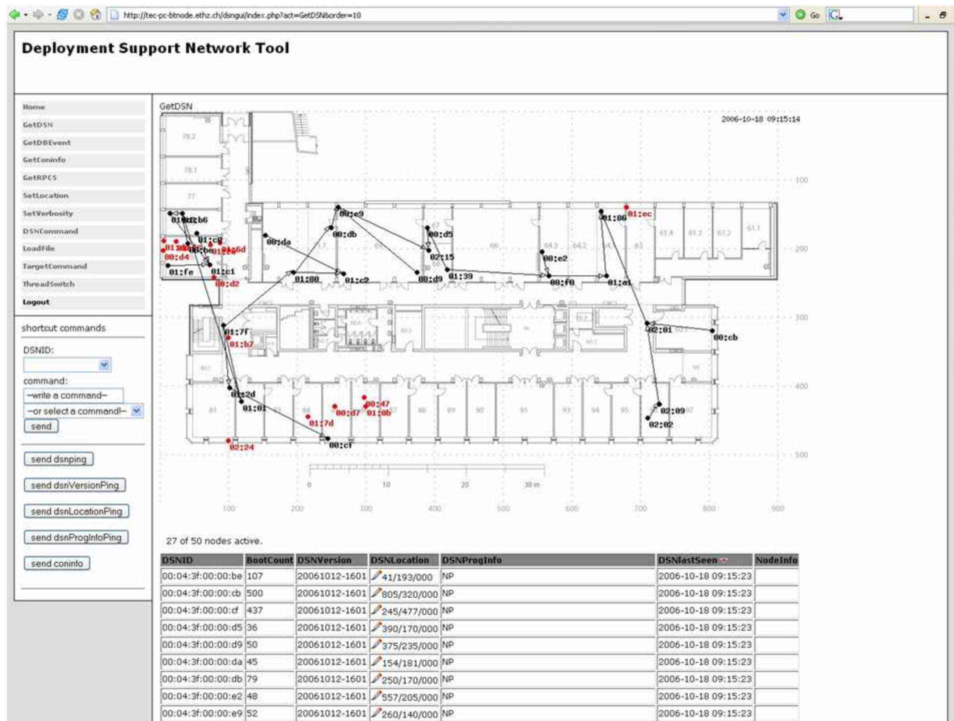
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Online Sensor Network Analysis Tools

Thomas Kalt



Masterthesis
MA-2006.17

Summer Term 2006

Betreuer: Jan Beutel
Matthias Dyer
Professor: Lothar Thiele

Contents

Abstract **vii**

1 Introduction **1**
1.1 Wireless Sensor Network 1
1.2 Deployment-Support Network 2
1.3 Jaws 3

2 Motivation **5**
2.1 Scenario 5
2.2 Requirements 5

3 Related Work **7**

4 DSN Server **9**
4.1 Concept 9
4.2 Technology 11
4.2.1 Java 11
4.2.2 MySQL 11
4.2.3 XMLRPC 11
4.3 Implementation 12
4.3.1 Communication 12
4.3.2 Parser 13
4.3.3 SQL 20
4.3.4 XMLRPC 21
4.3.5 Thread 21
4.3.6 LoggingThread.java 22
4.4 Functionalities 24
4.4.1 getDSN.getDSN 24
4.4.2 setNodeInfo.setNodeInfo 26
4.4.3 getDBEvent.getDBEvent 26
4.4.4 getRPCS.getRPCS 26
4.4.5 getConinfo.getConinfo 26
4.4.6 uploadFile.uploadFile 30
4.4.7 getFileList.getFileList 30

4.4.8 loadFile.loadFile 30
4.4.9 targetFlash.targetFlash 30
4.4.10 targetCommand.targetCommand 30
4.4.11 dsnCommand.dsnCommand 30
4.4.12 dsnVerbosity.dsnVerbosity 30
4.4.13 setDSNLocation.setDSNLocation 30
4.4.14 threadSwitch.threadSwitch 33
4.4.15 threadSwitch.get 33
4.4.16 getServerTime.getServerTime 33
4.4.17 user.user 33
4.5 Installation 35
4.5.1 Requirements 35
4.5.2 Installation 36
4.5.3 Server.properties 36

5 Web Based DSN Tool 43
5.1 Functionalities 43
5.2 Installation 45
5.2.1 Requirements 45
5.2.2 Installation 45
5.2.3 properties.php 46

6 Transport Performance Tests 47
6.1 Scenarios 47
6.2 Performance Evaluation 48
6.2.1 Sending Performance 48
6.2.2 Push Log Stream 49
6.2.3 Pull Log Stream 50
6.2.4 Push Log Burst with Retransmission 56
6.2.5 Pull Log Burst 57
6.2.6 Conclusion 58

7 Conclusion 63
7.1 Summary 63
7.2 Future Work 64

A XMLRPC Examples 65
A.1 Java Code 65
A.2 PHP Code 67

B Problem Task 71

List of Figures

1.1 The concept of the DSN 2
1.2 The graphical user interface for the DSN 4

4.1 Concept of the DSN 9
4.2 Concept of the DSN Server 12
4.3 Flow chart of the CmdParserLogging 1/2 17
4.4 Flow chart of the CmdParserLogging 2/2 18
4.5 Flow chart of the CmdParserRPCS 19
4.6 Linear interpolation of DSNTimetoTime 20

5.1 Web based DSN Tool 44

6.1 Log generating modes 48
6.2 Measured yield of correctly received log messages in push log stream mode 49
6.3 Used model of pull log stream. 51
6.4 Plot of the pull log stream simulation with the settings r=13 l=0,5 d=1 c=25 52
6.5 Plot of the pull log stream simulation with the settings r=13 l=1 d=1 c=25 53
6.6 Plot of the pull log stream simulation with the settings r=13 l=1,5 d=1 c=25 54
6.7 Plot of the pull log stream simulation with the settings r=13 l=2 d=1 c=25 55
6.8 Measured yield of correctly received log messages in push log burst with retransmission mode 56
6.9 Illustration of the pull log burst method. 57
6.10 Both methods of the log stream model. 59
6.11 Both methods of the log burst model. 60

List of Tables

4.1 The pattern that the Parser knows. 14
4.2 The header of a log message. 15
4.3 Parameter and Return of getDSN.getDSN 24
4.4 The fields of the table dsninfo 25
4.5 Parameter and Return of setNodeInfo.setNodeInfo 26
4.6 Parameter and Return of getDBEvent.getDBEvent 27
4.7 The fields of the table dsnlog 28
4.8 Parameter and Return of getRPCS.getRPCS 28
4.9 The fields of the table dsnrpcs 29
4.10 Parameter and Return of getConinfo.getConinfo 29
4.11 The fields of the table dsnconinfo 31
4.12 Parameter and Return of uploadFile.uploadFile 31
4.13 Parameter and Return of getFileList.getFileList 31
4.14 Parameter and Return of loadFile.loadFile 31
4.15 Parameter and Return of targetFlash.targetFlash 32
4.16 Parameter and Return of targetCommand.targetCommand . 32
4.17 Parameter and Return of dsnCommand.dsnCommand 32
4.18 Parameter and Return of dsnVerbosity.dsnVerbosity 32
4.19 Parameter and Return of setLocation.setLocation 34
4.20 Parameter and Return of setLocation.setLocation 34
4.21 The threads can be turned ”on” or ”off” with XMLRPC threadSwitch.threadSwitch 34
4.22 Parameter and Return of threadSwitch.get 34
4.23 Parameter and Return of getServerTime.getServerTime . . . 35
4.24 Parameter and Return of user.user 35
4.25 The fields of the table dsnpwd 35
4.26 Variables of the ”Server.properties” under the section ”General Settings” 37
4.27 Variables of the Server.properties under the section ”Threads” (Part 1/3) 38
4.28 Variables of the Server.properties under the section ”Threads” (Part 2/3) 39

4.29 Variables of the Server.properties under the section "Threads" (Part 3/3) 40
4.30 Variables of the Server.properties under the section "RXTX-Connections" 40
4.31 Variables of the Server.properties under the section "SQL Database" 41
4.32 Variables of the Server.properties under the section "Paths" . . 41
4.33 Variables of the Server.properties under the section "Target Commands" 41

5.1 Variables of the properties.php. 46

Abstract

In this master thesis a server was implemented for the use of the Deployment Support Network. With an easy interface of the DSN Server it is possible to attach and construct specific tools and GUI's for developing a sensor target network using the DSN.

The concept of the separation of the target layer, that includes the target sensor node and a GUI, and the DSN layer that includes the DSN Server and the DSN nodes is presented. Additionally the thesis gives an overview of the functionality of the DSN Server, therefore a target sensor network developer is able to attach or construct his own GUI.

Transport performance tests are made to see what the traffic maximum is that a DSN is able to process. Furthermore the tests give a user a basis to evaluate how many log messages in what time slices his sensor network can produce and how he should collect them.

Chapter 1

Introduction

The use of sensors in our life is increasing very fast. They are used to build smart environments, that can interact with human actions, to give alarm of environmental perils in good time or to observe environment changes. To observe wide areas without interfering with the environment, we need small autonomous sensor devices, which communicate wireless with each other. Such devices are termed Wireless Sensor Network (WSN).

1.1 Wireless Sensor Network

A WSN consists of up to thousands of sensor nodes, which are spread over a spacious area to collect different data without affecting the surrounded environment. A sensor node is a small computer including a radio and some sensors. In order to be used in hardly accessible, wide-spread areas, they often have their own power supply, for example in form of a battery. These sensor nodes organize themselves in a wireless ad hoc network to collect and process the measurement data in order to provide high-level sensing results.

It is quite difficult to develop and test a WSN due to the following reasons:

- Generally, debugging an embedded system is quite hard. Because of the small size and the resulting lack of energy and resources, there are not many possibilities for debugging.
- Resulting in the limited battery power the nodes are often in a low power mode, where they aren't able to communicate. Therefore the only way to find out at any time in which state they are, is a direct connection to the node.
- Sometimes it is not possible to use wired connections for debugging, for example if the test is placed in a real environment. Therefore, the sensors must contain additional functions, which generate extra debugging output. In return these functions can disturb the test because of

additional CPU time and memory, which the additional debugging functions need.

- If the sensor nodes send extra debugging messages over the radio, there could be an interference with the normal traffic of the WSN.

- Simulation is not the same as testing in real environment, because it is impossible to simulate the real world.

1.2 Deployment-Support Network

An interesting idea for deploying and developing a WSN is discussed in [1]. The idea in this paper is called Deployment-Support Network (DSN). The idea of a DSN is to replace the serial connection, which is used for getting debugging information out of special pins from the sensor to the PC, with a wireless connection. Over such a connection it is possible to have features including remote programming, debugging, monitoring and target control.

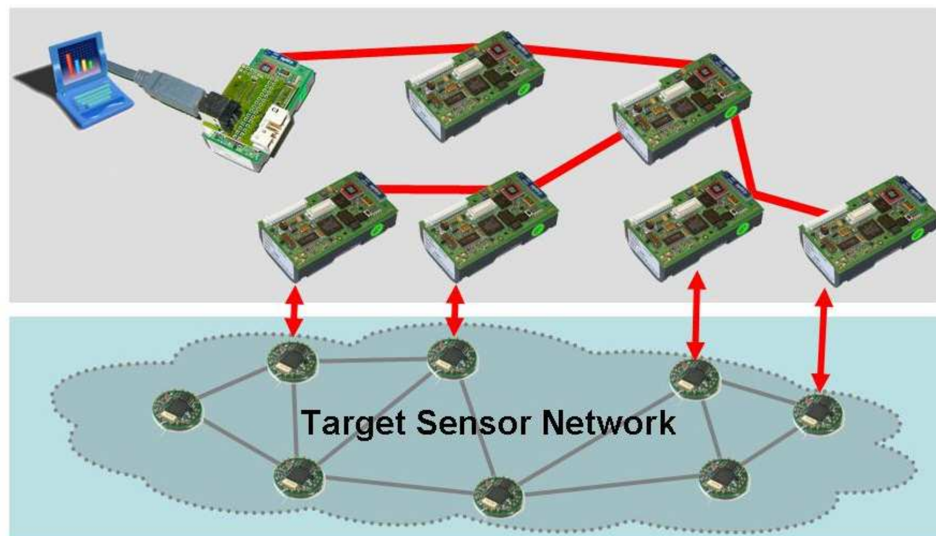


Figure 1.1: The concept of the DSN

To get rid of the wired connection from the PC and the sensor node, the sensor node is attached to another node, which can be placed next to the sensor node. Both, the DSN and the sensor node network build their own independent wireless networks. Therefore there exists a possibility to connect from a host, which is connected to a DSN node, to every other DSN node. Over the connection from the PC through the multi hop network of the DSN, it is possible to communicate with each node of the DSN. Therefore

the debugging traffic goes through the wireless network of the DSN, instead of a direct wired connection between a PC and a sensor node.

At the ETH Zürich a DSN is now operating. It is based on the BTnode platform. This first implantation of a DSN is described in [2].

1.3 Jaws

Jaws is the implementation of a DSN at ETH Zürich. It is running on the BTnode rev3 platform running NutOs, a multi-thread operating system for embedded devices. The Bluetooth device is used to autonomously interconnecting the DSN nodes. There are two different network topologies that can be established, a tree-based topology or a mesh based topology that is called XTC [3]. The Jaws implementation includes features for remote programming, logging, monitoring and target control.

A simple Graphical User Interface (GUI) is used to control the features of the DSN. In Figure 1.2 is a plot of this GUI.

With the GUI there exists a user friendly possibility to reprogram the sensor node targets or the DSN nodes, to get the logs from a DSN node or to send a command to a sensor node or a DSN node. Additionally, there is a feature implemented to display the connections between the DSN nodes.

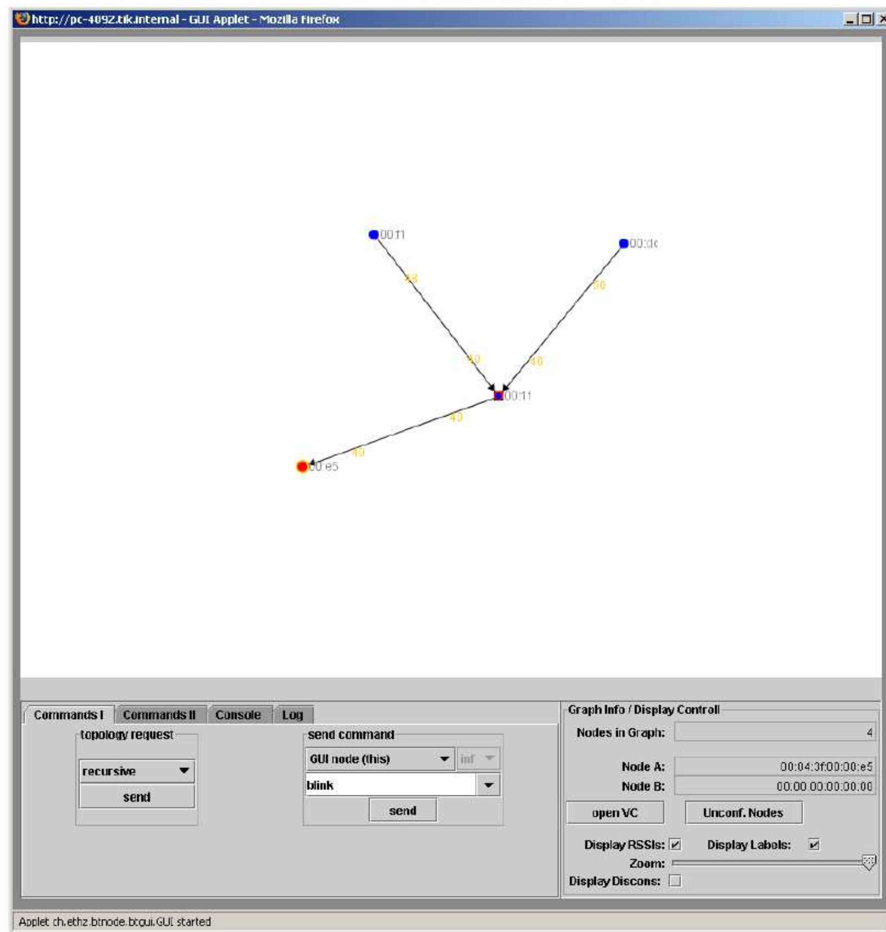


Figure 1.2: The graphical user interface for the DSN

Chapter 2

Motivation

2.1 Scenario

A typical scenario of a sensor network, which is descriptive to illustrate what is needed for developing it, is a wireless fire sensor network. A wireless fire sensor network is distributed in a building. Therefore the nodes aren't accessible very good, even if they are attached to the roof. It would be quite difficult to connect them with a wire to a PC to control them, because the wires would go through the whole building. For this purpose the use of the DSN can eliminate these wires. Next to every fire sensor node a DSN node is placed and it is connected to the fire sensor node. Therefore a sensor node developer has all the options, which are mentioned in Chapter 1.2.

