

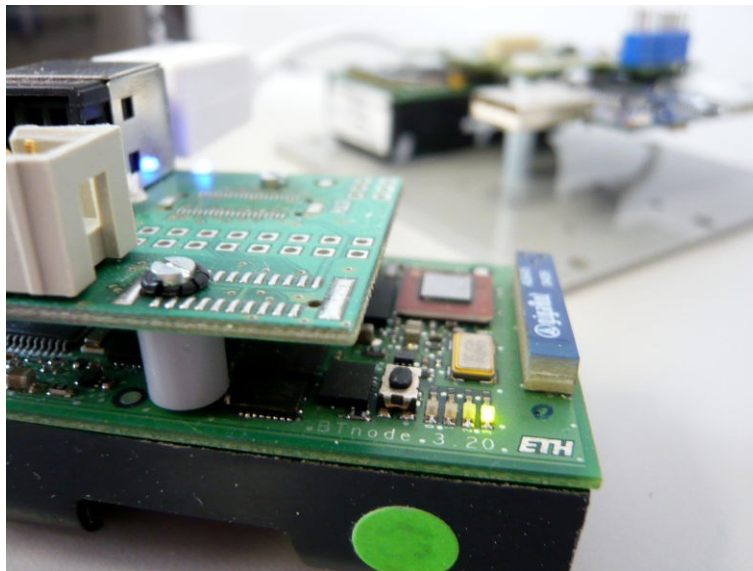


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester Thesis

Automating Worldwide Sensor Network Testing

David Frey; freyd@ee.ethz.ch



Spring 2008

Supervisor: Dr. Jan Beutel, j.beutel@ieee.org

Professor: Dr. Lothar Thiele, thiele@tik.ee.ethz.ch

Content

1 Introduction.....	1
1.1 Related Work.....	1
1.1.1 Wireless Sensor Network Testbeds.....	1
1.1.2 WSN Testbed Clients.....	7
1.2 Problem Setting.....	7
1.2.1 General Problem – What is missing.....	7
1.2.2 Specific Problem Statement – What is to be done.....	7
2 The Interface.....	8
2.1 Approach.....	8
2.1.1 Name.....	9
2.1.2 Notion of a Test Job.....	9
2.2 Operations.....	9
2.2.1 Get Testbed Description.....	9
2.2.2 Get Node List.....	11
2.2.3 Get Node Description.....	11
2.2.4 Submit Job.....	12
2.2.5 Get Job Status.....	13
2.2.6 Get Job Output.....	14
2.3 Error Handling.....	14
2.4 Authentication.....	14
2.5 Implementation.....	15
3 The Client.....	16
3.1 Requirements.....	16
3.2 Implementation.....	17
3.2.1 Configuration.....	17
3.2.2 Invocation.....	18
3.2.3 Output.....	19
3.3 Limitations.....	19
3.3.1 Special features.....	19
3.3.2 Multiple architectures.....	19
4 Integration in TinyOS Build System.....	20
4.1 Limitations.....	20
5 Summary.....	21
5.1 Future Work.....	21
6 References.....	22
7 Figures.....	23

1 Introduction

Over the past few years, the area of sensor networks has seen great interest in research and development. However, developing error free applications for sensor networks has proven to be very hard [1]. One important piece in a successful development process is repeated and thorough testing of the application and its building parts.

The problem of testing applications for sensor networks once again is not easy, because it includes tasks on many different abstraction levels: programming of the application on several nodes, executing the program in a realistic environment, gathering data during the test run and analysing it afterwards. To levitate this tasks, sensor network testbeds were developed by different universities, which automate most or all of these tasks.

These different implementations differ in many aspects: In their architecture, the hardware used, in how much they automate and how they are configured. No known testbed actually automates the complete testing process from programming over running to log acquisition from inside the development environment. For the developer this signifies a substantial effort to set up each test run on the same testbed, let alone the running the program on a different testbed implementation.

The aim of this work was to formulate a generic interface for sensor network testbeds which is compatible with any known implementation and to implement it in a prototype. On the server side the implementation was to be done for the JAWS Deployment Support Network developed here at ETH Zürich and the client was to be a standalone program, able to connect to any testbed server satisfying the presented interface.

1.1 *Related Work*

As mentioned before, the problem of testing applications for wireless sensor networks has been addressed by several research groups, who developed hard- and software to support and automate the process.

1.1.1 **Wireless Sensor Network Testbeds**

Wireless sensor network testbeds consist for one of a hardware architecture, connecting all nodes and providing a means to at least program all of them. Then they usually include a set of software tools to facilitate and automate recurrent tasks. As seen before, implementations differ widely in their approach and the extent they automate tasks. We will present some common desired properties of such a testbed and list a few examples.

1.1.1.1 *Properties*

A sensor network testbed automates the task of programming the target nodes. This is the most important task, because in order to get useful results, test runs need to be conducted on a sufficiently large number of nodes. Programming all these by hand is by no means efficient and would cause developers to do as few test runs as possible.

A testbed provides an additional channel to the nodes (apart from the existing wireless channel). This allows for introspection into the wireless network, without sending log messages over the wireless link and thereby affecting the system. Ideally, this channel allows for logging messages with synchronous timestamping, making it easy to analyse the chronological order of events.

A testbed should keep track of the nodes connected to it. As the small sensor nodes are prone to failure, the system needs to have an overview of the currently usable and not usable devices.

To allow for realistic testing scenarios, a testbed infrastructure should support dynamic configuration changes. By switching off or powering on nodes during the test run, failure and insertion of nodes can be simulated.

1.1.1.2 Features

As a starting point in this thesis, a small survey about currently existing wireless sensor network testbeds was conducted. We are going to present the different features treated in the survey followed by the results in tabular form. Three specific implementations will be described in somewhat more detail. Documentations of these testbeds is usually spread over several publications or webpages because most of these systems are in constant development. So if a feature is missing in the table, this does not mean that it is not currently implemented, but only that we found no document mentioning it.

Web Interface

A feature practically all testbeds implement is a web interface. It normally provides a status page listing the currently connected nodes and information about software versions, last successful communication and so on. This information is often visualized on a map, helping developers planning their program tests. For some implementations, this interface also offers the ability to interact with the testbed, i.e. programming nodes, switching them on and off or even scheduling a test job.

