DESIGNING AND ANALYZING AN EVENT SERVICE FOR SENSOR NETWORKS

by

SUMEET GUJRATI

B.Sc., Devi Ahilya University Indore, India, 2002
M.C.A., Indian Institute of Technology Roorkee, India, 2005

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2008

Approved by:

Major Professor
Dr. Gurdip Singh

ABSTRACT

This work is motivated by the OMG's CORBA Event Service Specification. CORBA is the acronym for Common Object Request Broker Architecture. In this research, we implemented and analyzed an event service using a model similar to the OMG model for sensor networks applications which are written in nesC programming language, an extension of C programming language. This implementation has been tested on a test bed created using Crossbow's TelosB motes and Crossbow's Stargate Netbridge modules as gateways. Event service interface implementations, which reside on the motes, are written in nesC. The data routing part, which is done through Stargate Netbridges, is written in the C language. This document contains experimental results obtained by deploying and running the implementation on the test bed.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

First and foremost I offer my gratitude to my advisor, Dr. Gurdip Singh, who has supported me throughout my thesis with his patience and knowledge whilst providing me the funding to work in my own way. I attribute the level of my Master degree to his encouragement and effort and without him this thesis, too, would not have been completed or written.

I am thankful to Dr. Daniel Andersen and Dr. Mitchell Neilson for kindly serving in my major committee. In my daily work I have been blessed with a friendly and cheerful group of fellow students. I would like to thank Dinesh Challa, Sandeep Pulluri and Lakshman Kaveti for providing me good comments on my work and helping me develop various parts of the implementation.

The department of Computing and Information Sciences has provided me the support and funding I have needed to complete my thesis. I am thankful to the department.

I am grateful to my uncle and aunt Mr. Kailash Mahajan and Mrs. Lata Mahajan of Ann Arbor, MI for their kind support throughout my stay in USA. I am thankful to my wife Sonal for her constant support and unbelievable patience throughout my research work. I am thankful to my best friend Minto Michael for always encouraging and supporting me, especially at terrible times.

Finally, I thank my parents for supporting me throughout my studies at K State. Without their support and encouragement I could not have attended the University.

CHAPTER 1 - Introduction to the CORBA Event Service

CORBA is the acronym for Common Object Request Broker Architecture. The CORBA event service specification is a communication model which allows a producer application to send event data to a consumer application. The model defines two different approaches for initiating event communication. Each of these two approaches again can take two forms. This chapter gives an overview of CORBA event service specification terminology and concepts, which are the basis of the implementation of our event service for sensor networks. This document uses terms *producer* and *supplier* interchangeably. Figures and some contents of this chapter are from [1] and [2].

1.1 Overview

The client-server communication model is fundamental to the CORBA architecture. In this model, a standard CORBA request by a client results in a synchronous execution of an operation on a specified object in a server. If the operation defines parameters or return values, then the data is communicated between the client and the server. For the request to be successful, both the client and the server must be available. This model is depicted in *Figure 1.1*

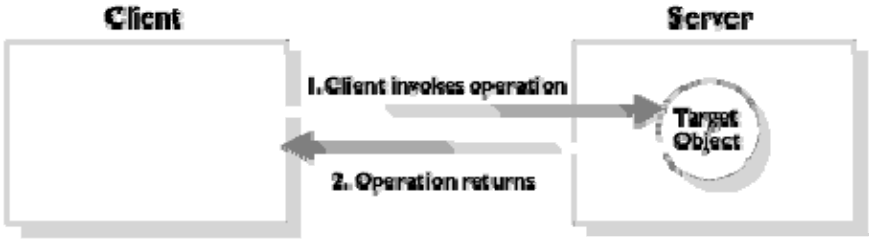


Figure 1.1 CORBA Model for Basic Client/Server Communication

If a request fails because the server is not available, then the client receives an exception and in such a case, it must take some appropriate action. However, some applications need a more complex and indirect communication style. For example (Examples 1 to 4 are taken from ref [1]):

1. A system administration tool is interested in knowing if a disk runs out of space. The software managing the disk is unaware of the existence of the system administration tool.

The software simply reports that the disk is full. When the disk runs out of space, the system administration tool opens a window to inform the user which disk has run out of space.

2. A property list object is associated with an application object. The property list object is physically separate from the application object. The application object is interested in the changes made to its properties by a user. The properties can be changed without involving the application object. That is, in order to have reasonable response time for the user, changing a property does not activate the application object. However, when the application object is activated, it needs to know about the changes to its properties.
3. A CASE tool is interested in being notified when a source program has been modified. The source program simply reports when it is modified. It is unaware of the existence of the CASE tool. In response to the notification, the CASE tool invokes a compiler.
4. Several documents are linked to a spreadsheet. The documents are interested in knowing when the value of certain cells has changed. When the cell value changes, the documents update their presentations based on the spreadsheet. Furthermore, if a document is unavailable because of a failure, it is still interested in any changes to the cells and wants to be notified of those changes when it recovers.

Clearly, this type of communication is indirect and cannot be handled by simple client-server style. Hence, we need event-based communication

1.2 Event Communication

The Event Service decouples the communication between objects. The CORBA event service defines the concept of *events* for CORBA communication. The application, at which the event is originated, is called an event *supplier* and the applications, which register themselves to receive event data, are known as *consumers*. Event data are communicated between suppliers and consumers by issuing standard CORBA requests. There are two approaches to initiating event communication between suppliers and consumers, and two orthogonal approaches to the form that the communication can take.

To support this model, CORBA event service introduces a new element, called an *event channel*. An event channel is a middleware which mediates the transfer of events between

suppliers and consumers. The steps for event communication between a supplier and a consumer are as follows:

1. Consumers register themselves with the event channel for the events in which they are interested.
2. Suppliers advertise the event data to the event channel.
3. After channel receives event data from the suppliers, it forwards the data to appropriate consumers.

An event channel allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events. *Figure 1.2* depicts this concept.

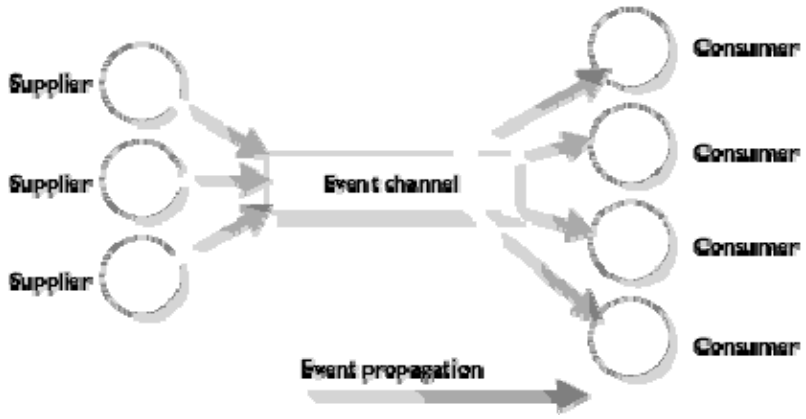


Figure 1.2 Suppliers and Consumers Communicating through an Event Channel

Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests.

1.3 Initiating Event Communication

The two approaches to initiating event communication are called the *push model* and the *pull model*. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model allows a consumer of events to request the event data from a supplier. In the push model, the supplier takes the initiative; in the pull model, the consumer takes the initiative.