During the developing phase it must be possible that the nodes can be easily reprogrammed. For monitoring, the sensor nodes can write log messages to the log buffer of the DSN nodes. These log messages should be accessible to several users at any time.

During a test the sensor node developer wants to send commands to different sensor nodes, for example to simulate a link break, that a fire sensor breaks or that a fire is detected on a node. In order to do the same tests several times, a test scenario has to be automated.

2.2 Requirements

The GUI that was mentioned in Chapter 1.3 is design for developing the DSN and not especially for the target sensor node network. Every sensor network has its own requirements and features therefore it should be possible to implement a specific target user interface without a huge knowledge of the DSN.

To create automated tests very easy, there has to exist a simple interface to access all the features of the DSN.

The DSN has a bottleneck that is given by the node that is directly

connected to the PC. All incoming and outgoing messages have to go over this node. Because several users would like to access the DSN at the same time, sending of messages must be coordinated. Due to the fact that the log messages are the largest amount of the traffic on the DSN, it is reasonable to store the log messages central that the log messages have to be requested once of the DSN nodes and can be distributed to different users several times at any time.

Because several users are able to access the DSN, there have to be an authentication that not everybody is able to reprogram the nodes over the internet or to send the nodes commands.

Chapter 3

Related Work

At UC Berkley there are three different testbeds [4] implemented and in operation. The *sMote* testbed consists of 78 Mica2DOT sensor motes, which are connected over ethernet for debugging and reprogramming. The *Omega* testbed consists of 28 Telos, which are connected to USB for debugging and reprogramming. The *Trio* testbed, an outdoor testbed which consists of 500 Trio motes temporary deployed in the wild at UCB's Richmond Field Station. sMote has its own web interface for uploading the program to the building-wide network.

The Harvard University implemented the *MoteLab* [5] that includes 190 TMote Sky sensor motes. They were connected to Ethernet for debugging, logging and reprogramming. MoteLab provides a public, permanent testbed where registered users can upload online executables for testing them on the MoteLab. During a test, all messages and other data are logged to a database that is accessible to the test owner.

There are several other test beds that are used for research. Most of them have a wired connection for power supply and reprogramming, for example over a Ethernet cable.

Another implementation of distributed sensors is *EarthScope* [6]. This is a collection of different geological sensors in North America. It is an interesting implementation for collecting different data to provide high-level sensing results and to manage a huge data pool. A part of the data out of this huge data collection is available through a web based interface.

SeNeTs [8] is a software environment to test senor network software on independent hosts such as PCs or evaluation boards. *SeNeTs* contains two communication channels, one for the sensor network application and the other for controlling and logging data. A server manages the communication

between test scripts and the node applications.

Chapter 4

DSN Server

4.1 Concept

Cause of the experience with running a DSN over a year and using the GUI, there were some good ideas to improve the usability of the DSN. Some of the problems were already mentioned in the Chapter 2.

In Chapter 2 is mentioned that for different implementation of sensor networks different GUI's are needed. Due to the fact that different target user interfaces will exist, we decided to make a clear separation between the sensor network, including analysis tools and GUI's and the target sensor nodes, and the DSN layer, including the DSN Server and DSN nodes. This separation of concerns is shown in Figure 4.1.

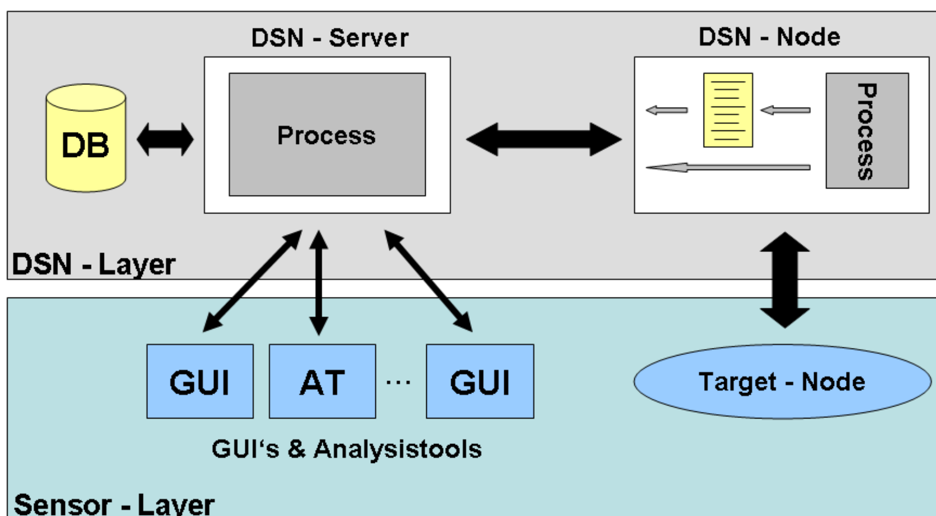


Figure 4.1: Concept of the DSN

The DSN layer consists of a DSN server that is directly connected to one DSN node, which is called GUI node. Over this connection the DSN server has the ability to send commands over the GUI node through a multi hop distribution to every other DSN node. A main task of the DSN server is to collect information, for example the log messages, regularly from the DSN nodes and stores them into a database. This has the advantage that no information gets lost if a node breaks. Moreover it is a method to save the slow connection between server and node, so every message must be sent just once and can afterwards be distributed several times over a fast Ethernet connection to a GUI and possible multiple users. All incoming and outgoing data is passing the GUI node, therefore the server needs a good strategy to safe this bottleneck.
The DSN Server provides some functions for the GUI's or analysis tools to get information about the sensor target nodes or the DSN nodes. Furthermore it is possible to send commands to every node by sending the command to the server that distribute the command to the write node without that the developer have to know the specific DSN commands.

4.2 Technology

In order to enable users to use the DSN Server on almost every platform and without great requirements, the server is built on widely used and free available technologies.

4.2.1 Java

The DSN server is written in Java, due to the fact that we had already some experiences with the communication over a serial port in Java. Additionally Java is an easy to use object-oriented programming language and there exist a lot of different packages to use that were needed for some technologies the DSN server needs.

4.2.2 MySQL

MySQL is a multithreaded SQL database with more than six million installation. It is available as free software under the GNU and there exists API's for different programming language to access to a MySQL database, including Java.

4.2.3 XMLRPC

The protocol XMLRPC is used for the communication between the DSN server and the GUI. It is a very simple protocol that can be printed out in two pages. XMLRPC uses http for transporting and XML to encode the calls. There are a lot of API's for different program languages, therefore GUI's can be written in almost any programming language.

4.3 Implementation

The DSN Server consists of 5 different parts that interact with each other, what is shown in the Figure 4.2. The communication module is responsible for the communication with the GUI node, it is reading and writing on the serial port. All incoming traffic form the GUI node is processed by the parser, it checks if the incoming line corresponds with one of the pattern it knows. The information won out of the incoming traffic is stored in a database. All the communication with the database is done by the SQL module. The XMLRPC module enables the communication with GUI's, therefore it provides some functionalities that are explained in Chapter 4.4. Thread is a module, which requests regularly information from the BTnodes.

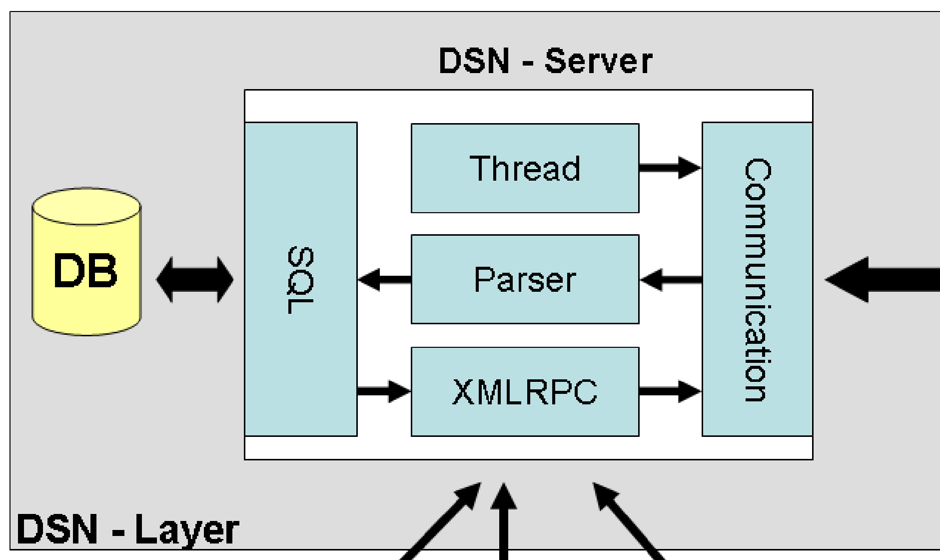


Figure 4.2: Concept of the DSN Server

4.3.1 Communication

The communication module is responsible for the communication with the GUI node. Communication.java is an interface for this module and defines the functions that are used. At the moment there is just one implementation of this interface available. RXTXComm.java is the used implementation of the interface Communication.java. The communication is done over the serial port. The function writeLine(command) is for writing something out of the serial port to the GUI node. This function has to be protected with a semaphore to guarantee that not more than one function is writing to the serial port simultaneously. This is very important, if a file is uploaded,

because during this procedure the BTnode would interpret every other line as a line of the .hex file as well and corrupt the data. The BTnode detect that the data is corrupted and stops the receiving process, therefore the upload procedure would fail. Additional the RXTXComm.java writes every incoming and outgoing line into the file msglog.txt.

4.3.2 Parser

The Parser is the module, which is checking every incoming line for some special patterns. For every known pattern a class exists that implements the interface CmdParser. If a pattern is recognized the corresponding CmdParser class is processing the line. A line is a string which is ending with a "\n".

The Parser.java is a thread that is started at the beginning. All the CmdParsers are instantiated in this class and are linked to the corresponding pattern. The parser thread is checking every line that is coming from the GUI node, if a pattern occurs, it passes the line to the corresponding CmdParser. Table 4.1 lists, which CmdParser is responsible for which line.

Parser.java provides two functions for every CmdParser that are used to force the Parser.java to pass the next arrived line to a special CmdParser.

SetParserState(CmdParser) this function sets a variable which indicates that the Parser passes the arrived line to the declared CmdParser without checking for the patterns.

resetParserState() resets the variable.

This functions are used if message consists out of several lines, therefore the CmdParser has the ability to say the Parser that the following lines belongs to the previous line and has to parsed by the same CmdParser. For example this functionality is used with a log message. A log message consists of a header, which starts with :SL, several lines of Data and is terminated with an endtag.

CmdParserDSNID.java / CmdParserVersion.java / CmdParserLocation.java / CmdParserDSNProgInfo.java

These CmdParsers parses the content out of the lines and update the corresponding fields of the table dsninfo including the field lastseen.

CmdParserGetAddr.java

This function is responsible for the answer of the command "addr", which the server writes directly out on the serial port and receives the address of the GUI node. CmdParserGetAddr.java provides to variables DSNnodeAddr

pattern	CmdParser	line
:DP	CmdParserDSNID.java	:DP <addr> <boot_count>
:DV	CmdParserVersion.java	:DV <addr> <version>
:DL	CmdParserLocation.java	:DL <addr> <location_str>
:XP	CmdParserDSNProgInfo.java	:XP <addr> PI <type> <version> <size> <name> :XP NP
Local	CmdParserGetAddr.java	Local bt_addr: <addr> (error code=0)
:T	CmdParserConinfo.java	:T <source-addr> <num-entries> :TE <neighbor addr 1> <con-state><rssi> :TE <neighbor addr 2> <con-state> <rssi> ...
:SL	CmdParserLogging.java	:SL <addr> <msg_id> <frag_nr> <dsn_time> <class> <level> <data> {<breaktag> or <endtag>}
:RP	CmdParserRPCS.java	:RP <addr> <data> {<breaktag> or <endtag>}
ready	CmdParserLoadFile.java	ready to receive hex data, press enter for quit
:LH	CmdParserLoadFile.java	:LH completed: <number> lines read :LH failed: <reason>
:TP	CmdParserTargetFlash.java	:TF OK :TF FAILED
:DT	DSNtimetoTime.java	:DT <bt_count>

Table 4.1: The pattern that the Parser knows.

and DSNpreAddr. DSNnodeAddr is the address string of the GUI node. Due to the fact that some functions on the BTnode returns only the last four bytes of the address, DSNpreAddr provides the first eight bytes that are the same on all BTnodes at the moment that the server can complete a short address.

CmdParserConinfo.java

It parses the coninfo replies that gives information, which nodes are directly connected and store it to the table dsnconinfo (see Table 4.11).

CmdParserLogging.java

All the Log messages are stored in the database, therefore CmdParserLogging parses the logs and stores the log messages to the database.

Log Header :SL <addr> <msg_id> <frag_nr> <dsn_time> <class> <level> <data> <breaktag or endtag>

The BTnode wraps every log messages with a header and a tag that indicates the end of the package. The following information, which is listened in the Table 4.2, is stored in the Header:

header	**explanation**
<addr>	The address of the BTnode that generated the log message.
<msg_id>	This is a increasing sequence number for every log. This number is set back to one after a reboot of the BTnode.
<frag_nr>	If a log message is longer than 86 Bytes, the log message has to be sent in several packages. The fragment number indicates, which part of the log message it is.
<dsn_time>	The BTnode time, when the log message was generated.
<class>	Class of the log message
<level>	Level of the log message

Table 4.2: The header of a log message.

On the next line after the header the <data> is starting. The data can consist out of several lines, but is 86 bytes long at most. The end of data is marked with one of the two tags, breaktag (0x03) or endtag (0x04). A breaktag indicates the end of the package but not the end of the log that means, that there will follow an other package with data that belongs to the same log message. This happens if the log message was longer than 86 bytes, therefore the log message will be split and sent in several packages. An endtag indicates that it is the last package of this log message.

Algorithm Due to the fact that it is possible that a log message can be split in several packages and packages can be lost, the CmdParserLogging must do some checks for every log package. The chain of checks of the CmdParserLogging is shown in Figure 4.3 and in Figure 4.4. CmdParser-Logging.java stores for every DSN node some variables, these has to be known to understand the flow chart.

Log Hashtable CmdParserLogging stores the header data and the fragment of the log message into the hashtable log and writes the log message into the database as soon as all fragments have arrived. (keys of the hashtable: DSNID, MsgID, FragNr, DSNtime, Class, Level, LogText)

Received List A List of the received MsgNr Request List A List of MsgNr, which have to be requested again.

Requested Hashtable it shows which MsgNr was requested how many times (is not used in CmdParserLogging.java)

Following assumption are made:

- If a package arrives with a fragment number that is not equal to the fragment number of the last package plus one, it is assumed that the package with the fragment number, which is one greater than the last package number, is lost, rather than the package order is out of order.

- If a log message with a higher MsgNr from the same BTnode arrives before the package with the endtag of the lower MsgNr arrives, it is assumed that the packages are lost of the log message with the lower MsgNr, rather than they could be overtaken by other messages.

CmdParserRPCS.java

The answer of a remote procedure call string (rpcs) command is parsed by the CmdParserRPCS.java and the parsed message is stored in the table dsnrpcs (see Table 4.9). An answer can be split in several packets, if it is too long for one packet. An answer can contain several lines. The end of a packet is marked with an endtag (0x04) or a breaktag (0x03). The endtag means that this was the last packet of this answer and the breaktag indicates that there will follow one more packet at least. If the whole message is stuck together, the Parser.java checks the message on known pattern, and starts the corresponding CmdParser. There is a flow chart of CmdParserRPCS in Figure 4.5.

