SWARMS - A Bi-Directional Interface For Sensor Testbeds

by

Charles Gruenwald

(No higher level degrees)

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Bachelor of Computer Science

Department of Computer Science

2006

This thesis entitled:
SWARMS - A Bi-Directional Interface For Sensor Testbeds
written by Charles Gruenwald
has been approved for the Department of Computer Science

Richard Han

Prof. John Bennett

Prof. Shivakaht Mishra

Date ____________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Gruenwald, Charles (B.S., Computer Science)

SWARMS - A Bi-Directional Interface For Sensor Testbeds

Thesis directed by Prof. Richard Han

Sensor Networks are becoming cheaper and more prevalent in the world of computing. Because of the availability and low expense, networks are being created with larger and larger numbers of nodes. Managing these many nodes proves to be quite a difficult process.

The current implementation provides a clean and elegant way to program many nodes contained in a large testbed, gather data sent from the nodes and to send data to each node individually. The system created to handle this setup is novel in its flexibility and usability.

Contents

Chapter

1 Introduction 1

1.1 Motivation 1

1.2 Related Solutions 2

1.3 Goals and Requirements 3

2 Architecture 6

2.1 Modular Pieces of System 6

2.1.1 Node 6

2.1.2 Node Mate 6

2.1.3 Cluster Controller 8

2.1.4 Head Server 9

2.1.5 Database 10

2.1.6 Web Server 11

2.1.7 Logger 11

2.2 Data Path and the 'Underlay' Network 12

2.2.1 Node Sending Data to Client 12

2.2.2 Client Sending Data to Node 13

2.2.3 Network Stack Figure 13

2.3 Control Path 13

2.4 Clients 14

2.4.1 Base Client 14

2.4.2 Standard Out Client 15

2.4.3 Node Browser / Network Visualizer 15

2.4.4 Existing MOS Pieces 16

2.4.5 Screenshot 17

2.4.6 Image Uploader 17

3 Future Work 19

3.1 Configurable Node Mates 19

3.2 Traffic Shaping and Multiplexing 19

3.3 Complete Build system Integration 20

3.4 Tiny OS Integration 20

3.5 Publish and Subscribe Interface for Clients 21

3.6 Access Control for External Clients 21

Bibliography 22

Appendix

A Solving the Goals and Requirements 23

Chapter 1

Introduction

Sensor nodes are small scale embedded systems run by a micro-controller. They are equipped sensors to gather information and a radio to send the information to other nodes. Additionally, many of the sensor nodes out on the market today are also equipped with a serial line to communicate directly with a computer. Nodes which are connected directly to the computer often act as the base-station to aggregate data from the nodes sending their information over the radio. We will explore using this serial connection as a means for diagnostics and controlling the nodes.

The focus of this thesis is to develop a testbed system which can manage many of these nodes at once. This involves attaching many nodes to a system via their serial link. The computer which is attached to those nodes can then communicate with the nodes over the serial line and reprogram the nodes with new images as needed.

1.1 Motivation

Currently, it is difficult to manage multiple nodes from one computer. Images are loaded through a terminal session, as a consequence one must open multiple terminals to interface with each node. When a developer wants to load a new image on the nodes, they have to execute the command which loads the new image separately for each node. This scheme of loading the nodes does not scale.

Additionally, the information coming from the node is viewed in a terminal which has the same consequence. In order to log the information coming from the node, the output from each terminal must be piped into a separate file. In this scheme, it is impossible to combine the logs from multiple nodes in real time.

Due to the fact that the node must be directly connected to the computer, one must have physical access to the node in order to interact with it. Since the sensor nodes are meant to communicate over the radio, it is common to use two or more nodes in a typical application. This means that each developer must have their own set of nodes, which will not be used when that developer is away. The solution proposed in this thesis solves the problems described above.

1.2 Related Solutions

There are existing solutions which solve some of the issues mentioned above. One of these is MoteLab [5]. MoteLab [5] is designed around sensor nodes connected to a network programming interface. The drawback to this system is the lack of hierarchy. As the testbed grows to a large size, it will be difficult to aggregate the data before the testbed. As a result, this system is not as scalable. Another problem with this system is it's lack of node heterogeneity. It would be difficult to add USB nodes or future nodes to this system without some modification. One goal in our system is to make it easy to add new nodes of different types. As such, it would be easy to add network nodes to our system as described below.

