SNSP: a Distributed Operating System for Sensor Networks

Jana Van Greunen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-158
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-158.html

December 18, 2007

Distributed OS for Sensor Networks

by

Jana van Greunen

B.S. (University of California, Berkeley) 2002
M.S. (University of California, Berkeley) 2004

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Jan Rabaey, Chair
Professor Dave Auslander
Professor John Wawrzynek

Fall 2007

The dissertation of Jana van Greunen is approved:

Chair Date

Date

Date

University of California, Berkley

Fall 2007

Abstract

A Distributed Operating System for Sensor Networks

by

Jana van Greunen

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Jan Rabaey, Chair

Sensor networks are an exciting new technology that promise to revolutionize the environment we live in by creating ambient intelligent spaces. In the current software model, applications are statically loaded onto the network at compile time. This means applications cannot react to changes in the underlying network and code reuse is rare because applications are so tightly coupled with hardware. Sensor network deployments suffer from a lack of standard software APIs that allows applications to interact with the network.

This dissertation presents SNSP, a distributed service-based operating system for sensor networks. SNSP presents an integrated interface to applications, which abstracts the distributed nature of the underlying network. It enables dynamic application and content management in the sensor network. SNSP's core consists

of seven OS-level services that manage content, discover network resources, monitor resource utilization, dynamically map network applications, provide fault detection and recovery, migrate applications and implement security for the sensor network. Programmers can write services that become a reusable library of SNSP code. The dissertation outlines a programming language and integrated development environment for programmers.

Further, the mechanisms for content management and replication and task allocation (mapping) are studied in more detail. Three replication schemes are compared via simulation. Results indicate that the probabilistic scheme has the best performance in terms of cost per data access and increased data availability. Moreover, three task allocation schemes are also compared. The third algorithm, a hybrid genetic search and market bidding protocol, outperforms the other two algorithms. However, due to its twelve times higher computation and 51% higher communication cost the greedy algorithm is preferred.

As a proof-of-concept, SNSP is demonstrated on a TelosB and Mica2 (with TinyOS) testbed implementation. The testbed allows applications on different hardware platforms (Mica2 and TelosB) to coexist. Measurements from the testbed indicate that the content management algorithm lowers data access cost and also the time to map a process onto the network.

Jan Rabaey, Dissertation Committee Chair

To my mother and Sterling

Contents

List of Figures

List of Tables

1 Introduction

Widespread sensor network deployment has been hampered by lack of a standard hardware abstraction and matching API for software development. For current sensor network deployments, each deployment is a one-off development effort. The software and hardware are developed together and the sensor network is then deployed in the field. When the application changes or new applications need to be deployed, the nodes are typically retrieved from the field and then new nodes are deployed[1]. If sensor networks are to become a ubiquitous part of infrastructure in smart spaces, the sensor network platform needs to be more extensible and flexible. Usage scenarios include hospital health monitoring, and energy monitoring (heating, ventilation and air conditioning HVAC) in buildings. In the first scenario, health monitoring, patients and diseases may change every day. The application in a particular room may change

[1] For small changes it is possible to reprogram nodes over the air. However, this reprograms the entire image and requires nodes to reboot

as a new patient enters the room and sensors and actuators may leave and enter the room with a patient. Further, there are some concerns for patient privacy and securing information retained on the sensor nodes. In the second example, applications and requirements may change with regulations or as new companies rent out the buildings. Health and indoor energy monitoring will be used as examples throughout this dissertation to provide illustrative examples of the flexible sensor network abstraction benefits.

While there have been successful abstractions for the individual sensor node hardware, for example, IEEE's sensor interface standard [1] and the instruction set abstraction defined in [2], abstractions for the distributed network are only now starting to emerge. This research presents the Sensor Network Services Platform (SNSP), a distributed operating system abstraction for the sensor network.

SNSP's goal is to allow an application developer to create modular portable code for sensor networks and to avoid building the application from the ground up every time the sensor network changes. Further, SNSP will help harness the potential benefits provided by distributed systems. The advantages of distributed systems include:

- Reliability/fault tolerance
- Improved resource utilization
- Scalability

SNSP will transform the sensor network into a truly smart environment. It describes a platform based on a set of clean abstractions for users, content, and devices. The platform's goals are to minimize device configuration and to adapt to different users in the environment. SNSP will be part of the sensor network infrastructure that remains in a building or location even if the people and applications change.