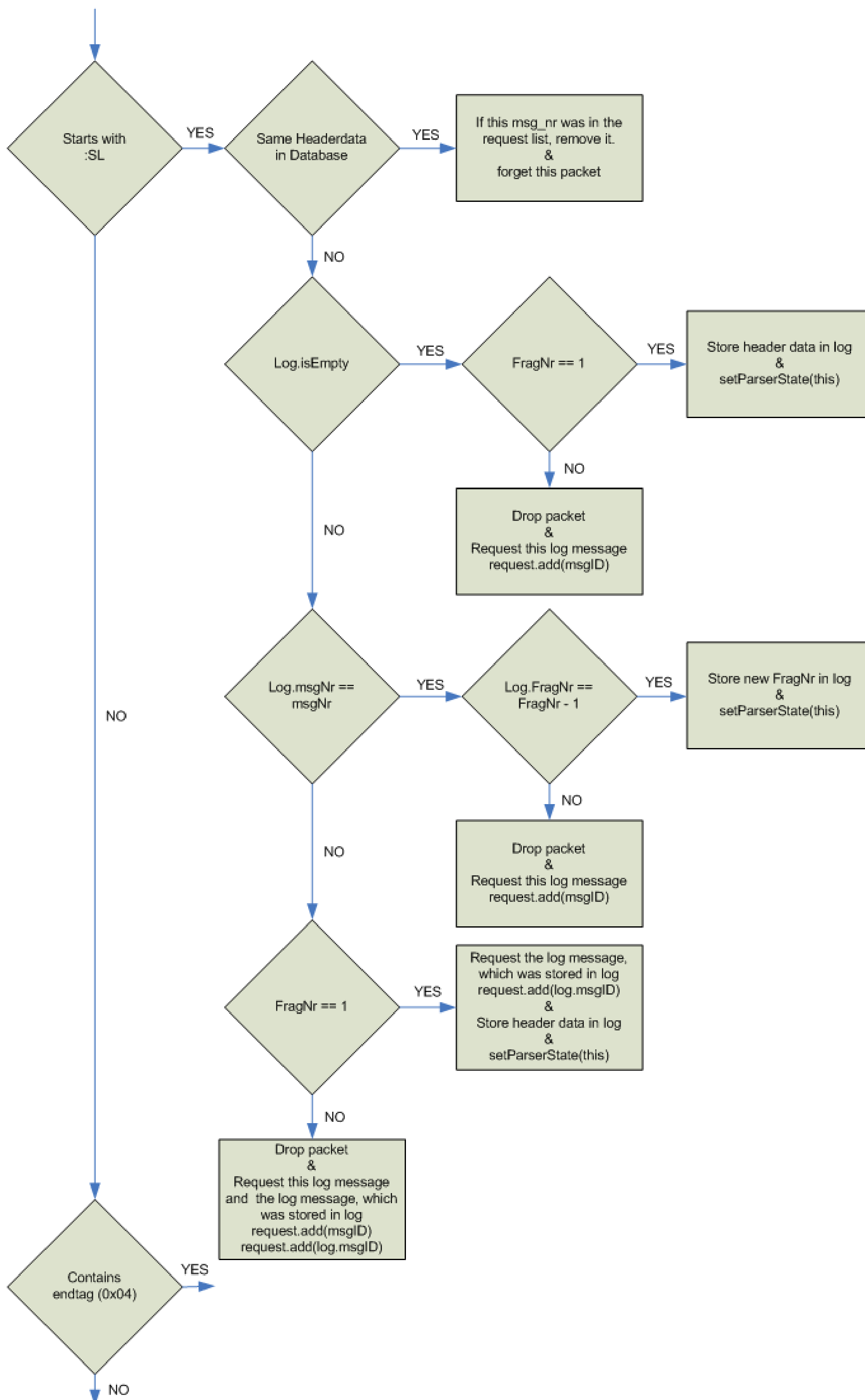


Figure 4.3: Flow chart of the CmdParserLogging 1/2

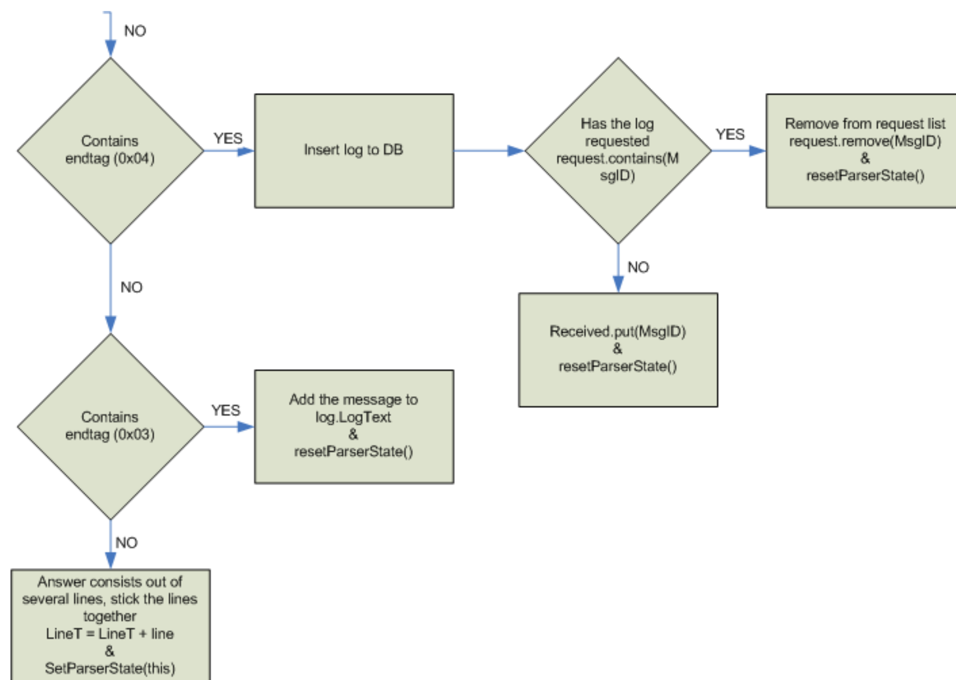


Figure 4.4: Flow chart of the CmdParserLogging 2/2

CmdParserLoadFile.java

The DSN provides the possibility to upload and distribute a .hex file to reprogram the target nodes or the BTnodes. Therefore the DSN server has to upload the .hex file to the GUI node. Before the server starts with sending the lines of the .hex file, it has to wait till the GUI node is ready to receive, hence CmdParserLoadFile.java is parsing the ready message and starts the send process afterwards. During the send process can happen some errors, if one of them occurs the CmdParserLoadFile.java parses them as well and stops the send process.

CmdParserTargetFlash.java

The BTnode sends back a message after trying to flash the target node. CmdParserTargetFlash parses this message and passes the line to the XmlRpcTargetFlash function.

DSNTimetoTime.java

The Btnode counts up a variable after a reboot, this tic count is used as internal time on the BTnode, for example for the log messages. DSNTimetoTime.java provies a function, which converts the BTnode time to a real

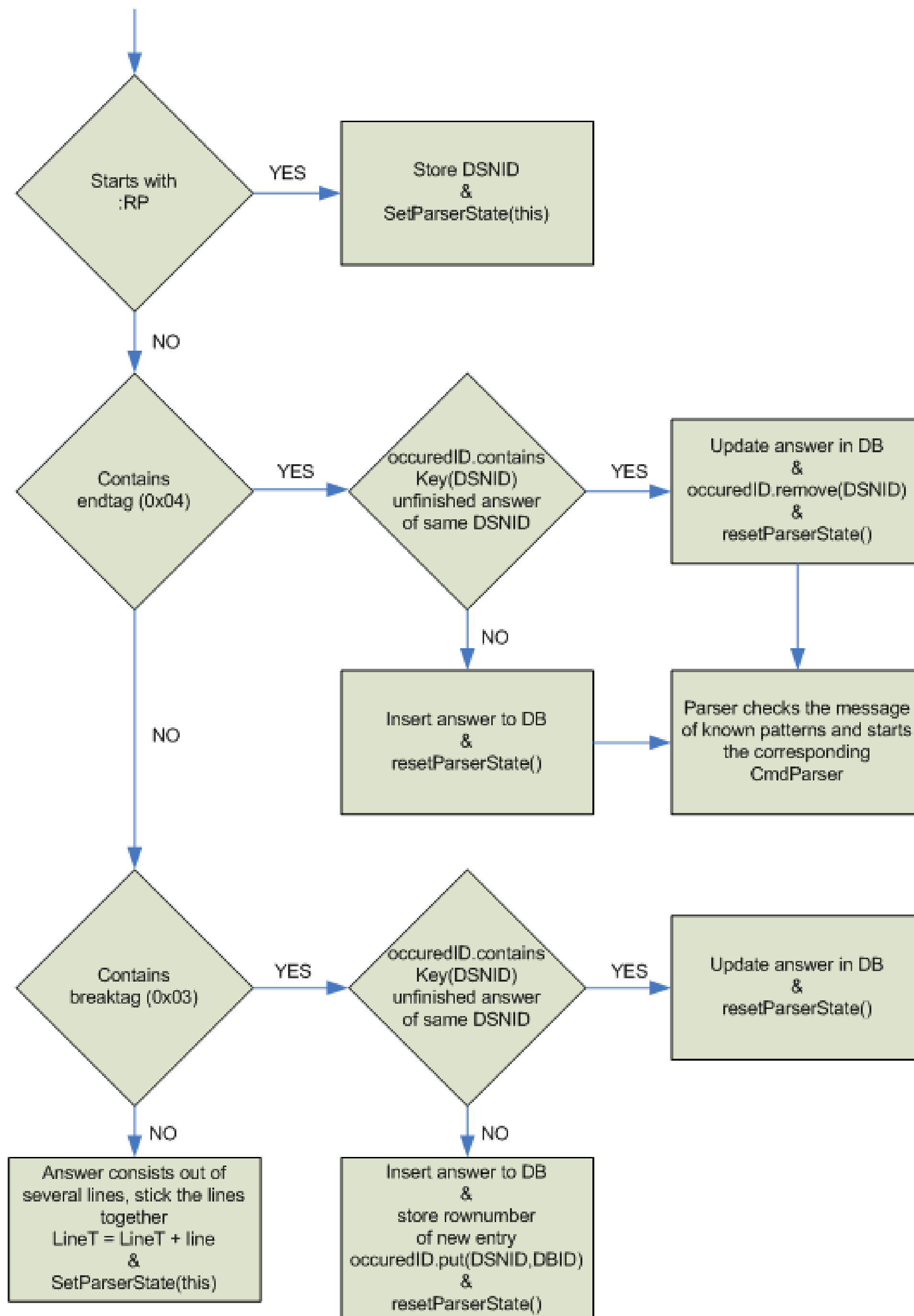


Figure 4.5: Flow chart of the CmdParserRPCS

time. This functionality makes only sense if the BTnodes are synchronized. Therefore the DSN Server requests regularly the BTnode time from the GUI node and stores them together with the real of the request. To convert a time, DSNTimetoTime interpolate or extrapolate linearly the time to the stored time, like it is shown in Figure 4.6.

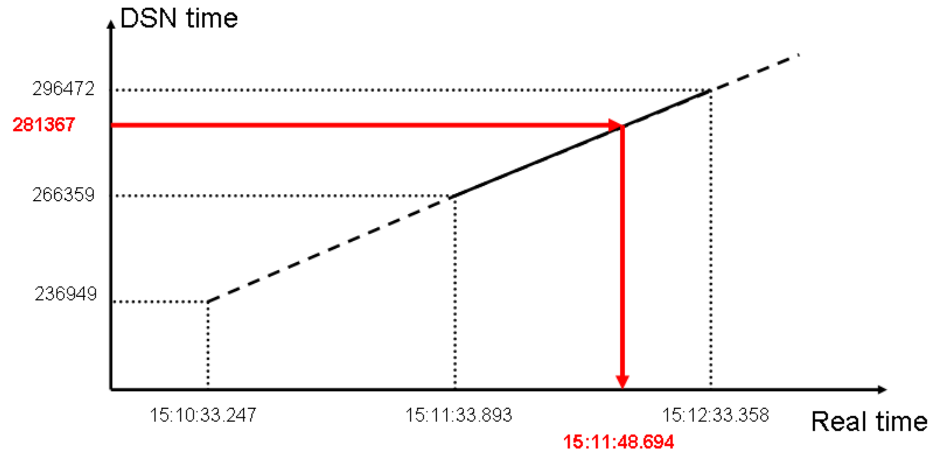


Figure 4.6: Linear interpolation of DSNTimetoTime

4.3.3 SQL

SQLDBCommunication.java

SQLDBcommuniction provides some functions to use the MySQL Database:

getResultfromDB(query) makes a query on the database, is used for SELECT statements

sendQuerrytoDB makes a query on the database, is used for the statements INSERT, DROP TABLE, CREATE TABLE

connectToDB opens a connection to the database

closeConnectionToDB closes the connection

makeTable makes all tables that the DSN Server needs, if they don't exist

lock acquire a semaphore

unlock release the semaphore

Java is not able to close a SQL statement with the garbage collector during the connection is open. A SQL statement object is used to execute a query. Therefore the server can do only one query at same time. The server must wait till the result is not used any more until the server can do the next query. Hence every resultset must be locked with a semaphore until it is not used anymore.

4.3.4 XMLRPC

The XMLRPC module is responsible for the communication between the server and every GUI. We use the Apache XMLRPC Version 2. The class XmlRpcServer.java starts the XMLRPC server on a specific port that can be defined in "Server.properties" (see Table 4.26). Furthermore the XMLRPC commands are specified in here as well.

Several XMLRPC commands need an authentication to use them. They are listed in Chapter 4.4.17. Each of these have a function checkPassword(user, password), it checks, if there exists an user with the corresponding password in the table dsnpwd and if this user is authorized for this specific function. In the field function of the table dsnpwd (see Table 4.25) can be the name of the function or "all", if the user has access to every XMLRPC function.

If you would like to implement a new XMLRPC function, you have to add a new Handler to the server in the file "XmlRpcServer.java" with a name that defines how the class of the new programmed function can be accessed through XMLRPC.

4.3.5 Thread

Threads are used in the DSN Server to request specific information regularly.

ThreadCommand.java

The ThreadCommand.java is a class to generate a thread, which sends a command on the serial port to the GUI node and waits a defined time till it sends the same command again. The time, the thread is waiting till it will send the command again, is defined in the file "Server.properties" (see Table 4.27, Table 4.28 and Table 4.29). The ThreadCommand.java is used for the following commands:

- "rpc 0 dsn ping" (to get a sign of life of every DSN node)
- "rpc 0 dsn version" (to get the version string of the DSN node, which indicates, which version is running on the BTnode)
- "rpc 0 dsn loc get" (to get the location string of every DSN node)

- "rpc 0 xbank get proginfo" (to get the program info of the .hex file which is on the BTnode to reprogram the Target node or the BTnode)

- "coninfo" (to get information about the connections between the BTnodes)

- "jaws time" (to get the tic count of the GUI node)

Every thread can be interrupted with the XMLRPC threadSwitch.threadSwitch. In the Server.properties it is defined, if the thread should be interrupted at the start of the DSN Server, but they can be started later with the XMLRPC threadSwitch.threadSwitch as well.

4.3.6 LoggingThread.java

The LoggingThread.java is responsible for requesting the log messages from the nodes. There are two different modes how the LoggingThread.java can work, pull mode or push mode. In the pull mode the server requests the logs of the BTnodes and detects, if a log message is missing and requests these logs again. In the push mode the logs are not requested, because the BTnodes send the logs automatically to the GUI node, but the server detects the missing logs as well and requests these logs. Thus the only difference between these two modes is, that in the push mode the request function, which requests every log, is not running but the same function, which is requesting the lost logs, is running in both modes. In the file "Server.properties" is defined in which mode the Server is working (see Table 4.29). This thread can be stopped with the XMLRPC threadSwitch.threadSwitch as well. It is important, that the thread, which sends out the dsn pings, is running, otherwise the server isn't requesting any logs.

To request the logs in the pull mode, the server takes all nodes, which have answered to the last dsn ping, out of the dsninfo table and stores these node addresses in a list. Afterwards the server goes through this list and requests for every node all new logs. Therefore the server knows from every node which is the log with the highest message ID, the node sent. The CmdParserLogging.java stores every received message ID for every node in a list. The LoggingThread assorts this list and detects the numbers which are missing in the sequence and requests these missing logs again. It is defined in the file "Server.properties" how many time the server should request the missing log till it gives up and writes into the file "msglog.txt" that this log is missing. If the start sequence is lost, the server is not able to request these logs, because the server can't know with which message ID the Btnode starts. The same is with a sequence of the last logs, if there no other log is following, the server will not detect that these logs are missing, because it doesn't know that these logs are sent.

The sever starts with the first node in the list of all nodes and sends the

request for the newer logs and the requests for the missing logs to the corresponding BTnode. The BTnode will process this requests and sends the requested logs back to the server. After a time D, which is defined in the file "Server.properties"(logging, see Table 4.28), the server haven't received any log messages of this node, the server assumes that the Btnode has finished the requests and take the next node in the list and does the same requests. If the server has finished the list, it builds up the list with all nodes, which has answered to the last dsn ping, again and starts the same process.

The difference between the pull and the push mode is, that in the push mode the server doesn't request any logs from the BTnode, but the server generates a list with the received message IDs as well. Therefore in both modes the server requests the lost messages.
In the push mode it is assumed that the BTnodes sends there log messages immediately to the server. Therefore the verbosity mask of the BTnode should be set "on" of all log levels, this can be done with the XMLRPC function dsnVerbosity.dsnVerbosity (see Chapter 4.18).
In the push mode the Server goes to the next node for requesting the missing log after a time D, which is defined in the file "Server.properties"(pushtime Table 4.28).

4.4 Functionalities

The DSN-Server provides some XMLRPC functions that can be accessed through the internet by using a normal XMLRPC client. There are a lot of implementations of XMLRPC in different languages. It doesn't depend which programming language is used to do an XMLRPC request to the DSN Server. The implemented XMLRPCs are specified below, including parameters and returned values. In Appendix A there are two code examples in different languages to do and process a XMLRPC request.