1.3.1 Push Model

In the push model, suppliers generate events and pass them to a consumer via event channel. In other words, suppliers "push" event data to consumers; that is, suppliers communicate event data by invoking `push` operations on the **PushConsumer** interface.

In order to set up push-style event communication, consumers and suppliers need to exchange **PushConsumer** and **PushSupplier** object references. Event communication can be broken in two ways, either by invoking a **disconnect_push_consumer** operation on the **PushConsumer** interface or by invoking a **disconnect_push_supplier** operation on the **PushSupplier** interface.

Figure 1.3 illustrates push-style communication between a supplier and a consumer.

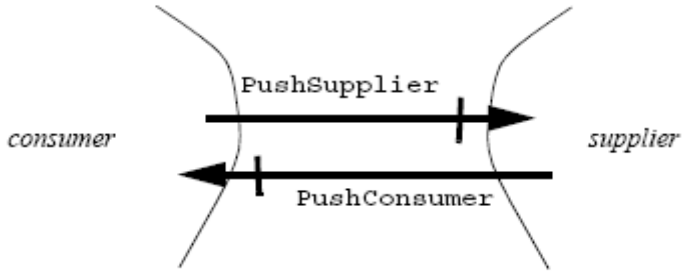


Figure 1.3 Push-style Communication between a supplier and a consumer

1.3.2 Pull Model

In the pull model, consumers request suppliers to generate events. In other words, consumers "pull" event data from suppliers; that is, consumers request event data by invoking **pull** operations on the **PullSupplier** interface.

In order to set up a pull-style communication, consumers and suppliers need to exchange **PullConsumer** and **PullSupplier** object references. Event communication can be broken in two ways, either by invoking a **disconnect_pull_consumer** operation on the **PullConsumer** interface or by invoking a **disconnect_pull_supplier** operation on the **PullSupplier** interface.

Figure 1.4 illustrates pull-style communication between a supplier and a consumer.

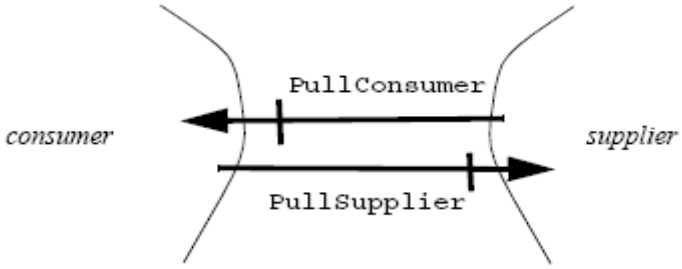


Figure 1.4 Pull-style Communication between a supplier and a consumer

1.4 Types of Event Communication

As mentioned above, each of these two approaches of event communication can take one of the two forms, *typed* or *untyped*. We briefly describe each of them in the following sections.

1.4.1 Untyped Event Communication

This form uses generic `push` and `pull` operations. An event is propagated by calling a series of `push` or `pull` operations. The `push` operation requires a single parameter which is event data of type `any`. The `pull` operation requires no parameters, but it transmits event data of type `any` in its return value. Both consumer and supplier applications must agree on the contents of the type `any`.

1.4.2 Typed Event Communication

In this type of event communication, the programmer specifies application specific interfaces through which events are transmitted. Instead of using `push` and `pull` operations and transmitting data of type `any`, the programmer specifies an interface and data type that suppliers and consumers use for transmitting events.

CHAPTER 2 - The Test bed

As mentioned in the abstract, our experiments are deployed and tested on a test bed. While creating the test bed, much of the time was spent on basic experiments to decide on the board (see *Figure 3.5*) dimensions and gateway to be used. Some of the time was spent on selecting USB hubs, USB cables and Velcro from a variety of vendors so that they all fit well on the board. This chapter describes various components and how they are integrated together to form the test bed.

2.1 Components

Test bed was created using various hardware components. The components are listed below:

Figure 2.1 Crossbow's TelosB motes

Figure 2.2 Crossbow's Stargate Netbridges

Figure 2.3 USB hub

Figure 2.4 USB Cable

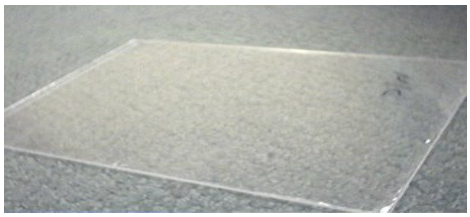


Figure 2.5 18” x 36” Plexi Glass board

Figure 2.6 Router

Figure 2.7 Ethernet cable

Figure 2.8 Velcro

The test-bed uses 12 plexi glass boards. Each board is made up of 8 TelosB motes, 1 Stargate Netbridge and one 13 port USB hub. Thus, the test-bed consists of 96 motes, 12 Stargate Netbridges and 12 USB cables. Motes are powered using the USB hub via USB cables. A USB cable is used to connect the hub to a Stargate Netbridge’s USB port, thus each mote can be accessed by a C program running on the Stargate Netbridge. Stargate Netbridges on each board are connected to the router via an Ethernet cable. This allows the Stargate Netbridges to communicate each other via TCP connections.

2.2 Determining Board Size

We wanted to create a mesh of 96 TelosB motes in the lab. The maximum communication range of TelosB motes is around 75 meters -100 meters if there is no obstacle in between. We wanted 96 motes to communicate each other in a mesh and since the lab is very

small, each mote will be able to communicate with every other mote and our purpose of creating the mesh will not be justified.

There are several interfaces provided by TinyOS, using which we can reduce the communication range of the motes. **CC2420Control** is one such interface. It provides a command **SetRFPower (int range)**. We used this command to solve our purpose. By hit and trial experiments, we found that if we use **SetRFPower(1)**, the two motes cannot communicate if placed more than 3-4 inches apart. In this case, the motes have to be placed very close to each other. We did not want that. We figured out that with **SetRFPower(2)**, the communication range is 8-10 inches and with **SetRFPower(3)**, the communication range is approximately 1 meter, which is very large. We decided to choose **SetRFPower(2)**.

To create a 2 x 4 mesh of 8 motes, we required no more than 18" x 36" space. We decided to have 12 18" x 36" plexi glass boards and motes placed on them to create 12 2 x 4 meshes. This way, we can arrange these boards to generate different type of meshes according to the requirements of the application. These 12 boards are easy to carry and can be placed in the lab. They can be hung on the wall as well. By creating the test bed in the lab, we simulated a larger deployed network of motes in the lab.

2.3 Selecting Gateway

Once the board size was determined, the next task was to decide on the gateway to be used on each board. We have some Crossbow's Stargate, which have WiFi. JVM is available for this platform which makes it easier to write TCP client/server application. Unfortunately, Crossbow stopped manufacturing Stargates and replaced them with a newer version, called Stargate Netbridge. It does not have Wifi; instead it has an Ethernet port. We could not find any JVM for this platform; however we could install C/C++ on it, thus we had to write TCP client/server application in C. By considering all these, we decided to use Crossbow's Stargate Netbridge as a gateway on each board. We will use term *gateway* or *node* for Stargate Netbridge throughout the document.

We also have HP some IPAQs which we are trying to make USB hosts. We are also studying Java on IPAQs so that in future, we can use them as gateways.

2.4 Creation of Test-bed

This section describes how these components are integrated together to form the test bed. The steps are as follows.