As an operating system, SNSP manages the system resources, specifically, it performs resources allocation e.g. memory and computation, allows multiple programs to co-exist, controls input and output devices and manages files or content. Further, SNSP will execute on a physically distributed platform and thus implicitly supports communication, concurrency, and synchronization, providing location transparency. SNSP provides basic, low-level functions for controlling parallel program execution, i.e., functions for loading a parallel program, for starting, halting and resuming. In addition, the notion of users is included in SNSP. As in traditional operating systems, a user's rights determine which programs, resources, and content they may read, write, and execute.

A service-oriented model is chosen for SNSP to separate the *function* of network services (i.e., what the services do) from their implementation, creating a reusable layer that can be implemented on any hardware platform. There are two kinds of services in SNSP: (1) Operating-system services, and (2) User-defined services. These services play a crucial role in abstracting the specific hardware

platform from the application, presenting any information required by an application through a unified service API.

From the application programmer's perspective, SNSP presents a familiar sequential programming model. The programming model is supported by pre-defined services that can be invoked to run in a parallel and distributed fashion. To facilitate programming, SNSP supports a parallel programming language, sensC, and environment abstractions. SensC is syntactically ANSI C with the addition of a few specific features, which are outlined in Chapter 4. At a high-level, this language supports parallelism up to the granularity of a process. Every application has at least one main process that executes on a single processor, when an application invokes a service, a second "process" is spawned which may be executed simultaneously anywhere in the network (constrained by timing, resources etc.). When application programmers write services, the service/application model provides the way for programmers to explicitly partition the program into parallelizable pieces. SNSP dynamically and automatically finds a place to execute these application pieces.[2]

This dynamic programming model is a departure from the static one normally used in sensor networks [3]. However, in examining the application space, it is clear that there is use for dynamic migration of programs. Mobile applications were first used in pursuer-evader games [4] where the processing follows the

[2] SNSP does not support implicit application partitioning.

evader through the network, i.e. nodes closest to the evader do the sensing and processing. In this case it is the same application and the same sensing capabilities that are activated in different regions of the network, so it could be programmed in advance. However, both applications and sensor network nodes are becoming more dynamic. Take for example healthcare applications. Change occurs when a new disease is detected, a different sensor is added, a different diagnostic test is devised, or when the patient changes location. All of these changes must be seamlessly incorporated into the system. The distributed operating solution is designed to manage these changes without explicit user interference.

SNSP, as an operating system with a built-in set of services, presents a departure from the usual design principles applied in networking. Typically, the network is designed to be 'dumb' and intelligence is placed only at the edges. The protocols used to communicate across the network are designed end-to-end [6], in other words, there is end-to-end reliability and end-to-end addressing etc. This traditional design has worked very well with powerful and stationary end-devices. However, with a more heterogeneous and mobile network, there is a push toward breaking end-to-end semantics and embedding more intelligence in the network to relieve the pressure of processing from the devices. Due to the constraints on the sensor nodes, and possible mobility of nodes, SNSP represents a similar break in end-to-end semantics.

Before describing SNSP in more detail, it is necessary to examine the goals of SNSP and state the underlying assumptions.

1.1 SNSP Goals

First, SNSP abstracts the internals of distributed sensor networks from applications and encourage sensor network deployment. There are three specific goals that SNSP aims to achieve:

- Enable applications to execute on different network configurations (agnostic of hardware) which will allow heterogeneous sensor network platforms to interoperate
- Shield applications (enable recovery) from node/hardware failures
- Optimize application execution by intelligently mapping the application onto the network (may involve run-time migration), which saves energy by reducing communication, increasing reliability and/or performance

SNSP is a general platform/middleware that must execute on all sensor network platforms, and therefore may have access to only a few parameters from any given platform. The following assumptions were made:

1.2 Assumptions

- The network contains heterogeneous nodes, which are globally asynchronous and run a kernel of SNSP middleware.

