ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Master Thesis

Controlling Wireless Sensor Networks

Otmar Caduff
caduffo@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Summer, 2006

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Nicolas Burri and Pascal von Rickenbach

Acknowledgements

After five years of studying, I am now about to complete my degree in Computer Science. This definitely would not have been possible with the support of some people I would like to thank.

Firstly, I would like to thank my parents for supporting me all the time and having made it possible for me to study. A big 'thank you' goes to Andrea for being appreciative of me being engaged in my studies. I also would like to thank my brothers, my friends and everyone being a good person to me.

With respect to this thesis, I especially would like to thank my advisors Nicolas and Pascal and Prof. Dr. Roger Wattenhofer for their support, discussions, ideas and feedback and for giving me the possibility to work on this thesis. Thanks for this learning experience!

Contents

Chapter 1

Introduction

1.1 Problem Description

Wireless sensor networks consist of small devices (called *nodes*) deployed in a geographic region to monitor the environment. The resources of these devices, such as processing capabilities, memory size, communication range or available power are limited. Finding out something about the state of a running application by only staring at the device is limited; the standard equipment of a node that allows exposing information directly to a human is often confined by several light emitting diodes.

Software operating on such devices is typically not only responsible for gathering information from its environment. More complex tasks such as managing the communication with other devices make developing such applications a non trivial undertaking. Therefore, developers should be given the possibility to fall back on tools that reveal state on different levels of granularity: single variables, a node or a whole network consisting of numerous devices. A mean to influence the devices from a single point would alleviate keeping track of the whole network.

This work aims at helping developers of applications for wireless sensor networks. This is accomplished by providing a framework consisting of different controlling components that can be added to the actual application on the node and facilities to inspect, watch and govern these components remotely.

1.2 Background Information

This section aims at giving basic information and forward pointers on environments the framework bases on. On the one hand, there is TinyOS, an operating system for devices running inside wireless sensor networks with the nesC programming language. On the other hand, there is the Eclipse platform, originally a development environment for Java that grew to an extensive application framework.

1.2.1 TinyOS, NesC

TinyOS is an operating system written in C and nesC, a component oriented language based on C. Software components running on TinyOS are implemented in nesC. NesC components consist of modules, which are wired to each other by connecting interfaces. The wiring of these components is defined in configurations. Modules can make use of interfaces or implement them, thereby offering their functionality to other modules. Modules and interfaces define a set of commands that can be called and a set of events that may be signaled. The implementation of commands and event handlers is provided by modules.

The following example briefly shows how interfaces, modules and configurations relate to each other. It is a shortened version of the configuration for the sample application used in appendix B. The application writes voltage measurements on a regular basis to the local flash storage.

```
1  configuration ControllingExample {
2  }
3  implementation {
4      components Main, VbatC, LoggerC, TimerC,
5          ControllingExampleM;
6
7      Main.StdControl -> VbatC.StdControl;
8      Main.StdControl -> LoggerC.StdControl;
9      Main.StdControl -> ControllingExampleM.StdControl;
10
11     ControllingExampleM.ADC -> VbatC.ADC;
12     ControllingExampleM.Logger -> LoggerC.Logger;
13     ControllingExampleM.Timer -> TimerC.Timer;
14 }
```

The configuration comprises five components (lines 4–5):

Main serves as the application's entry point. It initializes and starts the `VbatC`, `LoggerC` and `ControllingExampleM` components by using the `StdControl` interface provided by these components. Lines 7 to 9 define this wiring.

VbatC is used to measure the supplied battery power. It provides the `ADC` interface which is used by the `ControllingExampleM` component to get the actual voltage. This wiring is designated in line 11.

LoggerC logs data to flash memory. Line 12 indicates that the `Controlling ExampleM` component uses the `Logger` interface to access functionality provided by the `LoggerC` component.

TimerC triggers events after defined time periods. Here, these events are handled by the `ControllingExampleM` component, which uses the `Timer` interface provided by `TimerC` (line 13).

ControllingExampleM implements the missing logic: setting up the timer as well as getting and logging a voltage measurement whenever an event is triggered by the timer.

Further information on TinyOS can be found on the TinyOS website[1]. [1] and [2] introduce the nesC languge.

1.2.2 Eclipse Platform

The Eclipse platform bases on the Java language, what makes it independent of the underlying hardware architecture. Functionality is added by a plug-in system. Plug-ins can be made extensible by offering extension points. Essentially, an extension point is a description of what an extension should look like in order to make a contribution to the plug-in. It can be regarded as a contract any extending plug-in has to stick to, so that the host plug-in providing the extension point is able to make use of any extenstions. Figure 1.1 illustrates the extensibility model.

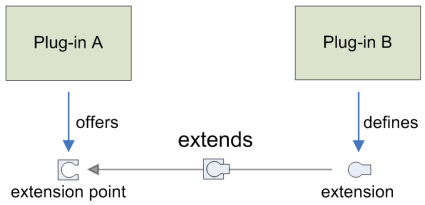

Figure 1.1: Plug-in B contributing to plug-in A by extending an extension point of plug-in A.

At runtime, host plug-ins offering extension points can consult the Eclipse platform registry containing all extensions and make use of them, e.g. by exploiting information they contain or making use of functionality they provide.

Apart from plug-ins, fragments present another method to make contributions to existing plug-ins. A fragment is similiar to a plug-in in that it is used to extend functionality, but it is not autonomous and depends on a specific plug-in. Furthermore, it does not support a life-cycle concept like plug-ins do.

The Eclipse website[2] offers tons of information. However, there are excellent books introducing developers to programming contributions to Eclipse. [3] is a good resource to start with.

[1] http://www.tinyos.net/
[2] http://www.eclipse.org/

Chapter 2

Related work

2.1 Nucleus Network Management

The Nucleus Network Management system was developed by Gilman Tolle at the University of California, Berkeley. Nucleus makes it possible to expose application information in three different ways: as RAM symbols (values residing in memory at a known position), as Nucleus attributes (values that can be queried by name) and as log events (not yet complete, planned for a future version). Once the development environment is set up correctly, making use of nucleus features is quite simple.

In order to enable RAM symbol querying (i.e. reading global variables), adopting the node application's `makefile` and calling an additional make target does the job; except for the inclusion of an additional module, no modifications to the application code are necessary.

In [4], the design of SNMS (Sensor Network Management System, a preliminary version of Nucleus) was restricted to "depend on the application as little as possible, to ensure that it will continue to function even when the application fails". Thus, Nucleus uses its own lightweight network layer operating in parallel with the application's networking layer. This implies that the nodes have to listen for new messages coming in at any time, which decreases the node's battery lifetime significantly. However, [4] mentions the possibility to use the applications network stack. Whether this option is available in Nucleus is not documented.

2.2 Pytos

Pytos ties in with the ideas of Nucleus. It is a an environment enabling the developer to inspect and modify application state at runtime. It was developed by various people at UC Berkeley.

As the name suggests, Pytos bases on Python. Global variables and commands declared appropriately residing in applications running on nodes can be governed through a command line interface. Pytos takes the necessary information to access variables and perform calls on nodes from two XML (Extensible Markup Language) files generated during the compilation process of the node

application: rpcSchema.xml and nescDecls.xml. Figure 2.1 shows an outline of the building process.

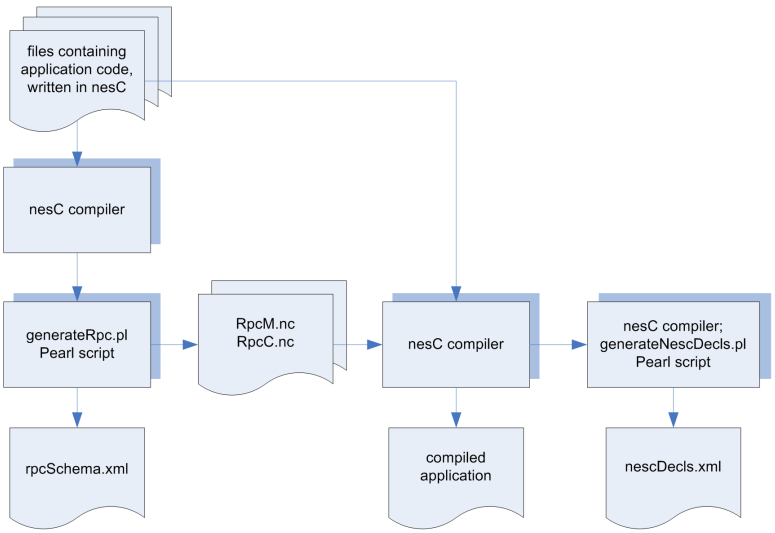

Figure 2.1: The building process of a NesC application with support for calling commands and getting/setting global variables remotely.

2.2.1 Embedded RPC

A core service Pytos makes use of is embedded RPC (Remote Procedure Call): In contrast to traditional RPC, emphasis is placed on low server load (in this case the node takes the position of the server) and small server code size.

The rpcSchema.xml file contains the information needed to request a remote call on a node, e.g. their arguments and its types and the type of the return value. Thus, the client application is able to send a message which a node is able to interpret, execute the corresponding command and send back a message containing the result value. The necessary logic is implemented in the RpcC.nc and RpcM.nc files, which are added by the building system (see figure 2.1).

2.2.2 Peek and Poke

Getting (peek) and setting (poke) global variables is realized on top of RPC: The position of a variable is retrieved by the client by inspecting the nescDecls.xml file containing all global variables. The nescDecls.xml file is generated by a Perl script (see figure 2.1) which in turn calls `objdump`, a command which displays information about object files, and parses its output. By executing a RPC call with a known memory address and in case of poke a new value for the variable, it is possible to get or set a global variable on the node.

2.2.3 Drip and Drain

The communication between Pytos and the nodes is performed over Drip and Drain. Drip, an epidemic protocol used to reliably disseminate messages through

the entire network, is used to send RPC messages to the nodes. Drain, on the other hand, is responsible for reliably sending messages from nodes back to the base station connected to the client running Pytos. Drain relies on a tree rooting at the base station, with every tree node representing a device. Thus, it is necessary that every device maintains some information about this tree (e.g. the device representing the parent node is needed in order to route a message towards the root node—the device acting as base station delivering messages to the client application). The drain layer has therefore to be updated periodically.

2.2.4 Pytos Installation

Pytos relies on a variety of tools, which makes its installation cumbersome. One point potentially consuming a lot of time is getting the needed JPype environment (a tool enabling access to Java classes from whithin Python) up and running—at the time of writing this document, the Pytos project website[1] still mentioned unresolved problems on Cygwin installations. In order to get Pytos running on Linux (Fedora Core 5), changes to the code were necessary.

2.3 MoteView

MoteView is a monitoring software for wireless sensor networks. It is developed by Crossbow Technology, a wireless network equipment supplier.

The MoteView software can be downloaded for free from the company's website[2]. The main features as presented on the Crossbow website include:

- historical and real-time charting
- topology map and network visualization
- data export capability
- print graph results
- node programming with MoteConfig
- command interface to sensor networks
- email alerts service
- Windows XP and 2000 SP4 compatible

The software works with networks consisting of nodes manufactured by Crossbow. The setup looks as follows: all nodes are preloaded with a specific software. Programming the nodes can be done from within the MoteView application. Basically, the applications running on the nodes send sensor reading data to the gateway node which posts the readings to a database. Different applications for different node configurations (sensing equipment, processor, power modes and frequency bands) are included with the MoteView software.

[1] http://nest.cs.berkeley.edu/nestfe/index.php/Pytos_Installation_Instructions; last accessed on August 29, 2006

[2] http://www.xbow.com/products/productsdetails.aspx?sid=88; last accessed on August 29, 2006

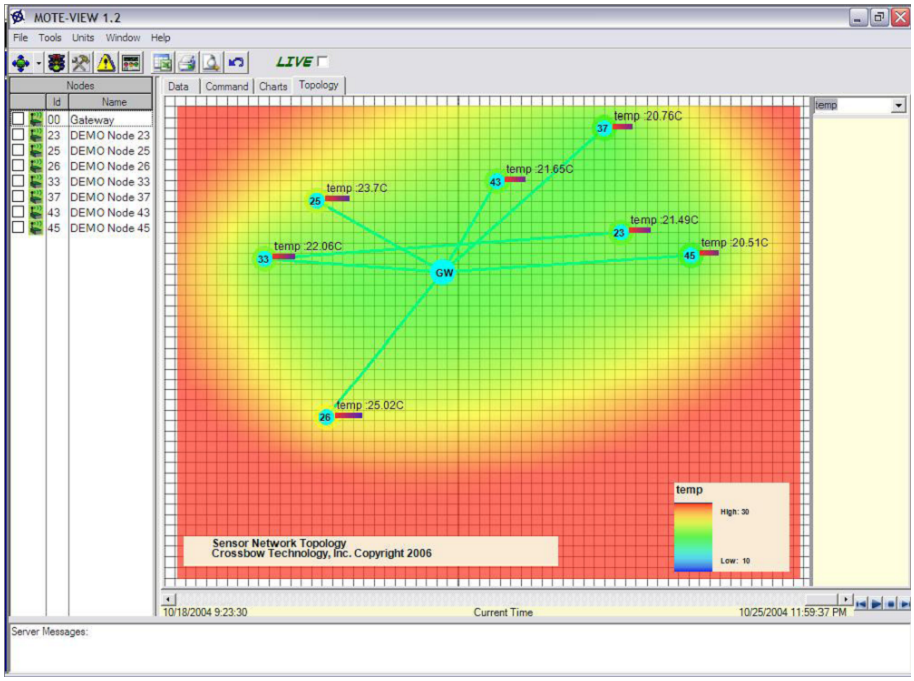

Figure 2.2: A screenshot of MoteView in action.