The main advantage of this kind of interface is clearly the ease of use even for people not acquainted with the application. The initial barrier to use a web interface is very low, given how widely used the web is today. However, it is usually not possible to automate tasks carried out over such an interface. This is especially painful for a task like testing, which should be done repeatedly after every change to the software. Going through a lengthy point and click procedure for every iteration will keep many developers away from testing their software regularly.

RPC Interface

While a web interface is aimed at humans as users, an RPC interface is meant to be used by applications. Having an interface suitable for programming allows the implementation of testbed clients which run locally on the developers machine and automate the task of testing an application. Until now only very few testbeds actually implement such an interface, and for these that do it is very technology specific, which hinders development of clients compatible with different testbed implementations.

Logging

A crucial part of testing and debugging applications for sensor networks is the generation of log files. A testbed can automate this task by providing the nodes with a channel on which they can send logging messages and by storing them in a database. This is mostly done on a central server, but depending on the testbed architecture, log collection can be done in a distributed fashion by devices further down in the hierarchy, which in turn report to a central server. (See 1.1.1.3)

Job Scheduling

While a testbed usually offers the automation of tasks such as reprogramming nodes, switching them on and off and collecting their log output, a more higher level service they can offer is the scheduling of complete test jobs. This includes two parts: On one hand side, the testbed server has to automate the succession of programming nodes, switching them on, collecting log output and terminating the test run. Thereby it eliminates the necessity of user interaction during such a test cycle. On the other hand side, there needs to be a way to avert collisions in testbed use, for example by allocating it for a certain user for a certain period, possibly in advance.

Scheduled Commands

In the notion of a test job, it can be possible to schedule commands to be sent to individual nodes while a test is running. This could be used to inject data or turn nodes on or off, thereby making a testing scenario more realistic.

Power Control

The ability to switch the power of certain nodes on or off during test execution adds the possibility to simulate failure and emergence of single nodes. This is an important scenario, as in real life conditions the small embedded sensor devices are prone to failure.

Power Logging

As wireless sensor networks are usually battery powered and low maintenance, they need to survive a long time with minimal energy consumption. To make sure that applications comply with these restrictions, some testbeds include the possibility to measure the power consumption of certain nodes (normally not all). This gives the developer another source of feedback, aside from the normal log messages.

Serial Port Forwarding

Many testbeds allow the developer to connect to the serial ports built into the sensor nodes over the network using a serial forwarder. This is an almost real time channel which can be used to gather information from the devices or send commands to them. For some testbeds however this is the only way to acquire log messages, meaning the user himself has to connect to all nodes he is interested in, a task which could be automated by the testbed. Connecting simultaneously to dozens of nodes over serial network links is a tedious task that is left to the developer to automate.

Image Format

In the TinyOS world, two formats for the image files that are to be loaded on the sensor nodes are prevalent: A binary ELF format (usually having an .exe extension) and a the Intel HEX format (usually with the extension .ihex). They are equivalent descriptions of the programs, but depending on the hard- and software used in a testbed, different implementations use one or the other.

1.1.1.3 Examples

JAWS - Deployment-Support Network

The Deployment Support Network was developed at ETH Zürich and relies on the BTnode platform [2]. This device, also developed at ETH, is equipped with a Bluetooth radio which is used by the testbed as the backbone channel. Each sensor (target) node that is part of the testbed network is connected to a dedicated BTnode device. One 'GUI' BTnode is wired to a server PC, on which the Java DSN server is running. Therefore the testbed is a distributed system, just as the applications being tested on it. The BTnodes are also able to switch the target nodes' power supply, making it possible to enable and disable nodes remotely [3].

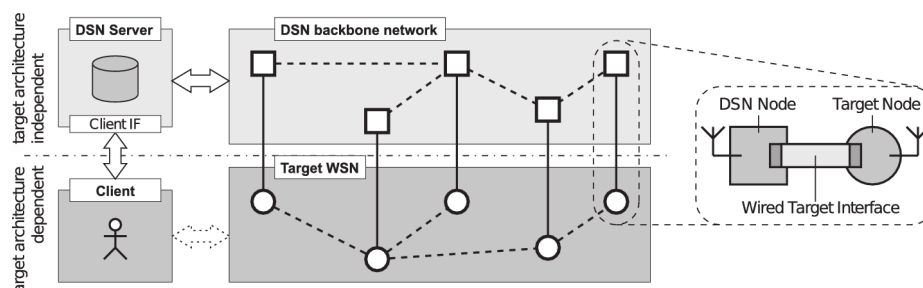


Figure 1: Architecture of the DSN testbed.

The Bluetooth channel is used to distribute image files in the testbed network, to send commands to single nodes and to gather log messages. Due to the nature of a wireless channel it is slower and less reliable than the wired equivalent, but it proved perfectly usable for this kind of usage.

The server provides a low level XML-RPC interface, which has commands such as 'distribute an image file in the network', 'flash a single node', 'turn the power of a node off' and so on and is hardly documented. There exists a web interface which translates this programming level interface to a graphical interface.

All log messages sent by the nodes are timestamped and saved in a database. Also the server keeps track of the currently connected nodes. Users can either directly query the database to search for log messages or use the RPC interface to fetch them automatically.

MoteLab

Developed at Harvard University, MoteLab is "a set of software tools for managing a testbed of Ethernet-connected sensor network nodes." [4] The sensor nodes are each connected to an Ethernet interface board, which provides a TCP forwarder for the serial port. A server then controls them over the network. It runs several separate pieces of software:

- A web interface lets users assess the status of the network and schedule jobs. Each user has a certain time quota for pending test jobs. (For the concept of a test job see 2.1.2.) All scheduled jobs are stored in a database.
- A job daemon is responsible for fetching jobs from the database when they are due, setting up the testbed hardware according to the job description and starting logging. After the test run has completed, cleans up and frees the resources for the next job.

- A Java logging program parses the log messages sent by the nodes over the serial links and stores them into the database.

A user acquires her log data by directly querying the database through a web interface. She also has the possibility to directly connect to the serial port of any node during her test job.