1. First we cut four Velcro of board size and stick them on the board shown in *Figure 2.10*.
2. We then take 8 motes and stick Velcro on them as shown in *Figure 2.9*.

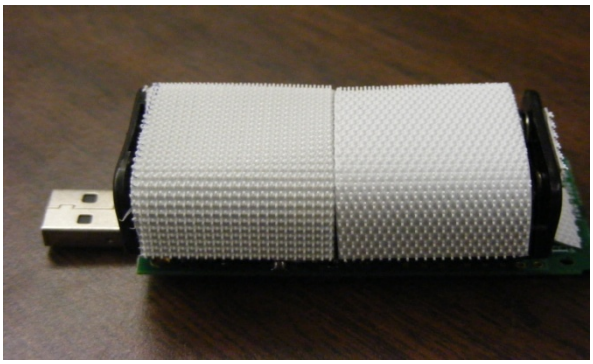


Figure 2.9 A mote with Velcro

3. Stick the motes on the board as shown in *Figure 2.10*.
4. Next, we take gateway and USB hub and stick them on the board using Velcro as shown in *Figure 2.10*.
5. Finally, we attach motes to USB hub using USB cables, and connect USB hub to gateway using USB cable.
6. Ethernet port of the gateway can be connected to the router using an Ethernet cable, if required. By using the router and a set of C programs running on gateway, we can create any topology for the gateways, either tree or ring or any other according to the experiment to be performed. In this thesis, we create a tree of gateways. See *Section 4.1* for details
7. Each of the gateways, USB hubs and router requires separate power supply.
8. A completed board without Ethernet cable and power supplies is shown in *Figure 3.10*.

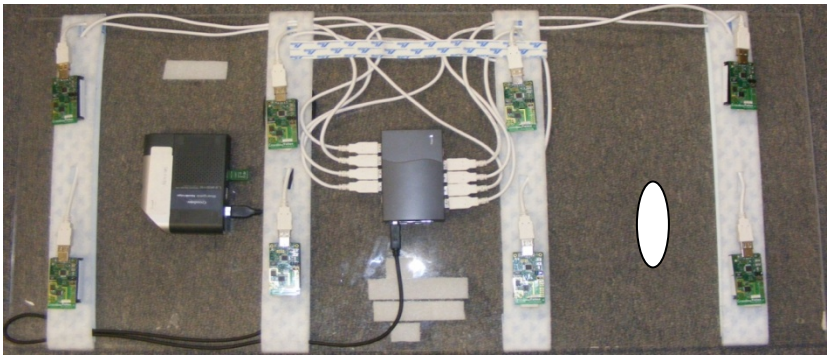


Figure 2.10 A complete Board

Distances of the motes can be adjusted according to needs. The four Velcro can be moved up and down, the motes can be stuck anywhere on Velcro we want, thus providing us flexibility to adjust their position in any direction.

Figure 2.11 shows an experimental test bed of 48 motes created using 6 boards. It forms a mesh of 4 x 12.

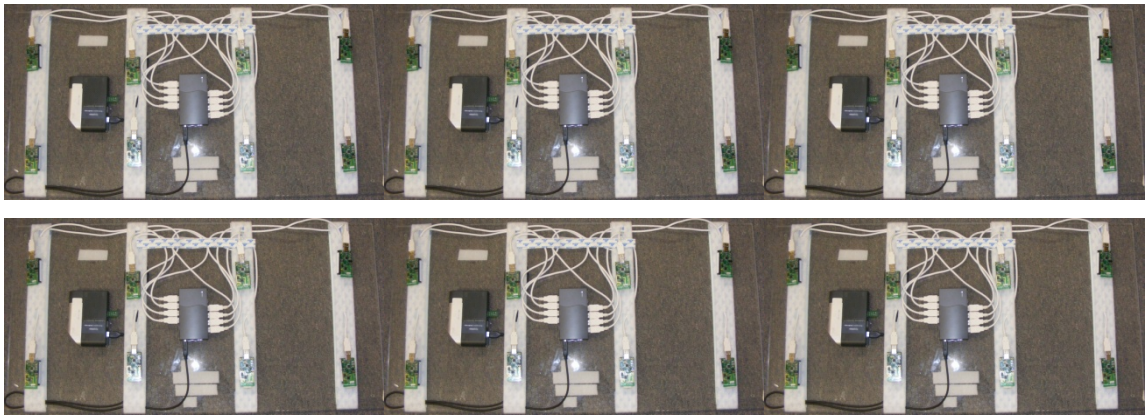


Figure 2.11 4 x 12 Mesh

2.5 Other Devices

Other devices, like Zigbee relays, as shown in *Figure 2.12*, can be attached to the boards. It can be glued to the board near the USB hub (see a vertical oval at the right hand side panel of the board, *Figure 2.9*). This type of relay can be used as an actuator, if required in an application. In this thesis, we did not use any such device.

Figure 2.12 Zigbee Relay

CHAPTER 3 - Event Service Interfaces and Architecture for Sensor Networks

In this chapter, we define the interfaces which are used to implement event service for sensor networks. The names of the interfaces are same as used in standard CORBA event service specification. As defined in *Section 1.2*, an event channel is a middleware which mediates the transfer of events between suppliers and consumers. The event channel decouples the communication between suppliers and consumers. In this research, we implemented event channel to work with the push model. The behavior of the event channel in this case is described in *Figure 3.1* below. Some figures and some contents of this chapter are from [1] and [2].

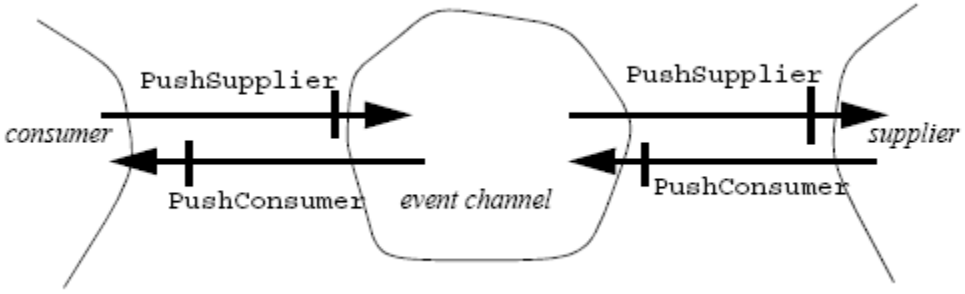


Figure 3.1 Push-style communication between a supplier and an event channel, and a consumer and an event channel.

The required module and interfaces for this model are described below.

```
Module EventChannel
{
        provides interfaces:
        push_consumer
        push_supplier
        command advertise(event id)
        command subscribe(event id)
}
interface push_consumer
{
```

```
        command Push(event_id, data)
        command disconnect
}
interface push_supplier
{
        command disconnect
}
```

3.1 The push_consumer interface

A push-style consumer supports the **push_consumer** interface to receive event data. A supplier communicates event data to the consumer by invoking the **push** operation and passing the event data as a parameter. The **disconnect** operation terminates the event communication; it releases resources used at the consumer to support the event communication.

3.2 The push_supplier interface

The **disconnect** operation terminates the event communication; it releases resources used at the supplier to support the event communication.

