An Architecture for On-Demand Wireless Sensor Networks

by

M.S.R.Fernando

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved February 2012 by the
Graduate Supervisory Committee:

Partha Dasgupta, Co-Chair
Amiya Bhattacharya, Co-Chair
Sandeep Gupta

ARIZONA STATE UNIVERSITY

May 2013

ABSTRACT

Majority of the Sensor networks consist of low-cost autonomously powered devices, and are used to collect data in physical world. Today's sensor network deployments are mostly application specific & owned by a particular entity. Because of this application specific nature & the ownership boundaries, this modus operandi hinders large scale sensing & overall network operational capacity.

The main goal of this research work is to create a mechanism to dynamically form personal area networks based on mote class devices spanning ownership boundaries. When coupled with an overlay based control system, this architecture can be conveniently used by a remote client to dynamically create sensor networks (personal area network based) even when the client does not own a network. The nodes here are "borrowed" from existing host networks & the application related to the newly formed network will co-exist with the native applications thanks to concurrency. The result allows users to embed a single collection tree onto spatially distant networks as if they were within communication range. This implementation consists of core operating system & various other external components that support injection maintenance & dissolution sensor network applications at client's request. A large object data dissemination protocol was designed for reliable application injection.

The ability of this system to remotely reconfigure a network is useful given the high failure rate of real-world sensor network deployments. Collaborative sensing, various physical phenomenon monitoring also be considered as applications of this architecture.

DEDICATION

To my parents, family, and friends.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help of many in various ways. It is with great pleasure I acknowledge the assistance of my advisors, Dr. Partha Dasgupta and Dr. Amiya Bhattacharya in the pursuit of this work. I also like to thank Dr. Sandeep Gupta for taking a special interest in this research work and for the support shown during my masters program. It was a great pleasure to work with Yuan Wang, Michael Cartwright and Vaibhav Jain in the operating systems lab. I thank National Science Foundation (NSF) for providing financial support for this research work. Also I would like to express my gratitude to everyone at Arizona State University for their support.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

According to Wikipedia wireless sensor network is a "spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature sound pressure, etc. and to cooperatively pass their data through the network to a main location" [1].

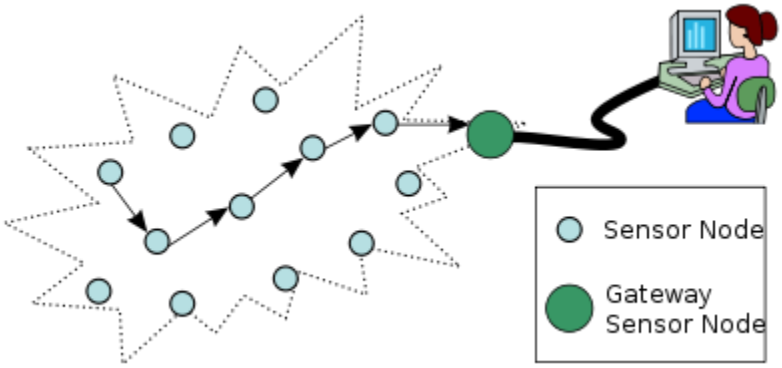

Figure 1 Typical sensor network

Even though there can be different types of sensor nodes, most of the time by sensor networks, we refer to a collection of low-power mote class devices. These types of networks are becoming popular in day to day life. Their usage varies from simple Body area networks to complex smart grid & energy control systems. Due to their small form-factor and the self-organizing nature they can be easily deployed in places that are inaccessible to humans and disaster relief operations. Currently most of the deployments are owned by individuals and are application specific. Even though sensor networks are capable of doing various tasks, because of the current usage pattern they are restricted to a predetermined set of functions. The popularity of the sensor networks is deteriorating because of these drawbacks. There can be a situation where a perfectly working sensor network has to be abandoned just because it does not support a particular functionality. In real life deployments it is impossible to predict the future demands and tailor the application according to that.

In this thesis we introduce an architecture that allows users to create virtual sensor networks using physical host sensor networks. In a high level overview we have created a software system that allows users to plug applications on demand to an already deployed physical sensor network. These injected applications at each node constitute a virtual sensor network. This virtual sensor network will be running an application without disturbing the application running on the host network. This solution mitigates above discussed issues by providing various services such as network reconfiguration, re-imaging of existing operating systems on nodes, fixing bugs etc.

The next chapter (chapter 2) describes the motivation behind this work. Related research work is described in chapter 3. Chapter 4 is about the core components of the support system (middleware). Application code dissemination is explained in chapter 5. Chapter 6 describes the changes I made to a widely used sensor network data collection protocol during the process of porting the said protocol to our system. Chapter 7 is a write-up about the supporting application which runs on a relatively powerful system connected to the sensor network. Chapter 8 describes the experimentation results & finally chapter 9 is about the future work (possible extensions).

CHAPTER 2

MOTIVATION

Most of the current sensor networks are application specific. The code that runs on the devices is optimized for processing platforms, sensors & application requirements. This is mostly due to the low power nature of the devices. Unlike general purpose computers, these devices do not have the capability to dynamically change the running application. The more application specific the code is, the more precious energy the device will be able to save. These devices are expected to run for approximately a year with just two AA batteries. This is because some deployments happen in places where human interaction with the devices, is not possible (For ex. monitoring a volcano). Without such human intervention, network maintenance, such as changing the power supplies, removing dead nodes from the network etc. becomes an impossible. To deal with these types of situations the applications running on the nodes must be robust & energy efficient. No matter how robust the developers make the applications, in real world there are always deployment issues with sensor networks [40]. The biggest issue is the high probability of the entire network becoming in-operable just because of the failure of few nodes in the network. This is due to the application specific nature of sensor networks. In this type of a failure scenario there is no fallback mechanism in order to get the network operational. The physical mote is considered in-operable just because of the application running on top of it is not working. There is no approach to remotely troubleshoot & maintain a wireless sensor network.

Even with the above issues, sensor networks are increasingly becoming popular. Currently in households there are various sensor networks used by security systems, garden monitoring systems etc. These systems are getting integrated in order to reduce the user's burden of having to deal with multiple systems. Even though the

"mote-class" low power devices are weak in processing power, if taken collectively they have a considerable amount of processing capabilities. With the introduction of smart phones & tablets, the capabilities of the devices are improving. Generally smart phones come with a lot of sensors integrated to them. Because of that one can easily create a network of sensors using smart phones. Smart phones have also been used as gateways/data sinks to mote based sensor networks.

The main issues that prevent the "mote-class" devices becoming even popular are their low-power nature and the inability to remotely configure/re-program after their deployment. These devices are constantly improving, with the introduction of powerful processors, long lasting batteries etc. There have been a lot of research to solve the latter issue. These methods vary from parameter changing to re-programming the entire application image. Parameter changing methods give basic control over a deployed sensor network. On the other hand image reprogramming gives a lot of control over the network but is an expensive operation. The above methods are discussed in-detail in the related work chapter.

The issues with the current technology motivated us to come up with an architecture that allows users to dynamically change the application running on the motes without any physical interaction with the device, still within the boundaries of resource limitations. This architecture takes the current technology a step further, by introducing a mechanism that can virtualize (create slices) a sensor network. This allows us to give us to give users "total access" to a sensor network, when compared to a limited "data only" sharing model provided by the current state-of-the-art. Currently users can only interact with the data and make modifications to it (mostly at the gateway), and not to the application itself which is running on the motes. If there is any application control given it was only limited to parameter changing (such as frequency of sensing or transmitting data). The idea for this architecture came

from community based testbeds such as Motelab [42] and PlanetLab [41], where remote clients are given a web interface to program the devices and exclusive access to those devices during their experiments. Since it is done using a reservation based mechanism, the downside is no multiple users are allowed at the same time. Since we slice (virtualize) a network, unlike Motelab there can be multiple users running applications at the same time. This model gives users a lot of flexibility in terms of control. We run multiple applications simultaneously by using TinyOS thread library, TOSThreads. The idea is to disseminate an application to the sensor network using the gateway. Once the injected application starts to run, all the running instances of the injected application constitute a virtual sensor network/sensor network slice. The formed virtual network will run totally independently from the other networks. It will have its own network identification number & each node within it will have a new node identification number. Therefore a node participating in multiple virtual networks will have multiple identifications associated with each virtual network. One advantage of this network virtualization is isolation of network related issues. This is extremely useful in creating resilient networks. For example if the users figures out that there are issues related to a particular virtual network, he/she can create another network even with the same functionality, without having to worry about the failed network. The other important feature is its ability to create virtual sensor networks across ownership boundaries. Users can stitch neighboring and with the help of tunneling, (described in chapter six) even non-neighboring networks and create a single large virtual sensor network. This hides the underlying platform dependencies. The idea is to inject platform dependent application code from respective gateways and give remote user a virtual sensor network abstraction. This will be useful in community-based sensing. Users will be able to inject various sensing and/or phenomenon monitoring applications, and once their requirements

are fulfilled, they can smoothly dissolve the network, making the network resources available to other users. There are lots of such applications that can make use of this architecture because of its unique nature of the design.

CHAPTER 3

Related Work

Since the architecture described in this thesis offers various services, there is plenty of related research work involved. In this section we are going to discuss the background and the research work related to the main aspects of this system.

Mote class Devices

Most of the mote-class devices are low power embedded systems. Popular devices in the current market are Telos revision B motes & Mica class motes. A basic mote consists of a board embedded with a microcontroller, a radio & sensors. In some motes the set of sensors are on a separate board which can be attached to the main board when needed (Figure 3). Microcontrollers are used because of the device's low power nature. Important technical specifications of a Telos Rev. B[2] mote are given below.

Technical specifications of a Telos revision B mote

- 250kbps 2.4GHz IEEE 802.15.4 CC2420 Radio
- 8MHz Texas Instruments MSP430 microcontroller (10k RAM, 48k Flash)
- Integrated onboard antenna with 50m range indoors / 125m range outdoors

Figure 2 Telos revision B with sensors on board

Figure 3 Telos revision B with detachable sensors

A mote is typically powered by two AA size batteries and expected lifetime of the power source is approximately one year. These devices run embedded operation systems such as TinyOS [3], Mantis [4], Contiki [5].

Communication model

Since these devices are networked, a single device is most likely not able to fulfill any of client's requirements. Because of that, interaction between these devices is important. This interaction is done by wireless radio communication. Both Mica & Telos class devices use the same radio (Chipcon CC2420) for communication. It operates on 2.4GHz band & it is 802.15.4 compliant. 802.15.4 is a standard for media access control/ physical layer of the network stack. Most of the 802.15.4 network topologies are either star or peer-to-peer based. But in order to make sensor networks more useful there are various other topologies implemented (such as mesh). According to 802.15.4 specifications a node in the network is identified by network id and node id combination. This allows flawless operation of multiple networks that are within communication range.

TinyOS

Since the implementation of this thesis is done in TinyOS, this section will contain an overview of the embedded operating system. TinyOS is based on an event-driven model because of the low power nature of the devices it runs on. The diagram below (Figure 4) gives an overview of the TinyOS structure [25].

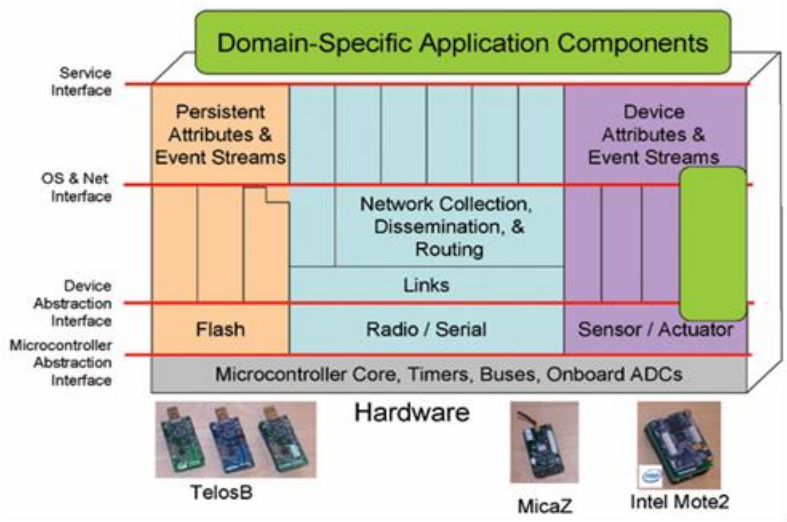

Figure 4 Components of TinyOS

The entire diagram can be considered a TinyOS image. When an image is made, the application, network, data related components are compiled together to form a monolithic operating system image. This is done to reduce the operating system image size because during compilation components that are only necessary for the application are added. This monolithic nature was slightly changed with the introduction of threads & dynamic loading of applications. Application level communication is done using Active Messages. Active messages, as the name suggest contain metadata about how to process the message at receipt.

The idea behind this architecture is to allow remote users to inject applications on to already established host networks. When a client wants to create a virtual sensor network he/she selects host sensor networks where the virtual network will span across. The client will be directly in contact with the selected sensor networks (host sensor networks). In our system a virtual network will be associated with a TinyOS application. This application will co-exist with the running application(s) on the host networks.