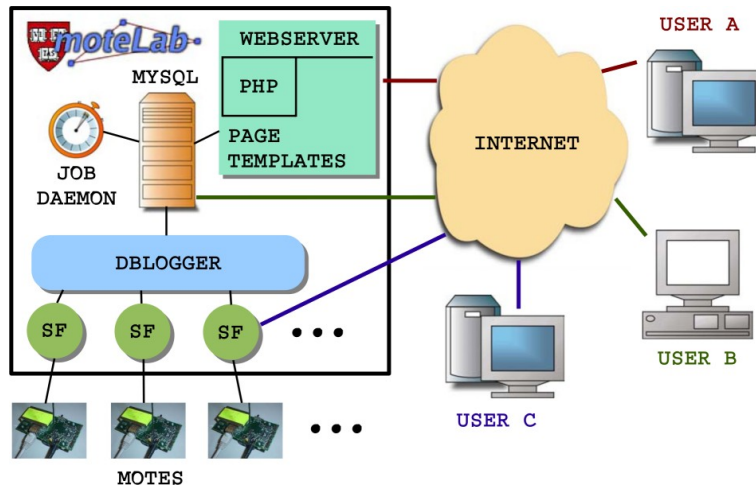


Figure 2: Architecture of the MoteLab testbed.

TWIST

TWIST, a testbed developed at Technische Universität Berlin, features yet another architecture. Between the testbed server and the sensor nodes, it introduces a layer of so called 'super nodes', which are embedded 32-bit devices running a part of the testbed logic. These super nodes are connected to the server over an Ethernet backbone and feature one or more USB ports. The nodes are connected to these super nodes by means of USB hubs, which by the USB Hub Specification 2.0 support power switching [5].

Commands and application images are sent by the server through the Ethernet backbone to the super nodes, which then, in parallel, forward them to the nodes. Thanks to the power switching abilities of the USB hubs, single nodes can be remotely turned on and off.

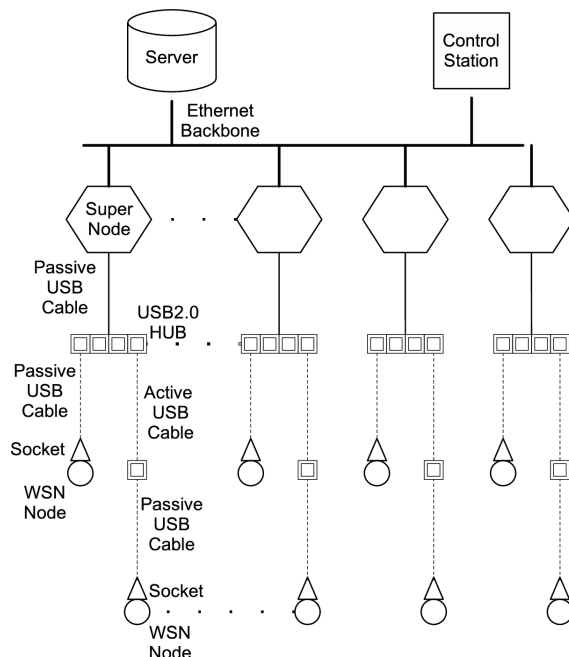


Figure 3: Architecture of the TWIST testbed.

Name	Institution	Scheduled commands	Serial port forwarding	Power control	Job Scheduling	Logging	RPC interface	Web interface	Power logging	Backbone	Image format
DSN [6]	ETH Zürich	x		x		local/central	x	x	x	Bluetooth	hex
MoteLab [4]	Harvard		x		x	central		x	x	Ethernet	binary
TWIST [5]	TU Berlin		x	x		central		x		USB / Ethernet	?
Tutornet [7]	USC		x			?		?		USB / WLAN	?
Mirage [8] [9]	Berkeley / Intel				x		(x)	x		?	?
Motescope [10]	Berkeley							x	x	Ethernet	?
Kansey [11]	Ohio State							x		Ethernet	?

Table 1: Results of the survey on wireless sensor network testbeds.

1.1.2 WSN Testbed Clients

1.1.2.1 DSNTargetLogger

The DSNTargetLogger is a Java program developed by Mustafa Yücel at ETH Zürich for the XML-RPC interface of the DSN server. It automates the task of distributing one image file in the testbed network and flashing it on a customizable number of nodes. It can then turn on the nodes for the duration of a test run and collect log messages while the application is running. It even features scheduled execution of DSN commands and switching nodes' power supply.

In short, this program can execute a test job according to a job description and collect the log output completely unattended. Opposed to the MoteLab implementation of test jobs, here the control logic is in the client, which gives the server low level commands of what to do and queries it about the success of operations, while MoteLab features a job daemon as a part of the server for this task.

DSNTargetLogger is a good attempt to completely automate running application tests on wireless sensor network testbeds. While it does the automation part well, it is very inflexible in that it only works with the application specific interface of the DSN testbed.

1.2 Problem Setting

1.2.1 General Problem – What is missing

As our little survey has shown, sensor network testbeds automate the task of testing applications for wireless sensor networks in various aspects and to various degrees. For a testing facility to be useful and used by developers, it is important however that the complete testing process can be automated from the command line. A client is needed that can with one invocation unattendedly execute a complete test run and return the results generated by the testbed.

Also to allow interoperability between testbeds and clients of different implementations, they need to support a common interface. Such an interface would need to be very generic, so that each existing implementation can fulfil it but yet flexible enough to support realistic test scenarios.

1.2.2 Specific Problem Statement – What is to be done

The Problem statement for this semester thesis consists of two main parts:

As a first part, a generic interface for wireless sensor network testbeds is to be developed. It has to be as generic that it can be implemented by any existing testbed, but extendible to support features not present in all implementations.

The second part consists of implementing this interface for the DSN testbed server. This will include adding a new interface and implementing the necessary control logic presently not existent in the server. Also a client is to be implemented that connects to any testbed implementing the new interface. While it should work on any testbed, it will be tested against the DSN server.

2 The Interface

2.1 Approach

One of the main tasks of this semester thesis was to develop a generic interface for communication with a sensor network testbed. This was done by analysing the traditional way of programming such a testbed.