Another solution to scaling to many nodes during development is using a simulator. There are several problems with using a simulator over a testbed. Since the nodes in a simulator are virtual, the radio model must be emulated. Since the radio on these small scale devices is highly unpredictable, it is difficult to model such behavior. Another drawback to the simulation environment is that the code must be written in that simulators language. As a result, there are issues when converting this code to actually run on the node which do not occur in a simulator. A simulator is not able to gather real sensor data either. Lastly, the simulator cannot emulate every piece of hardware on the node. As such, debugging complex interactions between the radio, the scheduler and a particular sensor for example becomes impossible in a simulator.

1.3 Goals and Requirements

The goal of the testbed, as mentioned previously is to allow a developer to access hundreds of nodes at once. The system will alleviate many of the problems and issues described in §1.1. The goals of the system are as follows.

(1) Program many nodes in parallel with ease

(2) Program only a select group of nodes from the testbed

(3) Ability to aggregate information sent from the node

(4) Provide remote access for developers

(5) Fault tolerance to any particular node crashing

(6) Node uniqueness to assist in replicating experiments

(7) Access control for testbed system

(8) Node Heterogeneity in terms of supporting multiple types of nodes

(9) Access to the data sent from the node

(10) The Ability to send binary data to the node

It is shown how the SWARMS system fulfills these requirements in Appendix §A. Below is the motivation for each of these requirements.

There are many benefits to having a system which is designed to manage many nodes at once. Instead of loading each node individually, a developer can simply upload a new image to be loaded on a group of nodes. The system can then manage the individual loading of each node. Since the system knows which nodes are attached to the computer, it can easily determine the ports necessary to send to the loading program.

Aggregation of the node information is another advantage of this system. Instead of outputting the data the node sends to a file (separate for each node), the information can be stored in a database. This makes it easy to combine the data from all nodes. By using a database, we also have the advantage of being able to query the database for this log information, allowing the system to filter out less important information.

Remote access to the testbed also solves problems faced in the current setup. Now the development can occur remotely, one does not have to own a node to test their sensor-net code. Since the testbed can be shared, each developer doesn't need their own set of nodes on their desk.

In order to gain meaningful data from the nodes, we must be able to distinguish one node from another. This means that each node needs a globally unique ID. Since we use this ID to identify which node data is coming from and to select a node to load it must be static.

Since a common setup is to load a class of nodes with the same image, the system must be able to create abstract groups of nodes which can be treated as a single entity.

Since there is no supervisor mode on the nodes and since the system will be used for development purposes, it must be possible to determine when a node has crashed and optionally recover from this situation. This functionality is provided by a scheme similar to a watchdog timer. Each node will periodically send information over the serial signifying that the application is still running properly. If the system has not heard this message in a specified amount of time, it can be assumed that image on the node is not functioning properly.

Due to the fact that the testbed has many nodes attached and will be accessible remotely, different users should be able to access the testbed at different times. This requires accounts and a reservation system in order to prevent conflicts.

Chapter 2

Architecture

The diagram below (Figure 2.1) is a graphical representation of the architecture used to support the swarms system. Initially, each piece will be described on it's own, then a description of how the data flows through the system and finally a description of the clients and what role they play in the system.

2.1 Modular Pieces of System

2.1.1 Node

The node contains an application which may talk over the radio and sense data. The code on the node may or may not be stable, as such it is a possibility that the node may freeze. In our system, the node has the responsibility of communicating with the Node Mate. While the Node and the Node Mate need to understand the particular protocol running on the node, the Node Mate translates this protocol into swarms packets to be passed through the system. Our aim is to be as hardware and OS independent as possible.

2.1.2 Node Mate

The node-mate is a ruby process which runs on a computer with a node attached. There is exactly one node-mate for every node connected to a computer. Each node-mate directly controls the starting, stopping and loading the node. It also reads information