This architecture gives a lot of benefits to the sensor network users and improves the operational flexibility of the sensor networks. Following is a discussion about few real-world scenarios where this system is useful.

The ability to use a sensor network without even owning it is a major advantage of this system. This is only possible if the real owners allow a remote client to use their networks. There is a high chance that a particular person does not own a sensor network where he/she wants to visit or know certain information about. Using this architecture, users will be able to use sensor networks to get up-to-date sensed information. For an example if a fire breaks out in a residential area the fire department will be able to use the sensor networks belonging to the residents to track down the fire.

Most real world sensor networks deployments are done in places inaccessible for humans. (Such as volcanoes, wild fire etc.)[4]. Because of environment they get deployed & the error-prone nature of the wireless communication have made the failure rate of network deployments high. But this architecture gives a convenient way to recover from any failures. A user can at least inject applications to operational nodes & collect data. In the above example there are chances that during a fire some motes in a sensor network will be inoperable. There are instances where the entire network becomes useless because of the failure of few motes. When coupled with a protocol that can route data around these network holes, this architecture can bring an inoperable network to life. Same applies to widely used application specific sensor networks. With this architecture we can re-use the sensor networks by just reprogramming them.

Even though the current implementation does not support, this architecture can be used to monitor moving phenomenon (patchy rain, smoke etc.). This can be pretty useful when the phenomenon can move across a very large area.

Architecture related research work

The most outstanding feature of the architecture described in this thesis is the ability to create virtual sensor networks in an on-demand fashion. Lots of real world sensor network [28] deployments do not work as planned. Failure of few nodes in the network (network hole) [29] might make the entire network unusable. There are also instances where [30], some design changes are needed after the deployment, and the sensor network needs to be re-configured or re-programmed. Because of this growing need to re-configure/re-program deployed sensor networks remotely, there have been lot of research work in this area. In a high level overview this work can be categorized into two types. Entire suites that are somewhat similar to the architecture described in this thesis, and some independent tools that when combined can be used to achieve the same goal. Even though we have mentioned re-configuration and re-programming together, they are different in many ways. Re-programming typically means changing the entire application running on the node whereas re-configuration, is changing some values in the application itself. For example changing the frequency of sensing a physical phenomenon is considered re-configuring a network. The main tasks of any re-programming or re-configuration effort are to propagate the necessary information or the application code to the network & to load them on to the devices.

Deluge [11] is a software suite that allows users to write sensor network applications at the gateway & reprogram the entire network with that application. Application writing is easy as long as the users are familiar with the NesC language. Users need to include the Deluge component when writing the applications so that the compiler knows to include Deluge library to the image. In the mote deluge applications are put in specially crafted external flash. Reprogramming commands are sent separately using a python based interface from the gateway. When the

command is received at the mote, the resident bootloader (TOSBoot – separate bootloader for Deluge) copies the program to program flash & executes it. The data propagation mechanism is Trickle [31] based & somewhat receiver-oriented. Since "large object" propagation is an expensive operation, they have done multiple optimizations to reduce it. Because of its costly nature Deluge is rarely used in current sensor network deployments.

There are various network re-configuration protocols [12][32] that are widely used today. Re-configuration easier when compared to re-programming because it needs fewer amounts of data to be disseminated to the network and a basic shared memory model will be sufficient for loading the re-configuration parameters to the application. Since it is a cheaper operation it is also less flexible. For example a network hole formed because of few dead nodes cannot be removed by reconfiguring. Most of these implementations use Trickle based consistency control method to disseminate data to the network.

As described in a previous paragraph, there are several independent tools that can be collectively used to achieve the same goal. For example, there are embedded operating systems [33] [34] that natively support dynamic loading of modules (applications). But they lack one of the required components of remote programming, which is the dissemination mechanism. Most of these features such as dynamic loading [20] of modules have been introduced in TinyOS which is the basis for the work in this thesis. When dynamic loading is involved also there should be a mechanism to create dynamically loadable modules. In current TinyOS, TOSThreads, TinyLD and a dissemination mechanism (such as Typhoon, TRD) collectively can be considered as an energy efficient alternative to Deluge. Instead of using various tools or suites to have true network virtualization, a pseudo virtualization can be achieved by using a concept called active message identifications (AMID) that has been

introduced later in TinyOS. Packet filtering based on AMIDs can create separate networks [35].

The above paragraphs discussed systems that are operationally somewhat similar to the architecture described in this thesis. Also there have been work related the applications of this architecture. For example this architecture gives a unique solution to community based sensor network sharing. By creating a network slices multiple users can simultaneously use (share) a host sensor network. The notable feature of this architecture is that it gives the users exclusive access to the sensor network from application programming to data collection. It is almost as if the user owns the sensor network. But it comes with inherent issues related to privacy. Because of the lack of security in current wireless sensor networks, there are various ways to gain unauthorized access to the sensor network. Community sensing paper [36] describes a unique way to preserve privacy while allowing users to share data. This is based on a producer-consumer model where consumer's requirements are bounded by the producer's application output. For an example the client cannot ask for sensed data related to temperature if the sensor network's owner's application does not provide temperature data even though the underlying sensor network is capable of delivering the data. Community sensing project is based on mote class devices, projects based on smart phones [37] are increasingly becoming popular.

CHAPTER 4

Middleware

Stock TinyOS is a tightly integrated system. Components are connected ("wired") with other related kernel components for smooth operation. "Wiring" makes sure unnecessary components are not built onto the application image, thus reducing the image size. During this research work a considerable amount of time was spent on understanding the TinyOS structure & figuring out the necessary changes to the core TinyOS code.

Main idea was to give a mote multiple "identities". Each identity is associated with a virtual network. Because of these multiple identities we identified three main areas of sensor node operations that must be linked to one particular identity.

1. Data (such as the results of temperature sensing)
2. Communication (such as sending & receiving messages)
3. Processing (such as processing of sensed data)

There is an optimization that can be done by sharing the above areas between the identities. Because of the complexity of implementation sharing is not done during this research work. Before the technical details regarding the implementation, the following paragraphs will discuss the drawbacks of the technologies described in the related work section & make a case for the on-demand virtual sensor network architecture.

AMID [38] as described in the related work section, can be easily used to create virtual networks. We can come up with a basic architecture where packet filtering happens based on AMID. Even though this is going to be relatively expensive, (because the packet needs to travel to the high level application layer for the filtering to happen) this is considerably less complex that the architecture described in this thesis. But AMID filtering comes with an inherent problem. In

sensor network applications AMIDs are basically used for a particular type of communication. That means in a reasonably big application there can be multiple AMIDs. (For example Beaconing is associated with one AMID & actual data is associated with another). Since AMID can contain only 256 unique numbers, when using multiple applications there are chances of AMID conflicts. If we are to save AMIDs by reusing the same AMID for multiple purposes in a single application, the application coding will end up being very complicated. In a multiuser model where lot of people have virtual sensor networks there need to be a more centralized mechanism which monitors the AMIDs being used, so that AMID conflicts can be avoided. AMIDs can be reusable in places where there is no radio range overlap. A centralized system managing this task will not be scalable with an increased number of users.

Currently most widely used reprogramming mechanism is Deluge. Sensor network deployments use Deluge under the assumption that deluging will not happen frequently. According to the description in the related work section, Deluge needs to propagate the entire TinyOS image each time it needs to inject a new application to the network. For example if we inject two applications to the network, these two applications contain a fair amount of common operating system related code. Unfortunately Deluge has to propagate the redundant code, which is an expense in terms of energy on mote class devices. This is the main reason Deluge is not used often. The other main drawback of Deluge is it is able to only run one TinyOS image at a given time. For an architecture described in this thesis, Deluge is not particularly good because we need to share the sensor network resources. In order to do something similar to sharing in Deluge, we need to add the old application's functionality to the new application so that the new application can perform both tasks. This is a complex task & each time we reprogram the TinyOS

image to be propagated will grow, because of the addition of old functionalities. This approach is not scalable as more applications are introduced in the system.

The widely accepted model of sensor data sharing is good, because it protects the privacy of the users. But it greatly reduces the flexibility of community sensing. It totally ignores the remote client's requirements. His/her requirements have to be fulfilled with the available set of sensed data. In our architecture remote client can write his/her application to suit their data requirements. The remote client will only have a problem when the underlying host sensor network does not have the capability to fulfill their requirements. This will be a problem with all the existing solutions too. Since according to the figure 36, the packet transmission & receive energy is considerably higher than any other task on a node, the users can run intelligent algorithms [39] to do in-network processing of sensed data in order to minimize transmission & receive costs. This is not really possible in a data-only sharing model, because the data is shared. Different users might want to apply different modifications to the data.

Following sections will discuss the implementation details of core components of this architecture.

Packet Handling

Any network link virtualization requires packet handling. An existing wireless link is sliced so that communication belong to multiple networks can take place simultaneously. Currently only high-level link slicing is done but channel switching type lower layer slicing can be done to improve performance. In our system by packet handling we mean to embed virtual network slice information to an outgoing packet so that the packet can be uniquely identified by the receiving nodes. Same way we forward the received data to the appropriate application thread. To

implement packet level handling we had to make changes to the send & receive interfaces of the TinyOS code base.

Send Interface

Sending a packet out involves several steps. Each packet has to go through different layers in the TinyOS kernel. In our architecture we let a single mote participate in different networks (network slices). Because of this a single mote will need to send out packets belonging to different networks. In the sending process special care must be taken to embed the network & node id information to the packet headers so that the receivers/forwarders know what network slices the packets belong to.

Since TOSThreads (described in the concurrency section) is an implementation of threads on top of an event-driven kernel, we need to use blocking system calls that are available for sending packets. Even though we ultimately use these blocking calls, we have to use functions that are again thinly-C-wrapped over the blocking calls. This is because the dynamic loader (TinyLD described in the dynamic loading section) supports only the C-wrapped version of the blocking calls. The figure 5 is the system call map for sending packets. Stock TOSThreads has an entire fork of the TinyOS kernel subsystems to a depth as long as CC2420ActiveMessageP. After that any call will start to refer to the real TinyOS implementation (event driven).

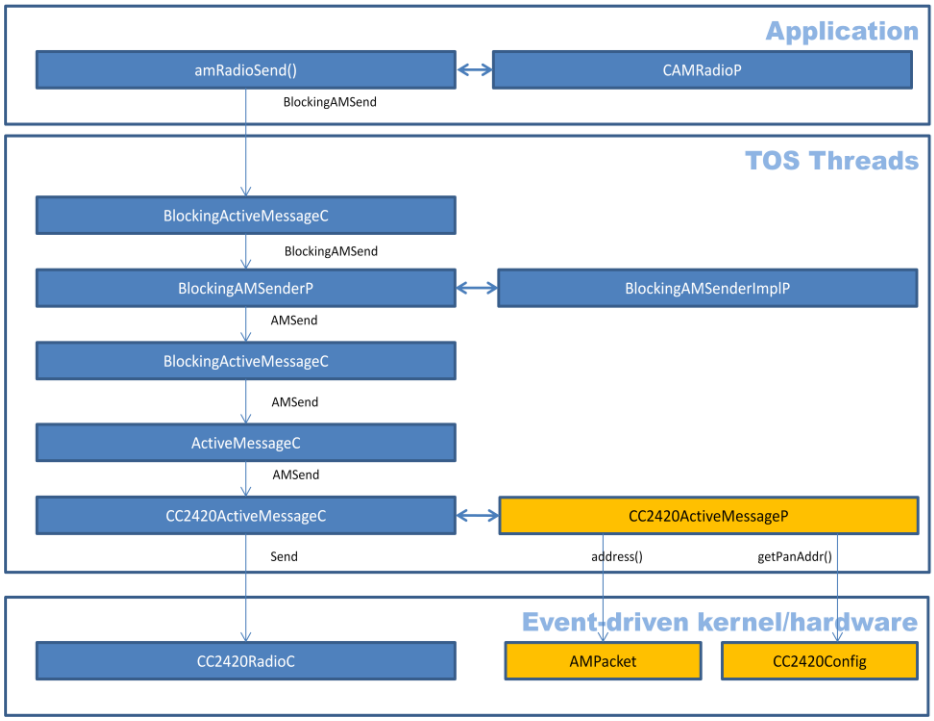

Figure 5 System call map for sending a packet in classic TinyOS

BlockingSenderImpleP is responsible for calling the syscall interface which will wake up the TinyOS kernel thread, to service the system call in an event-driven fashion. This is the boundary between thread & event-driven execution.

There are few changes that we had to make to the TinyOS kernel in order for it to work in our architecture. We have tried the minimalistic approach, so that we have to make minimal changes to the kernel for it to work in our architecture.