4.4.1 getDSN.getDSN

Parameter:	int order (0 - no order, 1 - (ascending) DSNID, 2 - (descending)DSNID, 3 - (ascending) DSNVersion, 4 - (descending) DSNVersion, 5 - (ascending) DSNLocation, 6 - (ascending)DSNLocation, 7 - (descending) DSNProgInfo , 8 - (descending) DSNProgInfo, 9 - (ascending) DSNlastSeen, 10 - (descending) DSNlastSeen)
Return:	Hashtable result (Keys: DSNID, BootCount, DSNVer, DSNLocation, DSNProgInfo, DSNlastSeen)

Table 4.3: Parameter and Return of getDSN.getDSN

The DSN Server generates a table in the database that includes all the DSN nodes, which were ever seen on the network. The fields of this table are explained in Table 4.4. The server stores additionally to every DSN address some information, which are explained in Table 4.4.

To update this table, the DSN-Server queries regularly a ”sign of live” of the nodes. The server sends out four different requests, a normal ping, a version request, a location request and a request to update the ProgInfo. With every answer of such a request the DSNlastSeen field is updated as well. The time of the iteration of each request is declared in the file ”Server.properties”. It is possible to switch those requests ”on” or ”off” per default in the file ”Server.properties” or switch them during runtime ”on” or ”off” with the XMLRPC function switchThread.switchThread, which is explained in Chapter 4.4.14.

field	explanation
DSNID:	The address of the BTnode.
BootCount:	The BootCount is a number, which is stored in the eeprom on the BTnode. This number increases every time if the BTnode reboots.
DSNVer:	The program version, which is currently running on the DSN node. It is a timestamp of the compilation time.
DSNLocation:	This is a string, which is stored in the eeprom on the BTnode and should give an information about the location, where the BTnode is located. It can be set by the XMLRPC setDSNLocation.setDSNLocation and it is a string of 16 Bytes.
DSNProgInfo:	The BTnode has the ability to reprogram itself or to reprogram the connected Target. DSNProgInfo declares the program which is stored on the BTnode for this functionality and looks something like this "PI 1 0608311052 32364 senso.btnode3.hex". The first number after "PI" declares whether the program is for the DSN node (0) or if it is for the Target node (1). It is followed by an identification number, which is used by the BTnode to see, if a program on an other BTnode is newer and therefore it is worth to update the saved program. Normally it is a timestamp (yymmddhhmm). The third number is the size of the program in byte. The last string is the name of the program, there is only 20 byte left for this string so it could happen that it is not the original string the file had.
DSNlastSeen:	It is a timestamp, that is updated, whenever an answer of a ping arrived.
NodeInfo:	It is a string that can be set with the XMLRPCSetNodeInfo. This string should give additional information about the Node and it is stored in the database only.

Table 4.4: The fields of the table dsninfo

4.4.2 setNodeInfo.setNodeInfo

Parameter:	String DSNID	DSNID of the node about which the information is.
	String NodeInfo	Information about the node. (255 byte)
Return:	int return	The number of affected rows in the database. (Should be 1)

Table 4.5: Parameter and Return of setNodeInfo.setNodeInfo

This XMLRPC is to store additional information about the node in the table dsninfo (see Table 4.4).

4.4.3 getDBEvent.getDBEvent

Due to the concept that no logs should get lost, the Server requests the log messages of the BTnode and writes them into a table in the database, the fields of the table are shown in Table 4.7. This XMLRPC gives back the log messages out of the database, which were requested, according to the set parameters that are explained in Table 4.6. In order to produce not too much traffic the DSN Server requests only the latest logs of the BTnode, which aren't yet in the table. The time of the iteration of each request is declared in the file "Server.properties". In the file "Server.properties" is defined, if the log requests should be done or not as well.

4.4.4 getRPCS.getRPCS

The DSN provides a function to send a command through the DSN to a specific BTnode, where the command sent will be executed. This function is called remote procedure call sting (rpcs). If the command, which was sent with the rpcs function to a BTnode, generates an answer, this answer will be wrapped into a specific packet and sent back to the server. All this answers are stored in the table dsnrpcs. getRPCS.getRPCS gives back the entries out of the table dsnrpcs.

4.4.5 getConinfo.getConinfo

To evaluate how stable the DSN network is, the DSN Server requests information about the connection of every node and store it to the table in the database that is explained in the Table 4.11. With this XMLRPC you can get the entries out of this table according to the set parameters. The time of the iteration of each request is declared in the file "Server.properties" and if the requests should be done, is declared in the same file as well.

Parameter:	String DSNID	
	int levelmask	a mask in decimal values that is generated out of set bit values (x01-Error x02-Warning x03-Info x04-Debug) (set all = 15, set Error & Warning = 3)
	int logclass	which log class you would like to get (0 = all classes)
	String logtext	it is looking for this String in the logtext in the DB, if the String is empty "", it gives back all rows which correspond with the other clauses
	String time_from	to get all messages that are newer or equal than this value (yyyy-MM-dd HH:mm:ss)
	String time_till	to get all messages that are older or equal than this value (yyyy-MM-dd HH:mm:ss)
	int order	how the entries should be ordered (0 - DBTime; 1 - DSNID; 2 - MsgID, DSNID; 3 - DSNtime, DSNID; 4 - time, DSNtime; 5 - class, DBtime; 6 - Level, DBtime; 7 - LogText, DBtime, DSNID)
	String ascdesc	if it should be ordered ascending or descending ("asc" or "desc")
	int start	it gives back the entries after this row
	int maxReply	the number of messages that you get, maxReply = 0 is to get all back
Return:	Hashtable result	Keys: DSNID, DSNtime, time, Level, DBtime, LogText

Table 4.6: Parameter and Return of getDBEvent.getDBEvent

field	explanation
DSNID:	The address of the BTnode.
MsgID:	The BTnode gives every log message a number. After a reboot the BTnode starts with one and increase the number on every log message and goes up to 65535.
DSNtime:	It is a tick count of the BTnode that is reset after reboot of the BTnode.
time:	It is the calculated real time out of the DSNtime. It works only if there is a time synchronization between the GUI node and the rest of the nodes and the CommandThread jawstime must run.
Class:	The Class of the log message.
Level:	The Level of the log message. There exists the following log levels: 1 = E: Errors, 2 = W: Warnings, 3 = I: Info, 4 = D: Debug (If none of the above levels is detected the message will be classified per default with Class = 18 and Level = 3 (Info))
DBtime:	The time when the log messages was stored into the database.
LogText:	The log message.

Table 4.7: The fields of the table dsnlog

Parameter:	String DSNID	
	String msg	The server looks for this string in the msg field in dsnrpcs table in the database, if the string is empty "", it gives back all rows that corresponds with the other clauses
	String time_from	To get all messages that are newer or equal than this value (yyyy-MM-dd HH:mm:ss)
	String time_till	To get all messages that are older or equal than this value (yyyy-MM-dd HH:mm:ss)
	int maxReply	The number of messages that you get, maxReply = 0 is to get all back and will be ordered ascending, if you specify maxReply the order will be descending and will start with the latest entry
Return:	Hashtable result	Keys: DSNID, time, msg

Table 4.8: Parameter and Return of getRPCS.getRPCS

field	**explanation**
DSNID	The address of the BTnode that sent the answer
Time	Time when the answer was written to the database
Msg	The answer of the sent command

Table 4.9: The fields of the table dsnrpcs

Parameter:	String Source_DSNID	The DSNID of the BTnode that sent this message to the server
	String Neighbor_DSNID	The DSNID of the BTnode that is the neighbor of the BTnode that sent this message
	String State	0: visible/not connected 1: neighbor is slave 2: neighbor is master 3: connection error all: to get all four states
	String time_from	To get all messages that are newer or equal than this value (yyyy-MM-dd HH:mm:ss)
	String time_till	To get all messages that are older or equal than this value (yyyy-MM-dd HH:mm:ss)
	int maxReply	The number of messages that you get, maxReply = 0 is to get all back
Return:	Hashtable result	Keys: time SourceAddr, NeighborAddr, State, rssi

Table 4.10: Parameter and Return of getConinfo.getConinfo

4.4.6 uploadFile.uploadFile

Before a hex file can be distributed in the DSN network, it has to be uploaded to the DSN-Server. In order to be able to send the file with XMLRPC to the server, it has to be converted to base64 at first. Base64 describes an encoding system to encode 8bit binary data into a string that consists out of ASCII characters only. It is allowed to upload files with a suffix .hex or .ihex using uploadFile.uploadFile.

4.4.7 getFileList.getFileList

To see which files are already on the DSN-Server.

4.4.8 loadFile.loadFile

A file, that is already on the DSN-Server, can be sent to the GUI node. When the file is correctly transmitted, the GUI node will distribute the file to the next nodes that will do the same. The parameter "type" specifies, if the uploaded file is for reprogramming the DSN node or the Target node.

4.4.9 targetFlash.targetFlash

After a file was uploaded with the XMLRPC loadFile.loadFile and the type was set to 1, the target node can be reprogrammed by flashing it. The command to flash a target is not the same for every Target application, therefore it has to be declared in the file "Server.properties".

4.4.10 targetCommand.targetCommand

To send a command to a target through the DSN.

4.4.11 dsnCommand.dsnCommand

To send a command to a DSN node.

4.4.12 dsnVerbosity.dsnVerbosity

The DSN node has the ability to send back special log messages immediately back to the GUI node. This has to be declared for every DSN node.

4.4.13 setDSNLocation.setDSNLocation

There is reserved 16 byte on the eeprom of the BTnode to safe a location string. This location string should give some hints on the location of this DSN node. Every string can be used, but it is recommended to use following format: xxx/yyy/zzz. This format is needed to plot the map in the DSN Tool that is introduced in Chapter 5.1.

field	**explanation**
time:	When it was stored to the database
SourceAddr:	The address of the BTnode that sent back the information
NeighborAddr:	The address of the BTnode that the sender of this message is connected to.
State:	Declare which state the neighbor node has. 0: visible/not connected 1: neighbor is slave 2: neighbor is master 3: connection error)
rssi:	It is the received signal strength. (range: -1 (good) to -127 (bad))

Table 4.11: The fields of the table dsnconinfo

Parameter:	String filename	The name, how the file should be stored in the folder hex/. Should not be longer than 20 Byte. (if this name already exists in the folder hex/, the file will be overwritten.)
	String base64	The file to upload. It has to be converted into base64.
Return:	String filename	The name of the stored file in the folder /hex

Table 4.12: Parameter and Return of uploadFile.uploadFile

Parameter:	none	
Return:	Hashtable result	Key: Filename

Table 4.13: Parameter and Return of getFileList.getFileList

Parameter:	String filename	The name of the file to upload to the GUI node. The file has to be already in the folder /hex.
	String type	0 = file is to reprogram dsn nodes 1 = file is to reprogram target nodes
Return:	Vector result	Key = "err" (the value of transmission_err)

Table 4.14: Parameter and Return of loadFile.loadFile

Parameter:	String DSNID	DSNID of the BTnode to which the command should be sent ("all" to send it to all nodes)
Return:	String msg	It is set to the output of the command "tg flash", if the command is not send to all nodes, otherwise it gives back "unknown". After a time, which is declared in "Server.properties" (Table 4.26), it gives back unknown as well.

Table 4.15: Parameter and Return of targetFlash.targetFlash

Parameter:	String DSNID	DSNID of the BTnode to which the command should be sent ("all" to send it to all nodes)
	String command	The command that will be sent to the target, must be shorter than 110 bytes
Return:	String return	If the command could be sent: "OK!" if the command was to long: "too long!"

Table 4.16: Parameter and Return of targetCommand.targetCommand

Parameter:	String DSNID	DSNID of the BTnode to which the command should be sent ("all" to send it to all nodes)
	String command	The command for the DSN node
Return:	String return	It returns an empty string ("")

Table 4.17: Parameter and Return of dsnCommand.dsnCommand

Parameter:	String DSNID	DSNID of the BTnode to which the command should be sent ("all" to send it to all nodes)
	int Class	Which class should be sent immediately back to the server
	int Levelmask	Which level of the class should be sent back immediately, it is a mask in decimal values that is generated out of set bit values (x01-Error x02-Warning x03-Info x04-Debug) (set all = 15, set Error & Warning = 3)
Return:	String return	it returns an empty string ("")

Table 4.18: Parameter and Return of dsnVerbosity.dsnVerbosity

4.4.14 threadSwitch.threadSwitch

There are running seven different threads on the server. Each of them is interruptible with the XMLRPC threadSwitch.threadSwitch. These threads can be interrupted per default in the file "Server.propertie", however the threads can be started afterwards with the XMLRPC threadSwitch.threadSwitch as well. All thread, which can be turn "on" or "off", are listened in the Table 4.21.

4.4.15 threadSwitch.get

ThreadSwitch.get returns "true" if the thread is running and "false" if the thread was stopped.

4.4.16 getServerTime.getServerTime

It gives back the internal time of the server.

4.4.17 user.user

Some of the XMLRPC commands need a username and a password to use them. The server has a table with all registered users that table is shown in Table 4.25.

For every function, the user is allowed to use, there is a row in the table with the name of the function. If the user is allowed to use every function, there is only one row with the entry "all" in the field function.

The XMLRPC user.user gives back all rows that correspond with the given username and password. The following XMLRPC functions need a password authentication:

- uploadFile.uploadFile
- loadFile.loadFile
- targetFlash.targetFlash
- targetCommand.targetCommand
- dsnCommand.dsnCommand
- dsnVerbosity.dsnVerbosity
- setDSNLocation.setDSNLocation
- threadSwitch.threadSwitch
- threadSwitch.get
- setNodeInfo.setNodeInfo

Parameter:	String DSNID	DSNID of the BTnode to which the command should be sent ("all" to send it to all nodes)
	String location	The location string that is stored to the eeprom of the BTnode (16 byte)
Return:	String return	It returns "OK!"

Table 4.19: Parameter and Return of setLocation.setLocation

Parameter:	String thread	The name of the thread which has to be switch "on" or "off" (logging, dsnping, dsnversion, dsnlocation, dsnproginfo, coninfo, jawstime)
	String state	"on" or "off"
Return:	String return	"on" or "off"

Table 4.20: Parameter and Return of setLocation.setLocation

field	**explanation**
logging	The thread that requests the logs from every BTnode. Before the server stops this thread it terminates the started round robin of the nodes and request the logs of the remaining nodes in the current round.
dsnping	The thread that sends a ping
dsnversion	The thread that requests periodically the program version, which is running on the BTnode
dsnlocation	The thread that requests the location of the BTnode
dsnproginfo	The thread that requests the information of the program, which is stored on the BTnode for reprogramming the BTnode or the target node
coninfo	The thread that requests the connections of every BTnode
jawstime	The thread that requests the ticcounts of the GUInode. This time is used to convert the ticcounts to a real time.

Table 4.21: The threads can be turned "on" or "off" with XMLRPC threadSwitch.threadSwitch

Parameter:	none
Return:	Hashtable result (Keys: LoggingSwitch, DSNPingSwitch, DSNVersionSwitch, DSNLocationSwitch, DSNProgInfoSwitch, ConinfoSwitch, JawsTimeSwitch)

Table 4.22: Parameter and Return of threadSwitch.get

Parameter:	none	
Return:	String time	(yyyy-MM-dd HH:mm:ss)

Table 4.23: Parameter and Return of getServerTime.getServerTime

Parameter:	String - username	
	String - md5pwd	password encrypted with the md5 algorithm
Return:	Hashtable result	Key: Function

Table 4.24: Parameter and Return of user.user

field	**explanation**
name	username of the user.
pwd	the password of the user, it is stored with a md5 encryption
function	the function the user is allowed to use

Table 4.25: The fields of the table dsnpwd

4.5 Installation

There are some requirements that are needed to run the DSN Server, but all of them are free available on the internet for the common operating systems.

4.5.1 Requirements

Concurrent Versions System (CVS) A CVS client is used to to checkout a complete copy of the project dsn_server.

Java Development Kit (JDK) JDK is used to compile the Java code of the DSNServer. It is recommended to use the jdk1.5.0 or higher.

Apache Ant Ant is used to build the project, like it is described in the file build.xml.
An installation instruction is available online on the Apache Ant Homepage [7].

MySQL Database MySQL is used as database server.
An easy installation of a MySQL Database can be done with XAMPP. XAMPP is a free software package containing the Apache HTTP Server, MySQL database, PHP, phpMyAdmin and other useful tools. An HTTP Server and the PHP module have to be installed to use the DSN Tool (see Chapter 5.1). And the phpMyAdmin tool is an useful extension for the administration of MySQL over a web browser.