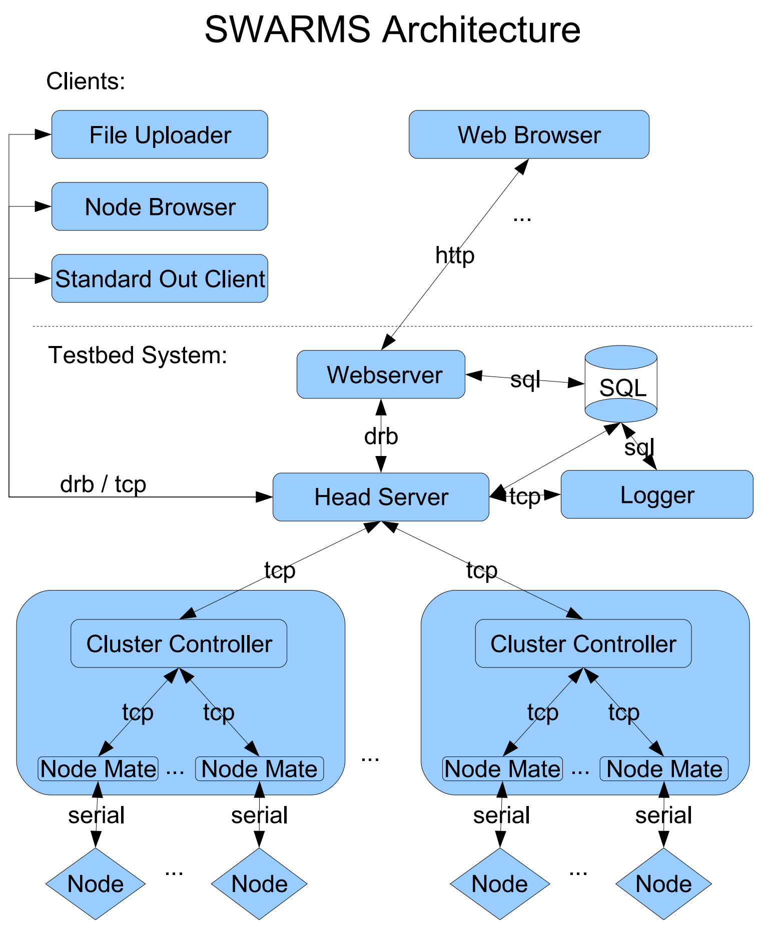


Figure 2.1: Swarms Architecture

in the node, parses it and forwards it through the system. Additionally, data passed to this process from the rest of the system must be translated by this process into the appropriate protocol running on the node. This part of the system is meant to be a simple layer which can translate packets between what the system understands and what the node understands. In order to add a new type of node to the testbed, one must implement a node-mate which can load images onto the node, start and stop the node, and finally translate packets. To make the process of creating a new node mate and to reduce the amount of copied code, the node mate was decomposed into separate orthogonal pieces. One piece is the hardware portion of loading/starting/stopping the node. The other piece is the translation layer specific to the OS on the node. As such, for new hardware, only the hardware portion must be created. Additionally, this modular design provides us with a bit of flexibility. Since the Node Mate is responsible for translation, it additionally has the opportunity to inspect or change the data before passing it through the system. It would be easy to create a custom Node Mate which performs aggregation/compression/encryption on the data before it passes through the system. An elegant interface for creating custom Node Mates was not created, however is slated for future work (see Chapter 3). This piece of the system provides the node heterogeneity. This process also provides fault tolerance. Once a node-mate hasn't received a message from a particular node after a given time, it can optionally reboot the node. The amount of time to wait for a keep-alive is configurable through the web interface.

2.1.3 Cluster Controller

The cluster controller is in charge of managing the node-mate processes. This is the process which can scan the ports on a computer looking for sensor nodes. Next it will kill old processes or start new node-mate processes as needed. There is one cluster controller for each computer which has node connected to it. The key to our scalability is the fact that we can handle multiple servers at this level. This means that many computers can be loaded with any number of nodes on them. Since they are all connecting to a larger computer we may choose the load of nodes per computer. As a result, many slower computers can be connected to only a few nodes and still function in the system. The cluster controller is also responsible for transferring the data between the Node Mates and the head server. When the cluster controller parses a packet sent from a node, it writes the data to the TCP connection open with the head server. Additionally, the cluster controller must perform the reverse process when it receives a packet from the head server. In this case, the cluster controller must query the underlay (local state information) to determine which process (Node Mate) to send data to.

2.1.4 Head Server

The relationship between the server and the cluster controller is what provides the high amount of scalability. Due to the limitations in USB addressing and furthermore a limitation in bandwidth, each computer can only handle a certain number of nodes. To handle more nodes past this limitation, multiple cluster controllers are connected to a single server. Another advantage of this setup is that smaller testbeds can exist in entirely different locations and still report the information to the same database. The Head Server has two basic roles in the testbed system. Below, each of these roles are broken down by functionality and described in detail.