The graphical user interface is straightforward to use and offers appealing views on sensor data. Figure 2.2 shows the topology presentation of MoteView. However, no source code is available and it is not possible to use the MoteView software with nodes running custom applications. The user is limited by the applications delivered with MoteView. Moreover, it requires a full fledged database running on the machine where the sensor data is stored. PostgreSQL is installed automatically if necessary.

Chapter 3

Design and Implementation

The controlling framework offers four different services to developers (figure 3.1): communication between the client workstation and the nodes in the wireless sensor network, inquiring variables on nodes, logging data, and controlling the network topology.

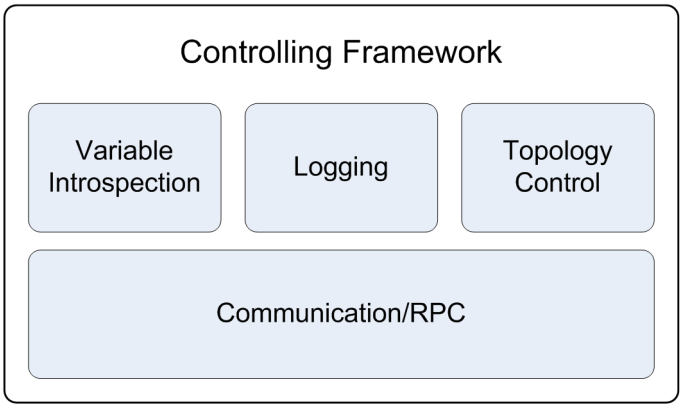

Figure 3.1: The services making up the controlling framework.

This chapter is organized as follows: first, an overview on the architecture and the user interface is given. The subsequent chapters introduce the individual services.

3.1 The Big Picture

The controlling framework can be divided in two parts in terms of architecture: The node side with all nodes running the application to be controlled, and the client side with the monitoring application.

There is a single node playing a special role, namely the gateway node. It runs an application that forwards all TinyOS messages received by the radio to the workstation attached (e.g. over a serial port or an IP network port) and vice versa. This way, communication between the workstation running the monitoring application and the wireless sensor network can be established. Regarding achitecture, the gateway node will be considered as a part of the client side. Figure 3.2 shows a sample setup with the client workstation, one node acting as gateway node and four nodes running the application.

The next two subsections will give a brief overview of these two parts.

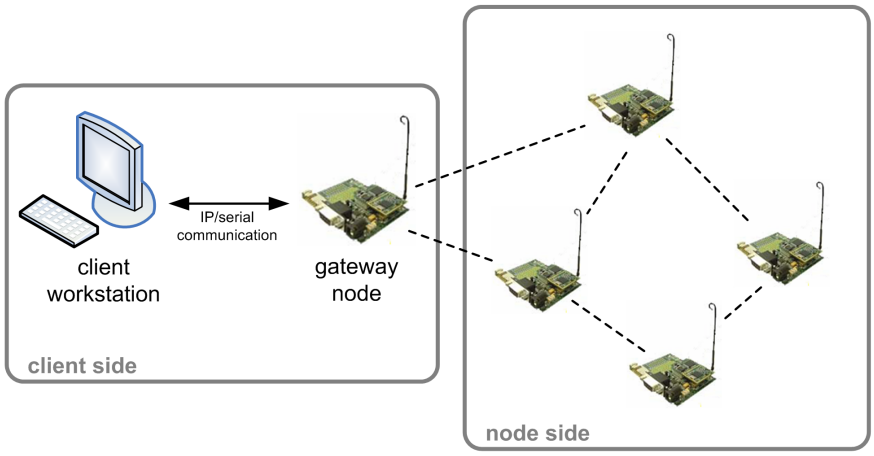

Figure 3.2: A sample setup visualizing the separation of client and node side.

3.1.1 The Client Side

The monitoring application running on the client workstation is written completely in Java and relies on the Eclipse platform. It is divided into several Eclipse plug-ins: the ones implementing the main functionality include the Serial Forwarder and the Sensornet Manager Plug-Ins. The former is responsible for sending and receiving messages, whereas the latter adds capabilities to manage the nodes. Figure 3.3 gives an overview.

User Interface

The Sensornet Manager Plug-In is responsible for keeping a model of the controlled wireless network and presenting it to the user. Figure 3.4 shows a screenshot of the client user interface in action[1]. The different views have the following purposes:

[1]The TinyOS Controlling perspective can be opened by selecting 'Window' → 'Open Perspective' → 'Other...' from the Eclipse main menu and then choosing 'TinyOS Controlling'.

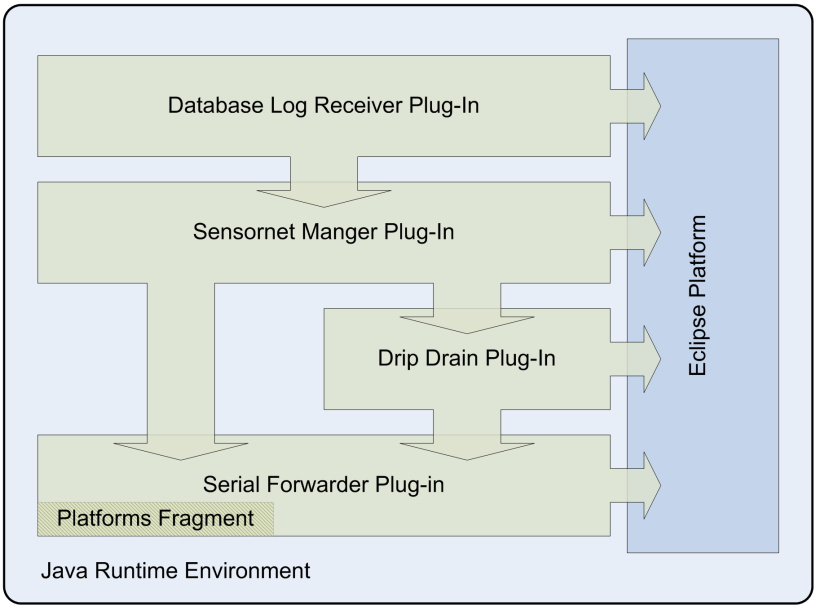

Figure 3.3: The client side architecture.

1. The navigator view is used to open and manage files.

2. Deployed nodes are shown in the editor view. The nodes can be moved so that they can be identified easier according to their actual physical position. Arrows indicate connectivity. The view can be zoomed and printed.

3. All available nodes are listed in the outline view with their installed modules, loggers, available variables and RPC commands.

4. Whenever information on a selected item (node, module, logger, RPC command or variable) is available, it will be shown in the properties view. The error log view (in the background) contains any errors or warnings.

5. The Serial Forwarder Plug-In contributes a view which permits setting up a connection to a gateway node and shows information about incomming and outgoing messages.

The used data model consists of objects representing nodes, their variables and RPC commands, the installed loggers and connectivity information. The whole data model can be stored as a file. A new data model can be created by selecting 'New' → 'Other...' from the Eclipse menu and then 'Sensornet' from the 'TinyOS Wizards' group. The subsequent dialog asks for a rpcSchema.xml and a nescDecls.xml file; these two files make up the installation information of the deployed nodes (see sections 3.2.3 and 3.3 for details).

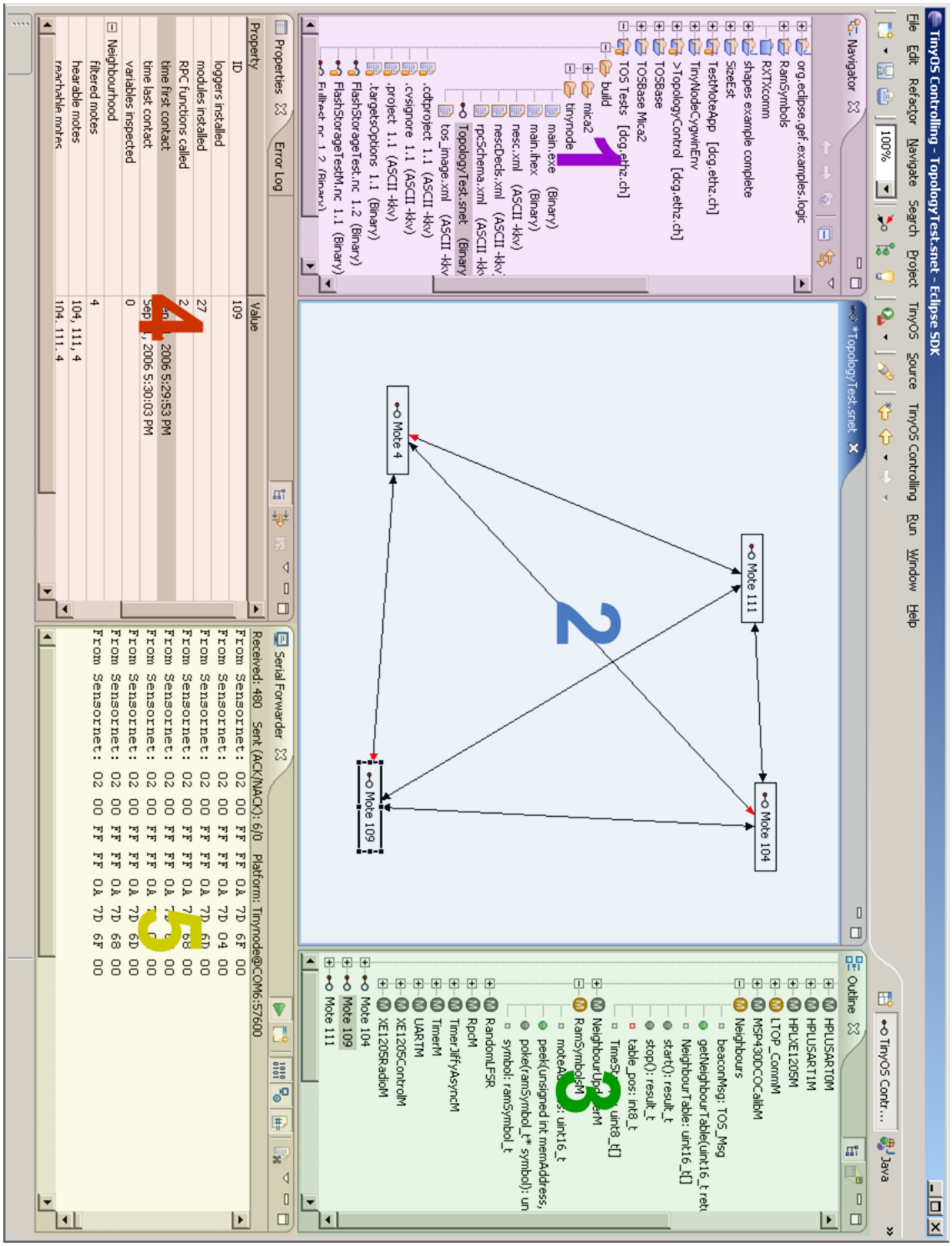

Figure 3.4: The client side controlling application.

3.1.2 The Node Side

On the node side, all components[2] of the architecture consist of nesC components. In figure 3.5, all implemented components are included. In order to save memory, components not needed for controlling a particular application may be omitted by the developer.

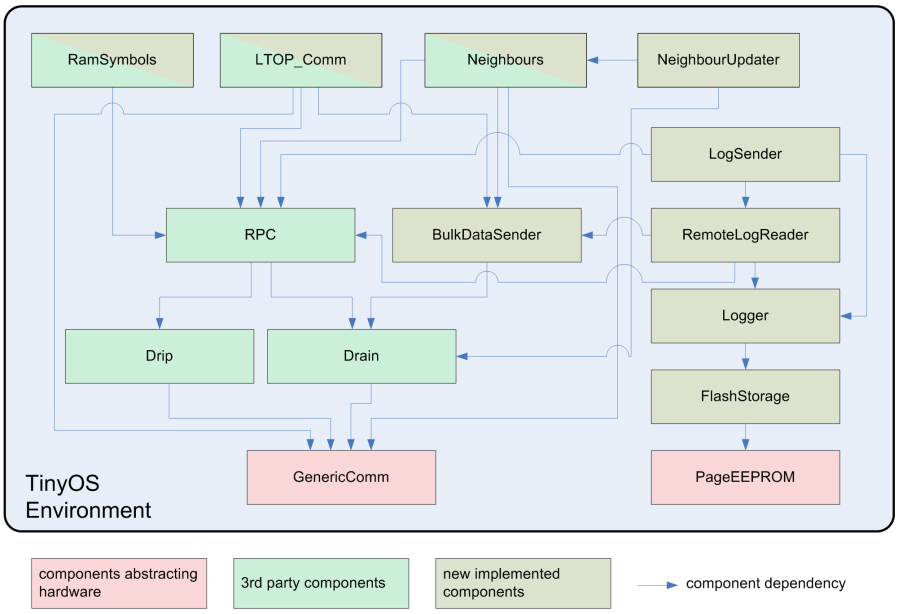

Figure 3.5: The node side architecture.