As an example, consider mote 1 which is interested in producing an event *temperature* and another mote 2, which is interested in receiving *temperature* data. The steps to establish this event communication are as follows:

1. Mote 2 executes *subscribe(temperature).*
2. Mote 1 executes *advertize(temperature).*

It is also possible that mote 1 executes *advertize(temperature)* before mote 2 executes *subscribe(temperature).* As soon as mote 1 executes *advertize(temperature),* it starts *push*ing temperature data to the event channel. Event channel then forwards the data to the destination mote, in this case mote 2.

Following sections discuss hardware and software architecture of the system.

3.3 System Architecture: Hardware Setup

In this section, we will describe how the test bed is configured to implement and deploy event service for sensor networks. We do not use all of the 12 boards; instead we took some of them and arrange them in a desired configuration. It doesn't matter if they form mesh or not. We

consider each board as a separate region and motes on the boards as forming a tree consisting of 7 motes, with the base station as the root. The root has two children and these two children again have two children each, thus summing 7 motes in the tree. We do not use the eight motes. A typical system hardware setup using 3 boards is shown in *Figure 3.2* below. The naming convention in the figure is as follows:

PC: Personal Computer (Desktop / Laptop)

SN: Stargate Netbridge (gateway)

BS: Base Station (it is a telosB mote)

○ : Other Motes (telosB)

The connections between motes and USB hub and USB hub to gateway are not shown as it will make the figure messy.

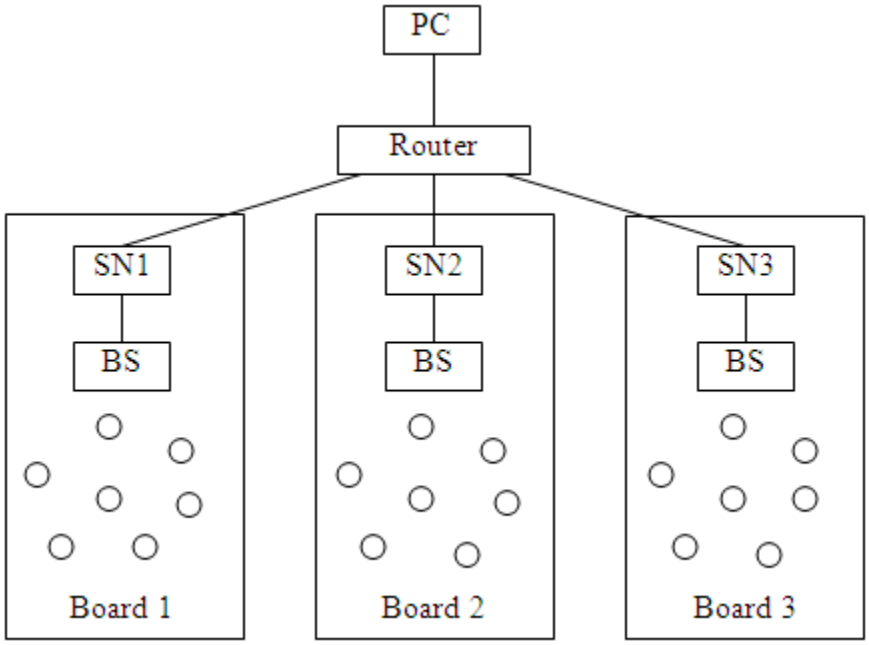


Figure 3.2 Typical System setup

Further, a tree of gateways is also formed. For example, with these three boards, we can form a tree with SN1 as root and SN2 and SN3 as its children. Or, we can form a tree with SN1 as root, SN2 as its child and SN3 is child of SN2. We form the tree of gateways using TCP connections.

Since all gateways are connected to the router, they can also communicate directly with each other. But in order to test our event service, we don't want them to communicate directly; instead, we want them to communicate as specified by the child-parent relationship.

Chapter 4 discusses an example in detail with how consumers and suppliers use the event service interfaces and calls the commands provided by them to establish event communication.

3.4 System Architecture: Software Setup

The software is written in C and nesC programming languages. The event service interfaces are implemented in nesC and reside on motes. Motes communicate with each other wirelessly via radio (using generic comm layer). The data routing functionality is written in C and is done via gateways. Gateways communicate with each other via TCP channels. Motes communicate with each other via radio links. The base station can communicate with the gateway via the UART interface. The UART interface can read data from or write data to USB using already existing software component called serial forwarder. It is written in C. All the data structures, which are discussed in Chapter 4, reside on gateway. Interactions of the various software components are shown in *Figure 3.3* below.

Apart from the programs for gateways and motes, a separate set of programs was written. This includes a program written in C and a program written in nesC, installed on a mote, which is different from the motes on the board. We call this mote as the *instructor*. This program gets the user input. When the user gives an input to the C program, it sends the input string to the mote. The mote then interprets the command and does the appropriate actions. Some of the actions are as follows:

1. cmd\> -3-10-temp- When the instructor mote receives this string, it tells mote 3 on the board to execute an advertise call for the event temp.
2. cmd\> -4-11-temp- When the instructor mote receives this string, it tells mote 4 on the board to execute a subscribe call for the temp event.
3. cmd\> -3-13-0300- When the instructor mote receives this string, it tells mote 3 on the board to set the push frequency to 300 milliseconds.
4. cmd\> -3-12-temp- When the instructor mote receives this string, it tells mote 3 on the board to execute the push call for the temp event. This call is executed 100 times, at the frequency set by the previous command.

However, once a mote executes an advertize call, it starts pushing the specified events. But for testing purposes, we have delayed the pushing of events until the supplier is instructed to do so by instructor.