1. Almost independently of the implementation, a first step would consist of getting to know the testbed. One would usually browse the website or a similar source of information about the testbed to find out of which nodes it consists and where they are located, to get an idea of what scenarios could be run.
2. After developing the testing scenario (set of nodes, on which the application should be run, time to run it, ect.), one would go about executing it on the testbed in question. The Motelab implementation actually allows to enter these parameters directly, to be read and executed automatically by a demon controlling the testbed. On DSN, you would either use DSNTargetLogger or the web interface, both of which send the testbed server a sequence of low level commands, causing it to upload the image files, distributing them in the network and flashing target nodes.
3. While the test scenario is running, the user might appreciate some kind of feedback about what is happening. The different implementations offer a number of different sources of information: from the live log output of the server, over a simple list of nodes which is currently in use, to forwarded data from the serial port of each node. In general, it would be reassuring to at least know if the test job is actually running, or information about possible errors.
4. After execution, the user expects to get back data for her to analyse. Be it logging information produced by the nodes or the result of power consumption measurement. Most testbeds store logging output in a database which the user can query, others rely on the user to connect to the nodes serial port and collect data himself.

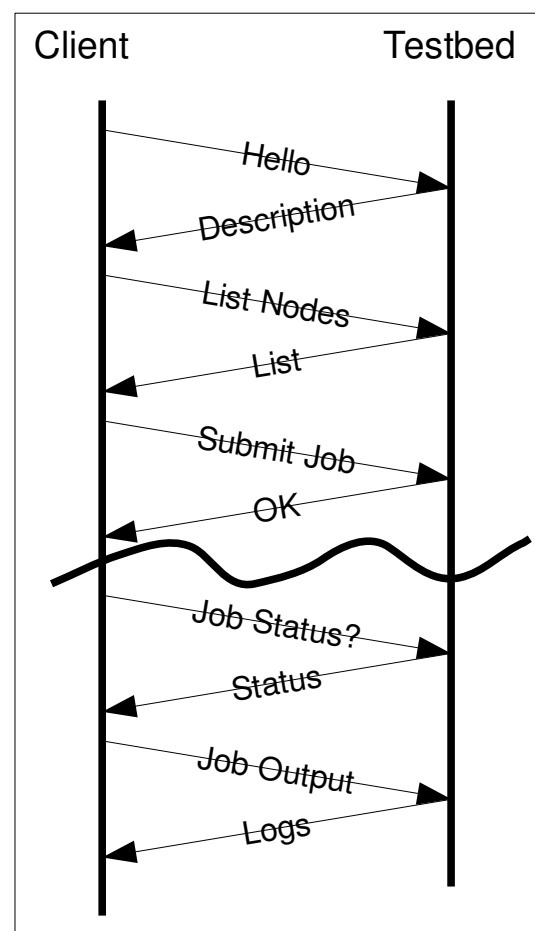


Figure 4: Simplified communication between client and testbed

For each basic step in this communication, one or maybe two standardised operations were defined. The resulting interface is very compact but still offers great flexibility to configure a testbed to ones needs.

2.1.1 Name

Just as every component the interface needed a name, even if it was only to name the Java classes and source files. Because a describing name would have gotten overly long and an acronym didn't seem appealing, I decided to use a fictional name. While brainstorming the different possibilities my eyes wandered over the desktop and found my trusty PET water bottle. Originally a container of apple juice, it carries the label RAMSEIER (a Swiss apple juice brand). So the interface came to its name *Rams*.

2.1.2 Notion of a Test Job

While searching for a generic view on the communication with a sensor network testbed, we deemed it appropriate to define the concept of a *test job*. This notion aggregates all information that is needed to program a testbed for one complete test run. Firstly, it contains the timing information, that is when the job will start and for how long it will run. Secondly it also contains the program images to be flashed onto the nodes. To allow arbitrary architectures in the testing scenarios, a test job can contain a free number of such images, each of which has a list of nodes associated, on which it is to be programmed.

Using this concept, one could for example easily construct a description of a test run for a harvester program with one application programmed to normal sensor nodes which gather and forward data to a special sink node running a different program, processing the data.

2.2 Operations

We go about listing the different operations of the Rams Interface. For each of these, a textual description is given, along with input and outputs, if applicable, and the possible error conditions that could occur while executing. Special data structures that were defined in the scope of the interface are also described under the method they are used in.

The data types of the different fields in these data structures are deliberately chosen to be very generic, they are all either strings, integers, timestamps or lists thereof. This imposes the least possible requirements to a technology the interface is to be implemented in. Also many data structures carry meta information like name and description. While these are not mandatory, they often make your life easier when managing a larger collection of such items, be it on the client side in a developers repository of testing scenarios or on the server side, where data is stored in a database.

2.2.1 Get Testbed Description

This operation is usually called first in any communication with a testbed. It provides general information about the testbed, its location, features and status, which could help troubleshooting possible problems that turn up later. It is also a good way to test if authentication works, where applicable.

Description	Retrieves a description of a testbed, providing general information about location, features and status.	
Input	None	
Output	Testbed Description	Custom data structure, see below.
Error conditions	None	

Testbed Description Data Structure

Field Name	Data type	Description
Description	String	A very general description of the specific testbed. Should contain the actual implementation (such as DSN, Motelab), and the location of the deployment.
Contact Information	String	Contact information of the person responsible for this testbed. Should contain at least an e-mail address or a telephone number.
Features	List of Strings	Each entry in this list describes a feature supported by this testbed. This is a simple way to communicate possible extensions to the described interface. An implementation could add more features to this list, which would signal a client that maybe additional commands are supported. Known entries: IMAGE_FORMAT_HEX testbed supports node images in hex format IMAGE_FORMAT_BINARY testbed supports node images in binary format SCHEDULED_EXECUTION testbed supports scheduling of test jobs IMMEDIATE_EXECUTION testbed supports immediate execution of test jobs Possible extensions: POWER_PROFILING testbed supports measurement of nodes' power consumption
Server Time	Timestamp	The current time of the server. Possible start times of scheduled jobs (see below) will always refer to this time. This should be no issue, as normally UTC is used for all implementations, but it helps to prevent obscure problems while scheduling jobs.
Status	String	The current status of the testbed. Must be one of the following: AVAILABLE the testbed is ready to be used IN_USE the testbed is currently in use and does not allow more users ERROR the testbed is in an error state and cannot be used

2.2.2 Get Node List

When constructing a test scenario for a specific application, one needs at least an overview of how many nodes are available and what their identifiers are. This operation therefore lists all nodes that *should* be available. This means that if there are some failing nodes in the testbed, they will be returned as well. This makes the interface more flexible, in that it is also possible to retrieve a list of failing nodes, however generating a list of working nodes is a bit more involved, as you will have to use the next operation to make sure each node is actually usable.