2.1.4.1 Controlling The Testbed

One main aspect of the testbed which the Head Server is responsible for is controlling the testbed. This is the main point of access to start and stop jobs. Both the webserver and external clients create a DRb connection to the head server and issue RPC commands to perform these actions. The Head Server also acts as a layer of indirection to the database. Clients may query the Head Server for information such as a list of nodes, list of jobs, current running job and the like.

2.1.4.2 Multiplexing and Demultiplexing Data

The other main function of the Head Server is routing swarms packets to/from nodes and clients. When the Head Server receives a packet from a cluster controller, it duplicates that packet for all clients connected. Additionally, when a client sends a packet destined for a node, the headserver inspect the packet then queries the underlay to determine which cluster controller to forward the data to.

2.1.5 Database

All of the information about a testbed is held in a database. Users, computers, nodes, jobs, log information is all stored here. This gives us the flexibility of showing the information in the database in many different ways. This makes it easy to access and filter the information sent from a node. For instance, this allows the user to download log-files for the entire system or for a particular node without having to create any external files. The database also eases the burden of handling such a large amount of information. One interresting problem we came across in developing this system was the database as a bottleneck. There were two main issues, one being that database inserts were taking too long and interferring with interactive clients. To solve this problem we added a configuration parameter for disabling the database logging feature for any given job. A second problem related to the database is that it was quickly filling with too many log entries and the indexing (and sorting) was slowing down the web interface. To solve this problem, we developed the concept of snapshots as described below.

2.1.5.1 Snapshots

Snapshots represent our solution to old data in the database causing decreased performance due to indexing. Instead of always keeping the logs in the current log table, we created a separate table known as 'snapshots'. In this table, we dump all the current logs as a single entry in the table. This has the advantage that indexing is not a problem as 300,000 (common area of slowdown in our experience) becomes a single entry in a separate table. On top of this, we provide an interface through the web client which allows users to view/download/delete old logs. We can autmatically create snapshots from prevous jobs so the current user has a fresh table to insert logs into. Lastly, this conveniently organizes the logs based on the job which was running.

2.1.6 Web Server

This piece of the system is the main interface in which a normal user interacts with the testbed. This allows the user to be located anywhere and using any OS. As long as they have the correct type of image, they may use the testbed. By having the web interface be the single point of entry to the testbed, it is easier to control access between multiple users. The registration system is contained in the web interface as well. The Web Server provides the cleanest interface to the current (and previous) logs along with the status of the system. When a user begins to program the nodes, they are brought to a page which periodically refreshes a table listing the nodes and their status. Once the nodes have all been programmed, the user can navigate to a page which contains a list of the current logs. This page which contains the logs is also updated every 15 seconds.

2.1.7 Logger

The logger is implemented as a client (see section 2.4) which resides on the same computer as the head server. It simply inserts each packet of data from the node into the database which can then be viewed by the web-client. The web client provides two views for accessing this data. One way to view the data is the 'live view' which refreshes the most recent entries into the database every 15 seconds. Additionally, these views are stored as 'snapshots' in a separate database for historical storage of node messages. This is the most basic way to get data from the node.

2.2 Data Path and the 'Underlay' Network

This section describes how actual data coming from/going to the node travels through the system and the Internet. The system was designed as an underlay[1] network which would pass packets from the nodes to the client and vise-versa. In MOS [2] all communication protocols send 'comBufs' [3] through the communications layer to a specific device [4] . Communication over the serial line (including the printf library) is abstracted away to the higher layers. As a result, to be agnostic to data in the buffer, we chose to send these as packets. Below the communications layer, the serial driver add a two byte preamble for synchronization. This can be seen visually in Figure 2.2.

2.2.1 Node Sending Data to Client

The Node Mate listens on the serial line and extracts the data from the com buf. Next it packs it into a swarms packet which contains the node id and type of message. The swarms packet is as follows:

```
~node_id~message_type~dataEOM
```

The Node Mate is given the node id when it is spawned. The message type is always 0 for node messages. The Node Mate then writes this message to the TCP/IP socket

[1] Underlay is a thin layer of cushioned material that is underlaid under a floor to provide sound, moisture and heat insulation. - wikipedia

[2] MOS [1] (The Mantis Operating System) - An open source OS for wireless sensor networks

[3] Like a Pascal string, a comBuf starts with single byte of length followed by that many bytes of data. This layer relies on lower layers for synchronization.