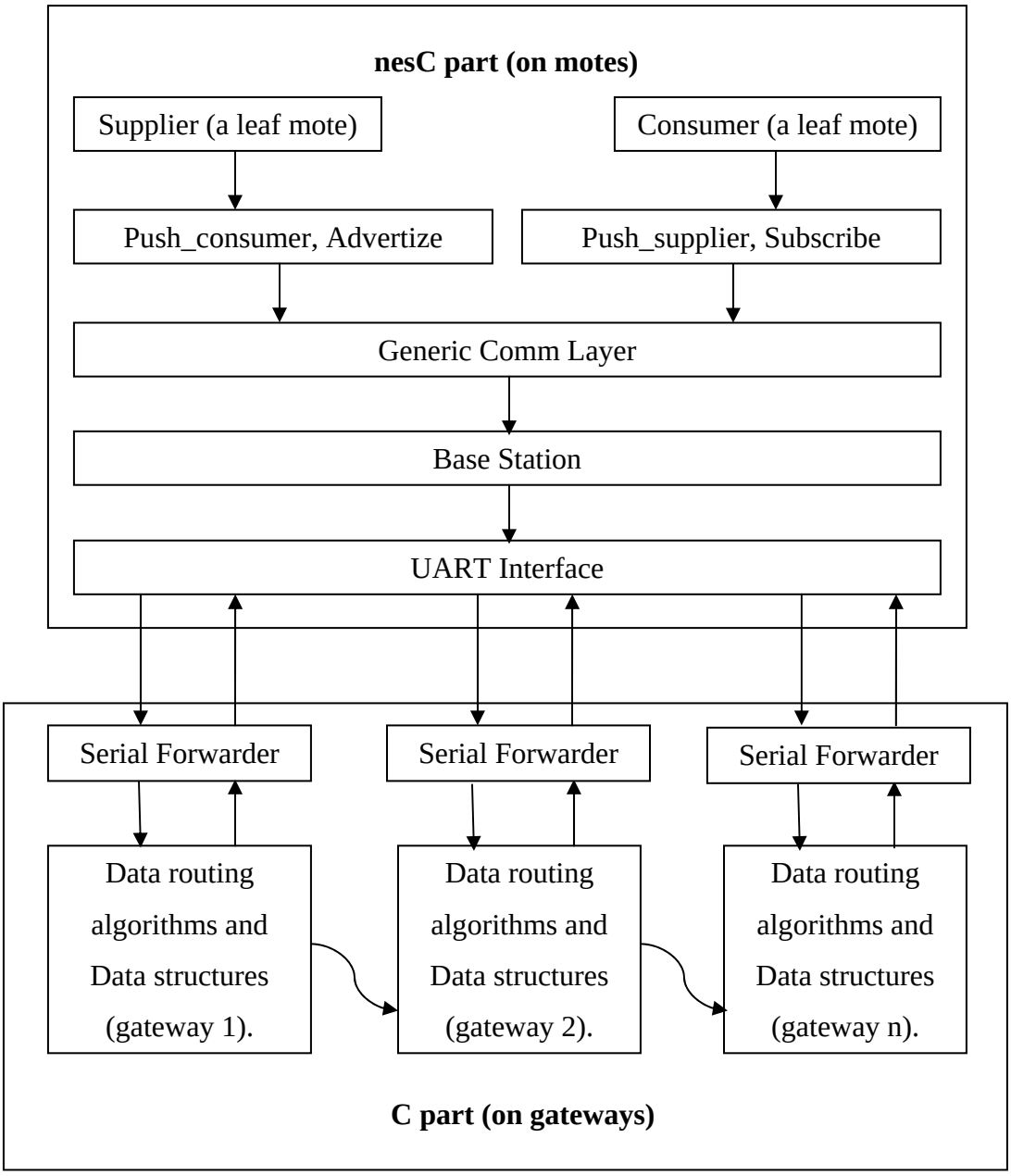


Figure 3.3 Interaction of various software components

CHAPTER 4 - An Example Illustrated with Data Structures

This chapter gives an overview of how the event service works. It highlights the steps to form a tree topology of gateways followed by the system operations using an example. This chapter concludes by giving an overview of the data structures used on the gateways.

Consider the 4 gateways (SN) and the tree structure we want them to form as shown in *Figure 4.1*.

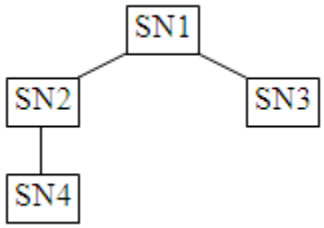


Figure 4.1 Example Topology

4.1 Steps to form a given topology

Following are the steps to be followed to create the given tree structure:

1. The PC opens a TCP socket (port 8000) and listens on that port. It also opens a file containing the tree structure.
2. When an SN gets connected to a router, it will execute a script to connect to the PC on port 8000. PC remembers its IP address.
3. As soon as all of the SNs are connected to the PC, it will instruct each one of them to open a TCP connection so that the given tree structure can be formed.

4.2 System Operation

Once the tree structure is formed, the actual operation of the system takes place. Let, mote 1 of SN4 be the supplier of the event temp and mote 1 of SN3 be the consumer of the event. Two system calls are used by the suppliers and consumers to indicate their role in the system.

The suppliers call *advertise(event type)* to indicate that they can produce event data. Consumers may call *subscribe(event type)* to indicate that they are consumers of the data.

Whenever a supplier produces the data, it has to be transferred to the consumer. When a supplier produces some event data, it makes a system call Push(event type, data).

So, in this case, when mote 1 of SN4 calls *advertise*, an entry is made in the *Advertise* data structure in SN. This data structure will store the mote id and the event type; in this case, it will look like (1, temp).

When mote 1 of SN3 makes the *subscribe* system call, the following sequence of actions takes place:

1. An entry is made in the *Subscriber* data structure in SN3. The entry will have mote id and event type, in this case (1, temp).
2. SN3 will then propagate the request further indicating that SN3-IP needs event temp. We maintain another data structure, called *Event_Request*. It has IP address and event type as fields. In this case, on SN1, it will look like (SN3-IP, temp). After SN1 updates this data structure, it further propagates the request in the network. In this case, it will only transfer it to SN2. SN2 will make an entry in the *Event_Request* data structure as (SN1-IP, temp). SN2 then propagates it further to SN4 and SN4 will have an entry (SN2-IP, temp).

When mote 1 of SN4 makes the *push* system call, the event data goes to SN4. On SN4, the following actions happen:

1. SN4 then checks its *Subscribe* data structure to check if there is any consumer in its local network. If yes, then it transfers the event data and proceeds to step 2.
2. It then checks the *Event_Request* data structure to check if some other SN has requested temp. In this case, SN2-IP has requested temp; so it forwards the data to SN2.
3. SN2 on receiving this data performs step 1.

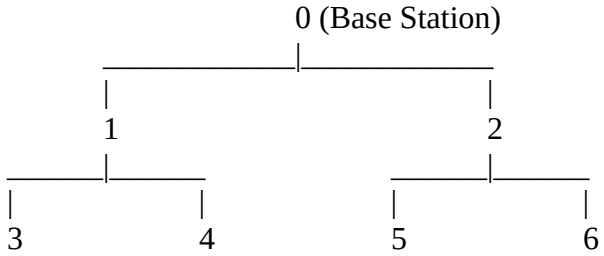
4.3 Data Structures

We infer from the above discussion that the following data structures are needed to implement the model.

1. Advertize (mote_id, event_type)
2. Subscribe (mote_id, event_type)
3. Event_request (SN-IP, event_type)

CHAPTER 5 - Basic Algorithms

This chapter discusses various algorithms which are implemented in either C or nesC to develop the entire system. We assume that following tree of motes on the board:



We also assume that only leaf nodes can act as suppliers or consumers. The algorithms are presented one by one in the following sub sections with the above assumptions in mind.

5.1 Algorithm to create tree of gateways

Input: A topology file.

Output: A tree of gateways is constructed. Gateways then wait for further actions.

Steps:

1. PC reads topology file. The topology file has number of nodes, say n, in the first line. PC waits for n connections.
2. Assign IDs to various nodes which request connection to PC in sequence.
3. Once all connections are established, the PC instructs each node, starting from the root, about their child-parent relationship. This relationship is read from the topology file. Once the message is sent to a node, it waits for response from that node.
4. Each node, after receiving this relationship information, waits for the children (if any) to connect. Once all children are connected, a message is sent to PC telling that the node has established all required connections. PC after receiving this message tells the next node to establish the connections.
5. Once the tree is formed, all nodes wait for data from either the base station or another node, either parent or child.

Pseudo Code:

Server (Running on PC):

read num_of_nodes;

for i ← 1 to num_of_nodes

accept connection from a node;

store socket descriptor in socket_fd[i];

for i ← 1 to num_of_nodes

send child-parent relationship to socket_fd[i];

wait for response from node i;

Client (running on nodes):

connect to server;

wait for child-parent relationship from server;

if (node has no parent)

wait for connections from children;

send acknowledgement to server;

else (node has a parent and has children)

connect to parent;

wait for connections from children;

send acknowledgement to server;

else (node has parent and no children)

connect to parent;

send acknowledgement to server;

5.2 Algorithm for advertize call

Input: A message from instructor to a leaf mote.