[2]When referring to components or subcomponents, they need not conform to nesC modules. The terms component and subcomponent are used to identify units of functionality on different granularity. However, in most cases, the actual nesC components implementing the described functionality resemble the structure as described in this document.

3.2 Communication, RPC

This section discusses communication between nodes and the client. This comprises several components as depicted in figure 3.6. First, an overview of the Serial Forwarder Plug-In is given. Then, the Drip/Drain layer used to route messages is introduced. The RPC and Bulk Data subsections will conclude this section.

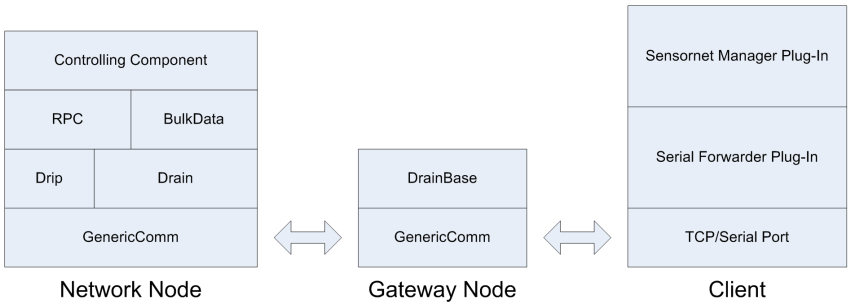

Figure 3.6: Communication between nodes and the client.

3.2.1 Serial Forwarder Plug-In

The Serial Forwarder Plug-In is responsible for the communication between applications running on the client workstation and the gateway node. The main functionality was ported from the existing stand-alone serial forwarder which comes with the TinyOS environment. The classic serial forwarder transmits packets comming in from the gateway node (connected to the workstation via a serial or IP connection) to any clients connected to a socket listening on a well known port and vice versa.

The Serial Forwarder Plug-In was implemented by paying special attention to the following points:

- A multitude of hardware platforms are able to run TinyOS. It should be easy to extend the Serial Forwarder Plug-In for new platforms.
- Other Eclipse plug-ins should be able to receive data packets by making use of extension points offered by the Serial Forwarder Plug-In.
- The Serial Forwarder Plug-In is able to act as a classic serial forwarder in that it forwards messages to and from any socket connected client application.
- The user interface should be intuitive and easy to use.

Extensibility

The platform extension point gives the developer the possibility to add support for additional platforms. Normally, communication between the connected node and the host running the serial forwarder takes place over a serial connection or

over an IP network. For this, a generic implementation is provided by the plug-in. Thus, the developer does not have to implement any Java code in order to add support for a new platform; providing an extension with connectivity details and a Java class describing the `TOS_Msg` struct (can be generated with the TinyOS message interface generator MIG) does the job.

The ch.ethz.dcg.controlling.platforms fragment extends this extension point in that it provides connectivity information for some known platforms such as Tinynode, the Mica family, Telos and Tmote.

However, it might be possible that the generic implementation for the platform is not appropriate, e.g. if the connection between the gateway node and the host running the serial forwarder is established differently. In such cases, a custom implementation can be provided.

Appendix C.1.1 gives information on the usage of the platform extension point.

Packet handling

There are three ways to hook into the plug-in to facilitate sending or receiving packets:

- Classic serial forwarder role: the client application connects itself through a socket to the serial forwarder.
- Raw message listener extension point: a software component implemented as Eclipse plug-in is able to get informed of any sent, received or failed (e.g. with invalid checksums or not acknowledged when supposed to) data by extending this extension point. See appendix C.1.2 for the extension point description.
- TOS message listener extension point: plug-ins making use of this extension point will be informed of any incomming TOS messages[3] addressed to the client (i.e. the ID of the gateway node or the broadcast ID). The header of the incomming message is processed so as to get the message type and the length of the message payload. Depending on whether extending plug-ins subscribed for the received message type, they are notified with a reference to the byte array of the entire message together with the position of the byte where the payload data starts and the length of the payload. Appendix C.1.3 describes this extension point.

Eclipse plug-ins receiving messages by making use of the latter two mentioned extension points send messages by calling the `send` method of the ch.ethz.dcg.controlling.messaging.TOSMessageSender class.

> **Note:** Message listeners must not change the provided buffer containing the packet data. The same received buffer will be used for all subscribed listeners in order to reduce overhead. Furthermore, listeners which consume time when processing incomming messages (especially when involving user interactions) should make a copy of the message and invoke a separate worker thread.

[3] A TOS (TinyOS) message defines the basic structure of any message. Defined fields include destination address, message type, length, payload, etc.

A plug-in extending a listener extension point has to ensure that the implementing listener does not throw any exceptions. If it does so, the listener will be considered as unsafe and will not be called anymore, i.e. it will be removed from the listeners list. In such a case, an appropriate entry is added to the Eclipse error log[4].

User interface

The user interface is realized as a view in Eclipse. If not present, the view can be made visible by choosing 'Window' → 'Show View' → 'Other...' from the Eclipse main menu and then selecting 'TinyOS Controlling' → 'Serial Forwarder'.

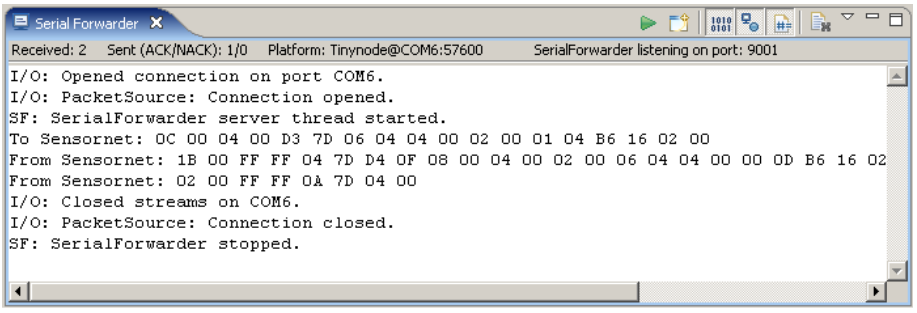

Figure 3.7: The serial forwarder view in Eclipse.

Figure 3.7 shows a screenshot of the serial forwarder view. The symbols in the upper right corner have the following functionalities:

- start/stop forwarding data
- select a platform and connectivity settings
- show/hide I/O messages
- show/hide messages rising from handling socket clients
- show/hide transmitted data
- clear output window

3.2.2 Drip and Drain

Same as in Pytos, multihop message routing between the client and deployed nodes is established over the drip and drain protocols. On the client side, drip and drain support is included in the separate Drip Drain Plug-In, because it primarly consists of code ported from the TinyOS Java tools. The Drain part of the Sensornet Manager Plug-In is only used for updating the drain tree. The Drain tree can be manually updated by clicking the 'Rebuild Drain Tree' ( ) button bellow the Eclipse main menu.

[4]The Eclipse error log can be made visible by choosing 'Window' → 'Show View' → 'Error Log' from the Eclipse main menu.

Drain messages received from nodes are interpreted by the Sensornet Manager Plug-In, because it knows the message format from the node's installation information. This makes it unnecessary to provide platform dependent Java classes describing the used `DrainMsg` struct.

Plug-ins that want to be notified of incomming drain messages can extend the ch.ethz.dcg.controlling.snetMgr.DrainMessageListener extension point. See appendix C.2.1 for its usage.

In order to reach as many nodes as possible, drip and drain resend messages on a certain intervall. However, if the network is small (i.e. the gateway node reaches all nodes directly), the user has the possibility to disable sending messages over the drip layer by deselecting the appropriate option[5].

3.2.3 RPC

The Sensornet Manager Plug-In inscpects the rpcSchema.xml file generated during the building process of the node application (figure 2.1). It extracts the following information:

- the name of the command
- the module it belongs to
- its return type
- the ID of the command. This ID is sent to the node in order to identify the command that has to be executed.
- for each parameter:
  - the name of the parameter
  - the type name of the parameter
  - whether the parameter is a pointer or not

RPC calls can be issued on any single node or on all nodes. Basically, calls can be initiated by right-clicking on the respective command in the outline view and selecting either 'execute RPC command...' or 'broadcast RPC command...'. The parameters have to be entered in hexadecimal format, least significant byte first, fitting the size of the referring type. For example, a parameter of type `uint_32` given the value `010A0000` would be interpreted by the node as 2561 in decimal.

RPC is not only used for giving the user the possibility to issue remote calls, it is also used to trigger actions on the node that send specific information to the Sensornet Manager Plug-In, such as logger data or information about a node's neighbourhood. This saves the developer from having to create specific messages.

[5]Choose 'Window' → 'Preferences...' → 'TinyOS Controlling' in Eclipse to access the TinyOS Controlling options.

3.2.4 Bulk Data

Under certain circumstances, the size of data to be sent from a node to the client is not known when developing the application and it possibly does not fit into a single TOS message. For this purpose, larger data structures can be split up into several bulk data messages on the node and reassembled on the receiving client. Figure 3.8 shows the message format of a single bulk data message.

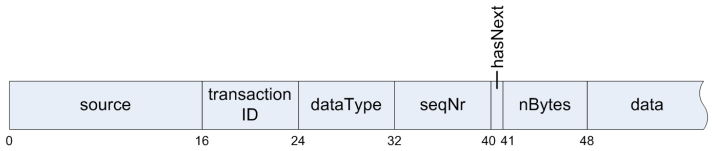

Figure 3.8: The bulk data message format.

The bulk data message fields are specified as follows:

- *source* – the local address of the node sending the message
- *transactionID* – an ID identifying messages that belong to the same data structure; will be incremented by the node for every new transaction
- *dataType* – an identifier for indicating the type of the data transmitted
- *seqNr* – the sequence number of this packet; used to detect lost messages and to reassemble the data structure in the right order
- *hasNext* – indicates whether more messages are following (1) or not (0)
- *nBytes* – the size (in bytes) of the data field
- *data* – the data portion

The Bulk Data component is kept simple and small. Therefore, it is not possible to request retransmission of single parts, i.e. if a single message is lost, the whole transaction failed.

Currently, this component is used to transmit neighbourhood information and the list of installed loggers.

On the client side, the Sensornet Manager Plug-In is capable of receiving bulk data messages and putting them back together to the original data structure. Figure 3.9 schematically depicts the involved steps. Bulk data transactions are identified by the address of the sending node and a transaction ID the node assigns to the transaction. Once a transaction is closed, it is removed from the list of open transactions.

The ch.ethz.dcg.controlling.snetMgr.BulkDataMsgListener extension point offers the possibility to add listeners for bulk data (see appendix C.2.2 for a description).

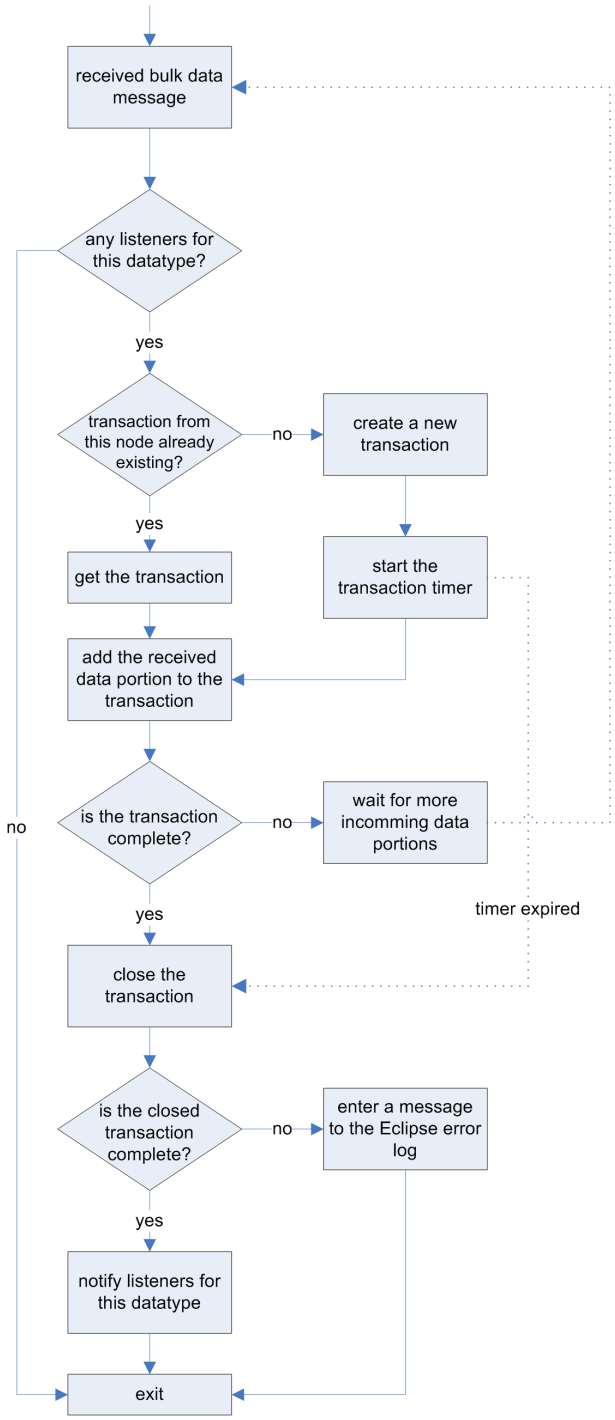

Figure 3.9: A flow chart of the bulk data message handling.