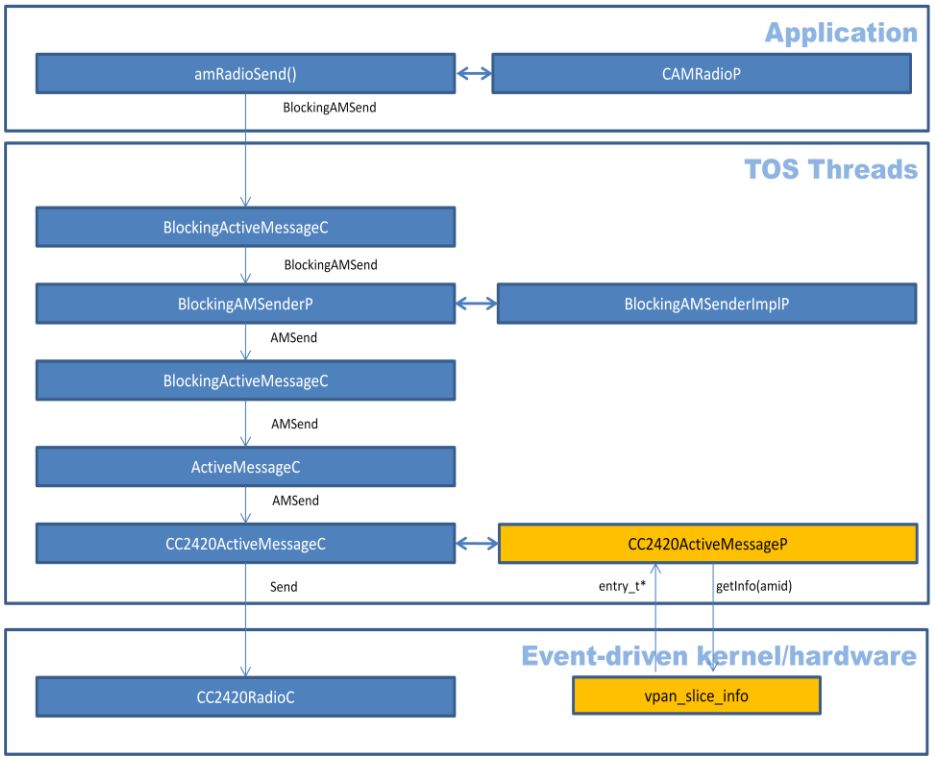

Figure 6 System call map for sending a packet using vpan_slice_info interface

The only difference between our implementation & the stock kernel is the injection of the header information. Unlike the stock kernel implementation which injects the header information from AMPacket & CC2420Config interfaces (Figure 5) we refer to the table (Figure 6, vpan_slice_info) which stores all the information about the virtual networks this particular mote is participating. Since the lower layer code is intact, event driven part of the kernel sees the packet as sent by the virtual network identity rather than the real host network's identity.

```
Command ret_type AMSend.send(arg0,arg1….)

{

//   Extract the CC2420 header
//  Contact the vpan_slice_info table for destpan and src
//  Load the information to the header
//   send msg* to lower layers

}
```

Figure 7 Pseudo code for AMSend command

Active message identification number (AMID) will be is sent to the table as an index (key). Older implementation had the application thread identification number as the key. Because of the widespread use of AMID we decided to use AMID instead of the thread ID.

Receive Interface

For the purpose of identification, each WPAN is associated with a network id. This id is injected to the packet header each time a packet is sent out. When there are multiple networks operating side by side (when their motes are within communication range), network id assures that nodes only accept packets that belong to their own network. Having this id serves two main purposes.

1. Application layer packet processing cost minimizes – If the packet does not belong to its network it is not forwarded to the upper layers. It is hard to avoid lower layer packet processing cost, because in order to discard the packet lower layers must receive it and inspect the header. So the physical layer, receive & inspection cost cannot be avoided.
2. Improves privacy – Other applications running in other networks do not receive packets that do not belong to them, hence maintaining privacy. But there are no measures to prevent a malicious user from listening to all the network ids that can be heard.

Our architecture is about having a single mote participate in multiple networks. So a node needs to accept packets from multiple networks. A node that blindly accepts all the packets that can be heard is said to be in promiscuous mode. But in our implementation we need to dynamically change the number of unique networks one node listens to. This type of listening can be called as a controlled promiscuous operation. We maintain the list of network ids a particular node needs to listen to locally in each mote in the form of a table. When a packet is received the receive interface checks the table to decide whether to accept or discard the packet.

CC2420 is the radio used by TelosB & Micaz family motes. CC2420receive interface is platform specific & is built into the TinyOS image when compiled with TelosB or Micaz options in the command line. Since our implementation is based on TelosB motes we chose this interface to make necessary changes.

CC2420receive sits in the lower layers (just above the hardware layer) & provides various functionalities to the upper layers. It is one of the first TinyOS modules to get bytes belonging to an incoming packet from the SPI (Serial Peripheral Interface) bus. This is the reason why we chose it to implement the controlled promiscuous mode so that we can avoid further processing costs. Because the more we let the packet travel to the upper layers, the more we have to spend energy for processing it.

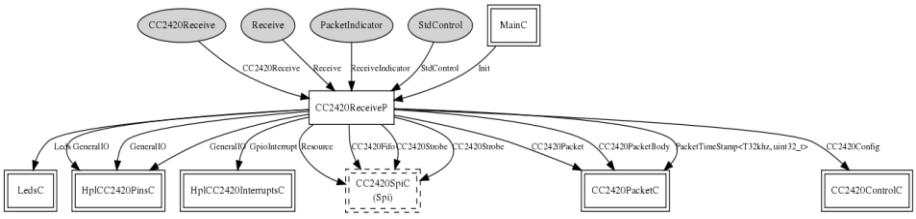

Figure 8 CC2420 ReceiveP interaction with upper & lower layers

CC2420 module contains the code necessary to process & forward packets to the upper layers. It deals with various cases such as longer packets, dropped acknowledgements etc. When all the bytes are received from the SPI bus correctly, receiveDone_task() is called. receivedone_task() contains passesAddressCheck() function. We modified it so that it checks whether the packet is from a valid network by consulting vpan_slice_info interface. vpan_slice_info interface contains the following information given in the table (Table 1).

Index	Thread_Id	AMID	PAN_Id	V_Node_Id
1	1	23	94	12
2	2	25	95	1
…	…	…	…	…
6	23	31	106	48

Table 1 Example of vpan_slice_info table

receivedone_task() calls passesAddressCheck(message_t *) with a pointer to the received message. As the name suggests passesAddressCheck(message_t *) basically checks the address of the incoming frame. By default stock TinyOS only checks the node ID. Stock TinyOS does not have a network wide packet filtering in place. Instead it deals with the situation with an application layer identifier called the Active Message ID (AMID)

We modified the passesAddressCheck(message_t*) function as depicted in the figure 9. Now it consults the table (Table 1) to find out whether the packet belongs to a PANID that the mote is supposed to listen or not, by calling the function check_PANID(uint8_t) provided by the interface vpan_slice_info.

The interaction between the CC2420Receive & the vpan_slice_info interface are depicted in figures 10.

```
ret_type passesAddressCheck (pointer to the received packet)
{

// Extract the header
// if(packet belongs to the control network OR any of the virtual networks this
mote is
//     participating (contact the vpan_slice_info interface)
//    {
 //        accept the packet
//     }
//  else
//    {
//        drop the packet
//     }
}
```

Figure 9Changes in the passesAdressCheck function (filter implementation)

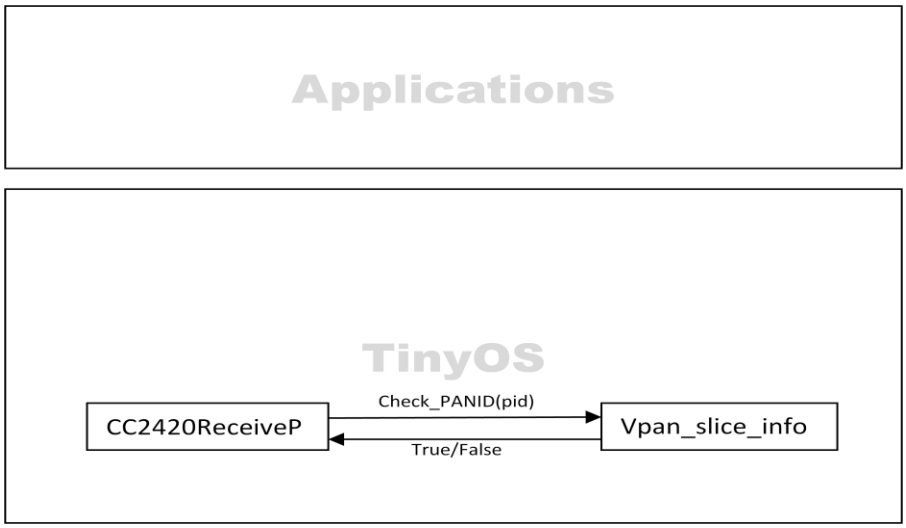

Figure 10 CC2420 Receive & vpan_slice_info interaction

According to the figure 9, besides checking the table using checkPANID(), we accept packets if the destination PANID is the mote's PANID. This is because we maintain a separate network that uses the middleware we developed to provide various services to the virtual networks we create on top of the host networks. All virtual network metadata related communications, such as application thread propagation, network id, virtual node id, etc information propagation happens

through this network. Once the packet passes the address check it is sent to the upper layers as usual.

Overview of Concurrency & dynamic loading on motes

Initially concurrency was not supported by TinyOS. Because of the high failure rate in real-world deployments of sensor networks there was a lot of research work done regarding remotely programming the network avoiding costly on-site visits. Application concurrency came as a solution for this problem. Most of the time, reprogramming the entire network with a new application (new operating system image) is better than trying to reconfigure the existing application.

Currently concurrency in motes is implemented using application threads. There are different implementations of threads in TinyOS [24] [21]. For the work done in this thesis TOSThreads is used. Each application runs as a single TOSThread. Since it is a "C" wrapper around event driven NesC code, all the TinyOS core services can be accessed by TOSThreads. There are some primitive interfaces available to access operating system components. We had to add/change available interfaces in order to add new services like collection tree protocol. Main challenge we found when using TOSThreads was packet handling (Multiplexing & demultiplexing) at the node. Since packets belonging to different application threads can arrive at any time, to avoid packet drop, those packets are queued using AMID & delivered to the corresponding thread when it starts the execution. These mechanisms are described in-detail in the following sections

Threads are great when it comes to concurrency but it does not entirely solve the remote programming problems. There should be mechanisms to create application code at a remote place, deliver it and load the code to the mote at runtime. Delivery mechanism is described in the application dissemination section of this thesis. As the dynamic loading mechanism TinyLD is used. In TinyLD [20] a

proxy service will be running on each mote to understand the application binary image. These binary images can be loaded from memory (RAM) or from data flash.

Middleware Data Integration at the Mote

For this discussion we assume the information related to AMID, PAN_id, v_node_id is already available at the node. Once the data is in the mote, the middleware needs to make several system calls to store the data in proper locations in the TinyOS kernel. According to the dissemination mechanism, AMID and PAN_id can be arrived in one packet. Since we are using a vector (Figure 11) to send the v_node_id along with the host network's node id (current node id), sometimes the mote will have to wait for more than one disseminated packets to receive the v_node_id information.

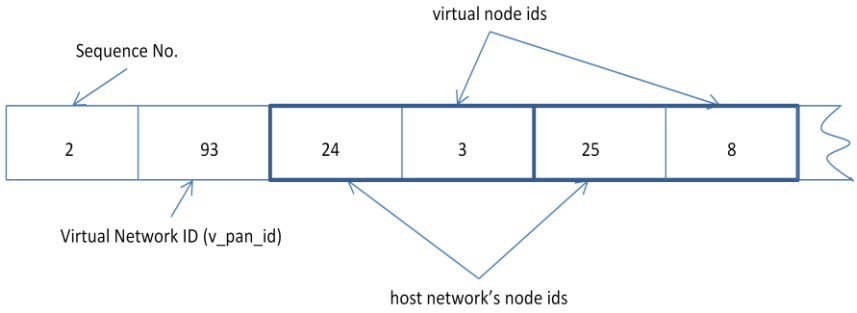

Figure 11 Virtual node id dissemination vector

As described in the network metadata dissemination section, each odd index in the array will point to v_node_id and every even index will point to mote's current node id (Please take a look at the network metadata dissemination section for more detail). We maintain three global variables to denote v_node_id, AMID and PAN_id. When the mote receives a control packet (network metadata information packet) we copy the necessary information to the above mentioned global variables & start vpan_slice_info_finalize timer.

This timer is implemented to deal with the situation when v_node_id is received later. It is efficient to get all the three variables available & then store in the table, when compared to storing each piece of data as it becomes available. In the latter case we need to search the table to find the matching information to insert the data. It will also result in inconsistencies of the network data. If there are inconsistencies, there are chances that all the virtual networks the particular node is participating will get affected. The receipt of the above data altogether at once will depend on the size of the host network. Since a typical TinyOS packet's payload length is 28 bytes, we can only accommodate data belong to approximately 10 nodes in one packet. If just one packet is used we can guarantee that v_node_id, AMID & PANID are arrived at the same time at a particular mote. If the host network is more than 10 nodes we use multiple packets to deliver this information. In this case parts of this information will be arriving at node at different times. The main task of vpan_slice_info_finalyze is to make sure the delayed arrival of data does not result in inconsistencies.