[4] The Radio or Serial

open with the cluster controller. The cluster controller then multiplexes the packets and sends them to the head server. Next the head server multiplexes the packets and sends them to each of the clients which are connected. The base client then parses the packet and calls the handle_message(node_id, message) function.

2.2.2 Client Sending Data to Node

For a client to send data to the node, it calls the send_to_node(node_id, message) function of the base class. Next the base class encapsulates the data in a swarms packet and writes it to the TCP socket of the head server. The head server parses the packet for the node_id and queries the database for which cluster controller this node belongs to. Next the server writes the packet to the cluster controller TCP connection. The cluster controller parses the packet then writes the data to the appropriate TCP connection based on a hash keyed on the node_id. Finally, the node mate parses the packet, converts it into a comBuf and writes two synchronization bytes along with the comBuf to the serial port. The serial driver passes the comBuf up to the communication layer. Next the communication layer passes the packet on to any thread which may be listening on that device.

2.2.3 Network Stack Figure

The following Diagram (Figure 2.2 represents this message passing process.

2.3 Control Path

An API for controlling the testbed remotely from a client is provided as well. For this aspect of the system we used DRb (distributed ruby objects). This provides an abstraction for connecting to a remote object and running methods from that object. Much of the functionality provided through the web server is exposed through this interface as well. Examples of these functions include: starting/stopping jobs, uploading

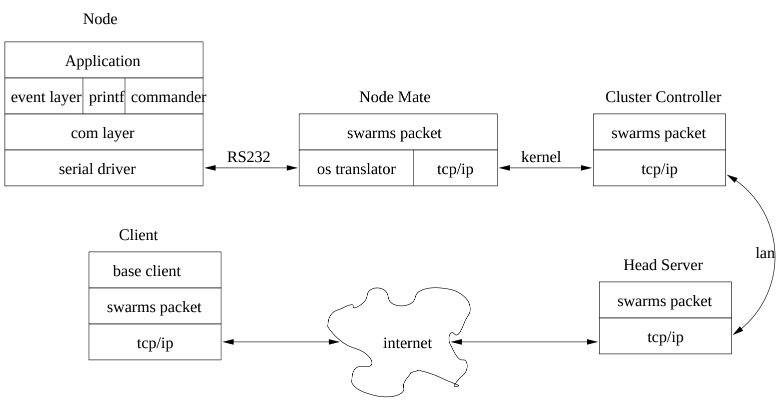


Figure 2.2: Network Stack Diagram

files, obtaining the current job, retrieving a list of nodes/groups, and restarting a single node. The following code snippet (Figure 2.3) demonstrates how a ruby client could restart the current job using this control interface.

2.4 Clients

This section describes the clients which and end user would run that connect to the testbed system. As described in section titled Data Path, clients transfer data from and to the node using the testbed underlay. Control of the testbed is managed through a DRb connection.

2.4.1 Base Client

The base client is an abstract class which each of the actual clients inherits from. It provides the connection to the swarms system and a convenient API for passing data to and from the node. The base client implements a function which accepts binary data

```
server = DRbObject.new(nil, ‘‘druby://swarms.cs.colorado.edu:6555’’)
current_job = server.current_job_id()
server.start_job(current_job)
```

Figure 2.3: Example DRb Code

to be sent to the node then packs the data into the appropriate underlay format and sends it through the Internet to the system. Additionally, the base client listens on the TCP socket for data coming from the nodes. Next, the base client parses the TCP byte stream for actual node packets and passes these packets on to a function which handles the packet. Each of the clients below simply override this function with their own method for handling the packets.

2.4.2 Standard Out Client

The standard out client is quite simple and is meant to provide an example implementation. It inherits the connection from the base client to provide the TCP streams to and from the system. The main point of interest in this client is the function for handling packets which simply spits out the node packets on the terminal.