3.3 Variable Introspection

On the node side, this service is implemented in the `RamSymbolsM` module by two RPC enabled commands: Peek (for getting a variable) and Poke (for setting a variable). The original version of this module included in the TinyOS distribution was slightly modyfied and now includes a variable holding the ID of the node, allowing the client application to identify deployed nodes.

To get rid of available variables, the Sensornet Manager Plug-In extracts the following information from the nescDecls.xml file (for each variable):

- the name of the variable
- the module it belongs to
- its type name
- its static address in the node's memory
- its length (i.e. the size it occupies in memory)
- whether it is an array
- whether it is a pointer

Only global variables ascribable to modules are taken into account. Variables introduced by TinyOS which do not belong to any module such as `TOSH_queue`, `TOSH_sched_full`, `TOSH_sched_free`, `LPMode_disabled`, `TOS_AM_GROUP` or `TOS_LOCAL_ADDRESS` are left out.

Getting or setting global variables remotely is as simple as calling RPC commands. Right-click on any variable in the outline view and choose one of 'retrieve value', 'retrieve value from all motes', 'set value...' or 'set value on all motes...'. In order to set a variable, the new value has to be entered in hexadecimal format, least significant byte first. When getting or setting a variable, the state of the underlying RPC call can be observed in the Eclipse properties view by selecting the respective call in the `RamSymbolsM` module.

3.4 Logging

The logging service is described in two parts: first, an overview of the node side is explained, then, focus is set to elements involved on the client side.

3.4.1 Node Side Implementation

When developing logger support for nodes, several criteria were considered:

Persistency: Log entries should be stored persistent on nodes. In case a node crashes or reboots, as few log entries as possible should be lost.

Space efficiency: Organizing data in memory introduces a certain amount of unavoidable overhead. However, this overhead should be kept as small as possible.

Minimize memory accesses: The included flash storage on the node is the only way to persistantly store larger portions of data. It bases on the EEPROM (Electronically Erasable Programmable Read-Only Memory) technology. The memory is not really read only how the name suggests, but writing and erasing worsens its lifespan. EEPROM memory typically wears out after some 10'000 erase-write cycles[6]. Therefore erase-write cycles should be rare.

Multiple loggers: A developer might want to observe different characteristics of an application. Therefore, multiple loggers should be supported.

Recovery: In case a node is rebooted, already existing log entries should not be overwritten. New log entries should be appended to the first available position.

Memory management: Other components might also make use of the flash memory. Thus, it should be possible to allow the developer to partition the storage.

Remote access: It should be possible to read logged entries remotely.

Overview

The Logging component is divided into subcomponents:

- The FlashStorage subcomponent, responsible for read/write accesses to the flash memory.
- The Logger subcomponent, implementing the logging functionality.
- The RemoteLogReader subcomponent, enabling the client to request (*pull*) log entries remotely.
- The LogSender subcomponent, which sends logged entries on a regular basis to the client (*push*).

[6]Source: http://en.wikipedia.org/wiki/EEPROM, last accessed on September 7, 2006

While the FlashStorage and Logger subcomponents are mandatory in order to make use of the logger mechanism, the RemoteLogReader and the LogSender components are optional. A possible scenario could be that the developer decides to run the nodes without the latter two components, collect the nodes after some time and install a different application to read out the log entries.

FlashStorage Subcomponent

The FlashStorage subcomponent relies on the PageEEPROM component provided by TinyOS to access flash memory. EEPROM memory is divided into pages. The Tinynode or Mica2 nodes both come with an Atmel AT45DB041 flash memory, which has 2048 pages of 264 bytes each.

The subcomponent has to be initialized before it can be used. The `init` command of the FlashStorage interface has to be called, providing a parameter indicating relative to which page the FlashStorage will compute the offsets.

Data that has to be written to or is read from the flash memory is buffered in memory. Figures 3.10 and 3.11 illustrate the steps involved for a read or write operation. A charistic of EEPROM is that in order to write a single page, the respective page in memory has first to be erased. This is also assured by the FlashStorage subcomponent.

Logger Subcomponent

As stated before, write operations to flash memory should be rare. On the other side, a requirement of the Logger component is to store log entries persistently. As a response to this trade off (persistency vs. sparing write operations), the Logger component organizes logged data in pages: a logger consists of a certain number of pages, each page containing a fixed number of entries, each entry having a fixed size. Thus, persistence is guaranteed on a page granularity. The developer is able to set the different parameters according to the above mentioned trade off.

However, page sizes exceeding the flash memory page size are not of big use, because once the write position of a new log entry exceeds the flash memory page bounds, the buffer has to be flushed and written to memory anyway. The only (for such sizes mostly neglectible) benefit is that—all in all—metainformation found at the end of every page (currently 4 bytes) will consume less memory for a certain amount of log entries. An example illustrating how to initialize a logger is given in appendix B.1.

The logger subcomponent stores its information to flash memory as follows: At the beginning, information about the installed loggers is written. For each logger, this comprises:

- the start position: the location in memory where the first log entry can be found
- the ID identifying the logger
- the length of a single log entry
- the number of pages to be reserved
- the number of entries per page

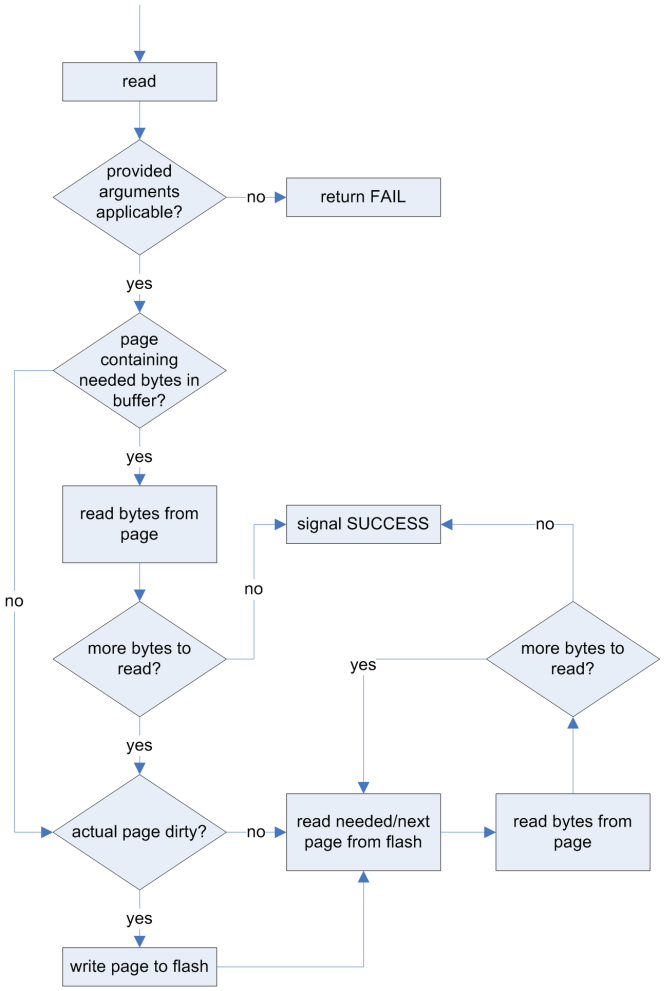

Figure 3.10: A flow chart of the FlashStorage read operation.

This information has to be updated only if a new logger is added. After this information, log data for the respective loggers is stored.

The logger is implemented as a circular logger: once the last page in memory was written, subsequent pages will overwrite the oldest pages. At the end of each page, a sequence number and the number of entries contained in the page is appended. The sequence number is needed to find the last used page of the circular logger. The sequence number is set as follows:

```
actualPage.seqNr = (prevPage.seqNr + 1) % (nPages + 1);
```

where `nPages` contains the number of pages reserved for this logger. Figure 3.12 illustrates a simple example.

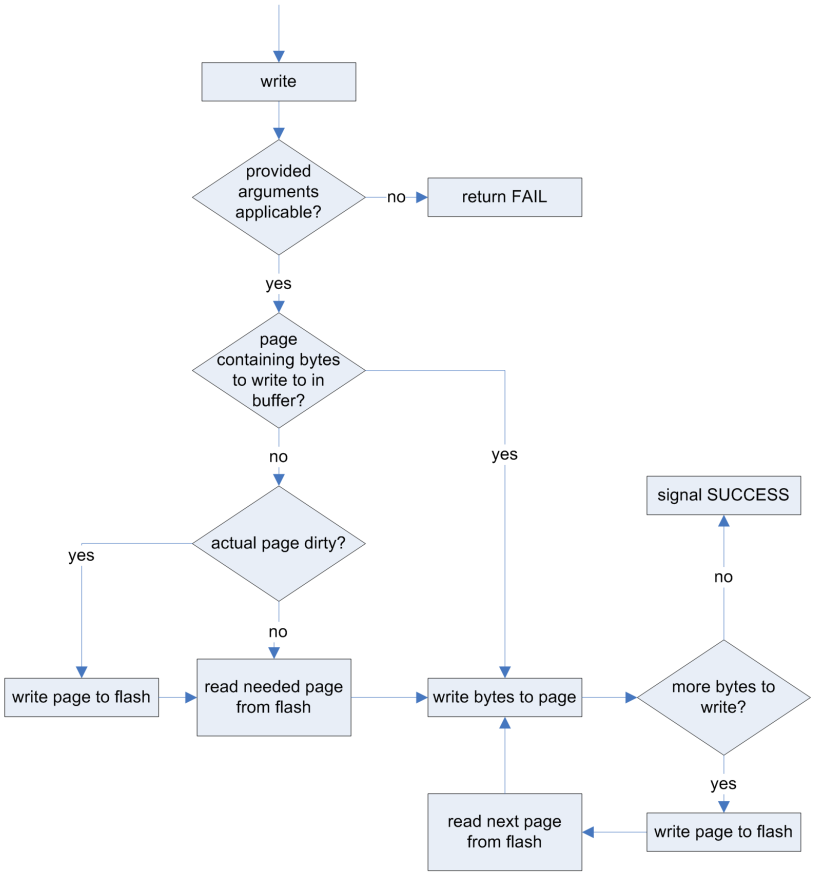

Figure 3.11: A flow chart of the FlashStorage write operation.

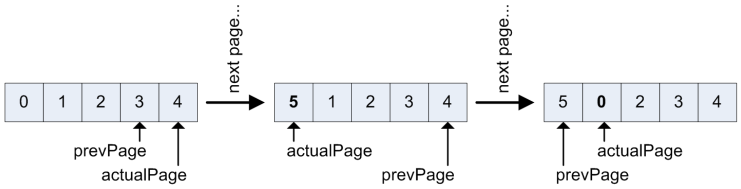

Figure 3.12: How sequence numbers are set. In this example, nPages is 5. Each box represents a page and its sequence number.

The newest page can then be found as follows:

```
go to next page until:
(actualPage.seqNr + 1) % (nPages + 1) != nextPage.seqNr
```

In case the logger tries to find the newest page (e.g. after a restart of the node) and only a part of the reserved space was used before, the logger assumes that the remaining space is clean, more precisely that it was erased before and

not changed afterwards[7]. When a flash memory page is erased, all bits are set to 1, because – and this is another characteristic of EEPROMs – bits can only be switched from 1 to 0; in order to reliably switch bits from 0 to 1, the corresponding page has to be erased. Therefore the maximal possible sequence number is hexadecimal `0xFFFE` (assuming two bytes reserved for sequence numbers). Thus, the number of pages must not be bigger than `0xFFFD` (=65533 in decimal).

The `Logger` interface comprises three commands: `init` initializes a logger, `append` appends a single log entry to the logger and `flush` forces writing the actual page to flash memory. Each operation returns `SUCCESS` or `FAIL`, depending on whether the operation could be started. The final result of the operation will be signalled as an event.

The `LogReader` interface provides commands to read entries with a given offset, get logger metadata, get the number of installed loggers and the possibility to lock or unlock the Logger subcomponent. The lock operation stops the logger from writing new entries. This enables reading entries in different calls without having to worry about changing offsets caused by possible appends.

RemoteLogReader Subcomponent

The RemoteLogReader subcomponent makes use of the `LogReader` interface. It provides RPC commands to get logger metadata, read logger entries and start/stop the Logger subcomponent. Basically, it exposes the `LogReader` interface to be used remotely.

Because the size of the logger metadata depends on the number of installed loggers, it can not be guaranteed that this metadata fits into a single message for transmitting this information to the client. Therefore, the Bulk Data component is used to transmit this data.

To transmit logger entries, the RemoteLogReader subcomponent uses the message format shown in figure 3.13. Its layout is basically the same as for bulk data, except the *nEntries* and *loggerID* fields. Opposed to the bulk data message where the *nBytes* field contains the number of bytes, the logger read message *nEntries* field contains the number of log entries that follow in the payload. The *loggerID* is the ID of the logger which stored the delievered entries. By checking the included message sequence number, lost messages can be identified and the missing entries may be read in a separate transaction by giving the correct offset.