4.5.2 Installation

1. Check-out with a CVS client the module "proj/dsn_server" from "btnode.cvs.sourceforge.net:/cvsroot/btnode".

2. Edit the file "Server.properties" in the main directory of the DSNServer. The settings are explained in the Chapter 4.5.3

3. Create a MySQL database with the name "BTnode" (This is the default name, which is set in the file "Server.properties", certainly the name can be changed)
   SQL-Statement: "CREATE DATABASE ‘BTnode‘ ;"

4. Create a user with all data and structure privileges on the BTnode database, use the same credentials as in the file "Server.properties".
   SQL-Statement: "GRANT SELECT , INSERT , UPDATE , DELETE , CREATE , DROP , INDEX , ALTER , CREATE TEMPORARY TABLES ON ‘BTnode‘ . * TO 'username'@'localhost' IDENTIFIED BY 'password';"

5. Compile the server with the commando "ant server"

6. Start the server
   Two different commands exists depending on the system used:

   - "ant run-server" for windows
   - "ant run-server-linux" for unix

7. Create a new XMLRPC user

   - Open the file src/testserver/NewUser.java and enter a new user and password with appropriate permissions (set the function to true or set "all" to true if the user have access to every XMLRPC function)
   - Run again "ant server" for compilation
   - Run "ant new-user"

4.5.3 Server.properties

In the following tables the variables of the Server.properties are listed.

name	value	explanation
truncate_Table	("yes" or "no")	If the tables in the database should be cleared, when the server starts.
reboot_node	("yes" or "no")	If the server should reboot all nodes in the network, when the server starts.
logFile	("yes" or "no")	If the value is set to "yes", the server puts all incoming and outgoing lines into the file "msglog.txt", which is placed in the root directory of the DSNServer.
deletLog	("yes" or "no")	The Server deletes the file msglog.txt if it is set to "yes".
clearLogBTnode	("yes" or "no")	If the server should send all the BTnodes in the network the command "log clear" to delete all log messages in the BTnode log buffer, when the server starts.
FlashTimeout	[milliseconds]	How long the server should wait for the acknowledgment of the flash process of the target. If the server doesn't receive an acknowledgment of the BTnode, the server gives back "unknown" to the XMLRPC request targetFlash.targetFlash.
xmlrpcPort	(between 1024 and 65535)	The port the XMLRPC-Server is listening to

Table 4.26: Variables of the "Server.properties" under the section "General Settings"

name	value	explanation
JawsTime	(”yes” or ”no”)	If the thread, that requests the DSNTime of the GUI node, should start, when the Server starts. The thread can be started later with the XMLRPC function threadSwitch.threadSwitch as well.
TimeSync	(”yes” or ”no”)	If a command should be sent to the GUInode that starts the time synchronization of the BTnodes.
getTime_Time	[milliseconds]	How long the interval of the thread, which requests the DSNTime of the GUI node, should be.
DSNPing	(”yes” or ”no”)	If the thread, which sends out a ”dsn ping” to all nodes in the network to get a sign of life, should start, when the server starts. The thread can be started later with the XMLRPC function threadSwitch.threadSwitch as well.
getDSN_Ping	[milliseconds]	How long the interval of the thread, which sends out a ”dsn ping” to all nodes in the network, should be.
DSNVersion	(”yes” or ”no”)	If the thread, which sends out a ”dsn version” to all nodes in the network to get there version string, should start, when the server starts. The thread can be started later with the XMLRPC function threadSwitch.threadSwitch as well.
getDSN_Version	[milliseconds]	How long the interval of the thread, which sends out a ”dsn version” to all nodes in the network, should be.
DSNLocation	(”yes” or ”no”)	If the thread, which sends out a ”dsn loc get” to all nodes in the network to get there location string, should start, when the Server starts. The thread can be started later with the XMLRPC function threadSwitch.threadSwitch as well.

Table 4.27: Variables of the Server.properties under the section ”Threads” (Part 1/3)

name	value	explanation
getDSN_Version	[milliseconds]	How long the interval of the thread, which sends out a "dsn loc get" to all nodes in the network, should be.
DSNProgInfo	("yes" or "no")	If the thread should be started that sends out a "xbank get proginfo" to all nodes in the network to get there programm info string of the programm which is stored on the BTnode to reprogramm the BTnode or the target node. The thread can be started later with the XMLRPC function threadSwitch.threadSwitch as well.
getDSN_ProgInfo	[milliseconds]	How long the interval of the thread, which sends out a "xbank get proginfo" to all node in the network, should be.
Coninfo	("yes" or "no")	If the thread, which sends out a "coninfo" to the GUI node to find out which nodes are connected, should start, when the Server starts. The thread can be started later with the XMLRPC function threadSwitch.threadSwitch as well.
getDSN_coninfo	[milliseconds]	How long the interval of the thread, which sends out a "coninfo" to the GUI node, should be.
LogThread	("yes" or "no")	If the LoggingThread that is explained in Chapter 4.3.6 should start, when the Server starts. The thread can be started later with the XMLRPC function threadSwitch.threadSwitch as well.

Table 4.28: Variables of the Server.properties under the section "Threads" (Part 2/3)

name	**value**	**explanation**
logMethod	("pull" or "push")	If the LoggingThread that is explained in Chapter 4.3.6 should run in the "push" or "pull" mode. Both modes are explained in the Chapter 4.3.6.
logging	[milliseconds]	This time is only used in the pull mode. If the server receives no log messages during this time, the server assumes that the BTnode has finished the requests and the server requests the log messages of the next node. This time depends on the estimated Round Trip Time of the used network.
pushtime	[milliseconds]	This time is only used in the push mode. After the server has received the last logs of a node, the server waits this time until he starts to request the missing logs of the next node.
numRequest	[Integer]	How many times the server should try to request a missing log message.

Table 4.29: Variables of the Server.properties under the section "Threads" (Part 3/3)

name	**value**	**explanation**
portName	[String]	The port to which the GUI node is connected. On a Windows system it is something like "COM3" and on a unix system something like "ttyUSB1"
baudrate	[bit/sec]	This is the rate, the server can write to the GUI node. This is defined in the jaws on the BTnode. Normally it is 57600.

Table 4.30: Variables of the Server.properties under the section "RXTX-Connections"

name	value	explanation
mySqlDriver	[String]	The JDBC driver (JDBC driver for MySQL = "com.mysql.jdbc.Driver")
mySqlUrl	[bit/sec]	The url to the database server (e.g.: "jdbc:mysql://localhost/BTnode" (the name after the last "/" is the name of the used database))
mySqlUser	[bit/sec]	The username, which has access to the database.
mySqlPasswd	[bit/sec]	Corresponding password to the username.

Table 4.31: Variables of the Server.properties under the section "SQL Database"

name	value	explanation
hexPath	[String]	A folder where the server can store the uploaded .hex files. The server must have writing permissions on this folder.

Table 4.32: Variables of the Server.properties under the section "Paths"

name	value	explanation
tgFlash	[String]	The command, which is used to flash a target.

Table 4.33: Variables of the Server.properties under the section "Target Commands"

Chapter 5

Web Based DSN Tool

The web base DSN Tool was developed to substitute the GUI, which was shown in Figure 1.2. Additionally we could show, how easy it is to develop a new GUI that is interacting with the DSNServer over the distributed XMLRPC interface.

The DSN Tool is written in php, because it is an easy to use web oriented programming language. Therefore we have the ability to access the tool with a web browser from every computer with internet access.

The functions, that are explained in Chapter 4.4, are implemented into the DSN Tool. Due to the fact that some XMLRPC functions need an authentication to use them, you have to login to the tool with username and password of a XMLRPC account to use all functionalities.

At the moment the web based DSN Tool can be used only with the web browser FireFox, this is the only web browser that we have tested with the tool.

The DSN Tool is shown in the Figure 5.1.

5.1 Functionalities

In this section we explain shortly some features of the DNS Tool

GetDSN In Figure 5.1 we can see, what is shown under GetDSN. It shows a picture with all the nodes on it and the corresponding links between two nodes.
The background picture can be changed, therefore you have to place into the root directory of the DSN Tool a picture in the jpg-format with the name "Bild.jpg".
The picture is recreated, if someone access GetDSN and the last picture is older than the time "coninfoUpdate" (Table 5.2.3) that is defined in the file "properties.php". The generated pictures are saved in the folder img/ and is named "conimg_YYYYMMDDHHmm.jpg".

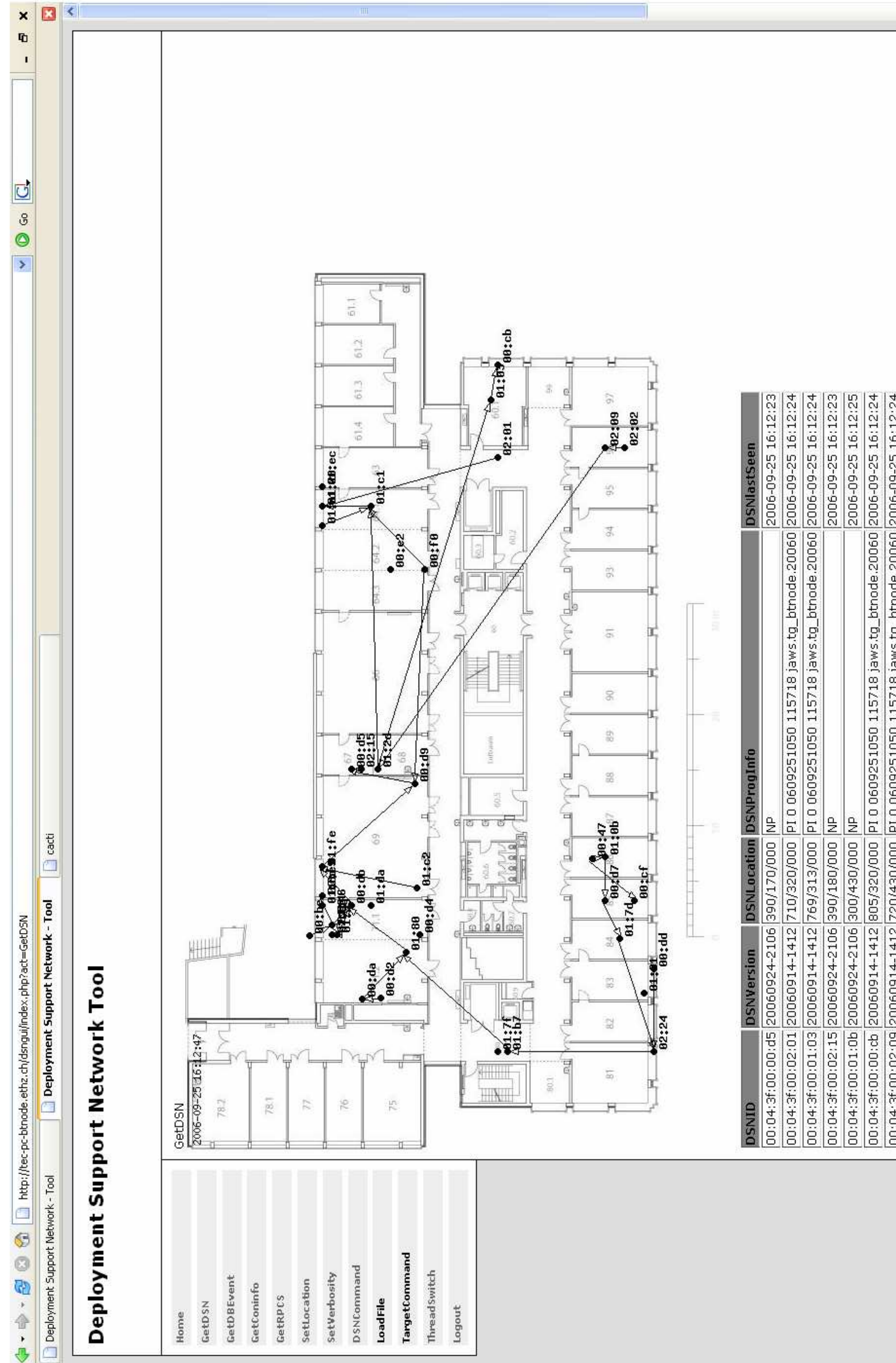


Figure 5.1: Web based DSN Tool

The Tool displays always the latest picture.
The arrows on the picture points from the slave node to the master node. This information is taken from the last entries out of the table dsnconinfo (see Table 4.11). If an arrow is red, it means that only one entry of both node is in the database. This can happen if only one node means that he has a connection to the other node, or a coninfo packet hasn't reached the server.
If a node in the picture is red that means that the node hasn't respond to the last ping request.
Under the picture there is a list with all nodes the server has ever seen. If a node hasn't respond to the last ping request the line is colored red. The rows can be sorted by clicking on the corresponding title you want to assort.

SetLocation We implemented a function to set the location of the BTnode relative to the picture, which is shown under GetDSN. Therefore you have to click with the mouse pointer into the picture and the coordinates of this point appears in the form. You have to choose the DSNID for this point and push "set Location".

5.2 Installation

5.2.1 Requirements

DSNServer The DSNServer has to run.

Webserver with PHP module A webserver has to run with a PHP module, but this server hasn't to run on the same computer as the DSNServer. We recommend for an easy installation for testing the toll an installation from the XAMPP project.

GD Library You need a GD PHP module that the tool is able to dynamic create the image on the GetDSN page.

5.2.2 Installation

1. Check-out with a CVS client the module "proj/dsngui" from "btnode.cvs.sourceforge.net:/cvsroot/btnode" into a directory of the webserver.

2. Create a directory in the root directory of the Tool with the name "img" and configure it that the webserver has write access to it.

3. Edit the file "properties.php" in the main directory of the Tool. The settings are explained in the Chapter 5.2.3.

4. Change the picture ”Bild.jpg” in the root directory of the Tool with your own picture of the environment, where the nodes are placed. The picture has to have the name ”Bild.jpg”.

5.2.3 properties.php

In the following table the variables of the properties.php are listed.

name	**value**	**explanation**
$host	String	It is the Host of the DSNServer, how the XMLRPC can be accessed. If the DSNServer and the Tool are on the same computer you can use ”localhost”.
$port	Integer	The port on which the XMLRPC is listening. This is defined in the file ”Server.properties” (see Chapter 4.5.3).
$radius	Integer	It defines the radius of the black circles on the picture under GetDSN, which indicates the location of a node.
$width_real, $height_real	Integer	It is the width and height of the picture in real scale. It is used to convert the location string of the BTnode to a coordinate in the picture.
$offset_x, $offset_y	Integer	Not needed anymore.
$coninfoUpdate	Integer	Time in seconds after a new coninfo request is sent to the GUI node. It is defined in the file ”Server.properties” (see Chapter 4.5.3).
$pingInterval	Integer	Time in seconds after a new ping request is sent out to the DSN network. It is defined in the file ”Server.properties” (see Chapter 4.5.3).

Table 5.1: Variables of the properties.php.

Chapter 6

Transport Performance Tests

An important thing of the DSN Server is the smart use of the bottleneck, we discussed earlier in Chapter 4.1. The largest amount of data in the DSN network comes from the collection of the log messages of the DSN Server. Therefore we have to make some considerations how the server should collect the log messages.
Hence we have two different method to collect the logs, push and pull method that are explained in Chapter 6.1
Moreover we are interested to do an estimation how much log messages can be generated on how many nodes that the server has the ability to store them all in the database.

6.1 Scenarios

As already mentioned there are two different methods to collect the log messages. An detailed explanation of these methods is given in Chapter 4.3.6.

Pull The Server goes in a kind of a round robin through all nodes in the network and requests the log messages of these. The server requests the log messages of the next node if it received no logs for a defined time of the current node.

Push As soon as a log messages is written to the log buffer, the BTnode is sending the message immediately to the DSN Server.

Hence for both methods we can discuss, if it is useful to do a retransmission for the lost logs.

We examine two different scenarios, how the logs are generated on a sensor node and written to the buffer of the DSN node.

Log Stream The log messages on a node are generated regularly, every specific time one log message is written to the buffer of the BTnode.

Log Burst After a pause some log messages are written together into the buffer of the BTnode.
This scenario can be described with a sensor node, which is sleeping a time, wake up, make some measurements, write these to the log buffer on the BTnode and sleep again.

In Figure 6.1 both modes are shown.

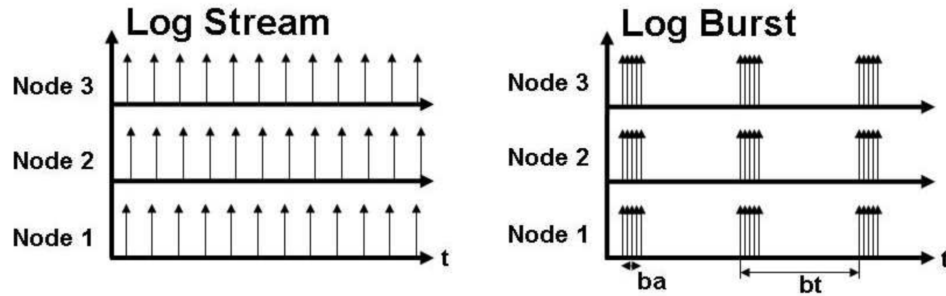


Figure 6.1: Log generating modes

Hence there are four different scenario each of them with or without retransmission of the lost messages:

- Push Log Stream (see Chapter 6.2.2)
- Pull Log Stream (see Chapter 6.2.3)
- Push Log Burst (see Chapter 6.2.4)
- Pull Log Burst (see Chapter 6.2.5)

6.2 Performance Evaluation

6.2.1 Sending Performance

We did a simulation for the model pull log stream (see Chapter 6.2.3) and a estimation for the model pull log burst (see Chapter 6.2.5) for the transport performance tests. Both models are dependent on the sending rate, which defines how many logs can be sent pro second from a node to the GUI node that has requested them.