Description	Retrieves the list of nodes the testbed disposes of. The returned node identifiers are of data type string and are not restricted to a certain format. Also they don't need to be globally unique, but only in the scope of the testbed in question. Possible choices for identifiers would be IP-addresses, MAC-addresses or other custom IDs. Note, that not all nodes returned by this method are usable in for a test job, some might be disabled or in an error state. See the next operation to gather more information.	
Input	None	
Output	Node List	List of strings, each uniquely identifying a node in the testbed
Error conditions	None	

2.2.3 Get Node Description

To gather more information about single nodes, this operation returns a detailed description containing most importantly the state of the device (if it is programmable at all) but also a host of other properties, such as hard- and software or special features, if the testbed knows this information.

Description	Retrieves detailed information about a specific node, data that could be useful to develop a testing scenario or monitor the testbed status.	
Input	Node ID	Unique identifier of a node.
Output	Node Description	Custom data structure, see below.
Error conditions	Node non-existent	

Node Description Data Structure

Field Name	Data type	Description
ID	String	Unique identifier of the node. Used to identify the node in the testbed.
Hardware	Strings	Name of the hardware platform of the sensor node. Possible values would be 'TinyNode', 'Tmote', etc.
Software	String	Software name and version running on the node, if applicable. On DSN this is used to report DSN software version of the BTnodes.
Status	String	The current status of the node. Must be one of the following: AVAILABLE the node is ready to be used

		<p>IN_USE the node is functional but currently in use</p> <p>DISABLED the node is disabled by the testbed administrator and cannot be used</p> <p>REMOVED the node was removed by the testbed administrator and cannot be used</p> <p>DEFECTIVE the node is not functional for any reason</p>
Features	List of Strings	<p>Each entry in this list describes a feature supported by this node. Currently no special node features are defined, but one could for example think of an implementation which supports power profiling only for certain nodes. This could be indicated by adding the following entry to the feature list of these nodes:</p> <p>POWER_PROFILING node supports power consumption measurement</p>
Location	String	Description of the nodes location. Again, no specific format is required, this could be a room number or coordinates or even a free textual description.
Last Seen	Timestamp	Point in time where the testbed server communicated with the node for the last time. Useful to identify nodes that are in some kind of error state.
Text	String	Free text to add additional information to the node description.

2.2.4 Submit Job

The most important part of every test run is to program the testbed with the information needed to run a test job. This notion was discussed before, and in this operation a data structure containing all necessary information is sent to the testbed, which then is in charge to execute the test run autonomously.

Description	Programs the testbed with a certain test job. Depending on the start time of the latter, it is to be executed right away or scheduled for later execution.	
Input	Job Description	Custom data structure, see below.
Output	Job Identifier	Unique identifier of the submitted Job. Has data type String, but will usually just be an integer value.
Error conditions	<p>Testbed not available at the specified time</p> <p>Testbed does not support scheduled execution</p> <p>Wrong image format (binary/hex)</p>	

Job Description Data Structure

All information needed to run a complete test cycle is contained in this one data structure.

Field Name	Data type	Description
Name	String	Short name of the test job.
Description	String	Textual description of the test job. Not required, but often useful when managing a larger number of test scenarios.

2 The Interface

Start Time	String	Point in time to start the execution of the test job. May be the string literal 'now' to denote immediate start of the execution or a point in time in ISO 8601 format[12].
Duration	Integer	Amount of time to run the experiment, in seconds.
Images	List of Image Records	Custom data structure, see below

The implementing technology should allow more fields to be added to this (and possibly other) data structures. This allows to pass additional parameters which might enable extensions only supported by some server implementations.

2.2.4.1 Image Record Data Structure

Each image record contains an image of the application to be tested and a list of nodes to be programmed with this application.

Field Name	Data type	Description
Name	String	Short name of the image.
Type	Strings	Type of the image file contained in the image record. Must be one of the following: HEX the image is in Intel IHEX format BINARY the image is binary
Image File	String	The actual image to be programmed to the nodes. If the image is of type HEX, this field contains exactly the content of the image file. If it is of type BINARY, the image file is encoded in Base64. Because of the given testbed implementations, this data type could not be standardized, but should be adapted to the testbed used. A client could automatically switch the right format, based on the testbed description.
Node List	List of Strings	List of unique node identifiers, denoting the nodes to be programmed with this image.

2.2.5 Get Job Status

Because execution of test jobs is done by the testbed completely without any interaction, a special operation is provided to gather information about the current status of the job. Calling it might be useful right after submitting a test job, to make sure it was scheduled correctly, or when waiting for a job to finish, not to ask for results before it is actually done.

Description	Find out the status of a certain job that was previously scheduled.	
Input	Job ID	Unique identifier of the job in question, as returned by Submit Job.
Output	Status	The current state of the test job. Must be one of the following: SCHEDULED the job is scheduled correctly but execution has not begun yet. RUNNING the job is currently running DONE the job was successfully executed ERROR An error occurred while executing the job. You might still get additional information using the next method.
Error conditions	Job non-existent	

2.2.6 Get Job Output

After execution of a test job, it is essential to retrieve the generated log data. This is to be returned in a set of log files, one for each node involved in the test run and possibly more added by the testbed itself. This operation should also return helpful data if the execution of the test job failed, for example because of a fault while setting up the testbed.

Description	Fetch the logging output of a job after it has run.	
Input	Job ID	Unique identifier of the job in question, as returned by Submit Job.
Output	Data	All log data produced by the job to be returned as list of pairs of strings. The first member of each pair denotes the origin of the log messages in the second member. This can be seen as a collection of log files, each of which has a name (the identifier of the originating node). The format of the log contents (the second member) is free, but will usually be a list of log messages including a timestamp and maybe more meta information. The testbed server is free to add additional entries tracing the execution of the test job itself.
Error conditions	Job non-existent Job has not finished execution	

2.3 Error Handling

We do not define special return values in the different operations to indicate errors that occurred, but we rely on the error reporting mechanism of the implementing technology. In our implementation in Java the method throw a custom exception and in XML-RPC the native error reporting mechanism is used, adding `<fault>` clauses to the response.