- SNSP is above the network layer, it cannot change the routing, link, or physical layer protocols.
- SNSP is below the application layer and does not know any application semantics.
- From the application's perspective, SNSP exposes geographic addressing, but for each hardware platform this is translated into the local addressing/routing scheme.
- SNSP knows/may observe the maximum and average point-to-point throughput (Mb/s) between two neighboring nodes in the network.
- There is a standard hardware abstraction for each sensor node which is:
    - MIPS number for CPU
    - Dynamic memory in KB
    - Storage for data in MB
    - Hardware (sensors/actuators)

Because SNSP does not control the underlying communication medium, there is inherent non-determinism and SNSP cannot guarantee communication latency. Moreover, guaranteeing correctness and consistency in an asynchronous system adds a considerable overhead that is not warranted for every sensor network deployment. Therefore, this is not part of the basic SNSP platform. SNSP adopts the *BASE* semantics: Basically Available, Soft State, and Eventual Consistency,

which were first introduced in [7] to describe distributed web-caches. BASE semantics are adequate for applications that can tolerate delays on the order of a second.

1.3 Research Contributions

The main contribution of this research is the design of SNSP. SNSP is a full-fledged operating system with memory management, location transparency, and resource allocation. SNSP enables the creation of applications that can be mapped onto the network at runtime and allows the programmer to build up a library of reusable sensor network services.

Further, the research focuses on two aspects of SNSP, file allocation and tasks allocation. These were chosen because they have a significant impact on sensor network performance and they have not been adequately addressed in existing work.

The contributions of the thesis are:

- Identified and designed the basic set of services for a distributed operating system
- Devised a programming model and user-interface that allows users to create reusable libraries of code
- Developed a novel file allocation algorithm
- Evaluated the performance of three file allocation algorithms via simulation

- Developed two novel task allocation algorithms
- Evaluated the performance of three task allocation algorithms via simulation
- Proof of concept implementation of SNSP on top of a TinyOS sensor testbed with select performance measurements

The rest of the thesis is organized as follows: Chapter 2 describes related work in sensor network operating systems and middleware. Chapter 3 outlines the architecture of SNSP, namely the abstractions and services that it provides programmers. The user interface and programming model is described in Chapter 4, followed by a problem formulation and detailed analysis of file allocation algorithms in Chapter 5. Chapter 6 presents a similar analysis of the task allocation problem. Chapter 7 describes SNSP implementation on a 30-node testbed, running TinyOS as the underlying operating system.

2 Related Work

The related work on sensor network platforms can be divided into three categories: 1) The single-node operating system, which focused mainly on small footprint and code size. 2) Sensor network middleware, which attempts to abstract commonly used functionality and standardize it. 3) Fully-fledged distributed operating systems for sensor networks. The distributed operating systems take the goals of middleware platforms a step further by not only providing a set of common functionality, but also controlling and optimizing the execution of applications on the network. They provide real-time coordination and control of the sensor network. Research in the three categories is outlined below.

2.1 Single-Node Operating Systems

The most popular single-node operating system is TinyOS [8]. TinyOS consists of a set of software modules in the NesC language [9]. Components are not divided

into user and kernel modes and there is no memory protection. TinyOS does not support preemption. The code is statically linked at compile time. Version two of TinyOS introduces some lower-level improvements to abstract the platform from the hardware. TinyOS is very widely used in sensor network test-beds and in published research results. Thus, it is an ideal platform to develop software on that other groups will use or compare results with. TinyOS was selected as the basis for the implementation platform as described in Chapter 7.

BTNodes [10] were developed by ETH Zurich. These nodes have their own hardware platform and operating system, called BTNut. BTNut is a multithreaded preemptive operating system written in C. It also has a TCP/IP stack built in. BTNut is the closest to other commercial RTOS systems, but it has not had the same traction in the academic setting as TinyOS.

MANTIS [11] is an open source, multi-threaded operating system written in C for wireless sensor networking platforms. It has automatic preemptive time slicing for fast prototyping and an energy-efficient scheduler for duty-cycle sleeping of sensor node. Another interesting sensor network operating system is SOS [12], which supports dynamic application modules. Application modules can be loaded or unloaded at run time. Modules send messages and communicate with the kernel via a system jump table, but can also register function entry points for other modules to call. SOS is a more extensible platform

than TinyOS, but it is not as widely adopted, and it supports fewer hardware platforms.