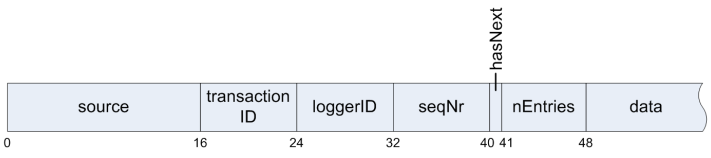

Figure 3.13: The logger read message format.

[7]To get sure that the flash memory is clean, the framework distribution supplies a simple application that erases the storage. See the installation guide (appendix A) for more information.

LogSender Subcomponent

The LogSender sends log entries to the client depending on a threshold value n it is initialized with. It counts the successful log appends. If the threshold value is reached, it sends the last n log entries to the client and resets the counter.

Limitations

The Logger component supports multiple loggers. However, using multiple loggers alternately will undermine the 'write as seldom as possible' rule, because it is unlikely that entries of two different loggers reside on the same page in flash memory, thus causing page writes in order to swap pages.

However, supporting multiple loggers still makes sense: different applications might initialize different loggers, without the need to affect the loggers belonging to previously installed applications.

3.4.2 Client Side Implementation

Logger information can be retrieved either in a *push* or *pull* manner: Depending on the configuration, a node might transmit logger entries on a regular basis. On the other hand, logger entries can be requested by the client[8].

The client has to be configured appropriatly by adding logger listeners in order to receive logger data residing from transmissions initiated by nodes[9].

Receiving Logger Entries

Logger entries are transmitted to the client similiar to bulk data, but without splitting single log entries among two messages, i.e. a single message contains a number of whole log entries. The Sensornet Manager Plug-In passes null values for lost log entries residing from lost messages. This way, the user can request the lost log entries by giving the correct offset. Because new entries might be appended to the log on the node once the transaction has completed, the offset for getting the lost entries changes. Therefore, the user has the possibility to stop the logger manually, so that the offset will not change until the user lets the logger resume its work[10].

If some messages are lost during a transaction, an entry containing the transaction ID and the node address is added to the Eclipse error log.

Received log entries are reported to listeners extending the ch.ethz.dcg.controlling.snetMgr.LogReceiver extension point. The Sensornet Manager Plug-In includes two extensions: one to print the received entries to a file, and one to print the received entries to the Eclipse console view. The Database Log Receiver Plug-In also receives log entries by extending this extension point (see appendix C.2.3 for details).

[8] A list of installed loggers can be obtained by right clicking on a node and then selecting 'Get Logger Info'. Logger entries can be retrieved by right clicking on the logger under the desired node in the outline view and then selecting 'Get Logger Data'.

[9] Logger listeners can be added by right clicking on the respective logger and then selecting 'Add Logger Listener...'.

[10] The two RPC commands 'stopLogger' and 'resumeLogger' and the 'loggerStopped' variable in the 'RemoteLogReaderM' module allow remote control of the logger module.

Database Log Receiver Plug-In

This plug-in enables inserting received log entries into a specific database. This way, any application capable of accessing an SQL (structured query language) database can make use of stored log entries. BIRT (Business Intelligence and Reporting Tools) is such an application. It can be used to produce reports and is also implemented as an Eclipse plug-in. An example is given in appendix B.3.

Extensibility There is a multitude of SQL based databases, each of them with different characteristics and ways to access it. The JDBC (Java Database Connectivity) API provides an abstraction to access databases in a uniform manner. In order to make use of the JDBC API, database specific drivers have to be provided.

Currently, the Database Log Receiver Plug-In includes the MySQL connector JDBC driver which facilitates access to MySQL databases. However, support for additional databases can be achieved by extending the ch.ethz.dcg.controlling.db LogReceiver.dbDriver extension point. Appendix C.3.1 describes this extension point.

Configuring a Database Log Receiver When a log listener is added or log data is requested from a node, the user is prompted with a window as shown in figure 3.14 to enter specific settings: the JDBC driver to be used and connection information needed to access the database. Furthermore, the user has to enter SQL `INSERT` statements that will be used to add log entries to the specified database. The transaction statement is executed once for each transaction, while the entry statement will be executed once for each log entry. Table 3.1 shows a list of strings inside these statements that will be replaced with the actual values.

substitutable	substituted by	SQL type
`<transactionID>`	ID of the transaction	`INTEGER`
`<moteID>`	ID of the node	`INTEGER`
`<loggerID>`	ID of the logger	`INTEGER`
`<entriesRequested>`	number of entries requested	`INTEGER`
`<offset>`	offset given with the request	`INTEGER`
`<dateStarted>`	date of when the transaction started	`DATE`
`<timeStarted>`	start time of the transaction	`TIME`
`<timeStartedMs>`	millisecond fraction of the time when the transaction started	`INTEGER`
`<callDuration>`	duration of the transaction in milliseconds	`INTEGER`
`<pos>`	position of the log entry relative to the given offset	`INTEGER`
`<value>`	log entry value	`VARBINARY` or `VARLONGBINARY`

Table 3.1: Strings that may be used inside INSERT statements that will be replaced by actual values.

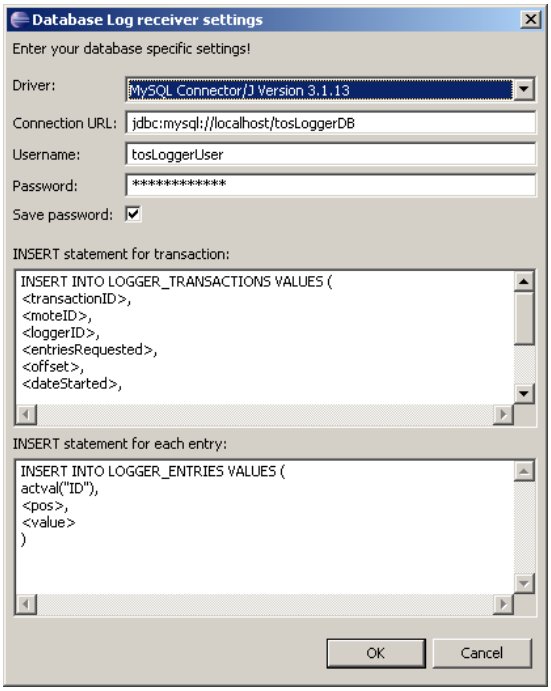

Figure 3.14: A screenshot of the database log receiver settings window.

3.5 Topology Control

The topology control component is based on previous work of Severin Winkler [8]. It is adapted to the controlling framework and now consists of three subcomponents, namely the Neighbours, NeighbourUpdater and the LTOP_Comm (logical topology communication) subcomponents. On the client, the topology can be influenced in the graphical editor.

3.5.1 Neighbours Subcomponent

The Neighbours subcomponent is responsible for collecting information on other nodes in a node's neighbourhood and exposing this information to the client.

This subcomponent periodically sends out a beacon message. It receives beacon messages from other nodes and maintains a list of recently heared nodes. This list can be retrieved by the client by calling a RPC command.

RPC commands to remotely control this component (i.e. start or stop sending beacon messages) were added. This way, precious battery power may be saved if topology information is not relevant over longer periods of time.

3.5.2 NeighbourUpdater Subcomponent

Whenever a new neighbour is detected or no more information about an existing neighbour is received, the Neighbours subcomponent changes its table accordingly and signals an event. The NeighbourUpdater handles such events and sends update messages to the client on any change.

3.5.3 LTOP_Comm Subcomponent

This subcomponent gives the developer the possibility to enable or disable certain links between nodes, e.g. when testing a new routing implementation, thereby establishing a logical topology.

Nodes whose messages should not be delievered to the application are included in a black list. RPC commands to add or remove node addresses from this black list and to send the black list to the client are implemented by this subcomponent.

The TinyOS GenericComm component is the usual component for sending and receiving messages. The same interface as the GenericComm component provides is offered by the LTOP_Comm subcomponent. This way, only minor changes to the code are needed in order to make use of the LTOP_Comm subcomponent.

The LTOP_Comm subcomponent can be seen as an additional layer on top of the GenericComm component. It includes an additional header to the TOS_Msg, therefore less space will be available for the payload data. Instead of using the `TOSH_DATA_LENGTH` constant to find out the length of the payload field the `LTOP_DATA_LENGTH` constant must be used.

Basically, LTOP_Comm and the Neighbours subcomponents are independent of each other. However, it makes sense to include the Neighbours subcomponent when including the LTOP_Comm subcomponent to determine which node addresses should be included in the black list.

3.5.4 Influencing the Topology

The topology is visualized in the client by the Sensornet Manager Plug-In in the graphical editor (figure 3.15). Black arrows indicate connectivity. A red arrow pointing to a specific node signifies that this node actually hears the node at the other end of the line, but messages residing from that node are not delievered to the running node application.

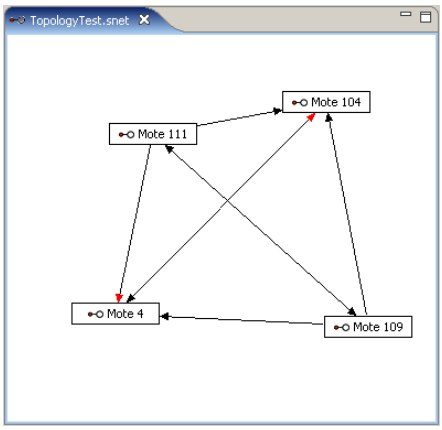

Figure 3.15: Visualization of topology information on the client.

Topology information can be retrieved by calling the `getNeighbourTable()` RPC command on the nodes (`Neighbours` module) or by clicking on the 'Broadcast request for topology information' ( ) button bellow the Eclipse main menu. The topology can be influenced by calling the respective RPC commands (`includeLink()` and `removeLink()`) in the `LTOP_CommM` module or by right clicking on the respective arrows in the editor and selecting 'put on black list' or 'remove from black list', respectively.

Chapter 4

Outlook

4.1 Future Work

The controlling framework offers functionality to govern a wireless sensor network. However, the framework is not (and will never be) exhaustive—needs of developers evolve! Some needs arising in the near future could be:

TinyOS 2.0 support: The components included in the controlling framework were developed against TinyOS 1.1.15. At the time of writing this thesis, TinyOS 2.0 is emerging and a beta version is already available. If porting the framework to TinyOS 2.0 makes sense, it has to be analyzed if and how different components could be ported and whether new components included in the TinyOS 2.0 environment could be leveraged.

Integration with TinyOS Plug-In: Another path for future work could be an integration with the existing TinyOS Plug-In[1] developed by Roland Schuler. This plug-in adds TinyOS application programming support to the Eclipse platform. Everytime an application is rebuilt, the available RPC commands and global variables change, making it necessary to reset the model used by the framework client side. The build features of the TinyOS plug-in could help out here.

Visualization: The graphical visualization feature offers information on which nodes can communicate with each other but is still in an early stage. New features could include the possibility to visualize the network on top of a map illustrating the environment. Furthermore, nodes not having given a sign of life for some period could be marked specially.

The gateway node is not included in the visualization; the client could interpret beacon messages received from neighbouring nodes and complete the graph accordingly.

Routing, power consumption: The RPC component relies on the Drip and Drain components to send and receive messages. The wiring of these components is hard coded into the `generateRpc.pl` script found in the `tools/scripts/codeGeneration` directory of the TinyOS environment.

[1] http://www.dcg.ethz.ch/~rschuler/

Thus, the developer has no way to influence this wiring except by editing this script. However, he may want to include other routing algorithms, e.g. to decrease power consumption. This could be acomplished by providing an other script that acts similiar to the `generateRpc.pl` script, but relying on components the developer includes by himself to the application.

Different applications in a single network: Currently, the framework requires all nodes to run the same application. Supporting multiple applications requires some effort; for instance, broadcasting RPC calls would no longer be possible this way, because nodes with different applications offer different RPC commands.

Variables hodling big structures: In the current version, variable values are sent to the client as the return value of an RPC call. Thus, bigger variables have to be read manually in multiple subsequent calls. The Bulk Data component could tie in here.

4.2 Conclusion

The wireless sensor controlling framework provides means to govern an application running on nodes deployed in a wireless sensor network. It was developed with focus on extensibility, usability and platform independence.

On the node side, this is achieved by different TinyOS components that can be included beside the actual application as the need arises. The functionalities these components feature include:

- calling commands on nodes remotely with remote procedure calls (RPC)
- inspecting and influencing application state by retrieving and setting global variables
- sending large data structures to a connected client
- logging any kind of data persistently to local storage and retrieve logged data remotely
- keeping track of the neighbourhood of every single node by sending and receiving beacon messages to or from neighbours, with the possibility to remotely inspect this information
- influencing the connectivity in that the actual application is only able to receive messages from a subset of hearable neighbours

The client side of the framework is implemented as Eclipse plug-ins. This way, the framework can be extended by implementing Eclipse extensions. Functionalities offered by the provided plug-ins include:

- communication with nodes in the network over a serial or IP connection
- utilize information generated at compile time in order to access application state on nodes
- remotely control nodes by making use of RPC

- receive large data structures from single nodes
- retrieve log data from nodes and store this data in files or insert it into a database
- graphically visualize a sensor network, including its topology

4.3 Personal Experience

To me, working on this master thesis was

instructive: I learned a lot during the six months of this thesis: on the topic of wireless sensor networks, how specific problems in this field are solved. I gained insight into the nesC programming language, was able to capture new programming techniques employed in the Eclipse framework, gained some insight into the Python programming language, learned a lot about user interface programming using SWT (the Standard Widget Toolkit) and GEF (the Graphical Editing Framework).

fun: Making progress, getting things working, the relief after having found solutions to peculiar problems.

frustrating: Because it sometimes took a lot of time finding a solution to such peculiar problems—or even the problem itself!

opened new horizons: Although I can claim to have some knowledge on programming, I constantly find new ways to solve problems. It's enlighting to see how different people solve different problems.

challenging: The field of wireless sensor networks was new to me and my C programming skills were basic. I had the chance to dive into new waters and gain more experience.