2.4.3 Node Browser / Network Visualizer

The Node Browser is a GTK 'thick' client created for an interactive session with the testbed. It demonstrates how application level protocols can be built on top of the testbed system. The main focus of this application was for visualizing the trees in a minimum spanning tree protocol. The node sends events describing the network which get passed to the application before being parsed. The event packet structure looks like the following:

sentinel	event_type	event_id	extra_params
0	network	new_parent	parent_id

The client receives these commands and performs some action based on the event. First the event_type is demultiplexed to the particular handler, in this case the event is passed on to the 'network event handler'. Next the network event handler determines that this node has obtained a new parent with the specified parent_id. As a result, the client draws a line from that node to it's parent if it exists. By using the initial sentinel character of 0, we are able to distinguish between event packets and printf messages from the node. As such, printf messages are handled differently by being passed into a per-node-log also viewable from the Node Browser.

2.4.4 Existing MOS Pieces

This client takes advantage of several pieces of software in MOS[1] described below.

2.4.4.1 Commander

To demonstrate data flowing from the client to the node, the Node Browser leverages an existing piece of MOS software, the commander. The commander accepts a string (human readable) command over the serial line and executes a command based on that string. During that command's execution, more parameters can be obtained from the serial line. To use this functionality, one simply selects a node and types the necessary command in the prompt. This provides a convenient way to cause events to happen on a specific node. One example where this is useful is if the user wants to watch events happen live. By sending the 'beacon' command to the base-station node, a beacon will be sent out over the network. This allows the user to control when events happen and observe the results in real time. This thread is meant as a flexible light-weight layer for human interaction. A more sophisticated distributing debugging environment such as Marionette[2] is desirable for debugging the nodes.

2.4.4.2 Printf library

The printf library is a lightweight library which facilitates the output of debug code in a human readable format. It simply converts numbers of different sizes to a different base (in ASCII format). This library has been used for debugging purposes in the past, as such the system is built with this backwards compatability in mind.

2.4.4.3 MST Protocol

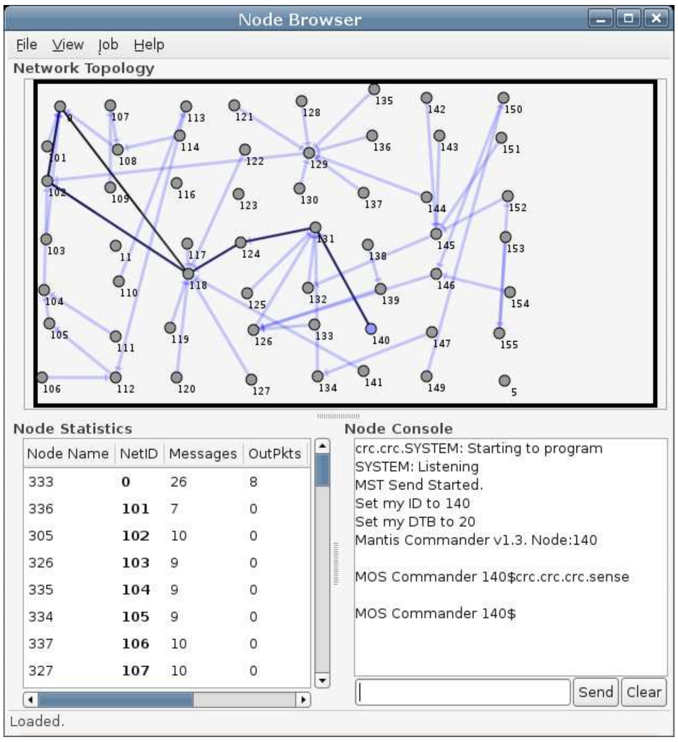
The MST protocol in MOS[1] is a simple beacon based vector routing protocol. The basestation node sends a broadcast packet out over the radio at a low transmit power. The nodes which hear this packet place the destination in their routing table, update the packet and retransmit the packet. This process continues until the packet leaves the system. To route data back to the basestation, nodes use their routing table to find their parent. The Node Browser facilitates a visual inspection of this parent relationship along with packet traces. One optimization (along with other bug-fixes) resulted from deploying the MST protocol on the visualizer. When the nodes setup their routes they turn down their radio output power, however when they send data they turn up their radio output power. In this case, the nodes setup parents which should be better than they need to ensure that the packet has a better chance of making it to the testbed.

2.4.5 Screenshot

A screenshot of the Node Browser was taken and is given in Figure [?]. Here we can see in blue arrows the nodes have setup their trees. The base station was issued two 'beacon' commands to establish these routes. Additionally, one can visually see the path a packet took when node 140 was issued the 'sense' command. The output in the 'Node Console' area is coming from printf commands, the line 'MOS Commander 140$' represents the prompt for the commander. The node had recieved 3 crc errors over the radio before the sense command was sent.