2.2 Sensor Network Middleware

Research on sensor network middleware aims to abstract common functionality and present it in reusable form. There are four main approaches to designing middleware for sensor networks: 1) Database, 2) Publish-subscribe 3) Services, 4) Mobile agents/clustering. The database involves abstracting the sensor network as a database and then allowing a user to write SQL-like queries to extract sensor data from the network. TinyDB [13], Cougar [14], and SINA [15] are the most well known of these approaches. The research focus here is on efficient query processing and routing. [16] writes wrappers for the underlying sensor network (single-node operating system) and then use an XML framework for syntactic description and then SQL for data manipulation.

The second approach, publish-subscribe, disconnects the data production from its consumption. A set of data producers publishes their data in the network. Interested parties can then subscribe to the data. [17] presents a set of operators that interested parties can use to describe patterns of data that they are interested in. Typically somewhat complex time stamping is needed to get any useful data. Mires Middleware [18] is another publish-subscribe mechanism. Initially sensor nodes publish the types of data that they can provide. Client applications then

select the data that they care about and subscribe to it. Only after receiving a subscription will sensor nodes publish their data. There is not much data available on how expensive the subscription language is, or what overhead it adds to the network. DSWare [19] is not a traditional publish subscribe mechanism, but it does cache data in the network and it provides a language for other nodes to subscribe to events.

The Milan platform [20] is an example of a service-based approach. Applications wishing to execute represent their requirements to Milan through specialized graphs that incorporate state-based changes in application needs. The graphs contain variables and QoS for each variable. Milan knows the level of QoS that data from each sensor or set of sensors can provide for each variable. For a given application, Milan can send and receive data from these variables. There is not much data on how the matching between applications and resources is done. [21] defines a set of roles in the sensor network that have rules associated with them. An application must choose one of these roles to play and the network knows how to load balance the application based on the role it is playing. Sensorware [22] defines a set of base services that the network must provide to an application. However, it does not define the mechanisms by which the network must deliver these services. These services are as follows:

- Communications Protocol Stacks
- Power Management

- User Interaction
- Network Synchronization
- Query Processing
- Configuration (e.g. health status and maintenance)
- Fault Tolerance
- Security/Authentication

The last type of middleware is agent-based. In Agilla [23], each application is a mobile agent that decides independently when and where to migrate. Agilla is based on the mate platform [24]. Nodes have Linda-like [25] tuple spaces and an acquaintance list that allows them to find a place to migrate. The shortcoming of this approach is that there is no attempt to achieve network optimality or analysis of what happens when there is inter-agent interaction.

2.3 Sensor Network Operating Systems

The first two operating systems presented both rely on applications that are comprised of identical code fragments, in other words, applications that can be decomposed into smaller, identical sub-problems. In [26] the main abstraction is that of Microcells. Microcells begin their life-cycle as inactive software components and are activated by stimuli in the environment, as well as by events in the computing system. The programmer must specify how a microcell reacts to stimuli. Once a microcell is activated, stimuli can cause it to perform self-

replication, migration, or grouping. For example, growth may be controlled to reach the required computing capacity or geographic coverage for different functions performed in the vicinity of an event. The distributed operating system determines where and when microcells execute. The system implements its own microcells, which implement fundamental activities in the infrastructure, such as information storage. [27] outlines Bertha, an operating system that can accommodate 11 process fragments at a time. Bertha manages processor startup, memory, access to hardware peripherals, communication with neighboring Pushpins, and provides a mechanism for installing, executing, and passing neighbors code and data to execute. The main drawback of these two approaches is that they only fit applications which can be broken down into small homogenous pieces.

Kaizen [28] focuses on a resource usage description and sandboxing strategy. CPU, memory, radio, and sensor/actuator usage contracts are outlined, in addition to methods to stop processes from consuming more resources than specified in their contracts. However, they do not address the optimal placement for processes or other services that may aid process execution.

The Eyes project [29] aims to achieve small code size and to solve the limited available energy problem by defining clear stop and start portions in the code after which the processor can be put to sleep. Nodes are able to make resources requests via remote procedure calls to their directly connected neighbors when

they do not have enough energy or processing power to fulfill a certain task. EYES also defines a distributed services layer which provides a lookup service and an information service. The lookup service supports mobility, instantiation and reconfiguration. The information service collects network data. The drawback of eyes is that nodes can only make local resource requests and they do not support a full mapping of applications onto the network.

[30] defines a service manager that is responsible for receiving and accomplishing service requests. There are two primitive request types: query and order, and two complex types: conditional and repetitive. The service manager uses the distributed service directory (DSD) to find a pairing for the request. Their DSD uses S-CAN [31], a distributed hashtable developed for content-addressable networks. The service manager and DSD is similar to the mapper and repository service presented in this research, but the services presented here can deal with a richer set of applications.