Appendix A

Installation Guide

Java

The controlling framework client runs inside the Java runtime environment, version 1.5 or above. Java can be downloaded from http://java.sun.com/javase/downloads/index.jsp. Please follow the installation instructions provided there.

Eclipse

The client side plug-ins base on the Eclipse platform, which can be downloaded from http://www.eclipse.org/downloads/. The plug-ins were developed against version 3.1, but they also run on the latest version 3.2. Eclipse is available on different platforms; the website offers more information on how to install it.

The Sensornet Manager Plug-In relies on the GEF (Graphical Editing Framework). Download and install the version of GEF corresponding to your Eclipse version from http://www.eclipse.org/gef/. The GEF runtime download will suffice, you don't have to download the whole SDK.

After the successful installation of Eclipse, copy the contents of the `/bin/eclipse` directory of the framework distribution to your Eclipse installation directory.

The installation can be checked by clicking 'Help' → 'About Eclipse SDK' from the Eclipse main menu and then clicking on 'Plug-in Details'. The following plug-ins should appear in the list: RXTXcomm, TinyOS Controlling Database Log Receiver, TinyOS Controlling Drip/Drain, TinyOS Controlling Platforms, TinyOS Controlling Sensornet Manager and TinyOS Controlling Serial Forwarder.

In Eclipse, the TinyOS Controlling perspective can be made visible by selecting 'Window' → 'Open Perspective' → 'Other...' from the main menu and then choosing 'TinyOS Controlling'.

TinyOS 1.x

The components running on the nodes were developed against TinyOS version 1.1.15. An installation distribution for version 1.1.0 exists, which you can down-

load from http://www.tinyos.net/download.html and install. This will prevent you from having to set up the toolchain manually. If you are using Microsoft Windows, you first have to update the Cygwin environment installed by the above distribution. See http://www.tinyos.net/scoop/special/howto_upgrade_cygwin for more information.

To get the most recent version of TinyOS, check out the `tinyos-1.x` module from the TinyOS Sourceforge website. If you used the installer, your TinyOS installation will be found in the `/opt/` directory inside your Cygwin environment. It is best to check out the TinyOS module to this directory. More information on how to check out the module can be found on http://sourceforge.net/cvs/?group_id=28656.

It is a good idea to keep the TinyOS installation up to date by checking for updates regularly. Depending on your platform, additional steps might be necessary in order to get the environment running. Check the manufacturer's installation notes!

In order to use the client side plug-ins, it is not necessary to compile the TinyOS Java tools.

NesC Compiler

To compile applications including provided components, the nesC compiler, version 1.2 or above is needed. The compiler can be downloaded from http://sourceforge.net/projects/nescc. After downloading the tar file, unpack it to a reasonable directory such as `/opt`, change to the `/opt/nesc-1.2.x` directory and run `./configure`, `make` and then `make install`.

The compiler version can be checked by calling `nescc --version`.

Additional Components

XML file generation

The XML file generation during the building process relies on Perl. Therefore, be sure that Perl is installed, including the `XML::Simple` module. Under Cygwin, this module can be installed by running `cpan install XML::Simple`, accepting default values.

Gateway Node

To enable communication between the client and the nodes deployed in the network, a single node has to play the special role of the gateway. This node is connected to the client workstation, normally through a serial connection or an IP connection. The node has to be installed with the `DrainBase` component by typing `make <platform> install` in the `beta/Drain/DrainBase` directory of the TinyOS environment.

`DrainBase` is a modified version of the better known `TOSBase` and supports message acknowledgement needed by the Drain layer. This feature is enabled in the radio module, so if you use another radio module than the one wired into the provided `DrainBase`, be sure that acknowledgements are enabled. For the Tinynode platform, a specific version can be found in the

src/tinyos/Controlling/DrainBaseTinynode/ directory of the framework distribution.

Cleaning Flash Storage

The src/tinyos/Controlling/EraseFlash/ directory of the framework distribution contains a simple application that deletes all data in flash memory. Once installed (by typing make <platform> install in this directory), the yellow led will light during the erase process, following by a green light indicating that the flash storage was successfully deleted.

Building TinyOS Components

The TinyOS components provided by the framework can be found in the src/tinyos/ directory of the framework distribution. It is recommended to copy the contents of this directory to the apps/ directory of your TinyOS environment.

Before building an application containing RPC commands and including the RamSymbolsM module that enbles getting and setting global variables, the following lines have to be added to the application's Makefile:

```
TOSMAKE_PATH += $(TOSDIR)/../contrib/nucleus/scripts
CFLAGS += -I$(TOSDIR)/../tos/lib/Rpc
CFLAGS += -I$(TOSDIR)/../tos/lib/Drip
CFLAGS += -I$(TOSDIR)/../tos/lib/Drain
CFLAGS += -I$(TOSDIR)/../contrib/nucleus/tos/lib/Nucleus/
```

To make use of the different components provided by the framework, add the following lines to your application's Makefile (assuming you copied the directories containing the implementations to the TinyOS apps directory before):

```
PFLAGS += -I$(TOSDIR)/../apps/Controlling/RamSymbols
PFLAGS += -I$(TOSDIR)/../apps/Controlling/TopologyControl
PFLAGS += -I$(TOSDIR)/../apps/Controlling/BulkDataSender
PFLAGS += -I$(TOSDIR)/../apps/Controlling/Logger
```

In order to enable the generation of the nescDecls.xml and rpcSchema.xml files during the building process of a TinyOS application, start the building process with the usual make command, but including the nescDecls and rpc parameters. An example for the Tinynode platform would look as follows (the application would be installed on the node attached to serial port 4 and the node would be given the address 32):

```
make tinynode nescDecls rpc bsl,3 install,32
```

If the building process completed successfully, the nescDecls.xml and rpc Schema.xml can be found in the build/<platform> directory of your application.

Enabling RPC support

To add RPC support to your TinyOS application, include and wire the `RpcC` component by completing your top level application configuration file as follows:

```
components Main, RpcC;

Main.StdControl -> RpcC;
```

More information on the usage of RPC can be found in appendix B.2 and on http://nest.cs.berkeley.edu/nestfe/index.php/Rpc.

Enabling getting/setting Global Variables

To be able to get and set global variables remotely, RPC support has to be enabled. Additionally, include and wire the `RamSymbolsM` module as follows (in your top level application configuration):

```
components Main, RpcC, RamSymbolsM;

Main.StdControl -> RpcC;
Main.StdControl -> RamSymbolsM;
```

Making use of Loggers

The `Makefile` of your application has to be completed by the following lines:

```
CFLAGS += -DMAX_LOGGERS=10
CFLAGS += -DLOGGER_STARTPAGE=0
CFLAGS += -DLOGGER_BASE_ADDR=0
```

The first line determines how many loggers may be installed at most on a node. This value influences memory consumption both in RAM and in flash memory. The second line indicates the starting page in flash memory where the logger component will store its information. The last line has only to be included if the `LogSenderC` component is included. It specifies the address where to send log entries, which normally is the address of the gateway node.

The `LoggerC` component has to be wired and included as follows:

```
components Main, LoggerC, YourApp;

Main.StdControl -> LoggerC;
YourApp.Logger -> LoggerC.Logger[YOUR_APP_LOGGER_ID];
```

And in order to enable remote access to the logger, complete your top level application configuration as follows:

```
components Main, LoggerC, RpcC, RemoteLogReaderC, YourApp;

Main.StdControl -> LoggerC;
Main.StdControl -> RpcC;
RemoteLogReaderC.LogReader -> LoggerC.LogReader;
YourApp.Logger -> LoggerC.Logger[YOUR_APP_LOGGER_ID];
```

To periodically send new log entries to the client include the `LogSenderC` component as follows (the `RemoteLogReaderC` component will be included by the `LogSenderC` component):

```
components Main, LoggerC, RpcC, LogSenderC, YourApp;

Main.StdControl -> LoggerC;
Main.StdControl -> RpcC;
LogSenderC.LogReader -> LoggerC.LogReader;
YourApp.Logger -> LoggerC.Logger[YOUR_APP_LOGGER_ID];
// if you want to initialize the LogSenderC
// on the node instead of remotely:
YourApp.LogSender -> LogSenderC.LogSender[YOUR_APP_LOGGER_ID];
```

Using the Topology Components

To be able to retrieve information about the topology, the `NeighboursC` component has to be included as follows to your top level application configuration:

```
components Main, RpcC, NeighboursC;

Main.StdControl -> RpcC;
Main.StdControl -> NeighboursC;
```

To notify the client workstation about any topology changes, the `NeighbourUpdaterC` component is added (`NeighboursC` will be included automatically). Additionally, the `Makefile` has to be extended by the line

```
CFLAGS += -DBASE_ADDR=0
```

which indicates where the update messages have to be sent (typically the gateway node address).

```
components Main, RpcC, NeighbourUpdaterC;

Main.StdControl -> RpcC;
Main.StdControl -> NeighbourUpdaterC;
```

And to make use of the `LTOP_CommC` component:

```
components Main, RpcC, LTOP_CommC, YourApp;

Main.StdControl -> RpcC;
Main.StdControl -> LTOP_CommC;
YourApp.SendMsg -> LTOP_CommC.SendMsg[AM_TYPE];
YourApp.ReceiveMsg -> LTOP_CommC.ReceiveMsg[AM_TYPE];
```

Possible Pitfalls

It will not take you long to find out that it's not that easy to get all things running quickly. Therefore, some problems and their solution are discribed here. Of course, and unfortunately, the list is far from being complete.

Initializing Components

Make sure to actually wire components providing the `StdControl` interface to `Main` or another component using this interface.

Keep in mind that the order of how you wire single components may matter. For example, the `LoggerC` component has to be initialized before setting up individual loggers.

Generation of nescDecls.xml

Generation of the `nescDecls.xml` file is performed by the `generateNescDecls.pl` Perl script located in the `tools/scripts/codeGeneration/` directory. The current version on the TinyOS CVS of this file includes a call to the `avr-objdump` command. This works fine with AVR based platforms. However, if you use another platform, this command may have to be changed. E.g. for Tinynode, replace this command by the `objdump` command.

Using the Tinynode Platform

If you arranged your TinyOS environment to the Tinynode platform, it may be possible that the `rpc` and `nescDecls` targets for `make` are not found. This may be due to the fact that Tinynode uses its adapted build mechanism which is located in `contrib/shockfish/tools/make` of the TinyOS distribution. A possible solution to this issue is to copy the `nescDecls.extra` and `rpc.extra` files from the `tools/make/` directory to the `contrib/shockfish/tools/make` directory.

Appendix B

Examples

The prupose of this chapter is to give examples on how to use services provided by the framework. We therefore consider a sample application that makes use of these functionalities.

The sample application logs the battery voltage every 5 minutes. By using RPC, two nodes running the application are set up with different settings: One node sends log entries to the client on a regular basis. The other node operates with the Neighbour component deactivated. The application logs the voltage for two days. Afterwards, the logged measurments are read out and visualized on the client.

Because the used `VbatC` and `XE1205RadioC` components are specific to the Tinynode platform, the example is not compilable for other platforms, unless these components are exchanged.

The sample application consists of two files: the `ControllingExample.nc` file defines the configuration, the `ControllingExampleM.nc` file the necessary logic. The sources of the sample application can be found in the `src/tinyos/Controlling\ControllingExample` directory of the framework distribution. To compile the example, copy the contents of the `src/tinyos/` directory to the `apps/` directory of your TinyOS environment and run

```
make tinynode nescDecls rpc install,<nodeID>
```

in the `apps/Controlling/ControllingExample/` directory.

B.1 Logging

To store voltage measurments permanently, we need to set up the logger accordingly. This requires four steps:

- Providing constants used by the Logger component: the first page in flash memory that will be used for storing logger data and the maximal number of loggers installed on a node. Furthermore, if the LogSender component is used, the address of the gateway node connected to the client must be specified, so that the node knows where to send recent log entries. These constants are set up in the application's `Makefile`.
- Initializing the Logger component by wiring its provided `StdControl` interface.

- Giving the logger to be used an ID: this is done in the application's configuration file where the Logger is wired to the ControllingExampleM module with a parameterized interface. In the example application, the value 121 is used.

- Initialize the logger to be used with appropriate parameters. This is explained in the next subsection.