2.4.6 Image Uploader

The image uploader is a utility program to assist in uploading new images to the system. This provides a systematic way of changing the image and updating it without going through the web interface. This simple interface could be integrated with the



Node Name	NetID	Messages	OutPkts
333	**0**	26	8
336	**101**	7	0
305	**102**	10	0
326	**103**	9	0
335	**104**	9	0
334	**105**	9	0
337	**106**	10	0
327	**107**	10	0

```
crc.crc.SYSTEM: Starting to program
SYSTEM: Listening
MST Send Started.
Set my ID to 140
Set my DTB to 20
Mantis Commander v1.3. Node:140

MOS Commander 140$crc.crc.crc.sense

MOS Commander 140$
```

Figure 2.4: Node Browser Window

build system quite easily. To use the client (to upload a new image) one simply specifies the job_id and the image to upload. The client creates a DRb connection to the system, uploads the file and re-starts the job. A small code sample is given in Figure 2.5 which performs this action. The 'scons -D ...' line builds the images. The next two lines tell the uploader to upload the given images to job items 13 and 14 respectively. The 'true' on the end of the last line tells the client to restart the system. With this simple script, one is able to rebuild, upload and restart their job. As discussed in Chapter 3, a more elegant integration into the build system is desirable.

```
#!/bin/sh
scons -D platform=telosb && \
cd build-telosb && \
file_uploader.rb 128.138.206.62 13 testbed_base.elf && \
file_uploader.rb 128.138.206.62 14 testbed_sense.elf true
```

Figure 2.5: Example Image Uploading Code

Chapter 3

Future Work

The goal of this section is to discuss the areas of the system which were considered out of scope. Additionally, this section discusses possible areas of future work to improve the testbed system.

3.1 Configurable Node Mates

As discussed earlier in the paper, one area of future work is expansion of the node mates. Since this piece was modularized into hardware/software dependent parts it would be easy enough to build a custom node-mate out of these parts. Additionally, the node mate is the perfect place to perform any aggregation, compression or encryption. A main advantage of putting this code here is that it has not entered the system yet, as a result we could greatly decreased the necessary bandwidth by compressing the data before it goes through the system. Another advantage to compressing the data here is that the application / client developer is familiar with the source data (for instance if a node is sending audio data) and can compress it accordingly.

3.2 Traffic Shaping and Multiplexing

One issue which arose with the testbed is the use of TCP and the risk of congestion. Since there are TCP connections connecting the different pieces of the system, when a given client takes too long to process the data, the head server's TCP buffer gets filled. Next, the cluster controller can't write data to its TCP buffer, as a result we see a cascading effect. One potential solution to this problem would be to perform some form of traffic shaping at the node mate. Additionally, the system is potentially being hit by large overhead from TCP/IP headers. To solve this problem, multiple packets could be placed into a single TCP/IP packet, thus reducing the overhead. A drawback to this solution is the delay incurred for buffering the packet. Neither of these solutions were explored in depth, as a result, the are both viewed as future work.

3.3 Complete Build system Integration

While a simple client was created to ease the pain of continuously uploading images during development, it was nowhere near complete. Ideally, a user would specify the name of a job and the images and groups needed. Next the build system could find that job based on the name (create one if it doesn't exist) then upload the images accordingly. Right now, the simple client requires the user to create a job on their own and upload the images once. Once they've uploaded the images they can determine the ID of each job item and place that in the script for uploading. This process should be made more automatic for the casual user.

3.4 Tiny OS Integration

The system was built from the ground up with OS independence in mind. In order to support TinyOS [6], the part of the Node Mate which translates between TinyOS packets and swarms packets must be developed. Next, some form of control or configuration interface could be provided by the web interface to control which os-translator would be run for each job. Implementing this piece of the system would make other debugging tools such as Marionette [2] fit right into the system.

3.5 Publish and Subscribe Interface for Clients

One issue at hand is that when any client connects to to testbed, they instantly recieve messages from every node. Ideally, the client should be able to tell the head server which nodes it is interrested in to limit the ammount of traffic potentially traversing the internet. This was considered out of scope for this thesis.