The vpan_slice_info_finalyze timer checks whether all the information is available & then invokes the system call vpan_slice_info.merge() which enters the data into the table.

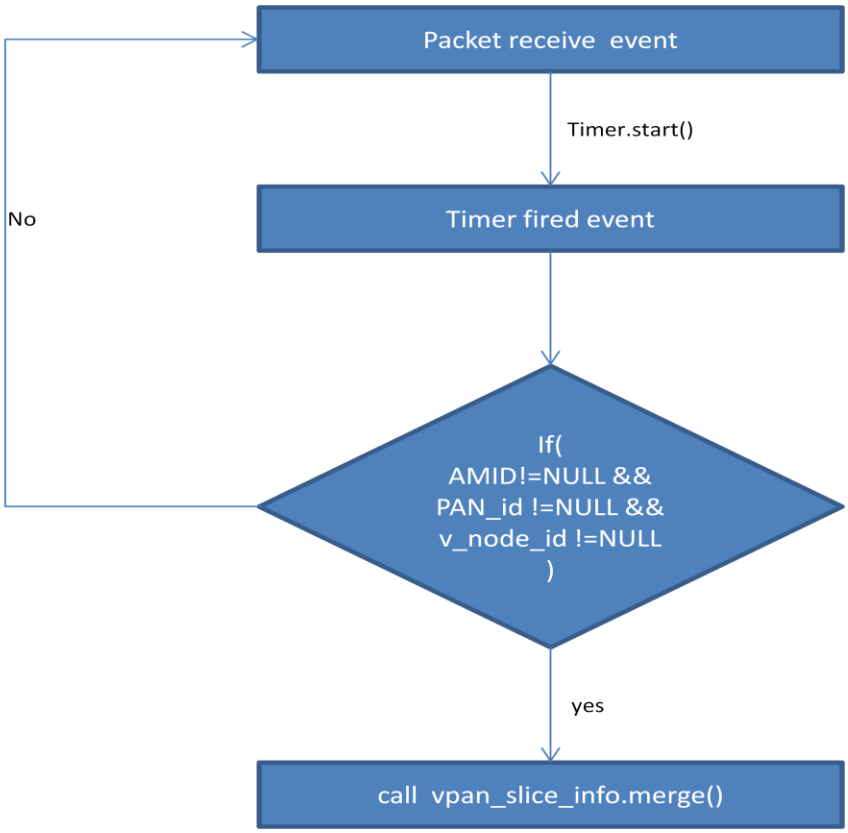

Figure 12 Handling the late arrival of v_node_id

```
command error_t vpan_slice_info.merge (entry_t* entry)
 {
     error_t err;
     err = call ENTRYQ.enqueue (entry);
    return err;
 }
```

Figure 13 vpan_slice_info.merge() function

Index	Thread_Id	AMID	PAN_Id	V_Node_Id
1	1	23	94	12
2	2	25	95	1
…	…	…	…	…
6	23	31	106	48

Table 2 vpan_slice_info table entries

```
typedef struct entry_t {
uint8_t thread_id;
uint8_t AMID;
uint8_t PAN_id;
uint8_t v_node_id;
} entry_t;
```

Figure 14 vpan_slice_info structure

This entire process needs to be completed before the application code starts to propagate from the base station, at least before the application thread starts to execute. But according to the current implementation it is completed before the application code starts to propagate.

According to the table 2 & figure14 there is one more field which is the thread_id. This id is only available after the application starts to execute. The following event will provide the application thread_id.

```
 event void DynamicLoader.loadFromMemoryDone (void *addr,
tosthread_t id,
     error_t error)
 {
          call vpan_slice_info.merge_thread_id(id);
 }
```

Figure 15 Calling thread id merge function

```
command error_t vpan_slice_info.merge_thread_id (uint8_t id)
 {
   entry_t *temp;
   int i;
   for (i = 0; i < call ENTRYQ.size (); i++)
    {
temp = (entry_t *) call ENTRYQ.element (i);
if (temp->thread_id == NULL)
 {
   temp->thread_id = id;
   return SUCCESS;
 }
    }
  return FAIL;
 }
```

Figure 16 vpan_slice_info.merge_thread_id() function

Inside this event we invoke the system call vpan_slice_info.merge_thread_id() to enter the thread_id to the table. For sending & receiving packets thread_id is not used, but it is helpful in killing the application thread, hence in dissolving the virtual network. The below diagram gives an overview of the system calls used in the integration of metadata section.

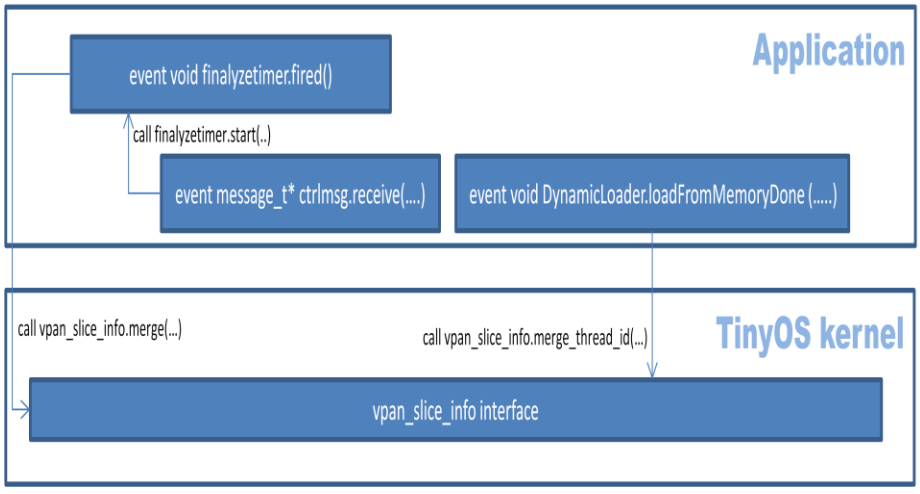

Figure 17 System call diagram for middleware data integration

Dummy Threads

In our initial design we injected the application code only to the motes selected by the client. So the application runs only on the selected nodes. Since this architecture is a true network-level virtualization, running the application only on selected nodes resulted in a routing issue which will be described in this section.

In the following diagram wireless links are depicted using solid lines. For example node 1 can only reach node 0 & node 2, node 6 can only reach node 3. So for packets originating from node 6 need to have node 3 in order to forward its packets to the base station. (i.e. node 0) The node ids given here do not correspond to the node IDs assigned by the TinyOS in the wireless sensor network. According to our architecture a single node will have several different node IDs depending upon the virtual networks the node is participating.

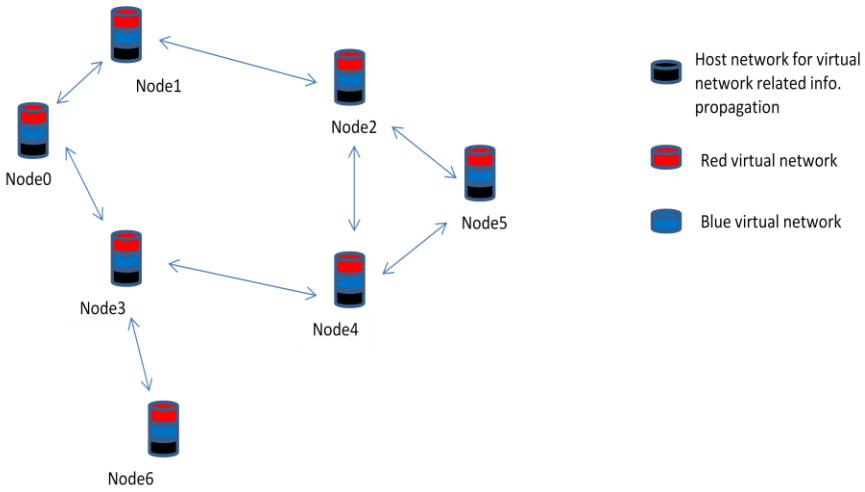

Figure 18 Sensor network with multiple virtual networks

Problem of Network Holes

According to the above setup the red & the blue virtual networks (WPAN slices) run on all motes. Since both the networks are properly connected all the packets belonging to red & blue networks flow towards the base station flawlessly.

(Assumption: Node 0 is the base station for both blue & red networks. Both the virtual networks can be configured to have different base stations.)

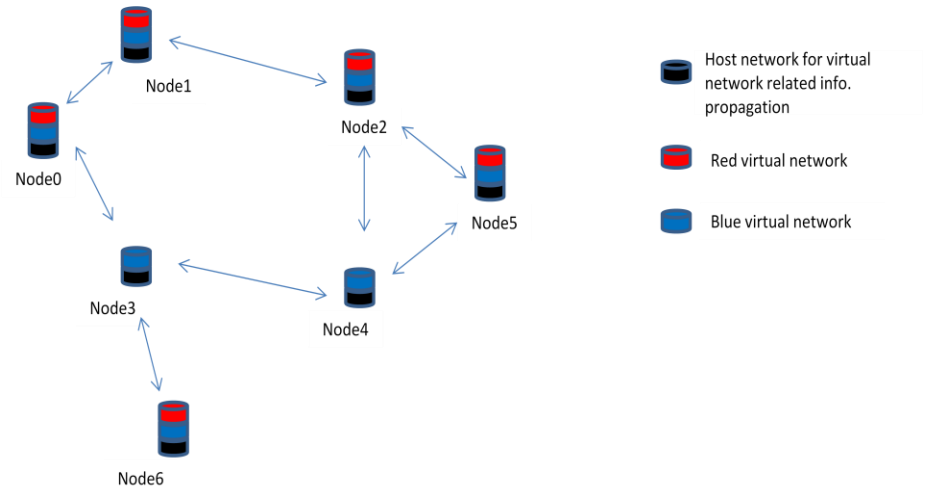

Figure 19 Red network routing hole

But according to the above diagram, even though all the nodes are participating in the blue network, not all the motes are participating in the red network. In this case node 3 is not participating in the red network, so the red application running on node 6 does not have a red neighbor running on node 3 to forward its packets. Even though there is a blue application running on node 3, it will not accept packets from red application running on node 3, because it belongs to a different network. Both the applications belong to different AMIDs & PANIDs. There can be instances where two applications use the same AMID (Described in the related work section). Since PANID filtering happens in a lower layer this architecture does not have to worry about possible AMID conflicts.

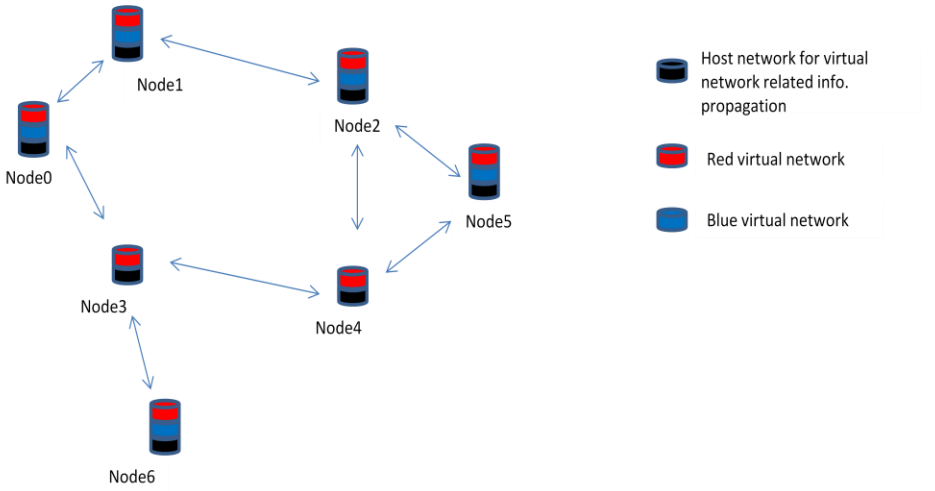

Figure 20 Blue network routing hole

The same holds true when there is no blue application running on node 3 & 4 (above figure). This situation arises regardless of the application running. The outcome will depend upon the location of the node involved. In the worst case this situation will make the entire virtual network unusable. This creates a routing hole in the network (red virtual network figure 21), possibly making the entire network unusable.

Possible Solutions

1. Making the blue network forward packets on behalf of the red network is a straightforward solution. In this example the blue application thread in node 3 will forward red application thread's packets towards the base station. But this solution has issues associated with it.The blue application needs to know the red application running on node 3's AMID, PANID & the v-node_id in order to forward packets. In this case all the applications need to know all the information about other concurrently running applications (virtual networks). These information need to be known before the any application starts to execute. This requirement greatly reduces the flexibility of creating dynamic virtual sensor networks. Even if we are to dynamically inject information the architecture will have another layer

of overhead for disseminating this information to the entire network as network slices get spawned. There can be situation where blue network is running CTP & the red network is running something else as its networking protocol. In this case there will be even greater problem in routing packets belonging to another network. Even if we allow blue network (in this example) to forward red network's packets with all the above mentioned drawbacks, this poses a serious security issue. In this case blue network application will be able to snoop into red network's data.