Initializing a Single Logger

Before any log entries can be appended to a logger, the logger itself has first to be initialized. This is performed by the `Logger.initLogger(uint8_t entryLength, uint16_t nPages, uint16_t nEntriesPerPage)` command. For the sample application, the three parameters were chosen as follows:

- `entryLength`: voltage measurements are of type `uint16_t`, therefore this parameter is set to `sizeof(uint16_t)`, which evaluates to 2.

- `nEntriesPerPage`: it is likely that the last started page before the node runs out of battery is not stored to flash memory, therefore `nEntriesPerPage` is set to 10. This way, not more than the last 9 log entries are lost.

- `nPages`: this parameter is set to 433, which allows reading voltage measurements for the last 15 days (if the batteries don't abandon earlier).

If no other loggers were initialized before, the flash memory occupied by the Logger component can be computed as follows:

$$size_{occupied} = size_{loggerIndex} + nPages_{logger121} \cdot (nEntriesPerPage_{logger121} \cdot entryLengh_{logger121} + size_{loggerPageTrailer})$$

$size_{loggerIndex}$ and $size_{loggerPageTrailer}$ can be derived from the `Logger.h` header file yielding the values 106 (by keeping in mind the 2 byte alignement of the Tinynode platform and assuming the `MAX_LOGGERS` constant set to 10) and 4, respectively. The above equation then evaluates to 10'498 bytes. These bytes are stored on the first 40 flash memory pages.

B.2 RPC

Additionally to the RPC commands declared by included components such as RemoteLogReader or Neighbour, the `ControllingExampleM` module provides four RPC commands:

- `result_t startLogging(uint32_t timeIntervalMs);`

- `result_t stopLogging();`

- `uint8_t getRFPower();`

- `result_t setRFPower(uint8_t value);`

The first command initializes a timer that is fired periodically with the given time interval, whereas the second command stops the timer. The latter two commands were added to control the sending power of the radio module.

These commands are declared at the beginning of the ControllingExample.nc file in the `module` block, each of them followed by the `@rpc()` tag. Note the `includes Rpc;` line at the beginning of the file!

```
includes Rpc;
module ControllingExampleM {
  provides {
    command result_t startLogging(uint32_t timeIntervalMs) @rpc();
    command result_t stopLogging() @rpc();
    command uint8_t getRFPower() @rpc();
    command result_t setRFPower(uint8_t value) @rpc();
  }
 ...
```

The commands are then implemented in the subsequent `implementation` block of the file.

After having installed the application on the nodes, imported the nescDecls.xml and rpcSchema.xml files into the client and looked up the running nodes, the RPC calls can be executed from the client.

- First, the sending power is increased on both nodes, so that the batteries will discharge faster. This is achieved by calling the `setRFPower()` command with the parameter 3, which indicates maximum power: a right click on the respective command (module `ControllingExampleM`) in the Eclipse outline view and selecting 'execute RPC command...' opens a dialog asking for the parameter. The parameter of size one byte has to be entered in hexadecimal format, so the value `03` has to be typed.
- Second, the LogSender is activated on one node by executing `init()` in the LogSender component with parameters `79` (decimal 121, the logger ID) and `0800`, indicating that after every 8^{th} entry, a log message with the last 8 entries will be sent to the client.
- Third, the Neighbour component is deactivated on the other node by calling the `stop()` command in the `Neighbours` module.
- Finally, the nodes can start measuring the voltage. This is initiated by calling the `startLogging()` command (in module `ControllingExampleM`) on both nodes with the parameter `E0930400`, which is 300'000 ms or 5 min in decimal.

B.3 Evaluating Log Data

This section describes how to download log data from nodes to the client, insert the data into a MySQL database[1] and visualize the data with the BIRT[2] (Business Intelligence and Reporting Tools) framework, which allows building reports for various output formats such as HTML or PDF.

[1]The MySQL database can be downloaded for free on http://www.mysql.org/
[2]The BIRT system is available on http://www.eclipse.org/birt/

Getting Log Entries

We can proceed by reading out the logged voltage measurments. The measurments are inserted into a previously set up MySQL database (the `src/scripts/` directory of the framework distribution contains a `dbsetup.sql` script that can be used to create a database as used in this example).

To actually get the entries, right click on the 'Logger 121' below the first node in the Eclipse outline view[3] and select 'Get Log Data'. In the opened dialog, select the 'Console Log Receiver' and the 'Database Log Receiver' (it is a good idea to select the console log receiver so that the retrieved entries can be seen immediately). For the 'number of entries' field, entering a high value (e.g. 4'500) will cause the node to send all available entries. The 'offset' field has to be specified, so 0 is entered here.

After having completed the dialog by clicking 'OK', another dialog (figure 3.14) for setting up the database is shown. The following `INSERT` statements are used:

```
INSERT INTO LOGGER_TRANSACTIONS VALUES (
  <transactionID>,
  <moteID>,
  <loggerID>,
  <entriesRequested>,
  <offset>,
  <dateStarted>,
  <timeStarted>,
  <timeStartedMs>,
  <callDuration>,
  nextval("ID")
)

INSERT INTO LOGGER_ENTRIES VALUES (
  actval("ID"),
  <pos>,
  <value>
)
```

The values are inserted into the database as soon as all entries were received successfully or a timeout as defined in the preferences[4] occurred. Check the console view for lost entries; they are indicated by `---LOST---`. Lost entries can be requested from the node by giving the respective offset; keep in mind to update the database accordingly (i.e. replacing logger entries with null values by the entries retrieved with a subsequent transaction).

The same procedure has yet to be done for the second node.

Visualizing Data

This subsection will not give a step by step guide to creating a report in BIRT. A brief tutorial that walks through building simple reports is available

[3] If no loggers appear below a node in the outline view, right click on the node and select 'Get Logger Info'.

[4] The timeout for log data can be set in the Eclipse preferences by choosing 'Window' → 'Preferences...' and then 'TinyOS Controlling' on the left side of the preferences dialog.

on http://www.eclipse.org/birt/phoenix/tutorial/basic/.

Interpreting Binary Data

The following SQL query was used to import the data into the report:

```
SELECT logger_transactions.moteID
       logger_entries.pos,
       logger_entries.value
FROM   logger_entries, logger_transactions
WHERE  logger_entries.ID = logger_transactions.ID
```

Log entries are stored in the database as `VARBINARY` or `VARLONGBINARY` and need to be converted so that they can be used for the report. In the case of the sample application, voltage measurments are logged as `uint16_t`, least significant byte first. A new 'Computed Column' is created with the following expression:

```
var voltage = row["value"][0] + row["value"][1]*256;
if (voltage > 2000 || voltage < 1000)
  null;
else
  voltage*(1.5/4096)/0.239;
```

The conversion is performed in the first line. In the if/else statement, any bursts are removed and the measurment is scaled appropriately.

The final result of the experiment is shown as a BIRT diagram in figure B.1. Battery power lowered faster on node 102, although it ran with the Neighbour component deactivated and without sending log entries regularly to the client.

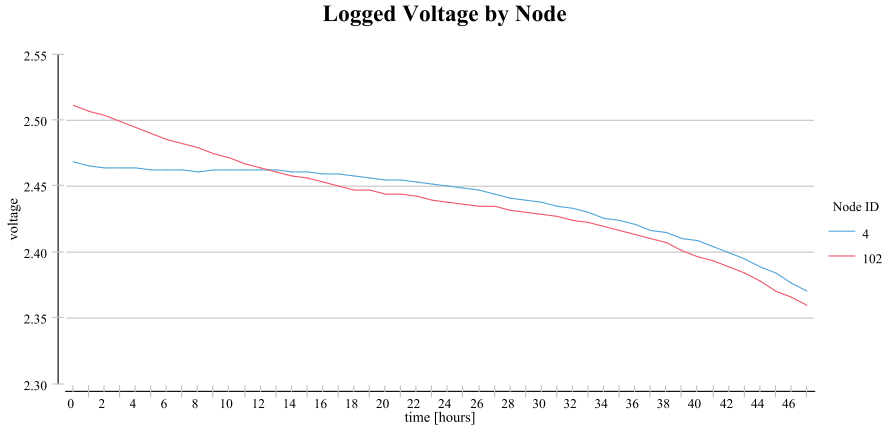

Figure B.1: Voltage measurments on nodes 4 and 102.

Appendix C

Extension Point Descriptions

C.1 Serial Forwarder Plug-In

C.1.1 Platform

Identifier: ch.ethz.dcg.controlling.sf.platform

Since: 1.0.0

Description: This extension point offers the possibility to extend the Serial Forwarder Plug-In with proprietary platforms running TinyOS. There are two possibilities to add platforms: by providing parameters in order to make use of the generic implementation that is included in this plug-in or by providing a proprietary implementation. The generic implementation offers the possibility to connect to the gateway node either over IP or over a serial connection.

Configuration Markup:

```
<!ELEMENT extension (platform+)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>
```

- *point* – a fully qualified identifier of the target extension point
- *id* – an optional identifier of the extension instance
- *name* – an optional name of the extension instance

```
<!ELEMENT platform (implementation , connectivity+)>
<!ATTLIST platform
  name        CDATA #IMPLIED
  description CDATA #IMPLIED
  image       CDATA #IMPLIED>
```

- *name* – the name of the platform
- *description* – a short description of the platform - will be displayed in the platform selection wizard
- *image* – the location of an image (preferably in GIF format) relative to the extending plug-in's/fragment's root directory. The image should represent the platform - it will be shown in the platform selection wizard.

```
<!ELEMENT connectivity (serial | networkIP |
  customConnectivity)>

<!ELEMENT customConnectivity EMPTY>
<!ATTLIST customConnectivity
  description CDATA #REQUIRED>
```

- *description* – a string describing the format of the parameters the user has to enter for using customConnectivity

```
<!ELEMENT networkIP EMPTY>
<!ATTLIST networkIP
  portnumber CDATA #IMPLIED>
```

- *portnumber* – if the portnumber of the IP connected gateway is always the same, specify it here

```
<!ELEMENT serial EMPTY>
<!ATTLIST serial
  baudrate (110|300|1200|2400|4800|9600|19200|38400|57600|
    115200|230400|460800|921600) >
```

- *baudrate* – the baudrate to be used when communicating over the serial port

```
<!ELEMENT implementation (generic | custom)>

<!ELEMENT generic EMPTY>
<!ATTLIST generic
  TOSMsgClass CDATA #REQUIRED
  platformID  CDATA #REQUIRED>
```

The generic element indicates that the generic implementation from the Serial Forwarder Plug-In will be used.

- *TOSMsgClass* – a fully qualified name of the class that extends net.tinyos.message.TOSMsg. This class is needed to find out offsets in order to communicate with the gateway node.
- *platformID* – the platform ID as found in the classic serial forwarder. This ID is needed to communicate with the gateway node.

```
<!ELEMENT custom EMPTY>
<!ATTLIST custom
  PlatformClass CDATA #REQUIRED>
```

The custom element indicates that the class given in the attribute PlatformClass will be used instead of the generic implementation.

- *PlatformClass* – PlatformClass - a fully qualified name of the class that implements ch.ethz.dcg.conrolling.moteIF.IPlatform. This class has to be implemented if you decide to provide custom connectivity.

Examples: The following is an example extension for this extension point:

```
<extension
      point="ch.ethz.dcg.controlling.sf.platform">
   <platform
         description="The Tinynode Platform by Shockfish"
         image="icons/tinynode.gif"
         name="Tinynode">
      <connectivity>
         <serial baudrate="57600"/>
      </connectivity>
      <implementation>
         <generic
               TOSMsgClass="net.tinyos.message.tinynode.
                  TOSMsg"
               platformID="5"/>
      </implementation>
   </platform>
   <platform
         description="The Mica 2 Platform"
         image="icons/mica2.gif"
         name="Mica 2">
      <connectivity>
         <networkIP portnumber="10002"/>
      </connectivity>
      <connectivity>
         <serial baudrate="57600"/>
      </connectivity>
      <implementation>
         <generic
               TOSMsgClass="net.tinyos.message.avrmote.
                  TOSMsg"
               platformID="1"/>
      </implementation>
   </platform>
</extension>
```

API Information: If a custom platform class is given, it has to implement the ch.ethz.dcg.conrolling.moteIF.IPlatform interface. The interface provides information on how the interface will be used.

Supplied Implementation: The ch.ethz.dcg.controlling.platforms fragment provides support for some existing platforms.

C.1.2 Raw Message Listener

Identifier: ch.ethz.dcg.controlling.sf.RawMessageListener

Since: 1.0.0

Description: This extension point is used to add a listener for raw messages sent to or from the wireless sensor network by the Serial Forwarder Plug-In.