2.4 Authentication

Just as error handling, authentication is intentionally left out in this specification and is assumed to be implemented on a lower protocol level. Our implementation in XML-RPC uses the HTTP basic access authentication scheme.

2.5 Implementation

As part of this semester thesis the described interface was implemented using XML-RPC[13]. We chose this protocol for its simplicity, portability and the availability of implementations for almost any programming language. On the server side, we expanded the existing DSN server written in Java to support the new interface and for the client side we wrote a standalone program to connect to any server implementing the interface.

The DSN server had so far no notion of a test job. The existing client, DSNTargetLogger had all needed control logic included to steer the relatively dumb testbed server through the test runs. To fulfil the Rams interface, this logic had to be ported into the DSN server itself. As for scheduling jobs, we decided only to support immediate execution of test jobs, unlike e.g. MoteLab, where jobs can be scheduled for execution in advance.

To prevent collisions of test jobs from different (or the same) users, a simple locking mechanism was added to the DSN server: Whenever one job is running, the Rams implementation is considered in use, and any jobs submitted in the meantime are rejected. In theory it would be possible to run two test jobs on two disjoint node sets, but this was not deemed any practical importance.

Using Java on both the server and the client side, we could share the code for the interface in both programs and encapsulate the protocol specific implementation on both sides. On the client side, there is a RamsProxy class, which implements the Rams interface and simply relies the method calls to XML-RPC calls. This component could be replaced by any other implementation of the Rams interface, allowing to easily switch the underlying protocol. In the server we use a RamsServlet, which accepts the XML-RPC requests and calls the corresponding methods on an implementation of the Rams interface. This servlet could be used by any testbed server implementing the Rams interface.

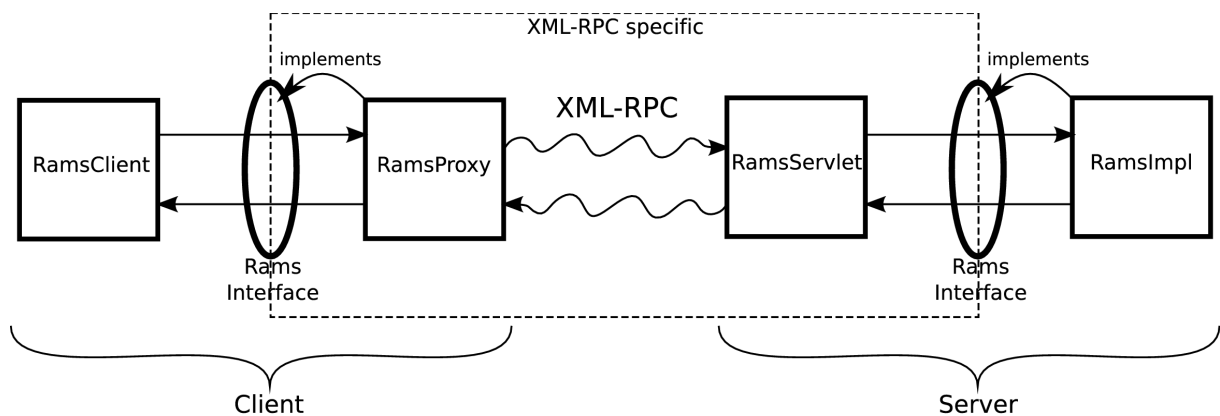


Figure 5: Calling diagram between the Rams client and the Rams implementation in the DSN server.

3 The Client

As mentioned before, one important part of this work was the implementation of a client for the Rams interface. Because this program is targeted at a very specific group of users, namely developers of applications for wireless sensor network devices, care was taken to find out what they expect from such an application and what functions it should implement to help them the most. So before starting to implement, we compiled a list of requirements for the client application.

3.1 Requirements

We present here the most important requirements in a non-formal textual description. While most of them are simple functional requirements, for some it is not that clear how they will affect implementation. In this respect, the requirements are listed in order of increasing 'fuzziness'.

3.1.1.1 *The client must be able to program the testbed.*

The most obvious and most important feature. Given a description of a test job, the program has to connect to the specified testbed server and submit the job using the respective method of the interface.

3.1.1.2 *The client must collect the output of test run.*

Once a test job is submitted to a testbed server, the program is able to determine when the execution of the job has ended and will then retrieve the log output of that test run. This output is then stored in several files containing the different parts of the log messages, in a configurable location.

3.1.1.3 *The client should be able to run in one or two phases.*

Given a test scenario that runs for e.g. six hours, it is desirable that the client can stop execution after the first phase of submitting the job and can later be reinvoked to collect the results of the test run. Thus the client has to support two modes: In the first one it submits a test job, waits for it to finish and collects the output in one single run. In the second mode, it exits after the first step, saving its current status (such as the identifier of the submitted job) in a status file, to be executed again later for the second step.

3.1.1.4 *The client should be able to explore a testbed.*

Given the address of a testbed server and valid credentials, where applicable, the client connects to this server, and lists all nodes with detailed information. It is also able to generate a sample configuration file as a starting point for the user to build his test scenarios.

3.1.1.5 *The client should be appropriately configurable.*

The configuration should be highly adapted to the problem at hand and support quick creation of new test scenarios, allowing to reuse parts of the existing configurations.

3.1.1.6 The client should be integrated into the TinyOS build system.

Since most of the prospecting users work on TinyOS applications, the client invocation has to be seamlessly integrated into the TinyOS build system. Ideally it would suffice only to add one additional argument to the usual make command line to invoke a completely automated test cycle.

3.2 Implementation

Just as the DSN server, the client was implemented in Java using the Apache XML-RPC library [14].

3.2.1 Configuration

The configuration is based on an XML file. We chose this standard for the availability of mature parsers and its well known syntax, allowing anybody with basic understanding of XML to edit a configuration file by hand. In the following discussion we will not use the term 'node' in the way it is usually used in the scope of the DOM, but instead use the more general term 'entity', to avoid confusion with sensor nodes.