The last two distributed systems are the most similar to SNSP in their goals and generality. The [32] design separates the core OS kernel, OS services, and distributed applications. The core kernel and the OS services control the behavior of a single node, while applications implement the distributed system behavior of the sensor network. Applications consist of processing elements connected via arcs. As SNSP does via equivalence classes, they allow for runtime reconfiguration of the model, including changing interconnections and

replacement of processing elements (PEs). Communicating PEs can be located on the same, or different physical nodes. There is not much data on how Omni achieves placement of applications, a core aspect of the system that this research focuses on.

Last, Magnet OS [33] provides a single system image of a unified Java virtual machine to applications. It partitions applications components statically along object interfaces. Magnet OS then finds an initial placement for these components on the network. MagnetOS uses the Java RMI interface for remote object invocation during run-time. During execution, two services adjust task placement to optimize communication cost. NetPull migrates components one hop at a time in the direction of greatest communication. NetCenter migrates components to the host that a given object communicates with the most. Magnet OS conceptually has the same goals and achieves them in a similar way as SNSP. However, it has the overhead of a full Java implementation in addition to not being able to reuse or share Java classes among applications at runtime. Note, classes are shared at compile time. SNSP's application construction is specifically designed so that applications can use existing components on the network.

3 Distributed OS Architecture

SNSP is a distributed operating system. SNSP code executes directly on the nodes in the sensor network. Each node runs its own native operating system and SNSP code runs as middle-ware on each node. SNSP is designed to run on a heterogeneous sensor network; a network comprised of nodes with different capabilities e.g. battery-powered nodes, constantly powered nodes and perhaps different bandwidth availability. It can also operate on a homogenous network, and a totally asymmetric network, where the base-station does all processing and the other nodes simply collect data.

As an operating system, SNSP manages system resources, performing resource allocation e.g. memory and computation, allowing multiple programs to co-exist, controlling input and output devices and managing files/content. Due to the nature of sensor networks, SNSP will execute on a physically distributed, heterogeneous platform and thus implicitly supports communication,

concurrency, and synchronization, providing location transparency. As in traditional operating systems, a user's rights determine which programs, resources, and content they may read, write, and execute.

There are several ways to design the architecture of a distributed operating system. The most common approach is the micro-kernel approach. In this approach, each component that comprises the distributed computing system runs the same kernel of code that takes care of basic services. A more flexible approach was desired for sensor networks as the entire computing system is not designed at one time, and parts may be incrementally added. Therefore, SNSP is actually middleware that runs on top of a basic operating system for each node. When SNSP layer is ported to a new operating system, it must be able to interface directly with that operating system and also be able to translate the routing and addressing.

SNSP is not only a distributed operating system for sensor networks, it also provides a uniform abstraction for applications running on it. This abstraction encompasses a common terminology to refer to applications, resources and people in the environment. An abstraction set was included in SNSP because sensor networks are intimately tied to the environment, and being able to compactly present aspects of the environment is essential to the success of reusable and general sensor network software.

A service-oriented model was chosen for SNSP to separate the *content* of network services from their implementation. These services form a reusable layer that can be implemented and shared across any hardware platform. The core layer of SNSP is comprised of services that handle concurrency, file allocation, security and resources. These services are a standard part of SNSP and are described in detail below. However, SNSP is also extensible via user-defined services. User defined service are meant to encapsulate higher-layer functionality so that they can be re-used across different sensor networks. A specific API is provided that allows users to write these modular components (or applications). The API is outlined in this section and further elaborated on in Chapter 4. Figure 1 shows a graphical representation of SNSP architecture.

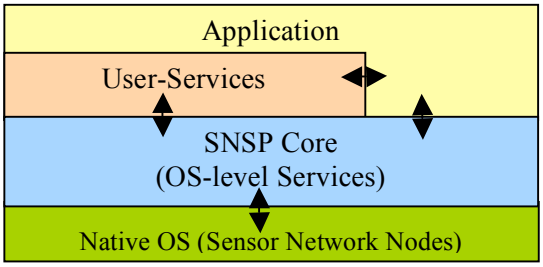

Figure 1: SNSP architecture layers.