3.6 Access Control for External Clients

Security and access control are both potential risks directly resulting from opening up an interface to the testbed to external clients. Currently, the system does not perform authorization or registration checks on clients. As a result, we find the potential problems of multiple clients controlling the testbed at the same time. Each client should be required to login much like the web-client must do before performing any actions. Ideally, the external clients and the web interface could use the same functionality for authentication and registration checks. This issues should be explored further as future work.

Bibliography

[1] H Abrach, S Bhatti, J Carlson, H Dai, J Rose, A Sheth, B Shucker, J Deng, and R Han. Mantis: System support for multimodal networks of in-situ sensors. In 2nd ACM Ineternational Workshop on Wirless Sensor Networks and Applications (WSNA 2003), 2003.

[2] K. Whitehouse; G. Tolle; J. Taneja; C. Sharp; S. Kim; J. Jeong; J. Hui; P. Dutta; D. Culler. Marionette: Using rpc for interactive development and debugging of wireless embedded networks. IPSN-SPOTS, 2006.

[3] P. Dutta; J. Hui; J. Jeong; S. Kim; C. Sharp; J. Taneja; G. Tolle; K. Whitehouse; D. Culler. Trio: Enabling sustainable and scalable outdoor wireless sensor network deployments. IPSN-SPOTS, 2006.

[4] L. Girod; A. Cerpa; T. Stathopoulos; N. Ramanathan; D. Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. USENIX Technical Conference, 2004.

[5] Patrick Swieskowski Geoffrey Werner-Allen and Matt Welsh. Motelab: A wireless sensor network testbed. Information Processing in Sensor Networks, 4:483–488, 2005.

[6] J HIll, R Szewczyk, A woo, S Hollar, D Culler, and K Pister. System architecture directions for networked sensors. In Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.

[7] Gilman Tolle and David Cullter. Design of an application-cooperative management system for wireless sensor networks. In 2nd European Workshop on Wireless Sensor Networks (EWSN), 2005.

Appendix A

Solving the Goals and Requirements

This chapter re-visits the requirements described in §1.3 and explains how each of them are met by the system.

(1) Program many nodes in parallel with ease - By distributing the cluster controllers across multiple computers, the system is able to program many nodes in parallel.

(2) Program only a select group of nodes from the testbed - Because we used the concept of 'groups' across the board, it is quite easy to specify only a subset of the total nodes to be programmed. A user simply creates a 'job' which includes a mapping of images to groups. This provides enough flexibility to specify any number of images to be programmed on any subset of the nodes in the system.

(3) Ability to aggregate information sent from the node - Due to the fact that the node-mate is a separate piece of the testbed, it is easy enough for this process to perform aggregation, compression or encryption before passing the data through the system. Because the data coming from the node-mate is treated like binary data, any data manipulation can be performed at this level without affecting the system.

(4) Provide remote access for developers - Both the web interface and the multiple client interfaces (DRb and TCP) provide remote access to the testbed system

without a physical connection.

(5) Fault tolerance to any particular node crashing - To solve this problem which occurs quite frequently, the node-mate implements a watchdog timer. For each job-item, the user may specify the maximum number of seconds to wait without receiving any data from the node before restarting it. This function can be turned off as well by specifying to wait 0 seconds before a reboot.

(6) Node uniqueness to assist in replicating experiments - One issue initially faced by the testbed software was specifying the same node for experiments when the usb device id's were being assigned randomly by the kernel. To fix this problem, we created a u-dev (unix's device file system) to create symlinks to the actual devices which were uniquely named by the nodes usb device id (which is unique across all nodes produced). By using this unique device id, we were able to always address the same node regardless of the generic id assigned by the kernel.

(7) Access control for testbed system - The web-interface requires a login and password and a reservation to access the system. As future work, this access control should be pushed out to the clients as well. Currently, the system assumes 'fair use' from the users which is a potential security risk.

(8) Node Heterogeneity in terms of supporting multiple types of nodes - Again, the node mate is the replaceable piece of software which provides a gateway between the sensor network and the ip network. By implementing a new node mate which can interface with any given piece of software allows that node to connect to the system.

(9) Access to the data sent from the node - Use of an underlay network solves this problem. Details are described more in the architecture section.

(10) The Ability to send binary data to the node - This is simply using the reverse direction of the underlay network to send data to the nodes and is described in the architecture section as well.