The configuration file contains one single *ramconfig* entry at the top level. On the next lower level a free number of *testcase* and *testbed* entries can be defined.

A *testbed* entity contains all information that is specific to one testbed instance. This is for one the name and URI of the testbed server, as well as the credentials used to log in. Furthermore, it also contains sets of node lists in entities called *topology*.

These entries all have a name as an attribute and contain one or more *nodelist* elements, each of which in turn contains a list of *node* entities. The node entities contain valid node identifier for the enclosing testbed, each of which can only occur once in all the *nodelists* of a *topology*. Thus each *topology* entity defines a partition of all nodes of a testbed in different named subsets.

The *testcase* entities on the other hand roughly contain the information needed to describe a test job (see 2.1.2). Apart from a name and a description (which is of no semantic importance) this is for one the *timing* information. Valid values are the same as for the Job Description data structure (see 2.2.4). A *scenario* entity specifies on which testbed a test case should be running and what topology is to be used. The content of both attributes must refer to the name of a testbed entity and a topology contained therein.

Furthermore, a *testcase* lists several image entities, which reference files located on the local filesystem. File names are always interpreted relative to the location of the configuration file, unless they are absolute. An *image* entry also has to specify the type of the image file (binary / hex) and on which *nodelist* this image is to be programmed. These attributes must refer to a name of a nodelist in the *topology* referenced by the *scenario*.

Figure 6: Sample configuration file in the Eclipse XML editor.

Last but not least each testcase has its own logfolder entry, which specifies a directory in which the log files are to be stored. Again the name is interpreted relative to the directory containing the configuration file, unless it is absolute.

While a textual description of the configuration sounds rather involved, it is actually quite simple to write and extend configuration files. A simple example should demonstrate this: For an application having one sink node and a lot of sources one could define a number of topologies containing two node lists: one named 'sources', containing several nodes, and one named 'sink', containing only one node. Different topologies could differ for example in the number of source nodes, forcing the application to set up multi-hop routes in one case, while all nodes can communicate directly with the sink in another.

One would then create a number of testcase entries, each of which contains the same *image* entries: one for the 'source' node list and one for the 'sink'. The different test cases would differ in the topology, that is selected, the log folder and maybe the timing setup.

3.2.2 Invocation

The Rams client is usually deployed as a Java jar file and a wrapper script, which completely hides the Java specific details of invocation, so that on UNIX compatible system execution is no different from any other program.

Except for the special functions of exploring a testbed and generating a sample configuration file, the user must specify a configuration file and a test case in each invocation, in this order. The order of the other options does not matter, however they cannot be concatenated.

Option	Meaning
-p1	Only perform the first phase (submit job). Saves the job identifier returned by the testbed in the file 'state.xml' (in the same directory as the configuration file) to be retrieved later.
-p2	Only perform the second phase (get job output). Tries to find the job identifier in the file 'state.xml'. If one is found, the testbed is queried for the log output and the identifier is removed from the state file.
-n	Don't remove the job identifier from the state file when performing the second phase.
-e	Explore a given testbed. The program expects the URI of a testbed as parameter, including the credentials. It will print a list of all nodes and detailed information, if the -v switch is specified.
-g	Generate a sample configuration file. Expects the same parameter as -e. The generated file is printed to standard output.
-v	Be verbose. Adds a lot of additional information in almost every run mode.
-d	Show debugging output. Shows backtraces to all exceptions caught internally, useful mostly for debugging.
-h	Show terse usage information describing the possible options.

3.2.2.1 Examples

We show two examples of how the Rams client is typically invoked.

Generate a sample configuration file for the Rams instance running on the machine pc-4720 on port 8889, using the username 'freyd' and the password 'lustig'. The output is saved in the file 'config.xml'.

```
$ ramsclient -g http://freyd:lustig@pc-4720:8889 > config.xml
```

Run the testcase with the name 'local_twonode_short' as defined in the configuration file 'config.xml'. Additional information about the job and the testbed used are printed.

```
$ ramsclient -v config.xml local_twonode_short
```

3.2.3 Output

As described in 3.2.1, every test case has its own folder for storing the log files. While this provides for separate folders for different test cases, another approach is taken to separate different runs of the same test case: Upon retrieval of the output, a new directory named after the current date and time is created. A run ending at June 5, 2008, 10:45 would produce a folder named '200806051045'. In this directory all log files are saved with the names as returned by the testbed.

3.3 Limitations

The specification of the Rams interface and this client were always intended to be as generic as possible. While this has the advantage that virtually every testbed could be adapted to offer the interface and thereby be programmed by the client, it makes it difficult to use testbed features not included in the specification.

3.3.1 Special features

There exists a deployment of the DSN server which was equipped with a power measurement infrastructure. Support for this feature currently only exists in the DSNTargetLogger, which simultaneously connects to the DSN server as well as to the power measurement device. The Rams client currently cannot support this feature, because it is not integrated into the DSN server.

Just as for other special features, an implementation could be achieved as follows:

- Integrate the functionality into the testbed server.
- Add an additional flag to the feature field of the Testbed Description.
- Define one or more additional fields for the Job Description data structure, that configure the extension.
- Return the output as additional log files.

3.3.2 Multiple architectures

The integration into the TinyOS build system (see chapter 4) has currently no specific support for different hardware architectures. It might be desirable to use the same test case when compiling for different architecture and select the testbed according to the chosen architecture, which is currently not possible. However, because a test case is mostly specific for a current architecture, it is probably necessary to define a new test case for every architecture anyway.

4 Integration in TinyOS Build System

Experience has shown that it does not suffice to have a standalone program for software testing, but that it needs to be integrated into the normal build process in order to be used. The more integrated and the easier the execution of a testing facility is, the more likely it is to be used by application developers.

Since most if not all applications for wireless sensor networks developed in the academic context use TinyOS, a seamless integration in the existing build system was necessary. Many conventions exist in that context, for example, the result of compiling an application is always called 'main.exe' and is saved in a subfolder called 'build/architecture'.

We deemed it therefore valid to add another subdirectory called 'test', which has to contain the configuration file called 'config.xml'. When using relative pathnames for the log folders, the output of the test runs are thus saved in subfolders of the test directory.