As an example scenario, healthcare monitoring will be used throughout this chapter. For healthcare monitoring the sensor network is installed in a hospital. Patients stay in the hospital for an average of three days. During the course of their stay, different sensor nodes are attached to their bodies. These nodes monitor different vital statistics, e.g. blood oxygenation, blood pressure, heart

rate, blood sugar level, breathing speed, stress levels etc. There is also a stationary sensor network in the hospital. These nodes may monitor the environmental conditions in the room (temperature, humidity), or be attached to larger machines, or to a system that alerts doctors and nurses when a patient needs them. As patients enter and leave the hospital, the applications executing change in accordance with the diseases being monitored. For example, the sensor network may be monitoring a patient for seizures. This application could combine heart-rate monitoring with stress level monitoring and it may also involve administering of certain anti-seizure drugs. Further, the application may require fine sampling of data from the sensors and thus it may be better to execute the control function locally, rather than route a large amount of data to a central base-station in the hospital. In addition, the data that the sensor network collects about the person must be interpreted and stored correctly.

SNSP abstractions are presented below, followed by a description of OS and user-level services.

3.1 Abstractions

There are three main abstractions in SNSP environment: *Personae*, *Content*, and *Resources*. These abstractions were first presented in [34]. First, a persona represents a single person, groups of people, or organizations (*e.g.* nurses organization). In addition, a persona with sufficient privilege may set rules of

operation for the environment (*e.g.* only nurses may have access to the medicine cabinet). A persona may be present or absent in a certain environment, but can still affect its operation even if they are not present. The run-time system uses personae rules to interpret, act on, and resolves conflicts between multiple users.

A persona consists of the following components:

- **Permissions** are user's access rights to devices, content, services, and applications. For example, permissions may limit a user's ability to control devices in the home such as lights. Further, permissions specify a persona's *category* or *priority* (e.g. ability to override another persona). Persona categories are described in more detail in the *equivalence* section
- **Properties** contain information that describe user(s), e.g. age or gender. Properties may help the system to detect and distinguish users from each other. The persona also has an authenticate function.
- **Preferences** contain information regarding user actions or default system configuration. Preferences may also be used to enforce certain user behavior. For example, in health monitoring it is important to determine whether a person has taken the correct medicine each day and to encourage them if they have not done so.

In order to make personae more concrete, an example persona, describing a homeowner is shown below:

```
Class: Persona
Name: Homeowner
Permissions: r,w,x for content, programs, devices
Properties:
      Sex: Female,
      Age: 60
Preferences: -
isPresent(location);
```

Table 1: Persona Example.

Second, the concept of *content* abstracts information that services and application can manipulate. Content may represent a range of data: media streams, sensor readings (light, temperature, motion, identification), security information, energy monitoring results, health data *etc.* Separating content from the sensors that generate it, and actuators that consume it, allows the system to cache and replicate content to provide fault tolerance. In SNSP, all content has unique identifiers, a size field and properties. In the health monitoring example, content is very important. The content must be stored to provide doctors with accurate medical data to look for potential problems as well as a record of patient care especially of medicines taken. It is important that this content is not lost in the sensor network. Content must also be kept private.

Last, *resources* uniformly abstract physical resources in the environment. Resources are typically categorized by functionality (*e.g.* sensor –"*current temperature" content source*, routing node – *connection from A to B*, transcoder – *conversion Celsius to Fahrenheit*). Resources do not have an invocation/execution

API. For each resource there is a corresponding service that indicates how to utilize it. There are two main types of resources: The first is a specific physical resource, for example a sensor or actuator. The second is computation and connectivity resources. The physical resources have an input and an output domain. The input/output domains may be physical (lumens, heat to measure temperature) or they may be numerical, ie reading from an on/off switch. A resource also has a description of how much connectivity, computation power and memory it can provide. An example of a heater is shown below followed by an example computation resource that has the heater attached to it:

```
Class: Resource
Name: Heater
Location: LivingRoom
Properties:
      Manufacturer: GE
      Power: 1000W
Input Domain: numerical (10 bit)
Output Domain: physical (heat)
```

```
Class: Resource
Name: Computation
Location: LivingRoom
Reliability: 99.9%
CPU: 1 MIPS
CPU Used: 0
Dutycycle: 10%;1 second
RAM: 10KB
Memory: 100MB
Current used: 10MB
Connectivity: 10kbit/s
Hardware: Heater
```

Table 2: Two resource examples.