Output: An entry is made in the *Advertize* data structure on the gateway.

Steps:

1. On receiving the instruction to execute the *advertise* call, a leaf mote keeps on sending a request to its parent (intermediate motes, *i. e.* 1 or 2) until it receives an acknowledgement from its parent.
2. Once a request is received by an intermediate mote, it keeps on forwarding the request to the base station (mote 0) until it receives an acknowledgement from the base station.
3. When the base station receives the advertize request, it sends the request to the gateway. The gateway, on receiving the request, updates its *Advertize* data structure.

Pseudo Code:

```
do
if (instructor mote)
        wait for user input;
        construct advertize(event) command;
        leaf mote ! advertize(event);
fi
if (leaf mote)
        ack ← 0;
        do
                instructor mote ? advertize(event);
                parent ! advertize(event);
                parent ? ack → ack ← 1;
                if (ack = 1) → break;
        od
fi
if (intermediate mote)
        do
                ack ← 0;
                child ? advertize(event);
                base station ! advertize(event);
                base station ? ack → ack ← 1;
                if (ack = 1) -> break;
        do
fi
if (base station)
        do
                child ? advertize(event);
                client on gateway ! advertize(event);
        do
if (client on gateway)
```

do

base station ? advertize(event);

update advertize data structure;

do

od

5.3 Algorithm for subscribe call

This is a split call. Half of the part is implemented in nesC on motes. The other half is implemented on gateways in C.

5.3.1 On motes (in nesC)

Input: A message from the instructor mote to a leaf mote.

Output: An entry is made in the *Subscribe* data structure on the gateway. The *Event_Request* data structure on all other gateways is also updated.

Steps: The algorithm is same as the algorithm described in *Section 5.2*.

1. On receiving the instruction to execute the *advertise* call, a leaf mote keeps on sending request to its parent (intermediate motes, *i. e.* 1 or 2) until it receives an acknowledgement from the parent.
2. Once the request is recdeived by an intermediate mote, it keeps on forwarding the request to the base station (mote 0) until it receives an acknowledgement from the base station.
3. When the base station receives the *subscribe* request, it sends the request to the gateway. The gateway, on receiving the request, updates the *Subscribe* data structure. The gateway then propagates the information to other gateways in the network.

Pseudo Code:

The pseudo code is exactly same as the pseudo code of algorithm in *Section 5.2*.

5.3.2 On gateways (in C)

Input: A message from node.

Output: The *Event_Request* data structure of parent and children gateways is updated.

Steps: The algorithm is the same as the algorithm described in *Section 5.2*.

1. Once a gateway updates its *Subscribe* data structure (Step 3 of *Algorithm 5.3.1*), it does the following. Assume that the *subscribe* call has just caused the temp event to be

inserted in the *Subscribe* data structure. If the gateway has already propagated the information for the temp event in the network, then nothing has to be done; otherwise it then sends the event information to its parent and all children.

2. On receiving the event information, other gateways updates the *Event_Request* data structure, and propagate this information further in the network. Thus, this information is propagated all over network.

Pseudo Code:

base station ? subscribe(event);

search subscribe data structure for event;

Start: *if (an entry of event is found in subscribe data structure)*

do nothing;

else

if (has parent and request not received from parent)

parent ? subscribe(event);

for i ← 1 to num_children

request not received from child[i] → child[i] ! subscribe (event);

if (has parent)

parent ? subscribe(event);

update event_request data structure;

for i ← 1 to num_children

child[i] ? subscribe (event);

update event_request data structure;

go to start;

5.4 Algorithm for the push call

This call is split into three (or TWO) steps. Half of the part is implemented in nesC on motes. The other half is implemented on the gateways in C.

5.4.1 On motes (in nesC)

Input: A message from a leaf.

Output: An event data is send to the gateway.

Steps:

1. A leaf node sends event data to the base station via intermediate motes. We do not ensure reliability, so we do not send acknowledgements as we did for *advertize* and *subscribe* calls.
2. On receiving the event data, the base station sends data to the gateway via UART interface.
3. Send event data to base station if *Subscribe* data structure has any entry for that particular event. Then execute *Algorithm 5.4.2*.
4. On receiving event data from gateway, the base station sends the data to the consumer via intermediate motes. Again, this operation need not be reliable.

Pseudo Code:

```
do
    if(leaf mote)
        parent ! event data;
        parent ? event data → process data;
    fi
    if(intermediate mote)
        child ? event data;
        base station ! event data
        base staation ? event data → forward it to destination mote;
    fi
    if(base station)
        client on gateway ! event data;
        client on gateway ? event data →
            forward it to intermediate mote;
    fi
    if(client on gateway)
        base station ? event data →
            if(subscriber data structure has entry for event)
                base station ! event data;
            fi
        execute algorithm 5.4.2;
```

other gateways ? event data →

if(subscriber data structure has entry for event)

base station ! event data;

fi

fi

od

5.4.2 On gateways (in C)

Input: An event data from gateway.

Output: An event data is send to gateway or base station.

Steps:

1. Send event data to all gateways for which an entry is found in *Event_Request* data structure.
2. Execute step 3 of *Algorithm 5.4.1*.

Pseudo Code:

for all gateways entries found in event_request data structure for event

gateway ! event data;

CHAPTER 6 - Performance Analysis

We tested our approach in number of ways. The simplest test is to assume that all suppliers and consumers are on the same board. In this case, actually there will no communication among gateways. We further divide this case into various sub cases, for example, one supplier and one consumer, one supplier and two consumers and so on. We tested all of these cases with various frequencies for generating the events. We then tested with multiple boards. In this case, there will be communication among gateways. *Section 6.1* contains results of one board and *Section 6.2* contains results of multiple boards.

6.1 Test with one board

We consider four different cases.

1. One supplier and one consumer of an event.
2. One supplier and two consumers of an event.
3. One supplier and three consumers of an event.
4. One supplier and one consumer of two different events.

Suppliers push 300 events in each case. We test each case with different frequencies of pushing events. For each case, we took seven readings, discarded two extreme readings and took the average of the remaining five readings. We then studied the average number of events received by the consumers. The following table contains the results.

In *Table 6-1*, *1 sup (temp)* means that there is 1 supplier of event temperature. *1 cons (pres)* means that there is one consumer of event pressure.

Number	Freq (ms)	1 sup (temp) 1 cons (temp)	1 sup (temp) 2 cons (temp)	1 sup (temp) 3 cons (temp)	1 sup (temp), 1 sup (pres) 1 cons (temp), 1 cons (pres)
1	1	6	3	3	3
2	5	24	18	18	21
3	10	51	51	48	48
4	25	102	99	93	102
5	50	126	120	117	123
6	75	195	138	123	132
7	100	216	132	126	129
8	150	231	228	195	201
9	200	231	231	222	225
10	250	234	234	222	222

11	500	231	234	234	231
12	750	234	231	228	228
13	1000	231	228	231	228

Table 6-1 Frequency Vs. Events received on a single board

Figure 6.1 shows the graph of the data from *Table 6-1*.