2. Letting the control (host) network forward packets on behalf of red network. This solution also will have the same security & flexibility issues related to the solution discussed previously.
3. Injecting the application (red in this case) to all the nodes & let the application take care of routing.

This solution also has the following drawbacks.

1. Usually client does not select all the motes.
2. There will be a performance penalty if we try to propagate the code to all the motes.
3. When running an application on a node which originally was not selected by the client will unnecessarily drain its battery.

When we looked into the problems associated with each solution, solution no.3 seems to be the most practical & cost effective to implement. The following is a discussion about why we feel solution 3 is cost effective considering all the three drawbacks associated to that solution.

1. Even though client selects a subset of nodes from the network the gateway PC will be able to inject the code to all the motes & keep track of the real participating motes. (Selected motes by the client)

2. According to our dissemination protocol, we divide the entire application code to small TinyOS packets that can be re-assembled at the destination. According to the simulation results during the dissemination almost all nodes in the network receive a major portion of the application code, regardless of whether they participate in the network or not. Because of this the additional energy cost to properly disseminate the entire application code to all the nodes will be minimal.
3. Even though running an application on nodes that are not selected by the client will unnecessarily drain the battery, these dummy applications participate in routing. This makes them useful in the maintenance of the virtual sensor network. The observation is if we do not run the same application on all the nodes, the battery drainage of the nodes in the network becomes uneven. This might make the entire network unusable regardless of whether we save battery in the unselected nodes.

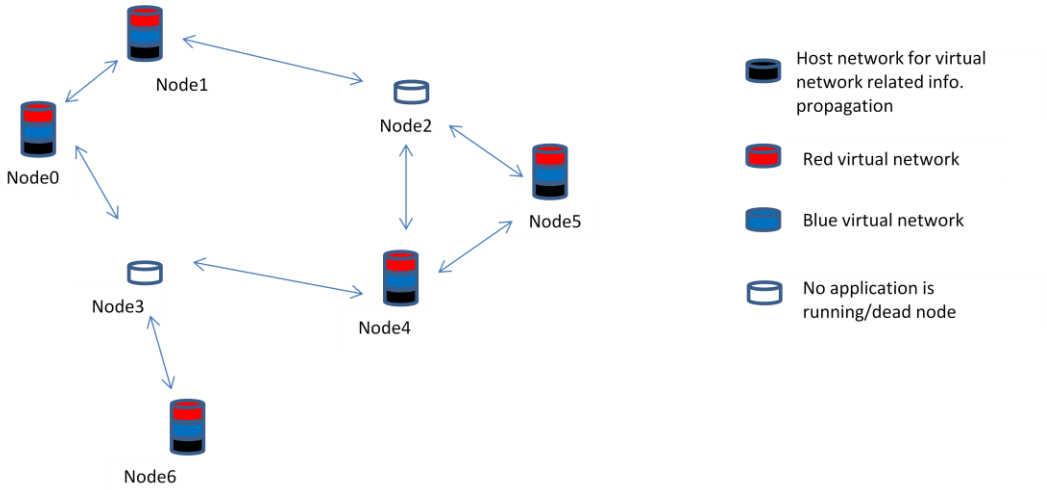

Figure 21 Partitioned network

According to the above diagram, if node 3 & 2 are dead because of frequent usage the entire network becomes useless. The current implementation is to send

the same application to all the nodes. But effective application will run only on selected motes.

There is a possibility to just let all the motes run application & filter out unselected node data at the gateway. But this will lead to trouble if the client injects a collaborative sensing/computation type of applications. This is when one node's activity will depend on its neighbor's (or other non neighboring node) output. Filtering out the results will be difficult in this type of a scenario.

Difference in application behavior on selected and unselected nodes

We came up with a solution that works flawlessly regardless of the type of application the client injects. The only difference between the selected node & the rest is, only selected node will actually execute the application section. (Such as sensing or computation)

In order to do that, we have divided all the client applications into different parts. Each part according to the figure is a dynamic TOSThread. For example the code in the figure has to result in three independent threads, but in the physical node only one dynamic TOSThread will get spawned. [20]

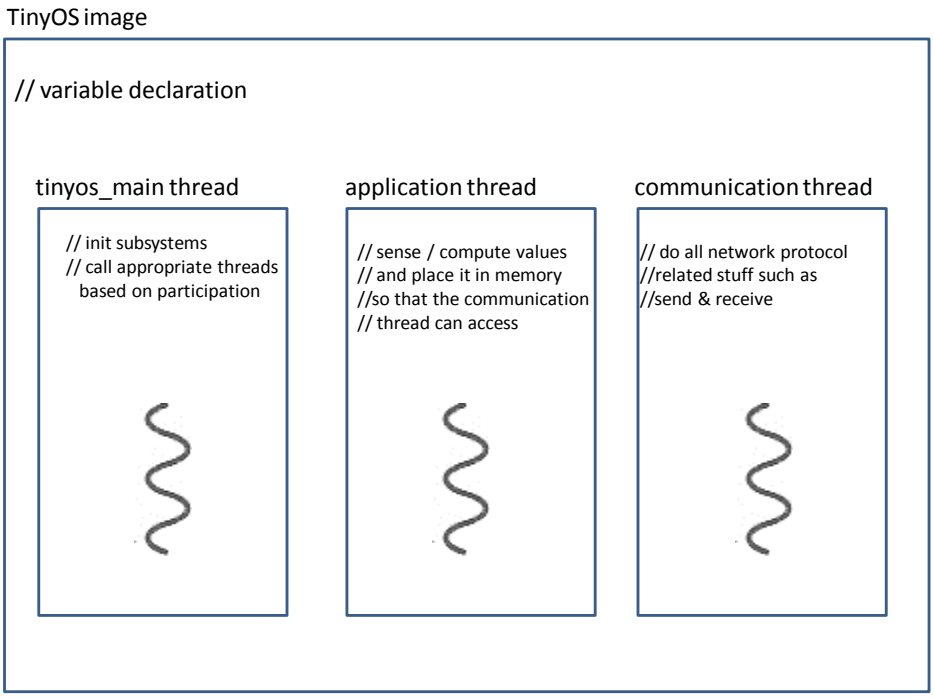

Figure 22 TOSThread Application

TinyOS_main thread – Most of the initialization is done in the main thread. This code is invoked when TinyLD loads the application. This function has to decide whether to call the application thread or not based on the selection by the client. If the client has selected the node to be participating in the virtual network, main thread will call the application thread & the communication thread, otherwise just the communication thread. Since it is vital for forwarding packets, communication thread will be called regardless of the participation. In our system the reduced version of the thread (main thread & the communication thread) is called dummy threads. In order to synchronize the operation of threads various synchronizing mechanisms such as semaphores & barriers are used.

According to the current TinyOS programming, there are no specific rules to where to put the initialization code as long as it is done before using the initialized components. Specifically for CTP, the CTP subsystem needs to be started & the root

information should be initialized before we use the collection tree protocol. This is true with other available protocols such as DHV, Drip & DIP.

Imposing code placement rules when creating TOSThreads C-based applications

In our architecture the client does not write the C code in the figure. Instead client will write the code using a high-level script-like language, which will be converted to a code like in the figure 24 (This feature is yet to be developed & not a part of this thesis). So our architecture can dictate where to put the code (without changing the client application's logical flow) so that an application can work as a dummy thread (in this example without sensing & sending packets) on motes that are not selected, and as a full blown application on selected motes.

CHAPTER 5

Dissemination

There are several dissemination mechanisms already available in TinyOS. Basically dissemination in TinyOS is divided into two categories.

1. Small object dissemination
2. Large object dissemination

Small object dissemination

According to the TinyOS enhancement proposals dissemination [7] is "reliably deliver a piece of data to every node in the network". As the description says this is about disseminating small values such as network configuration parameters etc.

Large object dissemination

This is basically the operating system image dissemination and the de facto protocol to do it is called deluge. Deluge can be considered as an entire suite because it not only takes care of the delivery, it also has its own boot loader and a storage mechanism. Usually deluge propagates an entire TinyOS binary. Since the payload is large this is a costly operation in sensor networks. This is because unlike dynamic threads (which we use in our architecture), it lacks the ability to dynamically link the function calls to the modules that are already existing making it disseminate redundant core TinyOS components each time it disseminate code. This type of protocols is designed based on the assumption that large object dissemination does not happen very frequently. But this assumption makes sensor networks not very flexible to use.

But in our architecture, object dissemination mechanism lies between the above two types. Since we are using dynamically loadable threads, our application code does not have to carry redundant TinyOS core component code, making the application

code to be propagated, smaller that what deluge propagates (see the experiments section). On the other hand our architecture cannot use small object dissemination protocols, because the items to be sent that are not application code is too big for stock small dissemination protocols to handle.

Because of the above reasons we could not find any available dissemination protocol which suits our needs. This led us to develop our own dissemination protocol.

We have identified the data objects that are wirelessly transmitted over the PAN in our architecture & believe that the data items will belong to either one of the following categories.

1. Application data
2. Application (thread) code.
3. Underlay (virtual network) related meta-data.

Majority of the exchanged data objects will belong to application data, because the type 2 & 3 data items will be only used during the formation & dissolution of virtual networks, which will happen rarely.

With the current software distribution we hope to present a dissemination protocol for the purpose of disseminating type 2 & type 3 data.

Push based guaranteed delivery mechanism.

Like Deluge & other stock dissemination protocols, this will push the data into the sensor network. Basically there are two main components to this protocol.

1. Trickle based neighbor discovery mechanism.
2. Link layer acknowledgement based data delivery mechanism.

Trickle based neighbor discovery mechanism

Most of the sensor network routing protocol build their routing tables before actually using the protocol for communication. (for example CTP, Deluge etc.) Similarly in this protocol nodes spend their initial time in discovering their neighbors. After a

successful completion of this state nodes move to another state where they start delivering the real data. A simplified version of a state transition diagram is below.

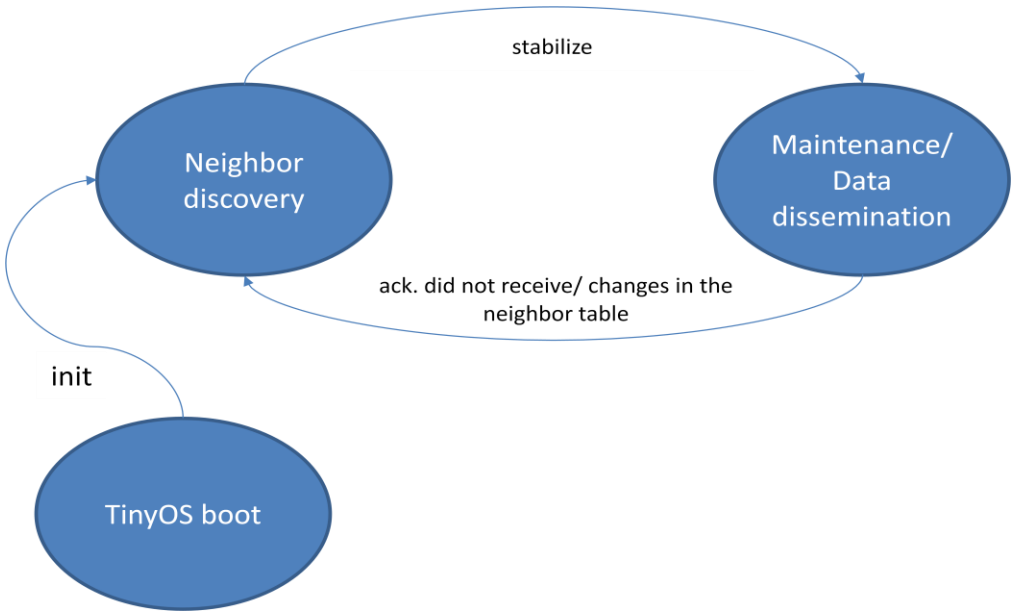

Figure 23 States of a node when running the dissemination protocol

When a node boots, it starts the beacon timer which will broadcast the beacon message. This beacon message will be sent periodically. Initially the interval between two beacon messages is set to a very low value. (128ms) this makes sure the beaconing process starts quickly in order to find out the neighbors.

Adaptive Beaconing.

The main purpose of the beaconing process is to find out the neighbors. Unless the nodes move (or die etc.) generally the neighbor table tend to be consistent over time. This is the reason why CTP like protocols employ adaptive beaconing. If the neighbor table is consistent, there is no real need to perform the beaconing task as it is an expensive operation in terms of energy. Therefore in our implementation the beaconing interval changes over time. When there is no need to send beacons (consistent routing table) the beacon interval increases & when there are changes to

the routing table the beaconing interval resets to its minimum to quickly capture the dynamic changes. The beacon interval resets to its minimum when one or more of the following conditions are true. This process only happens when the node is in maintenance mode. When the node is in neighbor discovery phase the following conditions do not apply.

1. An acknowledgement did not receive with respect to a beacon message sent by the node.
2. Beacon response message is received from a node which is not in the neighbor table.