3.2 OS-level Services

OS services are the lowest level services that keep track of the system at the device and connectivity level. OS services play several roles in the system, they:

- Resolve resource allocation conflicts between applications or persona.
- Support discovery of resources, persona, and content in the system
- Track of the utilization and availability of resources, persona, and content.
- Replicate code and content in the sensor network to maximize availability and minimize data access costs
- Map applications onto the nodes at runtime
- Detect application failure and take corrective action
- Manage security and trust for the system resources, persona, and content

These functions have been divided into seven services shown in the figure below.

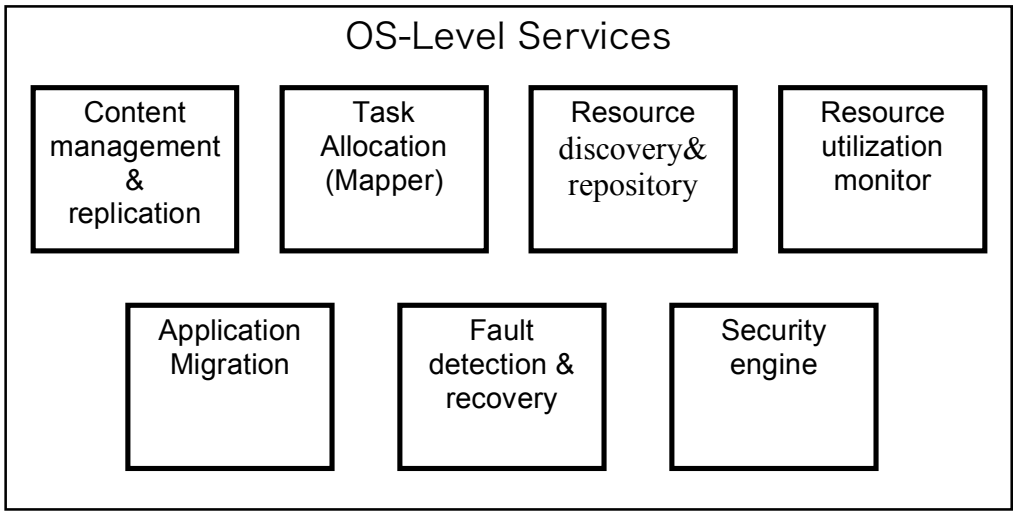

Figure 2: SNSP OS-Level Services.

3.2.1 Content Management & Replication

The content management and replication service distributes content throughout the sensor network. The underlying challenge providing this service is to optimize content placement for availability and minimize communication (both access and update) costs. This problem is known as the file allocation problem (FAP) and has been extensively studied for traditional distributed databases. The FAP may be based on static allocation or dynamic allocation; the access patterns may be deterministic, probabilistic or unknown. All versions of the FAP are NP complete [35]. This thesis evaluates two well-known file allocation algorithms in addition to developing its own. A formal problem statement as well as simulation results are presented in Chapter 5.

3.2.2 Task Allocation

This problem is closely related to the file allocation problem; in fact, the file allocation problem can be turned into a task allocation problem. The tasks comprising the application are the files. The application specifies an access pattern between the files. The tasks have additional constraints: namely, computation, dynamic memory, hardware, and location and bandwidth constraints. The optimal task allocation has been shown to be NP complete [36]. However, there is a large body of research on heuristic task allocation algorithms. This thesis presents and compares three heuristic task allocation

algorithms in the sensor network setting. The formal problem statement, with detailed constraints, is outlined in Chapter 6.

3.2.3 Resource Discovery & Repository Service

In SNSP we have chosen to do reactive resource discovery. The system does not proactively send out discovery messages, instead, nodes send out a register message when they join the distributed system. The register message contains information about the resources present on the node, as well as a content and/or service code that may be present on the node. This information is wrapped up as *repository content* then stored in the network. The distributed content management and replication service decides where to store this repository content. Further, the repository contents have soft-state. Thus, nodes must periodically re-announce their presence, otherwise it is assumed that they have died or left the network.