To calculate the sending rate, we built a chain with BTnodes with a maximal hop count of five and measured the time to get one hundred log messages with a packet size of 121 bytes. We measured the time between the first and the hundredth log.

We measured log sending rates between 12,98 and 13,7 $\frac{logs}{s}$.

Therefore we assumed for our simulations and estimations a log sending rate of 13 $\frac{logs}{s}$.

The measured log sending rate is astonishingly small, if we consider that we have a baud rate of 57600 $\frac{bit}{s}$ that corresponds with a rate of 60 $\frac{logs}{s}$ if the log packet size is 120 byte.

In the log sending algorithm of the BTnode is a delay implemented that is probably too big. Therefore we consider that we can increase the log sending rate up to 30 $\frac{logs}{s}$, if we decrease the delay in the log sending function on the BTnode.

This increased sending rate will have an impact to methods with a pull mode. If a node is able to send the log messages faster, the time can be decreased, which is needed to send all logs to the server. The methods with a push mode won't change, because they send immediately there logs if they are written to the buffer of the BTnode to the server that is independent how many logs a BTnode can send in a specific time.

6.2.2 Push Log Stream

In Figure 6.2 are the result shown of a test in the push log stream mode without retransmission, which were done by Matthias Dyer [9].

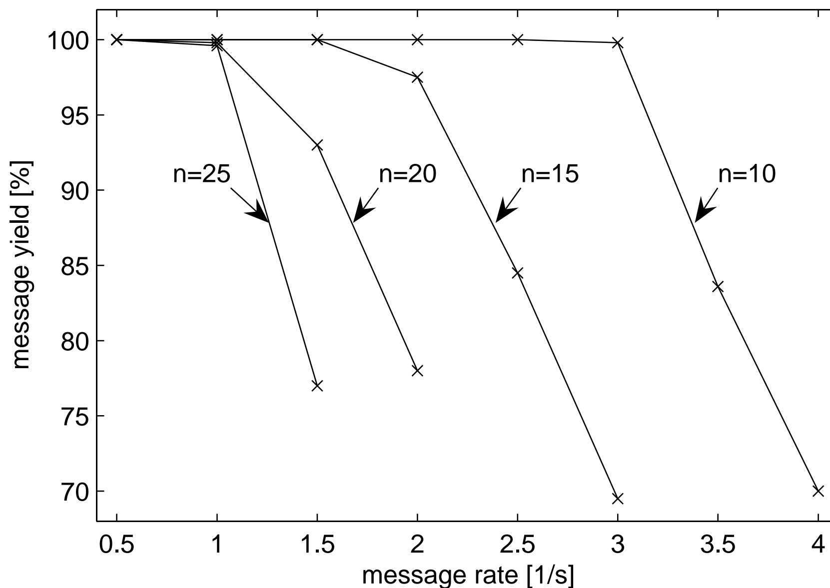


Figure 6.2: Measured yield of correctly received log messages in push log stream mode

Figure 6.2 shows the yield of correctly received log messages with a size of 86 bytes payload, which accords to a packet size of 121 bytes. During this test 100 log messages were generated on 10, 15, 20 or 25 nodes. Each test were processed with different message generation rates from 0.5 to 4 packets per node per second.

As we can see in Figure 6.2 a DSN can be run with following log generating rate on given nodes of the DSN to achieve a good message yield:

- 10 nodes $\Rightarrow$ l = 3,0 $\frac{logs}{s}$
- 15 nodes $\Rightarrow$ l = 1,5 $\frac{logs}{s}$
- 20 nodes $\Rightarrow$ l = 1,0 $\frac{logs}{s}$
- 25 nodes $\Rightarrow$ l = 1,0 $\frac{logs}{s}$

We don't think that there will be a gain, if we do the test with retransmission. If we are loosing messages, it is a sign that too much traffic is on the network and the traffic won't decrease later because the log messages are generated in a stream. Retransmission produces additional traffic, therefore it would have additional traffic and less messages will arrive at the server.

6.2.3 Pull Log Stream

For the pull log stream method we think that there will be no gain if we do it with retransmission of the lost messages. Because if the server request them, there should be no collision of the log messages.
Therefore we did a MatLab simulation of the method pull log stream without retransmission.

Model

With the MatLab simulation we would like to show if the time, which is needed to request the log messages of every node once, converges after some cycles or not. Therefore we added up the time that is needed to get the logs of every node over several cycles. On basis of Figure 6.3 we explain the model. As already mentioned we calculated the time which is needed to request all logs that is the time T in Figure 6.3.

- At the time zero all three nodes start to generate logs into the buffer of the BTnode.
- At the time zero the server requests the logs of the node 1 as well. But at this time the node 1 hasn't got any logs in the buffer, therefore no logs are sent to the server.

- The server waits the time D before it recognize that the node 1 won't send any logs.
- The server requests the logs of the node 2 and the node 2 sent its log to the server.
- When all logs are arrived at the server, the server has to wait the time D again, before it recognize that the node 2 finished the request.
- This goes now on and on.

MatLab Simulation

We calculated the times T and plotted them for several cycles to show, if it still increases or levels off. If the time T increases over all the cycles, it indicate that it is not possible to collect all logs. Because the number of logs on the buffer of the BTnode is increasing as well, consequently logs will be overwritten before they can be requested.
In the Figures 6.4 to 6.7 there are the plots for several log generating rates l. On the left side it is plotted the time T over the cycles and on the right side is plotted the maximal numbers of logs on a node over the cycles.

Conclusion

With the MatLab simulation we could show, how much nodes out of a network we can process, if they generate logs with a given rate l:

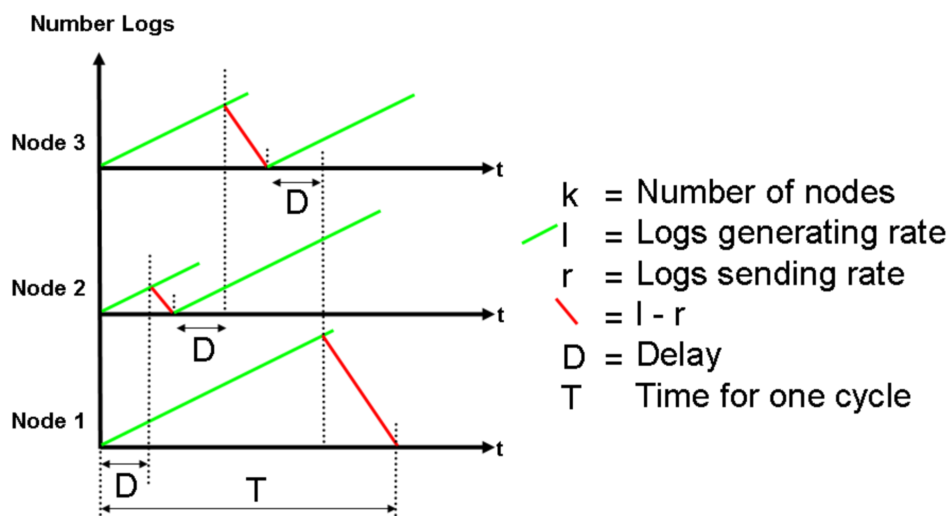


Figure 6.3: Used model of pull log stream.

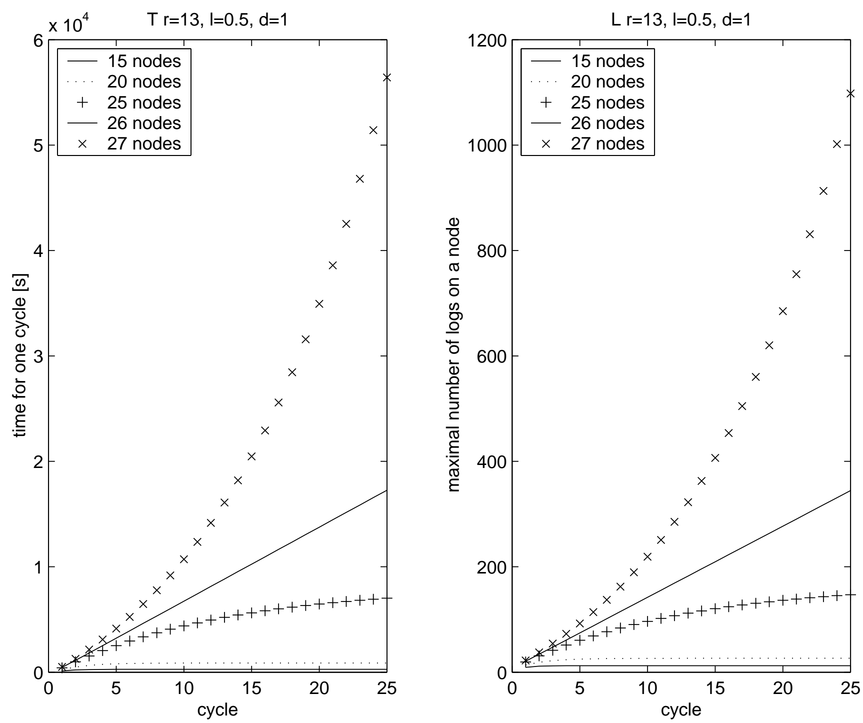


Figure 6.4: Plot of the pull log stream simulation with the settings r=13 l=0,5 d=1 c=25

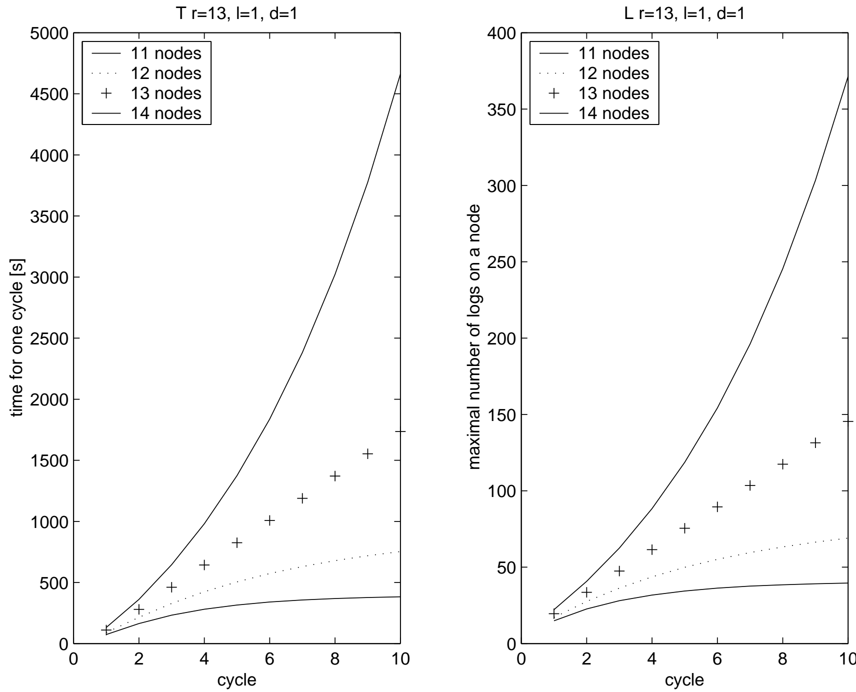


Figure 6.5: Plot of the pull log stream simulation with the settings r=13 l=1 d=1 c=25

- l = 0,5 $\frac{logs}{s}$ ⇒ 25 Nodes
- l = 1,0 $\frac{logs}{s}$ ⇒ 12 Nodes
- l = 1,5 $\frac{logs}{s}$ ⇒ 8 Nodes
- l = 2,0 $\frac{logs}{s}$ ⇒ 6 Nodes

We did our tests with a sending rate r = 13 $\frac{logs}{s}$ and a delay, which the server is waiting until it assumes that no more logs are coming, d = 1s. If we increases the value of r, we could process more nodes. The value of d has no influence on the numbers of nodes that can be processed, but the time T and the maximal numbers of logs on a node increases.

That it is possible to compare this simulated results with the results from the tests with the push log stream mode, we tested, which log generating rate l is possible with 5, 10, 15, 20 or 25 nodes.
With log sending rate r = 13 $\frac{logs}{s}$:

- 5 nodes ⇒ l = 2,5 $\frac{logs}{s}$
- 10 nodes ⇒ l = 1,2 $\frac{logs}{s}$

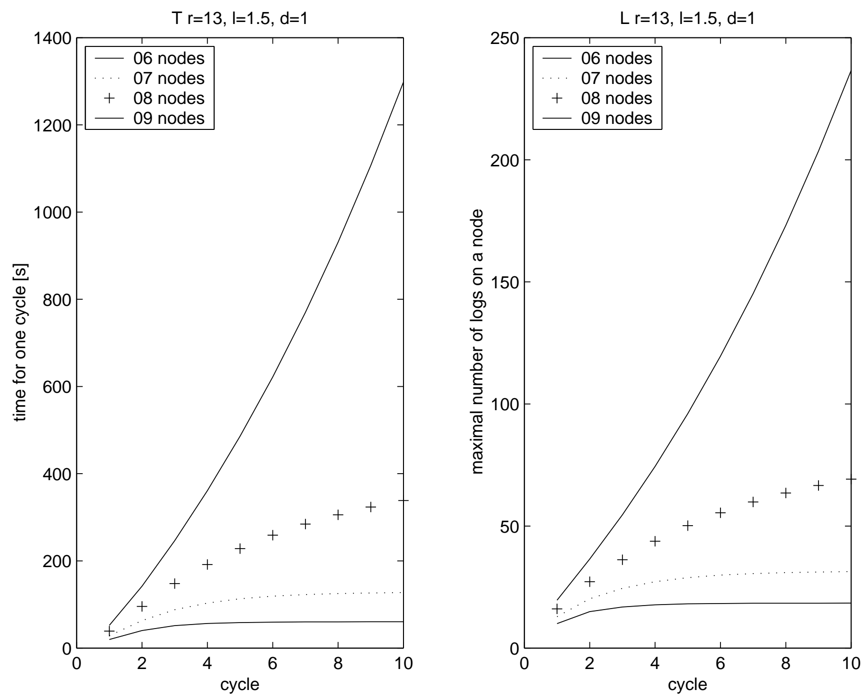


Figure 6.6: Plot of the pull log stream simulation with the settings r=13 l=1,5 d=1 c=25

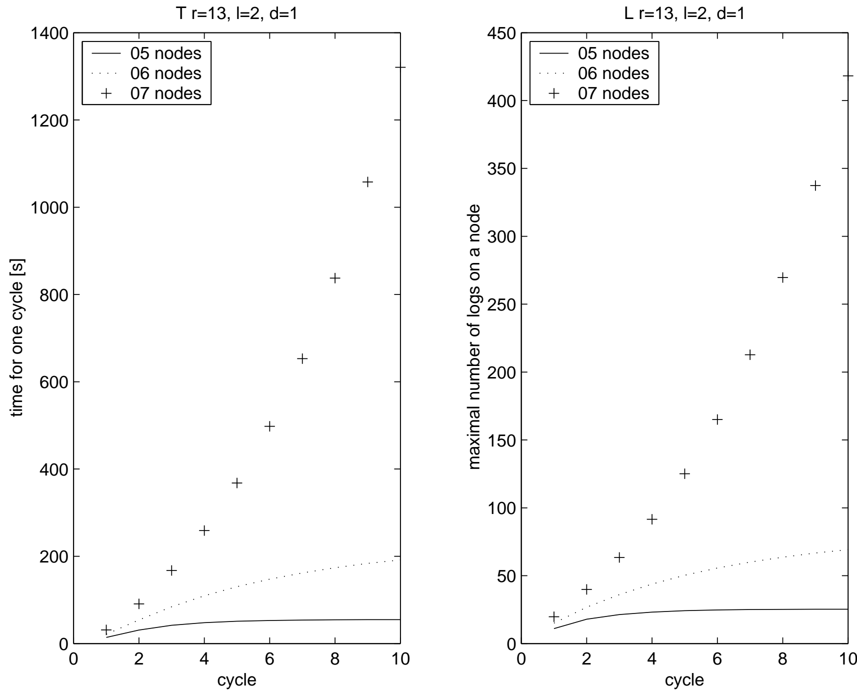


Figure 6.7: Plot of the pull log stream simulation with the settings r=13 l=2 d=1 c=25

- 15 nodes ⇒ l = 0,8 $\frac{logs}{s}$
- 20 nodes ⇒ l = 0,6 $\frac{logs}{s}$
- 25 nodes ⇒ l = 0,5 $\frac{logs}{s}$

As we already mentioned in Chapter 6.2.1, we expect that we can increase the sending rate up to r = 30 $\frac{logs}{s}$. This would have a great impact to this method.
With log sending rate r = 30 $\frac{logs}{s}$:

- 5 nodes ⇒ l = 5,9 $\frac{logs}{s}$
- 10 nodes ⇒ l = 2,9 $\frac{logs}{s}$
- 15 nodes ⇒ l = 1,9 $\frac{logs}{s}$
- 20 nodes ⇒ l = 1,4 $\frac{logs}{s}$
- 25 nodes ⇒ l = 1,1 $\frac{logs}{s}$

6.2.4 Push Log Burst with Retransmission

We tested the the performance of the DSN in the push log burst with retransmission mode on the DSN that was distributed on the G floor of the ETZ building at the ETH Zürich.