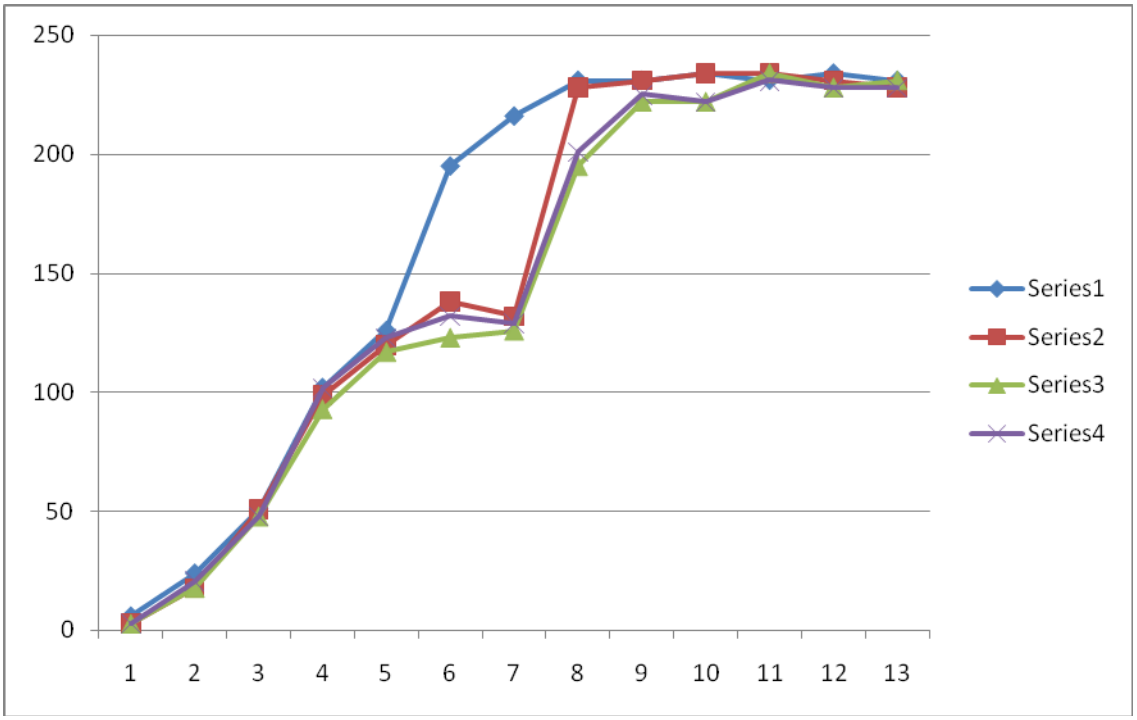


Figure 6.1 Frequency Vs Events received

We observe that after reading number 9 (corresponding to frequency of 200 ms), the number of events received tend to be constant (between 220-235). These results have been obtained when advertize and subscribe data structures are stored on gateway. But, on a single board, it is useless to store them on the gateway. Each time an event is pushed, the request has to go through the gateway via base station. The gateway then sends the data back to the base station if it finds any subscriber entry in subscriber data structure. If we store these data structures on the base station instead, then when single board is used, the cost of sending data to and from gateway to base station is saved. In this case, total number of events received is much higher; it is close to 275.

One more interesting case we observed on a single board. If we set up the system such that mote x is supplier of event e_1 and mote y is consumer of that event; mote y is supplier of event e_2 and mote x is consumer of that event, then the results change drastically. We found total

number of events received reduced to 165-180 for all frequencies above 200 ms. It is because there is more interference.

6.2 Test with three boards

We configured the system in many different ways using three boards and tested the event service on those configurations. Some of the configurations and corresponding results are given below. A figure is drawn corresponding to each configuration. A *P* next to a leaf mote indicates that it is a supplier and a *C* next to a leaf mote indicates that it is a consumer.

6.2.1 Configuration 1

1 Supplier on SN2

N Remote consumers on SN3, where N = 1, 2, 3, 4

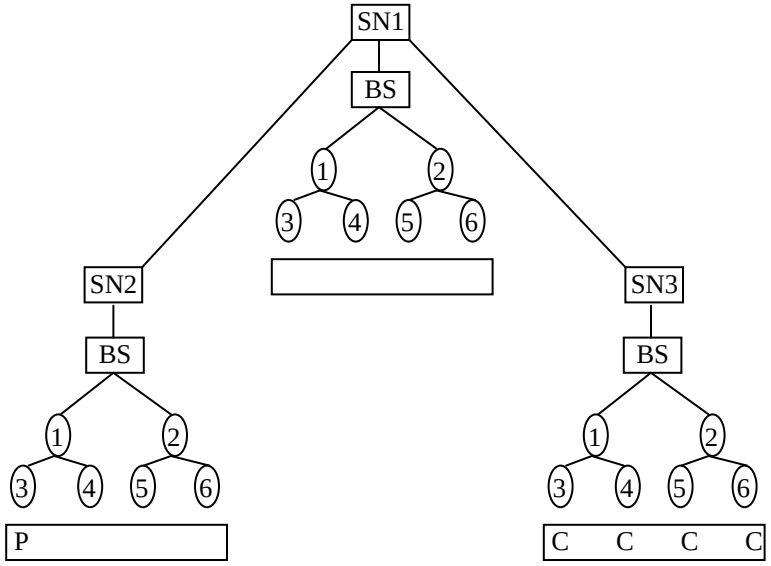


Figure 6.2 Multiple boards test: configuration 1

There are four possible experiments for this configuration for four different values of N. Supplier pushes 100 events in each case. We test each case with different frequencies of pushing events. For each case, we took seven readings, discarded two extreme readings and took average of remaining five readings. We then studied average number of events received by consumers. *Table 6-2* contains the results. *Figure 6.2* shows corresponding graph.

Number	Freq (ms)	N = 1	N = 2	N = 3	N = 4
1	1	4	3	3	2
2	5	12	12	11	10
3	10	30	28	28	25
4	25	70	66	66	60
5	50	71	67	66	64
6	75	74	74	73	73
7	100	75	76	75	75
8	150	76	77	76	76
9	200	78	78	77	74
10	250	76	77	75	78
11	500	79	77	78	76
12	750	78	76	77	78
13	1000	79	77	76	77