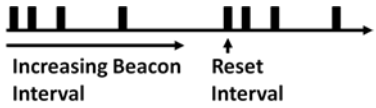

Figure 24 Beaconing effect. Time increases to the right.

According to the diagram after the boot event a node will always be in one of the two following states.

1. Neighbor discovery - In this phase a node will double the beacon interval each time it sends a beacon out until it reaches the maintenance mode. But the default interval will stay the same as long as there is no entry in the neighbor table. Initially nodes will broadcast beacon request messages. Upon receipt of the beacon request message other nodes will respond with a beacon respond message. On receipt of this message, nodes will add the source of the packet to its neighbor table. If an intermediate node (a node which is not in focus) hears the packet still it will add the node id to its neighbor table.
2. Maintenance - When the node enters this state it sends periodically beacons at the maximum beacon interval to check for any changes to the existing neighbor information. Even though the node is in maintenance mode with

respect to neighbor discovery, at the same time it is busy in delivering the real data.

Link layer acknowledgement based data delivery mechanism

The main idea behind this protocol is for each node to deliver the data to all the nodes in its neighbor table using link layer based packet level delivery guarantees. The observation is, if all the nodes can deliver packets to all of their neighbors, the entire network gets covered by the delivery mechanism.

In our implementation data is delivered using a special structure. Since we have categorized the payload types, we have selected an array of uint8_t values. Such an array was chosen for compatibility with other packet structures. When sending a data (application code or metadata) packet the node enables the PacketAcknowledgements field so that the receiver is signaled to send the sender an acknowledgement.

CHAPTER 6

Tunneling

The main idea of our architecture is to build virtual sensor networks. This could mean forming a virtual sensor network on top of a single host network, or stitching multiple host network clusters to build a single virtual sensor network. Stitching multiple host networks brings a new problem. This problem becomes obvious when the host clusters are not close enough. (i.e when they are not in communication range) This is when the virtual network cannot be considered a single network, even though we build the same virtual network on top of the host clusters. This is because the virtual network's corresponding pieces are not in communication range. This was a setback for our architecture, by allowing us to create a virtual network only when the host clusters are in communication range. We came up with a solution to this problem by emulating 802.15.4 wireless links with either Ethernet or Wi-Fi (802.11) links. This is because the P2P overlay which connects all the host clusters will use either one of them or both. By emulating the radio links, we can make the virtual network clusters "see" each other even though they are far apart.

Basically implementation of tunneling is protocol specific. That means in order to extend (i.e. to simulate a wireless link based on a particular network) a network we need to transfer network related information between motes using a different method other than the 802.15.4 based radio.

For our proof-of-concept implementation of tunneling we selected a widely used CTP (Collection Tree) protocol. Since most of the sensor network deployments are sensing and collection oriented, we hope this implementation will be useful for a wide variety of sensor applications, making lot of sensor applications which are based on CTP compatible with our architecture.

Overview of Collection Tree Protocol (CTP)

CTP [10] is an agile, efficient & reliable data collection protocol available primarily for sensor networks. The designers of this protocol have taken the error-prone nature of the wireless communication into account when implementing this protocol. This widely used protocol is based on the following main concepts.

1. Accurate link quality estimation – A pluggable component is responsible for link quality estimation & is done on a single hop basis. Link quality is measured using the number of estimated transmissions (ETX). The link quality is better when there is a less number of transmissions.
2. Adaptive beaconing – Beaconing is used by the link estimator to find the best single hop parent node. Each node will be associated with a parent node (like in a tree data structure) so that when a message is generated by the node or a message from another node is received, it will immediately be forwarded to the parent node. These beaconing messages are sent by each node periodically. If the network has settled down, the rate of beacon send will be reduced to save energy.
3. Route to the destination is validated – Based on each node's ETX value the route till the destination is validated. This will reduce the number of routing loops in the network.

Implementation

In order for CTP to work properly it has to establish the collection tree. The tree establishment phase results in broadcasting beacon messages so that nodes can figure out their neighbors and their respective link qualities.

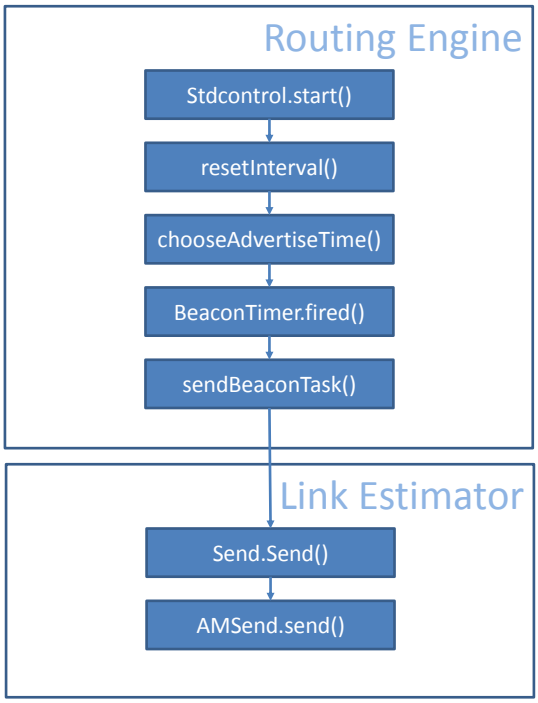

Figure 25 System calls between routing engine & the link estimator

Initially beacons are sent with a small timeout. Then eventually the timeout increases so that the mote does not spend too much energy on beaconing. This is an effect of adaptive beaconing.

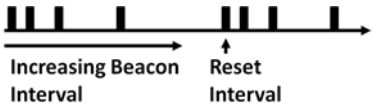

Figure 26 Beaconing effect

Figure 25 provides the functional diagram periodic beacon send task. In this example the 4-but link estimator is selected as the link estimator. The modular design of the protocol allows us to plug in any available link estimators (such as LQI)

The basic idea of the routing engine is to decide when to fire the beacon timer which will actually get served by the link estimator. The important task which the routing engine carries out in this scenario is the updateroute() task. During this task the routing engine will go through the routing table & find the best neighbor. If the,

found neighbor is different from the current parent it will change the current parent to the found neighbor. Routing engine calls updateroute() task every time before it sends out a beacon message to make sure neighboring motes get the most up-to-date information. Once the packet is handed over to the link estimator, it proceeds with the sending process.

We had to make some changes in the link estimator in order for tunneling to work. Since a wireless link needed to be virtualized using a wired (Ethernet) or a wireless (WiFi) link, we had to basically exchange the beacon messages using the virtualized links so that the nodes at the edges of the virtualized links act as if they were next to each other.

On order to calculate the beacon driven ETX value the following information need to be exchanged by the motes.

1. Current parent
2. Current ETX
3. Source of the packet (beacon sender)
4. Sequence number of the beacon.
5. Destination.
6. CTP header information

This implementation exchanges several information between the gateway application & the link estimator shown in the below figure. There are several system calls we created between these two components so that the link estimator can send data & probe for new data.

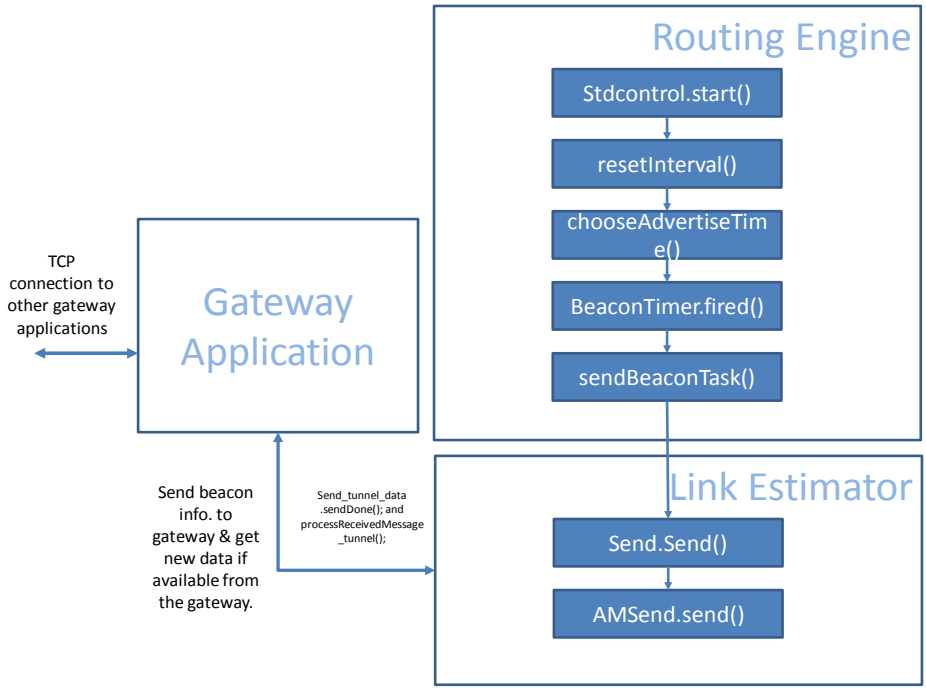

Figure 27 Tunneling system calls

When the link estimator is ready to send the beacon information, it will do the same task in two ways. It will send the information to the lower layers so that the radio can disseminate data & at the same time it will send data back to the gateway application using the function call Send_tunnel_data(). The latter part only works when the mote is connected to a serial cable. (Typically the base station)

Similarly according to the below code snippet when a message is received the function processReceivedMessage_tunnel() will take care of the processing & forwarding the necessary data to the upper layers.

```
void processReceivedMessage_tunnel (ctp_tunnel_info_t * msg)
 {
……

//Extract all necessary data from msg pointer

// update neighbor table with this information
// find the neighbor
// if found
   // update the entry
// else
  //  find an empty entry
  // if found
     // initialize the entry
  // else
    // find a bad neighbor to be evicted
    // if found
        // evict the neighbor and init the entry
    // else
      // we cannot accommodate this neighbor in the table

//once all the processing is done, send the packet to upper layers
…….  }
```

Figure 28 Processing tunnel data in the link estimator

CHAPTER 7

Gateway

In our architecture the gateway acts as a vital component. The typical gateway is considered to be a more powerful node than a sensor node & an interface between the personal area network & the rest of the architecture. The following figure gives an overview of the tasks in the gateway application.

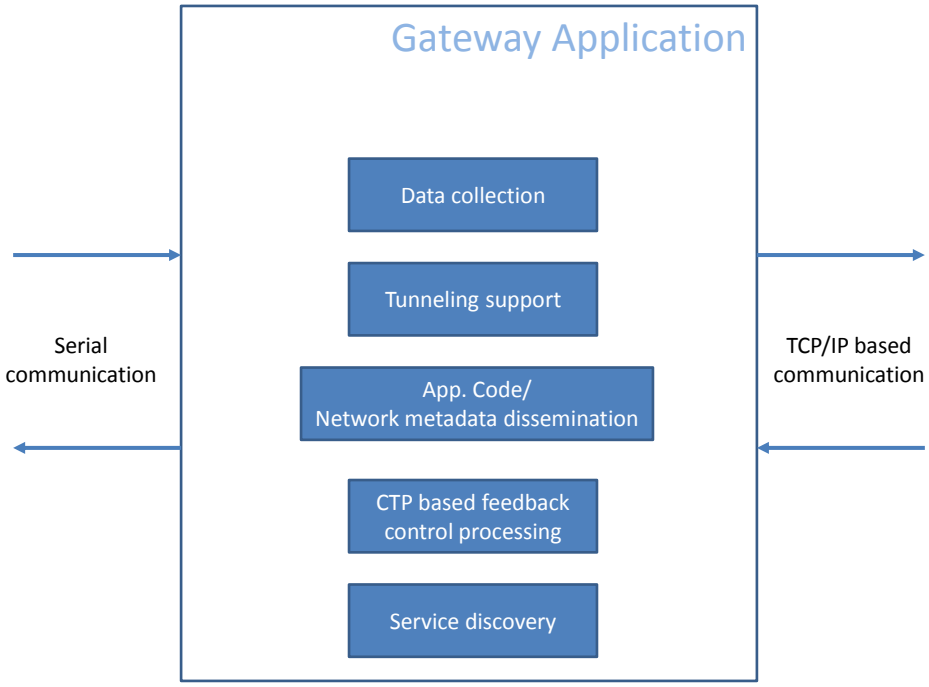

Figure 29 Typical gateway services

We have identified the following main tasks that a gateway application needs to fulfill. Current proof-of-concept gateway application is written in java.

1. Support application data collection – This is the primary task of any application that is running on a gateway connected to a sensor network. The java (or can be written in python) application does low level bit manipulation to convert raw bits to readable data. It also allows the user to manipulate data using the given accessors & mutators.