Therefore we sent to some nodes a command to generate a specific number of log messages at the same time. As soon as the log messages were written to the buffer of the BTnode, there were sent to the server. At the time of a burst a lot of messages were sent to the server and according to this some log messages were lost and had to request again after the burst.

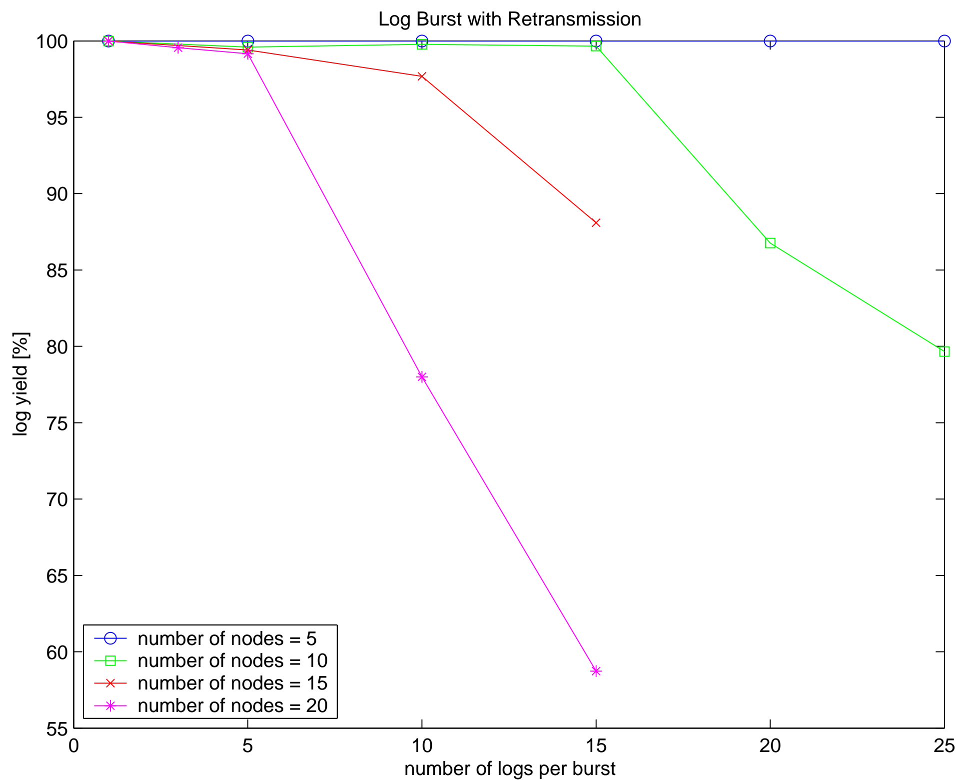


Figure 6.8: Measured yield of correctly received log messages in push log burst with retransmission mode

In Figure 6.8 we can see the yield of correctly received log messages. Therefore it is possible to get all log messages from 5, 10 15 or 20 nodes with following burst amount of log messages, if a burst occurs every 30 seconds:

- 5 nodes $\Rightarrow$ ba = 55 $\frac{logs}{burst}$
- 10 nodes $\Rightarrow$ ba = 15 $\frac{logs}{burst}$

- 15 nodes ⇒ ba = 7 $\frac{logs}{burst}$
- 20 nodes ⇒ ba = 6 $\frac{logs}{burst}$

6.2.5 Pull Log Burst

For the Pull Log Burst mode we found a condition that has to be achieved in order to request all logs properly:

$$bt \geq \frac{ba * k}{r} + k * d$$

This equation says that the time between two bursts must be greater or equal then the time that is needed to request all logs from the BTnodes. These two times are illustrated in Figure 6.9. If it is not possible to request all logs of the first burst before the second burst occurs then it won't be possible to request all logs of the second burst before the third burst occurs as well. Therefore the logs on the BTnode will still increase and the log buffer on the BTnode will overflow.

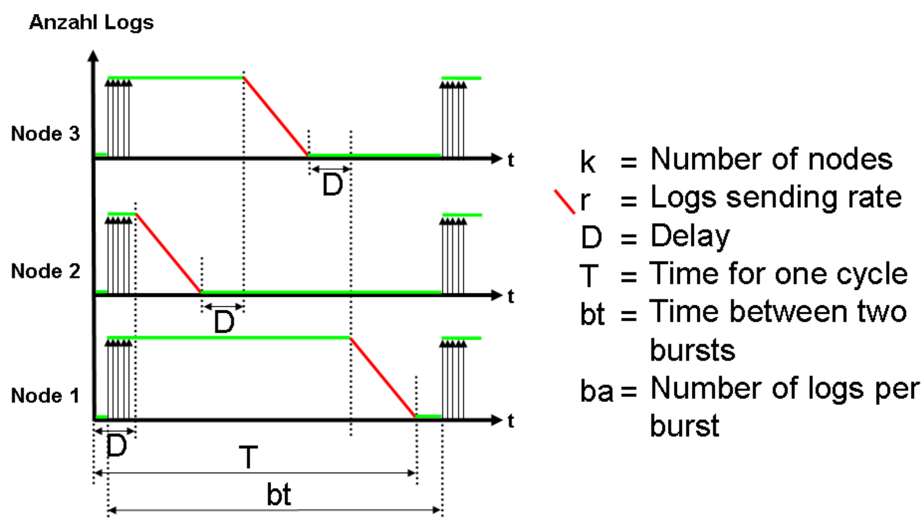


Figure 6.9: Illustration of the pull log burst method.

There won't be a gain if we do retransmission in this mode, because if the server is not able to request the log messages without retransmission, it

won't be able to do it with retransmission, due to the fact that retransmission needs some extra time.

To be able to compare the pull mode with the push mode directly, we calculated some values with the formula above.
With log sending rate r = 13 $\frac{logs}{s}$ and a burst every 30 seconds:

- 5 nodes $\Rightarrow$ ba = 65 $\frac{logs}{burst}$
- 10 nodes $\Rightarrow$ ba = 26 $\frac{logs}{burst}$
- 15 nodes $\Rightarrow$ ba = 13 $\frac{logs}{burst}$
- 20 nodes $\Rightarrow$ ba = 6,5 $\frac{logs}{burst}$

And again with the increased sending rate r = 30 $\frac{logs}{s}$ and a burst every 30 seconds:

- 5 nodes $\Rightarrow$ ba = 150 $\frac{logs}{burst}$
- 10 nodes $\Rightarrow$ ba = 60 $\frac{logs}{burst}$
- 15 nodes $\Rightarrow$ ba = 30 $\frac{logs}{burst}$
- 20 nodes $\Rightarrow$ ba = 15 $\frac{logs}{burst}$

6.2.6 Conclusion

Log Stream

In Figure 6.10 we show how many nodes can produce how many log messages per seconds that are correctly received at the server. This is done for the push log without retransmission and pull log without retransmission with two different sending rates.

As we explained in the previous chapter, we did tests for evaluating the push mode and we did a simulation for the pull mode. The curves in the Figure 6.10 have the same range of values and a similar characteristic of the curve. This indicates that our simulation is probably not far away from reality.

The Figure 6.10 points out that for the log stream model a push method is considerably better than pull method if the BTnode has a slow sending rate. But if it is possible to increase the log sending rate of the BTnode up to r = 30 $\frac{logs}{s}$, the pull method will be slightly better.

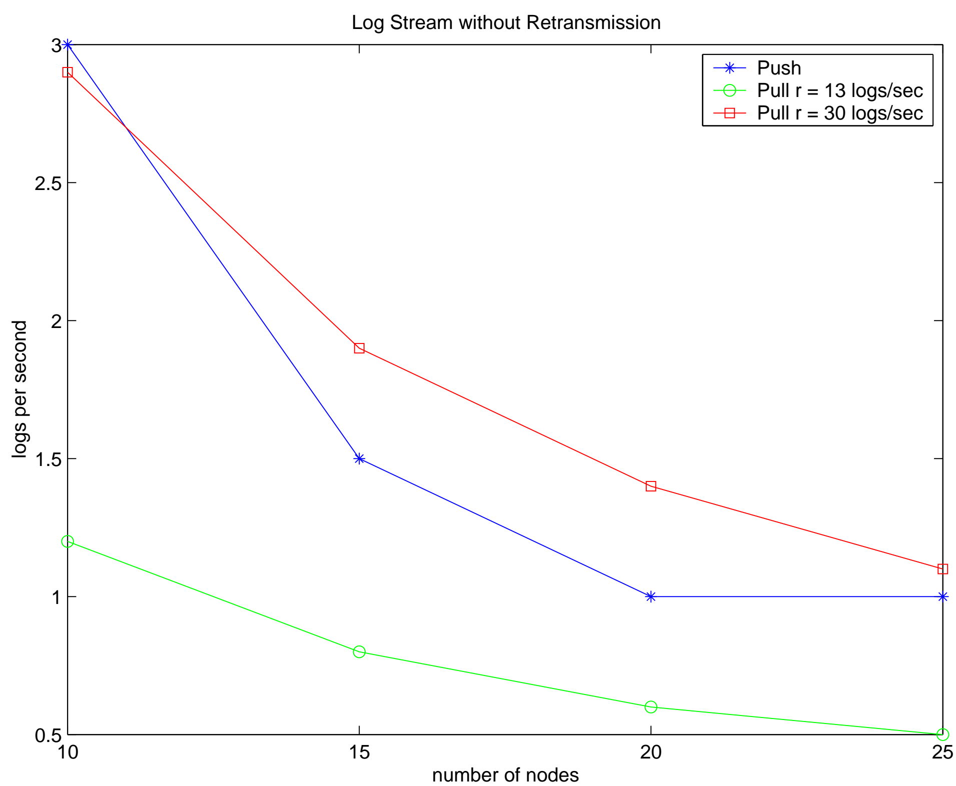


Figure 6.10: Both methods of the log stream model.

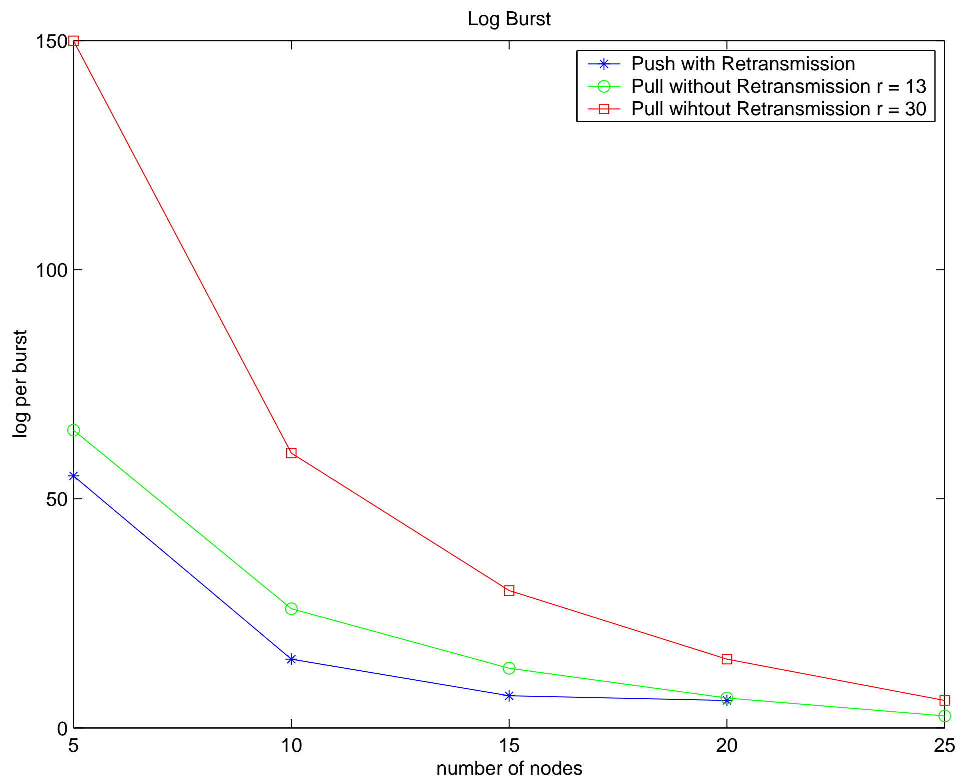


Figure 6.11: Both methods of the log burst model.

Log Burst

Figure 6.11 shows how many logs can be produced in a burst every 30 seconds of how many nodes, that it is possible to collect all packets correctly. This is done for the push log with retransmission and pull log without retransmission with two different sending rates.

In the Figure 6.11 we can see as well, that our test, which we did for the push mode with retransmission, and the estimation, which we did for the pull mode without retransmission, have the same range of values and similar characteristic of the curve as well.

The pull method is better as the push method, especially with the increased sending rate r = 30 $\frac{logs}{s}$.
The model that we have used for the test was fairly extreme, because the log burst started at the same time on all BTnodes. If the burst are randomly distributed over the time, the result of a push mode will be better then now. But we have to consider that nodes will be wake up at the same time slot because of communication reasons, therefore our model isn't far away from reality.

Chapter 7

Conclusion

The last chapter summarizes the achievements of this thesis and proposes some topics for future workings.

7.1 Summary

In this master thesis, a server could be implemented for an easy use of the DSN functionality and a Tool, which communicates with the server, which provides an easy use of all the functionalities of the server.

The server provides a method to collect all the logging data from the BTnode and store it to a database that it can be accessed later as well. Some status information are collected from the server regularly to provide them to the user of the DSN. There is the possibility to give restricted access to users.

Using the XMLRPC interface it is possible to attach and construct tools and GUI's to be used with the DSN. Therefore every developer of a sensor network can build his own tool with special functions that are necessary to develop, debug and test this network. There is already a GUI in use which is using this server [10].

During the developing process of the server we had to change and build some function on the jaws application on the BTnode, due to the fact that with this thesis some new concept were built, e.g the log messages were adjusted that the server had the possibility to notice lost messages.

In this thesis we did transport performance tests to estimate how many log messages can be collected from the BTnodes.

Generally it is now quite easy to write and execute test scenarios for the DSN, if we look in Appendix A how easy it is to do a XMLRPC request.

7.2 Future Work

The server has to be maintained during the time the jaws application on the BTnode is deployed. If there will be a new command implemented on the BTnode that allows the server to get more information about the network, which has to be stored into the database, there must be written a new CmdParser.

Additionally a XMLRPC function has to be updated or there must be written a new one.

During the time the DSN Tool is used, there will be some requirements that would be nice to have and therefore have to be implemented.

It would be interesting to do more performance tests. At the moment it occurs very often that a node doesn't respond anymore, if there was a lot of traffic in the network. If the network isn't stabile, it is very exhausting to do meaningful performance tests.

There is a watchdog implemented that reboots the BTnode, if it is in a deadlock situation. It should be measured how much this reboot occurs and how big the impact is due to the transport performance, if it has an impact of loosing packets.

The tests that are described in Chapter 6, should be repeated, because there were some changes on the server that could have an impact to the test results.

The estimation of the pull log burst and the MatLab simulation of the pull log stream should be tested on the real DSN network as well. It would be interesting if the measured results would be the same as the estimated results.

During the transport performing tests there was taken a random topology of the network and the tests run several time. It must be tested how big is the factor of the network topology.