Table 6-2 Results for Configuration 1

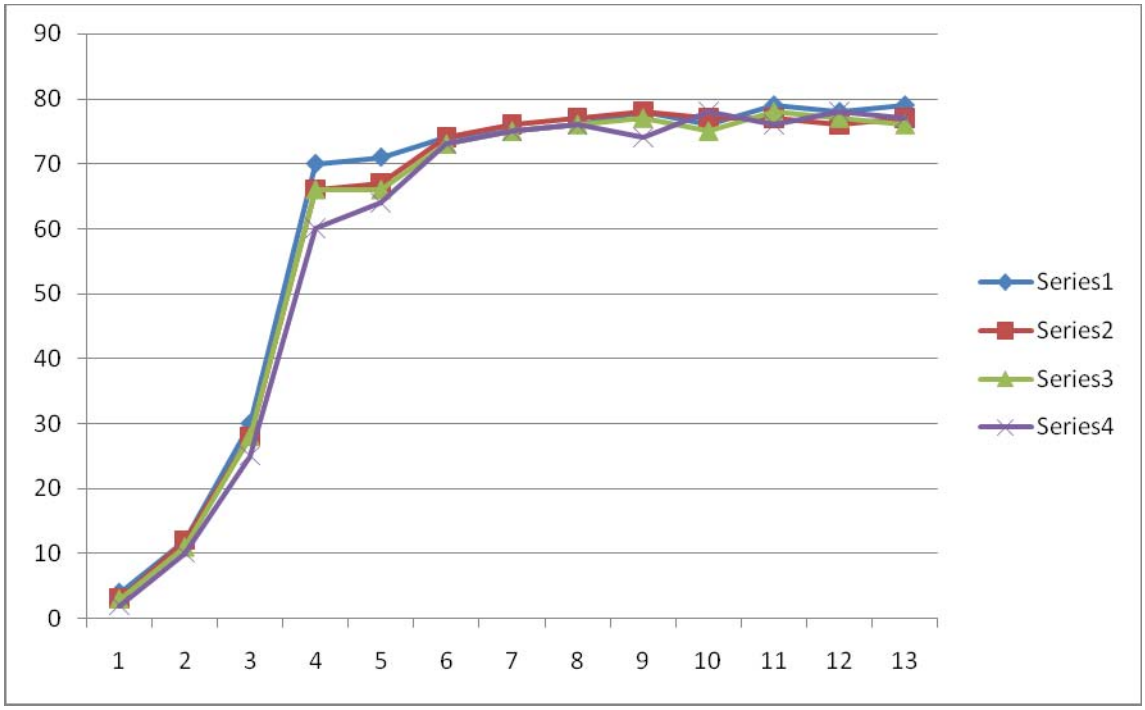


Figure 6.3 Frequency Vs Events received for configuration 1

We observe that after reading number 7 (at frequency of 100 ms or more); total number of events received tends to be constant, *i. e.* between 75-79. Note that, this frequency was 200 ms while all the consumers were on the same board. If remote consumers are receiving an average of 75 events with a frequency of 100 ms, local consumers should definitely receive 75 or more events with the same frequency. We are still trying to figure out what causing this difference.

6.2.2 Configuration 2

1 Supplier on SN2

N Remote consumers on SN3, where N = 1, 2, 3, 4

1 local consumer on SN2

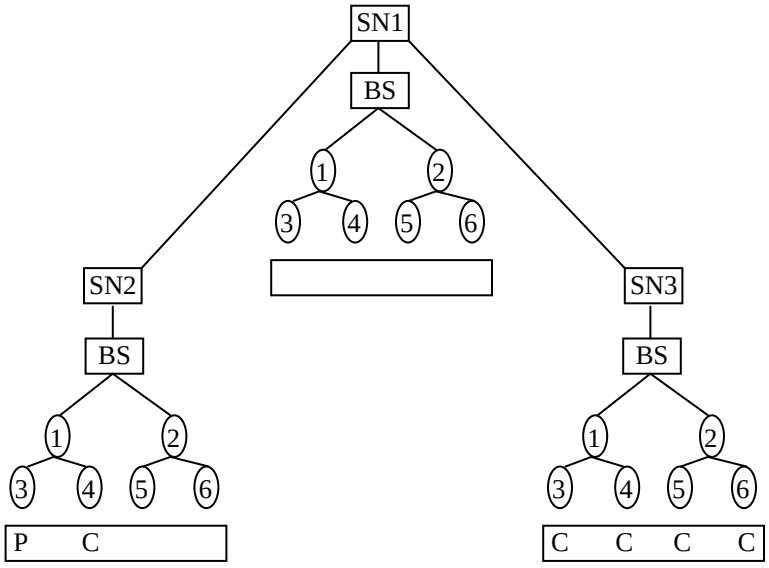


Figure 6.4 Multiple boards test: configuration 2

There are four possible experiments for this configuration for four different values of N. Supplier pushes 100 events in each case. We test each case with different frequencies of pushing events. For each case, we took seven readings, discarded two extreme readings and took average of remaining five readings. We then studied average number of events received by consumers. *Table 6-3* contains the results.

Number	Freq (ms)	Local Consumer	N=1/2/3/4
1	1	1	1
2	5	3	2
3	10	3	8
4	25	6	20
5	50	25	33
6	75	27	36
7	100	30	37
8	150	32	36
9	200	33	37
10	250	32	36
11	500	31	38
12	750	32	37

13	1000	33	38

Table 6-3 Results for Configuration 2

It is observable that by introducing just a single local consumer, the performance degrades drastically. Results for N = 1, 2, 3, 4 are almost same. We need to look into this case more deeply as to why it is behaving this way. We are still trying to figure out the reasons behind this behavior.

CHAPTER 7 - Conclusion and Future work

7.1 Conclusion

We draw the following conclusions.

1. If all the consumers are on the same board, then the implementation works fine for frequencies greater than or equal to 200 ms. In this case, consumers receive an average of 75%-80% of events.
2. If consumers are on remote board and there is no local consumer, then consumers receive an average of 75%-80% of events for all frequencies greater than or equal to 100 ms.
3. If we introduce a local consumer, then the performance drops down to 35%-40%.

7.2 Future work

There are few areas which we are going to implement in the future. These are

1. **Failure Management:** While developing and testing this model, we assumed that neither the base station nor the gateway will ever fail. In case, if the base station fails, then we need to choose a new base station. Gateways open a serial forwarder for each base station on a particular port. At present, this port is hardcoded in the program. This needs to be taken care of if we select a new base station.

   If a gateway fails, we need to reconstruct the tree of gateways. We need to merge the existing tree of motes of the failed gateway to some other gateway.
2. **Algorithm Optimization:** There are lots of places where we can optimize our algorithms. One such optimization is discussed in *Section 6.1*.

   We observe from *Section 6.2.2* that by introducing a local consumer, the overall performance degrades. We need to optimize the program code which processes a *push* call.
3. **Movable Motes:** At present, we have studied the event service for static motes only. We will test this event service for movable motes also. For example, a robot with a mote on top of it is moving around, which is subscriber for the temperature event. Then, we will study how our event service should be modified to handle this case.
4. **Complex Events:** We can imagine a set of complex events. For example, a mote is interested in an event, for example, temperature > 50 and temperature < 60 and humidity

> 40%. Then how various data structures should be modified to accommodate these needs.

5. **Tree structure of motes and wireless routing:** In the present work, we have assumed a fixed tree of seven motes. Wireless routing is also fixed. For example, if leaf mote wants to send some data to base station, it will send data to its parent, *i. e.* mote 1 and mote 1 will forward the data to base station. Future modifications to this static structure will be to have variable tree of motes and wireless data routing through *TinyOS Multihop* component.

References

[1] OMG event service specification manual.

http://www.omg.org/docs/formal/04-10-02.pdf

[2] OrbixEvents Programmer's Guide.

www.iona.com/support/docs/orbix/gen3/33/html/orbixevents33_pguide/intro.html

[3] TinyOS 1.x tutorial.

http://www.tinyos.net/tinyos-1.x/doc/tutorial/

[4] Beej's guide to network programming.

http://beej.us/guide/bgnet/

[5] Crossbow's Stargate Netbridge User manual.

www.xbow.com/Support/Support_pdf_files/Stargate_NetBridge_Users_Manual.pdf