To invoke testing, a new special target was added to the TinyOS build system: `test`. It has to be followed by a comma and the testcase to use. The build system will compile the application as usual and then invoke the Rams client.

```

[1] Done
freyd@pc-4720:/opt/tinyos-2.x/apps/Blink$ ll test/
total 100
-rw-r--r-- 1 freyd freyd 7603 2008-05-26 01:41 blink.ihex
-rw-r--r-- 1 freyd freyd 2850 2008-05-26 01:43 config.xml
-rw-r--r-- 1 freyd freyd 2849 2008-05-26 01:41 config.xml~
-rw-r--r-- 1 freyd freyd 29976 2008-05-26 01:41 DsnEchoTestApp.20070301.tos2.tmote.hex
-rw-r--r-- 1 freyd freyd 30 2008-05-26 01:41 node.ihex
drwxr-xr-x 11 freyd freyd 4096 2008-05-19 16:06 quick
-rw-r--r-- 1 freyd freyd 43439 2008-05-26 01:41 TmoteDsnTestApp.20061218.tos2.tmote.hex
freyd@pc-4720:/opt/tinyos-2.x/apps/Blink$ make tmote test,local_twonode_short
mkdir -p build/telosb
  compiling BlinkAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-hwmmul -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc
inkAppC\" -DIDENT_USER_ID=\"freyd\" -DIDENT_HOSTNAME=\"pc-4720\" -DIDENT_USER_HASH=0x518916d6L -D
  compiled BlinkAppC to build/telosb/main.exe
      2654 bytes in ROM
      55 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex
ramscient test/config.xml local_twonode_short
submitting Job, recieved ID 7
sleeping until job is done
job still running
testjob is done
retrieving job output
saved 2 logfiles
done.
freyd@pc-4720:/opt/tinyos-2.x/apps/Blink$

```

Figure 7: Invocation of a test run directly through the TinyOS build system.

4.1 Limitations

The configuration file currently has to be written by hand to comply with the TinyOS build system. For example, the build system cannot signal the Rams client, where the image file is located after compiling it (the location depends on the architecture compiled for), but the location is always read from the configuration file.

Support for split phase execution (where the program returns after submitting the test job to be invoked again later) is currently not present.

5 Summary

Designing and implementing applications for wireless sensor networks is not an easy task. One important tool in the development process are testbeds, which can evaluate performance and validate the correct working of applications.

To gain an overview about the current state of testbed implementations we conducted a little survey. It showed that efforts to facilitate testing had been taken by many institutions, but the actual implementations differ widely in feature richness, the amount and the quality of offered services. In particular, no two testbeds are compatible and general purpose testbed clients are missing.

To cope with this shortcoming, as a first step a generic interface for wireless sensor network testbeds was developed. It is intentionally kept very simple so that for every existing testbed an implementation should be possible, be it only in parts.

The new interface called Rams was implemented for the DSN testbed server in use at ETH Zürich. Substantial control logic had to be added to support the unattended execution of test jobs. Also, a client was written, that runs test jobs on every testbed supporting the Rams interface. It was tested on the DSN server but should be compatible with any future implementation.

Last but not least, the developed client was integrated seamlessly into the TinyOS build system. Using just one additional target in the invocation of the make tool, a future developer can trigger a fully automated test run, which delivers back neat log files into his source tree.

5.1 Future Work

One aim of developing a generic interface for wireless sensor network testbeds was to improve interoperability between the different existing solutions. The next step now is to actually implement the interface for as many testbeds systems as possible. In particular for MoteLab the necessary changes should be rather straightforward, as it already supports the notion of test jobs. Other testbeds could require more work, depending on the amount of control logic already present.

6 References

- [1] Jan Beutel, Matthias Dyer, Roman Lim, Christian Plessl, Matthias Wöhrle, Mustafa Yücel, Lothar Thiele, Automated Wireless Sensor Network Testing. In *Proc. 4th International Conference on Networked Sensing Systems (INSS 2007)*, IEEE, Piscataway, NJ, June 2007
- [2] <http://www.btnode.ethz.ch/>, May 2008
- [3] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin, Philipp Blum, Deployment Support Network: A toolkit for the development of WSNs, *Proc. 4th European Workshop on Sensor Networks (EWSN 2007)*, ser. *Lecture Notes in Computer Science*, Vol. 4373, pp. 195–211., January 2007,
- [4] Geoffrey Werner-Allen, Patrick Swieskowski, Matt Welsh, MoteLab: A Wireless Sensor Network Testbed. In *Proc. of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, April 2005
- [5] Vlado Handziski, Andreas Köpke, Andreas Willig, Adam Wolisz, TWIST: A Scalable and Reconfigurable Testbed for Wireless Indoor Experiments with Sensor Networks. In *Proc. of the 2nd Intl. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality, (RealMAN 2006)*, Florence, Italy, May 2006
- [6] <http://www.btnode.ethz.ch/Projects/Jaws>, May 2008
- [7] <http://enl.usc.edu/projects/tutornet/>, May 2008
- [8] Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, Amin Vahdat, Mirage: A Microeconomic Resource Allocation System for Sensornet Testbeds. In *Proc. of the 2nd IEEE Workshop on Embedded Networked Sensors*, May 2005
- [9] <https://mirage.berkeley.intel-research.net/>, May 2008
- [10] <http://www.millennium.berkeley.edu/sensornets/>, May 2008
- [11] Anish Arora, Emre Ertin, Rajiv Ramnath, William Leal, Mikhail Nesterenko, Kansei: A High-Fidelity Sensing Testbed, , Vol. 10, No. 2, pp. 35-47, March 2006,
- [12] http://en.wikipedia.org/wiki/ISO_8601, May 2008
- [13] <http://www.xmlrpc.com/>, May 2008
- [14] <http://ws.apache.org/xmlrpc/>, May 2008

7 Figures

Figure 1: Architecture of the DSN testbed.....	4
Figure 2: Architecture of the MoteLab testbed.....	5
Figure 3: Architecture of the TWIST testbed.....	5
Figure 4: Simplified communication between client and testbed.....	8
Figure 5: Calling diagram between the Rams client and the Rams implementation in the DSN server.....	15
Figure 6: Sample configuration file in the Eclipse XML editor.....	17
Figure 7: Invocation of a test run directly through the TinyOS build system.....	20