2. Application code dissemination – The gateway application must be able to receive application code from the client, break it into serial packets & send them to the base station node via serial interface. The base station node will handle converting them again to TinyOS packets & propagating them to senor nodes.
3. Network metadata dissemination - This is similar to application code dissemination. Corresponding tasks will take place.
4. Feedback control – All the information about underlay creation & maintenance will be monitored by the P2P overlay for smooth operation of this architecture. When the delivery of different components related to the virtual network formation is properly done, the base station node will notify the gateway about the completion. The gateway (in this case the java application) can take further actions depending upon the outcome. (For example notify the client or restart the process again etc.)
5. Support tunneling – (figure 30) this part is discussed in detail in the tunneling section of this thesis. Support for tunneling has to be established by consulting both P2P network and the attached WPAN at each gateway. This brings the P2P overlay network & the WPAN underlay closer together whereas in other parts WPAN & P2P work more or less independently. When supporting tunneling all the participating gateways must be connected to each other. There are several messages that the P2P (JXTA based) will be sending during this process. Since the P2P overlay control system is still under development & is not part of this thesis, the structure of these messages will not be discussed. In the current implementation all the participating gateways are hard coded so that they have TCP connections between them.

6. Service discovery – This task is mainly about communicating with other peers (participating gateways) and the remote client regarding the services offered by the particular sensor network.

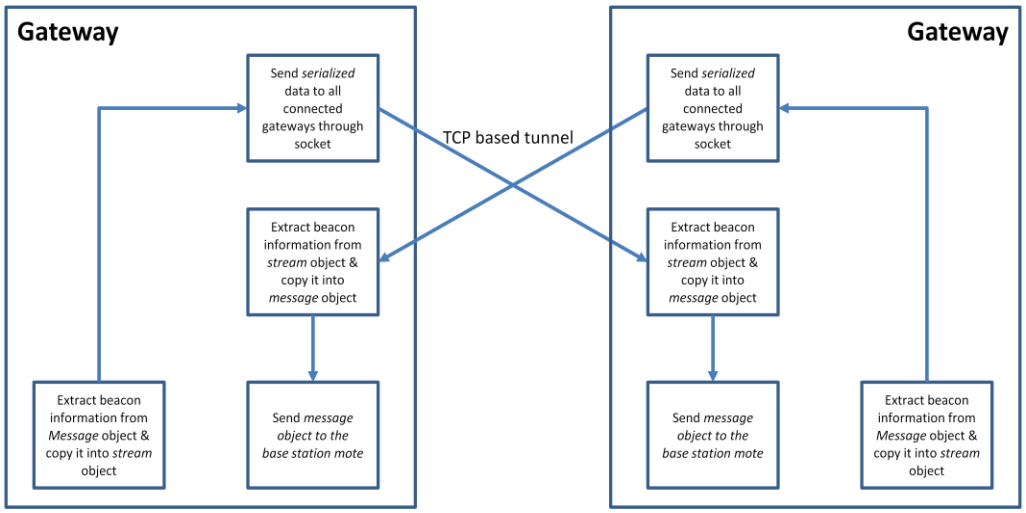

Figure 30 Tunneling between two participating gateways

According to the above diagram, the main purpose of the tunneling support section of the gateway application is to act as an interface between the socket object & the serial interface object. Extracting data from the socket object & feeding it to serial interface object & vice versa is done here. For simplicity only one way is shown.

CHAPTER 8

Experimentation

Any research work will be a tradeoff between the advantages the work provides & the overhead to implement it. Because of this we decided to evaluate various aspects of the middleware implementation

Program image size comparison with Deluge

The main advantage of our system in comparison to Deluge (Described in the related work section) is the ability to send incremental code updates. Unlike Deluge in this architecture does not send the entire TinyOS image to the nodes. In order to verify that, we had to run an experiment, that captures the application code size. By "application" here we mean the code that needs to be propagated in order to reprogram the network (in Deluge) or to create a virtual network (in our architecture). A Deluge enabled application can be depicted as following. These applications are propagated from the base station to all the nodes in the network.

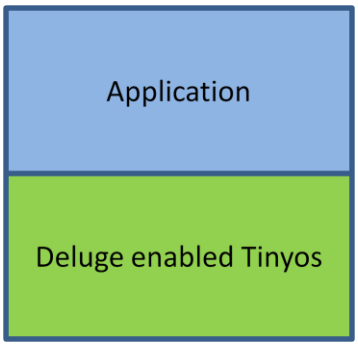

Figure 31 A Deluge enabled application.

The following diagram shows a node which has received two applications via Deluge.

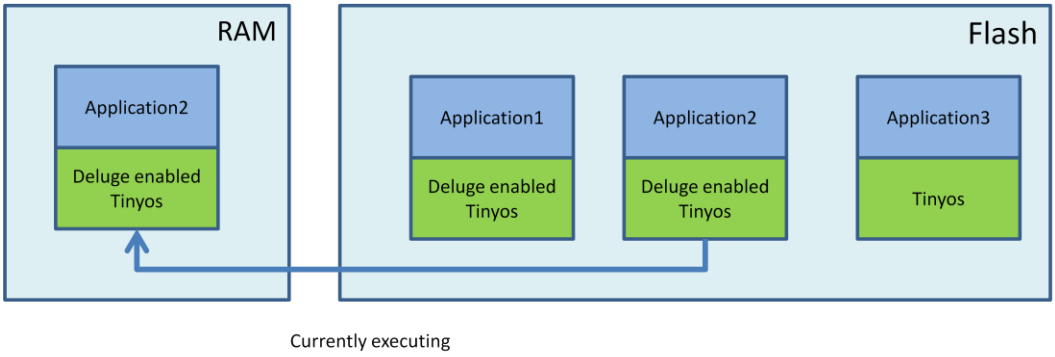

Figure 32 Node containing two Deluge based images

In Deluge one can create multiple volumes so that multiple applications can be stored. When compared to Deluge, a node that supports our architecture will have the following format.

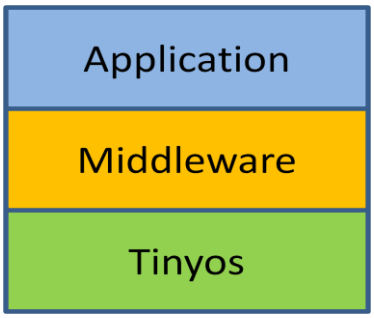

Figure 33 Operating system image of a node that supports virtual sensornet architecture.

All the applications will be hosted on top of it as dynamic threads.

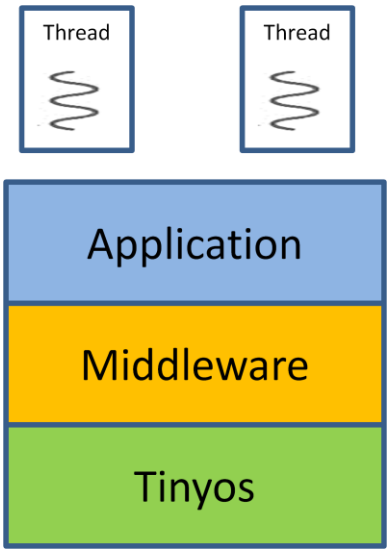

Figure 34 Deployed Threads on top of the system image

In table 3, we have compared the program image that must be installed (By physically connecting the node to the computer) to support the two architectures. In Deluge, the bottom part of figure 32 must be physically installed on every node that supports Deluge. All the application images (figure 31) are added later. In our architecture the image described in figure 33 must be installed. There is a possibility that the image in figure 33, itself can be Deluge enabled. Such a configuration will look like figure 35.

Underlying support system image	Size (bytes)
Deluge – Base Station	29186
Deluge – Golden Image	28640
Virtual sensor network architecture	34818
Virtual sensor network architecture – Deluge enabled (Figure)	39274

Table 3 Underlying support system size

There are two types of support systems in Deluge. All the nodes except the base station will run the Golden Image support system. For this experiment we have used a null application which does not do anything. This Golden image is just for invoking the Deluge system. The same thing applies to the base station. Only differences are related to the capabilities of Deluge.

When compared to Deluge our architecture's underlying support system's size is relatively high. This is because our underlying system has to support various services such as dynamic loading, virtual sensor network middleware, code dissemination mechanism etc. When it is coupled with Deluge it becomes even bigger, because it need to have Deluge related components built onto it.

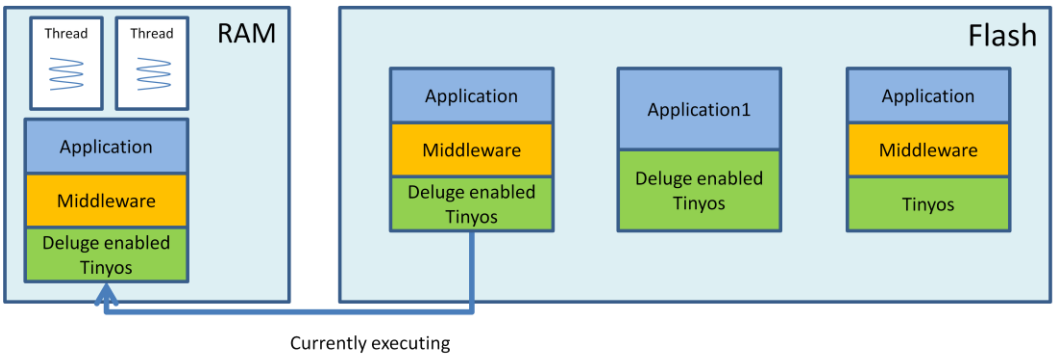

Figure 35 Deluge enabled VSN images

In table 4, we have compared the images that need to be sent in order to re-program (or create virtual networks) sensor networks. This is comparing thread binary to operating system image depicted in figure 31. For this experiment we have selected a simple Blink application.

Application image	Size (bytes)
Deluge – Blink	23772
Virtual sensor network architecture - Blink	16016

Table 4 Loadable Application size

By looking at the results, we can see that the thread based application's size is smaller than the Delugeble application's size. This makes our architecture better because we need to send fewer amounts of data to the network in order to create a virtual network. The other most important feature is the thread does not kill any other threads that are concurrently executing, whereas in Deluge only one application image can run at a given time.

Energy overhead for establishment & maintenance of the virtual network

Energy consumption by the nodes (or the energy consumption in the network as a whole) is one of the most critical properties to consider when designing an architecture that involves wireless sensor networks. The main activity that consumes majority of the node's energy is communication (Figure 36).

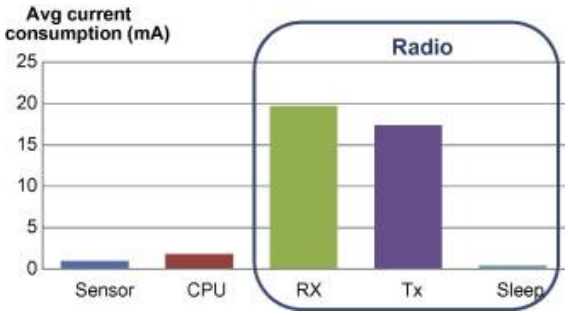

Figure 36 Energy consumption of a typical sensor node

The main assumptions for this experiment are the following.

1. Concurrently there is no other underlay establishment in progress.
2. There is no other application running during the establishment process.

During normal operation of our architecture the first assumption always holds true. But there can be many applications (virtual networks) already running during a virtual underlay establishment (second assumption).

For simplicity we have selected the metric as the number of messages transmitted during the establishment of the underlay. Establishment solely depends on the dissemination protocol. Unfortunately this metric does not cover energy consumption of the microcontroller, sensors, energy related to listening etc. In real world scenarios receive energy accounts for more than even the transmission energy (Figure 36), because the radio needs to be on in order to receive packets. To reduce the difference between the real world scenario & the experiment results we have reduced the time taken to establish the underlay so that nodes spend less time in listening. More details about this can be found in the dissemination section of this document.

Experiment setup

This experiment is performed on two 6x4 host sensor networks. Nodes used here are Telos rev. B [2]. By default they use CC2420 radio & a typical CC2420 header includes 11 bytes. Generally a TinyOS packet allows 28-byte long data payload

length. For this experiment we have not used any packet footers. We have monitored three different stages during the establishment. Beaconing, metadata dissemination & application code dissemination. Each experiment is run three times to get an average measurement. The power level of the CC2420 radio is set to the default level which is 31 dBm. Measurements were taken based on the bytes transmitted instead of the packets because of the transmitted packet size differences. Radio power changes according to the actual packet size.

Two virtual network applications were used. For simplicity the initial application (experiment I) is a basic blink application, in which when the virtual network is formed all the nodes in both host networks (two 6x4= 48 nodes) will blink red LED every second. The size of the application code is approximately 100 bytes. Second application is close to a real world sensor network, because the application used here is a sensing & data gathering application. Same host networks as the above experiment were used. All the nodes will be sensing the temperature every 30 minutes & reporting the data back to the base station which is connected to a Linux host. There can be several variation of this experiment. The size of the application code is approximately 700 bytes.

Type of Message (size in bytes)	Avg. no. of bytes transmitted per node (App - Blink)	Perc. of bytes (%)	Avg. no. of bytes transmitted per node (App - Sensing)	Perc. of bytes (%)
Beaconing(dissemination protocol) (11+1)	96	0.49 %	72	0.05 %
Underlay metadata information (11+28)	819	4.23 %	780	0.64 %
Application code (11+28)*4	18408	95.26 %	119,652	99.29 %

Table 5 Number of bytes transmitted during formation of underlay

For both above experiments application data is ignored. Since the application related to the virtual network is supplied by the client, the current architecture has no control over the application's energy requirement. Also in the above experiments the last application code packet is not exactly 28 bytes. For simplicity in calculation we have included the final packet in application code as a full 28-byte long packet (a negligible addition).