Configuration Markup:

```
<!ELEMENT extension (RawMessageListener+)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>
```

- *point* – a fully qualified identifier of the target extension point
- *id* – an optional identifier of the extension instance
- *name* – an optional name of the extension instance

```
<!ELEMENT RawMessageListener EMPTY>
<!ATTLIST RawMessageListener
  listenerClass CDATA #REQUIRED
  packetKind    (inPacket|outPacket|badPacket) >
```

- *listenerClass* – a fully qualified name of the class that implements the ch.ethz.dcg.controlling.messaging.IPacketListener interface. Upon the first processing of a message, the class will be instantiated. For every message to be processed, the receivePacket method will be called with the message data as a parameter.
- *packetKind* – one of inPacket, outPacket or badPacket, depending on the kind of packets the listener wants to receive

Examples: The following is an example extension for this extension point:

```
<extension
      id="SerialForwarder"
      name="Serial Forwarder listener"
      point="ch.ethz.dcg.controlling.sf.RawMessageListener">
   <RawMessageListener
         listenerClass="ch.ethz.dcg.controlling.utils.
            SerialForwarder$PacketListener"
         packetKind="inPacket">
   </RawMessageListener>
   <RawMessageListener
         listenerClass="ch.ethz.dcg.controlling.sf.views.
```

```
            MessageReceiverImpl$InPacketListener"
          packetKind="inPacket">
    </RawMessageListener>
    <RawMessageListener
          listenerClass="ch.ethz.dcg.controlling.sf.views.
            MessageReceiverImpl$OutPacketListener"
          packetKind="outPacket">
    </RawMessageListener>
    <RawMessageListener
          listenerClass="ch.ethz.dcg.controlling.sf.views.
            MessageReceiverImpl$BadPacketListener"
          packetKind="badPacket">
    </RawMessageListener>
    <RawMessageListener
          listenerClass="ch.ethz.dcg.controlling.messaging.
            TOSMessageDispatcher$PacketListener"
          packetKind="inPacket"/>
</extension>
```

API Information: It is common practice to declare the class implementing the ch.ethz.dcg.controlling.messaging.IPacketListener interface as an inner class which calls a method of a singleton instance of the outer class in order to process a packet.

Supplied Implementation: The ch.ethz.dcg.controlling.sf.RawMessageListener extension inside this plug-in extends this extension point.

C.1.3 TOS Message Listener

Identifier: ch.ethz.dcg.controlling.sf.TOSMessageListener

Since: 1.0.0

Description: This extension point is used to add a listener for TOS messages received from the wireless sensor network by the Serial Forwarder Plug-In.

Configuration Markup:

```
<!ELEMENT extension (TOSMessageListener+)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>
```

- *point* – a fully qualified identifier of the target extension point
- *id* – an optional identifier of the extension instance
- *name* – an optional name of the extension instance

```
<!ELEMENT TOSMessageListener (messageType)>
<!ATTLIST TOSMessageListener
  listenerClass CDATA #REQUIRED>
```

- *listenerClass* – a fully qualified name of the class that implements the ch.ethz.dcg.controlling.messaging.IPacketListener interface

```
<!ELEMENT messageType EMPTY>
<!ATTLIST messageType
  AMtypename CDATA #IMPLIED
  AMtype     CDATA #REQUIRED>
```

- *AMtypename* – the name of the active message type the listener will be registered for
- *AMtype* – an integer indicating the active message type of the TOS messages the listener wants to be notified upon receipt. Insert a * if the listener has to be notified of all incomming TOS messages.

Examples: The following is an example extension for this extension point:

```
<extension
      point="ch.ethz.dcg.controlling.sf.TOSMessageListener">
   <TOSMessageListener
         listenerClass="ch.ethz.dcg.controlling.rpc.
            RPCTransport$PacketListener">
      <messageType
            AMtypename="AM_RPCRESPONSEMSG"
            AMtype="212"/>
   </TOSMessageListener>
</extension>
```

API Information: It is common practice to declare the class implementing the ch.ethz.dcg.controlling.messaging.IPacketListener interface as an inner class which calls a method of a singleton instance of the outer class in order to process a packet.

The dataOffset parameter provided when the receivePacket method of the ch.ethz.dcg.controlling.messaging.IPacketListener interface is called will point to the first byte of the data field of the TOS_Msg struct inside the received data byte array.

Supplied Implementation: The Sensornet Manager Plug-In (ch.ethz.dcg.controlling.snetMgr) makes use of this extension point in order to receive different types of messages such as RPC responses or Drain messages.

C.2 Sensornet Manager Plug-In

C.2.1 Drain Message Listener

Identifier: ch.ethz.dcg.controlling.snetMgr.DrainMessageListener

Since: 1.0.0

Description: This extension point is used to add a listener for messages comming in over the Drain routing layer. Its usage is identical to the ch.ethz.dcg.controlling.sf.TOSMessageListener extension point.

Configuration Markup:

```
<!ELEMENT extension (DrainMessageListener+)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>
```

- *point* – a fully qualified identifier of the target extension point
- *id* – an optional identifier of the extension instance
- *name* – an optional name of the extension instance

```
<!ELEMENT DrainMessageListener (messageType)>
<!ATTLIST DrainMessageListener
  listenerClass CDATA #REQUIRED>
```

- *listenerClass* – a fully qualified name of the class that implements the ch.ethz.dcg.controlling.messaging.IPacketListener interface

```
<!ELEMENT messageType EMPTY>
<!ATTLIST messageType
  AMtypename CDATA #IMPLIED
  AMtype     CDATA #REQUIRED>
```

- *AMtypename* – the name of the active message type the listener will be registered for
- *AMtype* – an integer indicating the active message type of the Drain messages the listener wants to be notified upon receipt. Insert a * if the listener has to be notified of all incomming Drain messages.

Examples: The following is an example extension for this extension point:

```
<extension
      point="ch.ethz.dcg.controlling.snetMgr.
            DrainMessageListener">
   <DrainMessageListener listenerClass="ch.ethz.dcg.
         controlling.rpc.RPCTransport$PacketListener">
      <messageType
            AMtypename="AM_RPCRESPONSEMSG"
            AMtype="212"/>
   </DrainMessageListener>
</extension>
```

API Information: It is common practice to declare the class implementing the ch.ethz.dcg.controlling.messaging.IPacketListener interface as an inner class which calls a method of a singleton instance of the outer class in order to process a packet.

The dataOffset parameter provided when the receivePacket method of the ch.ethz.dcg.controlling.messaging.IPacketListener interface is called will point to the first byte of the data field of the DrainMsg struct inside the received data byte array.

Supplied Implementation: The Sensornet Manager Plug-In (ch.ethz.dcg.controlling.snetMgr) makes use of this extension point in order to receive different types of messages such as RPC responses or log messages comming in over the Drain layer.

C.2.2 Bulk Data Message Listener

Identifier: ch.ethz.dcg.controlling.snetMgr.BulkDataMsgListener

Since: 1.0.0

Description: This extension point is used to add a listener for bulk data comming in. Bulk data is data that potentially doesn't fit into one single TOS message and needs to be split up into several messages by the node and put back together when received.

Configuration Markup:

```
<!ELEMENT extension (BulkDataMessageListener+)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>
```

- *point* – a fully qualified identifier of the target extension point
- *id* – an optional identifier of the extension instance
- *name* – an optional name of the extension instance

```
<!ELEMENT BulkDataMessageListener (messageType)>
<!ATTLIST BulkDataMessageListener
  listenerClass CDATA #REQUIRED>
```

- *listenerClass* – a fully qualified name of the class that implements the ch.ethz.controlling.bulkData.IBulkDataListener interface

```
<!ELEMENT messageType EMPTY>
<!ATTLIST messageType
  dataTypename CDATA #IMPLIED
  dataType     CDATA #REQUIRED>
```

- *dataTypename* – the name of the bulk data type the listener will be registered for
- *dataType* – an integer indicating the bulk data type of the bulk data messages the listener wants to be notified upon receipt. Insert a * if the listener has to be notified of all incomming bulk data.

Examples: The following is an example extension for this extension point:

```
<extension
      point="ch.ethz.dcg.controlling.snetMgr.
            BulkDataMsgListener">
   <BulkDataMessageListener
         listenerClass="ch.ethz.dcg.controlling.snetMgr.
               model.Sensornet$NeighbourhoodDataListener">
      <messageType
            dataType="25"
            dataTypename="BULK_NEIGHBOURTABLE"/>
   </BulkDataMessageListener>
</extension>
```

API Information: It is common practice to declare the class implementing the ch.ethz.controlling.bulkData.IBulkDataListener interface as an inner class which calls a method of a singleton instance of the outer class in order to process a packet.

Supplied Implementation: The Sensornet Manager Plug-In (ch.ethz.dcg.controlling.snetMgr) makes use of this extension point in order to receive data about a node's neighbourhood, its filter or its installed loggers.

C.2.3 Log Receiver

Identifier: ch.ethz.dcg.controlling.snetMgr.LogReceiver

Since: 1.0.0

Description: This extension point is used to add listeners for log entries received from nodes.

Configuration Markup:

```
<!ELEMENT extension (logReceiver+)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>
```

- *point* – a fully qualified identifier of the target extension point
- *id* – an optional identifier of the extension instance
- *name* – an optional name of the extension instance

```
<!ELEMENT logReceiver EMPTY>
<!ATTLIST logReceiver
  receiverClass       CDATA #REQUIRED
  receiverName        CDATA #REQUIRED
  receiverDescription CDATA #IMPLIED>
```

- *receiverClass* – a fully qualified name of the class that implements the ch.ethz.controlling.logging.ILogReceiver interface

- *receiverName* – the name of the logReceiver. Is used in the UI for the user to identify the logReceiver.
- *receiverDescription* – a short description of the logReceiver

Examples: The following is an example extension for this extension point:

```
<extension
      point="ch.ethz.dcg.controlling.snetMgr.LogReceiver">
   <logReceiver
         receiverClass="ch.ethz.dcg.controlling.logging.
               FileLogReceiver"
         receiverDescription=
               "Prints the received entries to a file"
         receiverName="File Log Receiver"/>
</extension>
```

API Information: The ch.ethz.controlling.logging.ILogReceiver interface contains comments on how the interface is used by the framework.

Supplied Implementation: The sensornet manager plug-in (ch.ethz.dcg.controlling.snetMgr) extends this extension point by supplying log readers that print received logger entries to System.out or to a file.

The database logReceiver plug-in (ch.ethz.dcg.controlling.dbLogReceiver) extends this extenstion point by supplying a log reader that inserts received logger entries into a specified database.

C.3 Database Log Receiver Plug-In

C.3.1 Database Driver

Identifier: ch.ethz.dcg.controlling.dbLogReceiver.dbDriver

Since: 1.0.0

Description: This extension point is used to extend the database log receiver plug-in with proprietary JDBC drivers.

Configuration Markup:

```
<!ELEMENT extension (databaseDriver+)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>
```

- *point* – a fully qualified identifier of the target extension point
- *id* – an optional identifier of the extension instance
- *name* – an optional name of the extension instance

```
<!ELEMENT databaseDriver (library+)>
<!ATTLIST databaseDriver
  driverClass         CDATA #REQUIRED
  name                CDATA #REQUIRED
  sampleConnectionURL CDATA #IMPLIED>
```

- *driverClass* – a fully qualified name of the class that implements the java.sql.Driver interface
- *name* – the name of the driver. Will be shown in the UI.
- *sampleConnectionURL* – a sample URL as a help for the user on how to enter the connect URL

```
<!ELEMENT library EMPTY>
<!ATTLIST library
  jarFile CDATA #REQUIRED>
```

- *jarFile* – the location of the jar file needed in order to load the driver relative to the extending plug-in's/fragment's root

Examples: The following is an example extension for this extension point:

```
<extension
      point="ch.ethz.dcg.controlling.dbLogReceiver.
            dbDriver">
   <databaseDriver
         driverClass="com.mysql.jdbc.Driver"
         name="MySQL Connector/J Version 3.1.13"
         sampleConnectionURL="jdbc:mysql://&lt;server&gt;
               :&lt;port&gt;/&lt;dbName&gt;">
      <library jarFile=
            "lib/mysql-connector-java-3.1.13-bin.jar"/>
   </databaseDriver>
</extension>
```

Supplied Implementation: The Database Log Receiver Plug-In (ch.ethz.dcg.controlling.dbLogReceiver) extends this extension point by supplying a JDBC driver for MySQL databases.

Bibliography

[1] David Gay, Philip Levis, David Culler, Eric Brewer: *nesC 1.1 Language Reference Manual*. Available at: http://nescc.sourceforge.net/papers/nesc-ref.pdf (last accessed on September 13, 2006)

[2] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer and D. Culler: *The nesC language: A Holistic Approach to Networked Embedded Systems*. Available at: http://nescc.sourceforge.net/papers/nesc-pldi-2003.pdf (last accessed on September 13, 2006)

[3] Erich Gamma, Kent Beck: *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addison-Wesley, Boston, 2003, ISBN: 0-321-20575-8

[4] Gilman Tolle, David Culler: *Design of an Application-Cooperative Management System for Wireless Sensor Networks*. Second European Workshop on Wireless Sensor Networks (EWSN), Istanbul, Turkey, January 31 - February 2 2005.

[5] Gilman Tolle: *Nucleus Network Management*. Available at: http://www.cs.berkeley.edu/~get/nucleus/nucleus-manual.pdf (last accessed on August 29, 2006)

[6] Crossbow Technology, Inc.: *MOTE-VIEW 1.2 Users Manual; Revision B, January 2006*. Available at: http://www.xbow.com/Support/Support_pdf_files/MoteView_Users_Manual.pdf (last accessed on April 25, 2006)

[7] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler: *Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks*. IPSN'06, April 19 – 21, 2006, Nashville, Tennessee, USA.

[8] Severin Winkler: *Kontrolle und Visualisierung eines Sensornetzwerks*. Semester thesis at ETH Zurich, Distributed Computing Group.