Appendix A

XMLRPC Examples

A.1 Java Code

```
import java.io.IOException;
import java.net.URL;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;

import org.apache.xmlrpc.DefaultXmlRpcTransportFactory;
import org.apache.xmlrpc.XmlRpcClient;
import org.apache.xmlrpc.XmlRpcException;


public class XmlRpcExample {

  XmlRpcClient client;

  public XmlRpcExample(String urlstring) throws Exception {
    URL url = new URL(urlstring);
    DefaultXmlRpcTransportFactory tf =
        new DefaultXmlRpcTransportFactory(url);
    tf.setBasicAuthentication("username", "password");
    client = new XmlRpcClient(url, tf);
  }

  public static void main (String args[]) throws Exception {
    if (args.length < 1) {
      System.err.println ("Usage: java Client URL");
    } else {
      XmlRpcExample client = new XmlRpcExample (args[0]);
```

```
    client.run ();
  }
}

public void run () {
  try {

    Vector args0 = new Vector();
    args0.add(new String("root"));
    args0.add(new String("coninfo"));
    client.execute("dsnCommand.dsnCommand", args0);

    Vector<Object> args1 = new Vector<Object>();
    args1.add(new String("all"));
    args1.add(new String("all"));
    args1.add(new String("all"));
    args1.add(new String("0001-01-01 00:00:00"));
    args1.add(new String("9999-01-01 00:00:00"));
    Vector result1 = (Vector)client.execute("" +
        "getConinfo.getConinfo", args1);

    System.out.println("Result of getConinfo has "
        +result1.size()+" elements.");
    System.out.println("------------------"
        +"------------------");
    Iterator iter = result1.iterator();
    while(iter.hasNext()){
      Hashtable t = (Hashtable)iter.next();
      System.out.println("time            = "
          +t.get("time"));
      System.out.println("SourceAddr   = "
          +t.get("SourceAddr"));
      System.out.println("NeighborAddr = "
          +t.get("NeighborAddr"));
      System.out.println("State            = "
          +t.get("State"));
      System.out.println("rssi            = "
          +t.get("rssi"));
      System.out.println("----------------"
          +"--------------------");
    }
  } catch (IOException e) {
    System.out.println("IO Exception: "
         + e.getMessage());
```

```
    } catch (XmlRpcException e) {
      System.out.println("Exception within XML-RPC: "
           + e.getMessage());
    }
  }
}
```

A.2 PHP Code

```
<?php

require_once 'xmlrpc.inc';

$_RPC_CLIENT = new xmlrpc_client('/', 'localhost', 8887);
$_RPC_CLIENT->setCredentials('kaltt', 'mypwd');

//==========================================================
function dsnCommand($destination, $command){
  global $_RPC_CLIENT;

  $args = array(
    new xmlrpcval($destination, 'string'),
    new xmlrpcval($command, 'string'),
  );
  $req = new xmlrpcmsg('dsnCommand.dsnCommand', $args);
  $resp = $_RPC_CLIENT->send($req);

  return $resp;
}
//==========================================================
function getConinfo($Source_DSNID, $Neighbor_DSNID, $State,
                    $time_from, $time_till, $maxReply){
  global $_RPC_CLIENT;

  $args = array(
    new xmlrpcval($Source_DSNID, 'string'),
    new xmlrpcval($Neighbor_DSNID, 'string'),
    new xmlrpcval($State, 'string'),
    new xmlrpcval($time_from, 'string'),
    new xmlrpcval($time_till, 'string'),
    new xmlrpcval($maxReply, 'int'),
  );
  $req = new xmlrpcmsg('getConinfo.getConinfo', $args);
```

```
  $resp = $_RPC_CLIENT->send($req);

  return $resp;
}
//==========================================================



dsnCommand('all', 'log clear');

$r = getConinfo('all', 'all', 'all', '0001-01-01 00:00:00',
               '9999-01-01 00:00:00', 10);

if ($r->faultCode()) {
  print "An error occurred:<pre>";
  print "Code: " . htmlspecialchars($r->faultCode()) .
        "\nReason: '" . htmlspecialchars($r->faultString()).
        '\'</pre><hr/>';
}
else{
  $v = $r->value(); // array
  $max = $v->arraysize();
  if($max == 0){
    print "no entries found.\n";
    return;
  }

  echo'
    <table border="1">
    <tr>
      <th>Time</th>
      <th>Source</th>
      <th>Neighbor</th>
      <th>State</th>
      <th>rssi</th>
    </tr>';

  for($i=0; $i < $max; $i++){
    $rec = $v->arraymem($i);
    $time_temp = $rec->structmem("time");
    $time = $time_temp->scalarval();
    $source_temp = $rec->structmem("SourceAddr");
    $source = $source_temp->scalarval();
    $neighbor_temp = $rec->structmem("NeighborAddr");
    $neighbor = $neighbor_temp->scalarval();
```

```
    $state_temp = $rec->structmem("State");
    $state = $state_temp->scalarval();
    $rssi_temp = $rec->structmem("rssi");
    $rssi = $rssi_temp->scalarval();
    print "<tr><td>".htmlspecialchars($time).
          "</td><td>".htmlspecialchars($source).
          "</td><td>".htmlspecialchars($neighbor).
          "</td><td>".htmlspecialchars($state).
          "</td><td>".htmlspecialchars($rssi).
          "</td></tr>\n";
  }

  print "</table>\n";

}
?>
```

Appendix B

Problem Task

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik (TIK)

Sommersemester 2006

MASTERRARBEIT

für
Thomas Kalt

Betreuer: Jan Beutel
Stellvertreter: Matthias Dyer

Ausgabe: 24. April 2006
Abgabe: 20. Oktober 2006

Online Sensor Netzwerk Analyse Tools

Einleitung

Eine bekannte Vision für ad hoc Netzwerke [16] geht davon aus, das unendlich viele, kleinste "Sensorknoten" kollaborativ ein Netzwerk und somit eine Applikation bilden. In anderen Visionen [13, 6] wird davon ausgegangen, das solche System weite Anwendungsbereiche abdecken können und das die einzelnen Komponenten unterschiedliche Ressourcen aufweisen.

Die BTnodes [5] bestehen aus einem Atmel AVR Mikrokontroller, einem Bluetooth Modul und ein Low-Power Radio. Zusammen mit der im NCCR-MICS [31] entwickelten BTnut System Software bilden sie eine sehr kompakte programmierbare Platform für die Entwicklung mobiler ad hoc und Sensornetze. An diese Knoten können diverse Peripheriegeräte (z.B. Sensoren) angehängt werden. Mit der geeigneten Software bauen viele Sensorknoten selbstständig ein Sensornetzwerk auf, worüber die Sensordaten transportiert werden können. Andere, vergleichbare Sensorknoten sind zum Beispiel die Mica Motes [12] und das TinyOS Software Framework [19, 20].

Heute werden Applikationen für Sensornetze meist explorativ entwickelt. Hierzu ist relativ viel Aufwand von Personal, Know-How und entsprechenden iterativen Designzyklen notwendig. Erfahrungsberichte von solchen Experimenten gibt es von Szewcyk [24, 25, 23], Cerpa [7], Hemingway [11], Mainwaring [21] und anderen. Erste Ansätze die koordinierte Methoden und Verfahren eines ganzheitlichen Entwicklungsprozesses zum Ziel haben gibt es bereits. Insbesondere sind in den Teilbereichen der Simulation [18, 22, 17], Emulation [10], Entwicklung [9, 5], Inbetriebnahme [15], Test [27], Validierung und Verifikation [4] Lösungen vorhanden. Einer besonderer Ansatz stellt hier das sogennante Deployment-Support Netzwerk (DSN) [3, 2, 4] dar, welches als temporäres Werkzeug während des Entwicklungs- und Inbetriebnahmenprozesses sowie zur Überwachung und Validierung angewendet werden kann. In einem solchen Deployment-Support Network wird mit speziellen DSN Knoten ein zweites, temporaeres Backbone Netzwerk aufgespannt über das jeglicher debugging Verkehr abgewickelt wird. Von einem zentralen Server aus kann das DSN gesteuert werden und die anfallenden Daten werden dort in einer Datenbank (MySQL) geloggt.

In dieser Arbeit sollen geeignete analyse Tools entworfen und mit dem am TIK betriebenen DSN basierend auf den BTnodes, experimentell ausgewertet werden. Dazu wird es notwendig sein sowohl eine geeignete Steuerung des DSN mittels eines graphischen User Interfaces zu erstellen, ein logging in die Datenbank, sowie Auswertungsmethoden mittels online (z.b. Cacti, Nagios) oder offline (z.b. Matlab) zu erstellen.

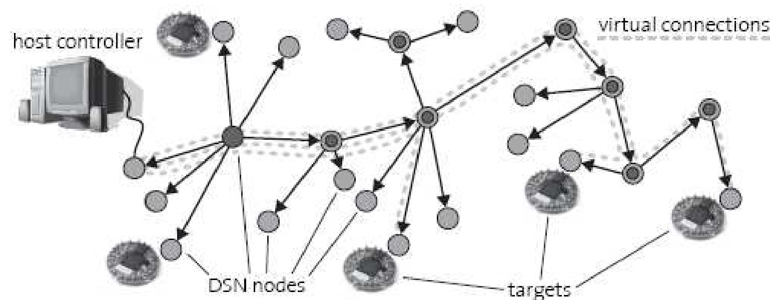


Abbildung 1: JAWS: Deployment-Support Network. Die BTnodes (DSN nodes) bilden und unterhalten selbstständig ein Backbone-Netzwerk über welches die Targetnodes (z.B. die Berekley Motes) programmiert bzw. überwacht werden können.

Aufgabenstellung

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest [28]. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.
2. Machen Sie sich mit den relevanten Arbeiten im Bereich Sensornetze, Platformen, Systeme, Software und Fast-prototyping vertraut. Führen Sie eine Literaturrecherche durch. Suchen Sie auch nach relevanten neueren Publikationen. Vergleichen Sie bestehende Demonstratoren anderer Universitäten (Motelab [26], Smote [1], Kansai, Mirage, EmStar [10]). Prüfen Sie welche Ideen/Konzepte Sie aus diesen Lösungen verwenden können.
3. Arbeiten Sie sich in die Softwareentwicklungsumgebung der BTnodes [29] ein. Machen Sie sich mit den erforderlichen Tools vertraut und benutzen Sie die entsprechenden Hilfsmittel (Versionskontrolle, Bugtracker, online Dokumentation, Mailinglisten, Application Notes, Beispielapplikationen).
4. Nehmen Sie das JAWS Deployment-Support Network auf einigen Knoten in Betrieb und testen Sie dieses auf Zuverlässigkeit und Leistung. Erstellen Sie eine Liste der noch fehlenden Eigenschaften die für einen zuverlässigen Dauerbetrieb notwendig sind sowie der noch vorhandenen Fehlfunktionen. Benutzen Sie hierzu soweit Möglich computergestütze tools (Bugtracker).
5. Entwickeln Sie in Zusammenarbeit mit der Masterarbeit Fire-Sensor Network Demonstratorentsprechende online analyse Tools zur Steuerung und Überwachung eines DSN.
6. Führen sie Tests und Benchmarks durch. Interpretieren Sie die Resultate.
7. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

Durchführung der Masterarbeit

Allgemeines

- Der Verlauf des Projektes Masterarbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.

- Sie verfügen über PC's mit Linux/Windows für Softwareentwicklung und Test. Für die Einhaltung der geltenden Sicherheitsrichtlinien der ETH Zürich sind Sie selbst verantwortlich. Falls damit Probleme auftauchen wenden Sie sich an Ihren Betreuer.
- Stellen Sie Ihr Projekt zu Beginn der Masterarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums Ende Semester.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern. Verfassen Sie dazu auch einen kurzen wöchentlichen Statusbericht (email).

Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichtes spätestens am *20. Oktober 2005* dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.
- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen usw. bestehen bleiben. Der Programmcode sowie die Filestruktur soll ausreichen dokumentiert sein. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.

Literatur

[1] UC Berkeley. Smote: Berkeley network sensor testbed. http://smote.cs.berkeley.edu/.

[2] J. Beutel. *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*, chapter Location Management in Wireless Sensor Networks. CRC-Press, Boca Raton, FL, 2004.

[3] J. Beutel, M. Dyer, M. Hinz, L. Meier, and M. Ringwald. Next-generation prototyping of sensor networks. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 291–292. ACM Press, New York, November 2004.

[4] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable topology control for deployment-sensor networks. In *Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN '05)*, pages 359–363. IEEE, Piscataway, NJ, April 2005.

[5] J. Beutel, O. Kasten, F. Mattern, K. Römer, F. Siegemund, and L. Thiele. Prototyping wireless sensor network applications with BTnodes. In *Proc. 1st European Workshop on Sensor Networks (EWSN 2004)*, volume 2920 of *Lecture Notes in Computer Science*, pages 323–338. Springer, Berlin, January 2004.

[6] L. Blazevic, L. Buttyan, Capkun S., S. Giordano, J.P. Hubaux, and J.Y. Le Boudec. Self organization in mobile ad hoc networks: The approach of Terminodes. *IEEE Communications Magazine*, 39(6):166–174, June 2001.

[7] A. Cerpa, J.E. Elson, M. Hamilton, J. Zhao, D. Estrin, and L. Girod. Habitat monitoring: application driver for wireless communications technology. *ACM SIGCOMM Computer Communication Review*, 31(2):20–41, April 2001.

[8] Chipcon. *CC1000, Single Chip Very Low Power RF Transceiver*, April 2002.

[9] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation (PLDI 2003)*, pages 1–11. ACM Press, New York, June 2003.

[10] L. Girod, J. Elson, A. Cerpa, T. Stathapopoulos, N. Ramananthan, and D. Estrin. EmStar: A software environment for developing and deploying wireless sensor networks. In *Proc. USENIX 2004 Annual Tech. Conf.*, pages 283–296, June 2004.

[11] B. Hemingway, W. Brunette, T. Anderl, and G. Borriello. The Flock: Mote sensors sing in undergraduate curriculum. *IEEE Computer*, 37(8):72–78, August 2004.

[12] J.L. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. 9th Int'l Conf. Architectural Support Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104. ACM Press, New York, November 2000.

[13] J.P. Hubaux, T. Gross, J.Y. Le Boudec, and M. Vetterli. Toward self-organized mobile ad hoc networks: The Terminodes project. *IEEE Communications Magazine*, 39(1):118–124, January 2001.

[14] T. Hug and F. Süss. Mote/TinyOS meets BTnode, February 2004.

[15] J.W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 81–94. ACM Press, New York, November 2004.

[16] J.M. Kahn, R.H. Katz, and K.S.J. Pister. Next century challenges: Mobile networking for smart dust. In *Proc. 5th ACM/IEEE Ann. Int'l Conf. Mobile Computing and Networking (MobiCom '99)*, pages 271–278. ACM Press, New York, August 1999.

[17] O. Landsiedel, K. Wehrle, and S. Götz. Accurate prediction of power consumption in sensor networks. In *Proc. 2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, page to appear. IEEE, Piscataway, NJ, May 2005.

[18] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*, pages 126–137. ACM Press, New York, November 2003.

[19] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, Brewer E., and D. Culler. The emergence of networking abstractions and techniques in TinyOS. In *Proc. First Symp. Networked Systems Design and Implementation (NSDI '04)*, pages 1–14. ACM Press, New York, March 2004.

[20] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *Ambient Intelligence*, chapter TinyOS: An Operating System for Sensor Networks, pages 115–148. Springer, Berlin, 2005.

[21] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proc. 1st ACM Int'l Workshop Wireless Sensor Networks and Applications (WSNA 2002)*, pages 88–97. ACM Press, New York, September 2002.

[22] V. Shnayder, M. Hempstead, B. Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 188–200. ACM Press, New York, November 2004.

[23] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 214–226. ACM Press, New York, November 2004.

[24] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin. Habitat monitoring with sensor networks. *Communications of the ACM*, 47(6):34–40, June 2004.

[25] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proc. 1st European Workshop on Sensor Networks (EWSN 2004)*, volume 2920 of *Lecture Notes in Computer Science*, pages 307–322. Springer, Berlin, January 2004.

[26] Harvard University. MoteLab: Harvard network sensor testbed. http://motelab.eecs.harvard.edu/.

[27] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A wireless sensor network testbed. In *Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN '05)*, pages 483–488. IEEE, Piscataway, NJ, April 2005.

[28] E. Zitzler. Studien- und Diplomarbeiten, Merkblatt für Studenten und Betreuer. Computer Engineering and Networks Lab, ETH Zürich, Switzerland, March 1998.

[29] BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. http://www.btnode.ethz.ch.

[30] Crossbow Technology Inc. http://www.xbow.com.

[31] NCCR-MICS: Swiss National Competence Center on Mobile Information and Communication Systems. http://www.mics.org.

Bibliography

[1] J. Beutel, M. Dyer, L. Meier, M. Ringwald, and L. Thiele. Next-generation deployment support for sensor networks. Technical Report 207, Computer Engineering and Networks Lab, ETH Zürich, Switzerland, November 2004.

[2] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable topology control for deployment-sensor networks. Technical Report 208, Computer Engineering and Networks Lab, ETH Zürich, Switzerland, November 2004.

[3] K. Martin, Adaptive XTC on BTnodes, Masterarbeit, ETH Zürich, Switzerland, Wintersemester 04/05

[4] http://www.millennium.berkeley.edu/sensornets/

[5] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A wireless sensor network testbed. In Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN '05), pages 483488. IEEE, Piscataway, NJ, April 2005

[6] http://www.earthscope.org/

[7] http://ant.apache.org/manual/install.html

[8] Jan Blumenthal, Frank Reichenbach, Frank Golatowski, Dirk Timmermann: Controlling Wireless Sensor Networks using SeNeTs and EnviSense, 3rd IEEE International Conference on Industrial Informatics, INDIN 05, ISBN: 0-7803-9095-4, Perth, Australien, August 2005

[9] M. Dyer, P. Blum, L. Thiele: Deployment Support Network, A toolkit for the development of WSNs, submitted to EWSN 07, Delft, The Netherlands, January, 2007

[10] P. Oehen, DSNAnalyzer: Backend for the Deployment Support Network, Master Thesis, ETH Zürich, Switzerland, September 2006