Measuring virtual network formation delay

The response time for between the underlay creation initialization is an important aspect to measure. Especially this is vital when we want to create virtual networks that are mobile. In this experiment we measure the time between the user initialization & the completion of the underlay formation.

The completion is reported by the control CTP network running on the host networks. Each above experiment is run thrice & taken the average time in milliseconds.

$$\text{Completion time} = \Sigma \begin{pmatrix} time\ taken\ for\ beaconing + \\ time\ taken\ for\ metadata\ dissemination + \\ time\ taken\ for\ application\ code\ dissemination \end{pmatrix}$$

First version of the experiment is run when the nodes are approximately 7 inches apart. For the second version we brought all the nodes closer (3 inches apart) & executed the same above experiment. Here we can see a slightly high latency mainly because of the contention between nodes during transmissions.

Application	Time (milliseconds) (Sparse Deployment)	Time(milliseconds) (Dense Deployment)
Blink	3507	4263
Temperature sensing app with CTP	4855	5520

Table 6 Time taken for the dissemination

CTP overhead to establish tunneling

The basic idea behind tunneling (as described in the tunneling section) is simulating a sensor network radio link with an Ethernet or wi-fi link. In this experiment we measured the total CTP control message overhead to establish & maintain the tree. In the first step we used a 6x8 grid with a single collection tree running. In the second step we divided the same network (as in step1) into two 3x8 networks & kept them at a distance so that the motes belonging to separate networks are not in communication range. But both the networks were belonging to one collection tree. These two networks were only connected by an Ethernet tunnel. We observed at,

initially the motes in the network which does not have a root do not have parents selected until the tunnel is made. Once the tunnel is made, the tree starts to behave normally. To plot the following graphs we selected four motes at different locations of the network. By looking at the following graphs we can safely conclude that the overhead is negligible.

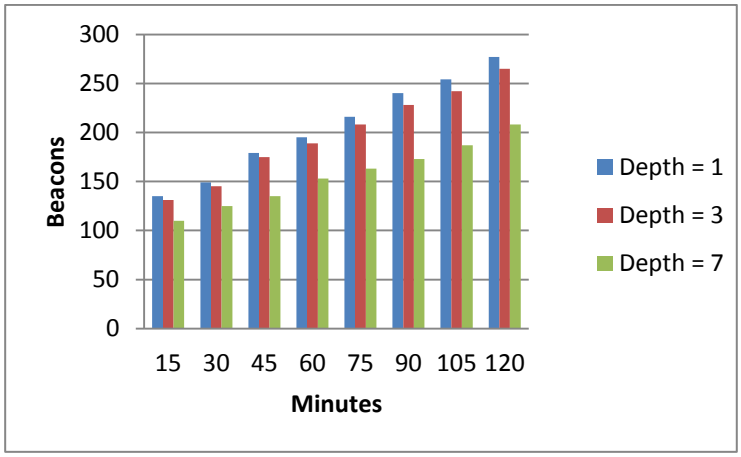

Figure 37 Single 6x8 network.

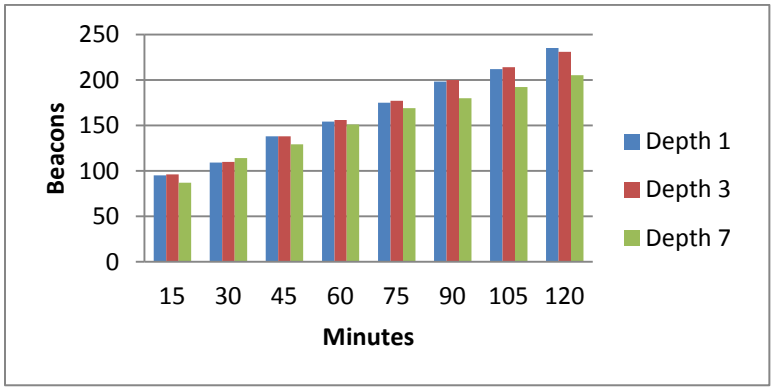

Figure 38 Two 3x8 networks

We hope to extend this experiment by bringing the 3x8 networks close by so that we can observe the CTP algorithm's behavior. In this case motes will end up having the tunnel & the radio to communicate with the other part of the tree.

CHAPTER 9

Future Work

During the development & experimentation of the prototype we have come across various ways to improve the architecture. A brief discussion of the main points is given below.

1. Energy profiling of applications – The main architecture has a component (not part of this thesis) which allows users to write sensor network applications. Currently there is no real mechanism to profile the energy usage of these applications. This can be done in the overlay or at the gateway. Poorly developed applications can drain the battery & make the entire network unusable.
2. Improve the data delivery mechanism – Features like frequency reuse, spatial multiplexing can be integrated to the protocol used to reduce the energy usage.
3. Improve the architecture so that the virtual network can be used for monitoring a moving phenomenon – Even though we have tested the creation & dissolution time, rapid creation & dissolution of virtual networks is not observed. Robust behavior during these testing will help in real world scenarios where the phenomenon is moving (such as smoke).
4. Improve the security [26] of the architecture – Since this is a proof-of-concept implementation, by default it does not employ any security mechanisms. But the nodes in the network need to accept install & run application code sent by their neighbors which can be exploited. It is necessary to at least have a basic security mechanism like a pre-shared key.

5. Run variations of the CTP tunneling experiments – This will help in understanding the behavior of the modified CTP protocol. The results can give more details about the improvements that can be made to the sensor network side of the tunneling implementation.

REFERENCES

[1] http://en.wikipedia.org/wiki/Wireless_sensor_network

[2] http://moss.csc.ncsu.edu/~mueller/rt/rt11/readings/projects/g4/datasheet.pdf

[3] http://www.tinyos.net

[4] http://mantisos.org/index/tiki-index.php.html

[5] http://www.contiki-os.org/

[6] Hartung, Carl, Richard Han, Carl Seielstad, and Saxon Holbrook. "FireWxNet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments." In Proceedings of the 4th international conference on Mobile systems, applications and services, pp. 28-41. ACM, 2006.

[7] http://docs.tinyos.net/tinywiki/index.php/TEPs

[8] http://code.google.com/p/tinyos-main/source/browse/trunk/apps/MultihopOscilloscope/MultihopOscilloscopeC.nc

[9] http://code.google.com/p/tinyos-main/source/browse/trunk/apps/tosthreads/capps/SenseAndSend/SenseAndSend.c

[10] Gnawali, Omprakash, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. "Collection tree protocol." In Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, pp. 1-14. ACM, 2009.

[11] Hui, Jonathan W., and David Culler. "The dynamic behavior of a data dissemination protocol for network programming at scale." In Proceedings of the 2nd international conference on Embedded networked sensor systems, pp. 81-94. ACM, 2004.

[12] Dang, Thanh, Nirupama Bulusu, Wu-Chi Feng, and Seungweon Park. "Dhv: A code consistency maintenance protocol for multi-hop wireless sensor networks." Wireless Sensor Networks (2009): 327-342.

[13] Liang, Chieh-Jan, Răzvan Musăloiu-e, and Andreas Terzis. "Typhoon: A reliable data dissemination protocol for wireless sensor networks." Wireless Sensor Networks (2008): 268-285.

[14] Culler, David, Deborah Estrin, and Mani Srivastava. "Guest editors' introduction: overview of sensor networks." Computer (2004): 41-49.

[15] http://research.microsoft.com/en-us/projects/senseweb/

[16] Gnawali, Omprakash, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. "The tenet architecture for tiered sensor networks." In Proceedings of the 4th international conference on Embedded networked sensor systems, pp. 153-166. ACM, 2006.

[17] http://research.cens.ucla.edu/projects/2007/Systems/Tenet/

[18] Tham, Chen-Khong, and Rajkumar Buyya. "SensorGrid: Integrating sensor networks and grid computing." CSI Communications 29, no. 1 (2005): 24-29.

[19] Ciciriello, Pietro, Luca Mottola, and Gian Pietro Picco. "Building virtual sensors and actuators over logical neighborhoods." In Proceedings of the international workshop on Middleware for sensor networks, pp. 19-24. ACM, 2006.

[20] Mike, Razvan Musaloiu-E. Chieh-Jan, and Liang Andreas Terzis. "HiNRG Technical Report: 21-09-2008 A Modular Approach for WSN Applications."

[21] Klues, Kevin, Chieh-Jan Mike Liang, Jeongyeup Paek, Razvan Musaloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. "TOSThreads: thread-safe and non-invasive preemption in TinyOS." In Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, pp. 127-140. ACM, 2009.

[22] Chan, Haowen, and Adrian Perrig. "Security and privacy in sensor networks." Computer 36, no. 10 (2003): 103-105.

[23] http://www.sensor-networks.org/index.php?page=0823123150

[24] McCartney, William P., and Nigamanth Sridhar. "Abstractions for safe concurrent programming in networked embedded systems." In Conference On Embedded Networked Sensor Systems: Proceedings of the 4 th international conference on Embedded networked sensor systems, vol. 31, pp. 167-180. 2006.

[25] http://www.capsil.org/capsilwiki/index.php/TinyOS

[26] Perrig, Adrian, John Stankovic, and David Wagner. "Security in wireless sensor networks." Communications of the ACM 47, no. 6 (2004): 53-57.

[27] Arch Rock Corporation, 2007. A sensor network architecture for the IP enterprise. In proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN 2007, Cambridge, Massachusetts, April 2007), 575. DOI=10.1145/1236360.1236447.

[28] Ringwald, Matthias, and Kay Romer. "Deployment of sensor networks: Problems and passive inspection." In Intelligent Solutions in Embedded Systems, 2007 Fifth Workshop on, pp. 179-192. IEEE, 2007.

[29] Fang, Qing, Jie Gao, and Leonidas J. Guibas. "Locating and bypassing holes in sensor networks." Mobile Networks and Applications 11, no. 2 (2006): 187-200.

[30] Arms, S. W., A. T. Newhard, J. H. Galbreath, and C. P. Townsend. "Remotely reprogrammable wireless sensor networks for structural health monitoring applications." In ICCES International Conference on Computational and Experimental Engineering and Sciences, Medeira, Portugal. 2004.

[31] Levis, Philip Alexander, Neil Patel, David Culler, and Scott Shenker. Trickle: A self regulating algorithm for code propagation and maintenance in wireless sensor networks. Computer Science Division, University of California, 2003.

[32] Lin, Kaisen, and Philip Levis. "Data discovery and dissemination with dip." In Proceedings of the 7th international conference on Information processing in sensor networks, pp. 433-444. IEEE Computer Society, 2008.

[33] Dunkels, Adam, Bjorn Gronvall, and Thiemo Voigt. "Contiki-a lightweight and flexible operating system for tiny networked sensors." In Local Computer Networks, 2004. 29th Annual IEEE International Conference on, pp. 455-462. IEEE, 2004.

[34] Han, Chih-Chieh, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. "A dynamic operating system for sensor nodes." In Proceedings of the 3rd international conference on Mobile systems, applications, and services, pp. 163-176. ACM, 2005.

[35] Jayasumana, Anura P., Qi Han, and Tissa H. Illangasekare. "Virtual sensor networks-A resource efficient approach for concurrent applications." In Information Technology, 2007. ITNG'07. Fourth International Conference on, pp. 111-115. IEEE, 2007.

[36] Krause, Andreas, Eric Horvitz, Aman Kansal, and Feng Zhao. "Toward community sensing." In Proceedings of the 7th international conference on Information processing in sensor networks, pp. 481-492. IEEE Computer Society, 2008.

[37] Ra, Moo-Ryong, Bin Liu, Tom F. La Porta, and Ramesh Govindan. "Medusa: A programming framework for crowd-sensing applications." In Proceedings of the 10th international conference on Mobile systems, applications, and services, pp. 337-350. ACM, 2012.

[38] http://docs.tinyos.net/tinywiki/index.php/Mote-mote_radio_communication#Active_Message_Interfaces

[39] Dimitriou, Tassos, and Dimitris Foteinakis. "Secure and efficient in-network processing for sensor networks." In First Workshop on Broadband Advanced Sensor Networks (BaseNets). 2004.

[40] Ceriotti, Matteo, Matteo Chini, Amy Murphy, Gian Picco, Francesca Cagnacci, and Bryony Tolhurst. "Motes in the jungle: lessons learned from a short-term wsn deployment in the ecuador cloud forest." *Real-World Wireless Sensor Networks* (2010): 25-36.

[41] http://www.planet-lab.org/

[42] Werner-Allen, Geoffrey, Patrick Swieskowski, and Matt Welsh. "Motelab: A wireless sensor network testbed." In Proceedings of the 4th international symposium on Information processing in sensor networks, p. 68. IEEE Press, 2005.