Querying the repository service accesses the repository contents. The repository service provides the following types of information:

- Available content, personae, and resources
- Available services in the network and their API's

3.2.4 Resource Utilization Monitor

The resource utilization service gives information about resources that are used by applications and services currently executed in the network, in addition to the

unallocated resources remaining in the sensor network. The mapper and content management service use the resource utilization service to determine what is available for allocation. The resource utilization service receives the resource record from nodes that send their periodic registration updates (as part of resource discovery). This provides the service with information about currently available resources.

The resource utilization service also works with the mapping service to keep track of resources that have been allocated to processes, but that have not yet been "consumed". To facilitate this, SNSP has a coupon system. A coupon is placed on each resource that the mapper decides to allocate to a process. Once the process has been initialized, it "consumes" the coupon and the nodes' resource state gets updated. The resource utilization service also keeps track of coupons issued to resources to assure that a resource is not over-provisioned. Resource utilization records are distributed and managed in the same way that content is.

3.2.5 Application Migration

To enable applications to migrate from one node to another node, the application's state must be captured. This is known as checkpointing an application. When the mapper wants to relocate an application, it will request that the application checkpoints itself. Note, both applications and services may

migrate from node-to-node, thus the application/service will be referred to as a remote process.

Checkpointing involves the creating of a checkpoint object. The checkpoint object stores the remote process's current execution position, any local content that may have been generated as part of the process state, the services that the process are currently using (including specific queries to these services that the process has made and not yet received responses to), and services/applications that have sent queries to the process (there are referred to as *waiting services*). This state is captured in the RemoteProcess object. The interfaces for the RemoteProcess and checkpoint objects are given in Table 3.

```
Class: RemoteProcess
ID: XXX
Location: XXX
start(void* args);
stop(void* args);
suspend();
resume(location);
servicesUsed(name, loc, queries);
waitingServices (name, loc, queries);
stateContent(content);
processDescription(ast); //abstract syntax tree rep of proc code
processStackPointer(void *);

Class: Checkpoint
ProcessID:
commit(); //update complete
initialize(processID);
recover(); //returns last committed checkpoint
recoverLast(); //returns last data
update(void* args); //updates part of a remote process e.g. stack
ptr
```

Table 3: RemoveProcess and Checkpoint objects for application migration.

3.2.6 Fault Detection and Recovery

The advantage of distributed systems is that it allows applications to be more tolerant of partial failures (there is redundant hardware for applications to recover from failure or faults). Due to homogeneity of different platforms and various energy constraints in sensor networks, it is not practical to require a universal fault tolerance/recovery standard for sensor networks. However, SNSP supports three loosely defined levels of fault tolerance: recoverable processes, fault detecting processes and none. Recoverable processes have the maximal support from the underlying network to both detect and recover from faults. Fault detecting processes will simply be restarted when a fault is detected, but are not guaranteed to preserve state. Last, there are processes for which the system guarantees no tolerance at all.

The first step of fault tolerance is fault detection. Due to energy constraints, SNSP does not, by default, actively monitor for faults in the network. Instead, it relies on a combination of information from the application and the resource discovery/utilization services. First, SNSP provides applications with a mechanism to signify a fault in one of its sub-processes (i.e. a service that it has invoked) Given the application composition (the task graph described in Chapter 4), a component that is using a service can naturally detect if a fault has occurred (e.g. data is not arriving according to specification or an incorrect service was

invoked). This mechanism can also be used to validate semantics of requests in the system.

The second method of fault detection is relying on information from the resource discovery and utilization services. Through these services, SNSP keeps track of resource states in the network. A variety of faults, i.e. nodes dying, nodes not executing a process, etc., can be detected by combining information from these two built-in services.

Once a fault has been detected, the system must recover from the fault. From the fault detection information, SNSP will know what process failed. If that process is still executing, it will immediately be terminated. If the program is a recoverable process, SNSP must resume the process instead of simply re-starting it. Recoverable processes must create checkpoint objects (described in the application migration service). The processes determine how often to update their checkpoint objects. The checkpoint objects are stored and replicated by the content management service to ensure that an accessible copy exists in the network. Once SNSP has located the process's checkpoint, the process is remapped onto the network.

The re-mapped process starts executing again from its last checkpoint. However, the other tasks in the application may receive old or duplicate information from the re-mapped process, which may lead to race conditions. In order to avoid this, SNSP notifies all tasks belonging to an application when the process is re-

mapped and provides them with the checkpoint sequence number. Further, it is recommended that communicating tasks include sequence numbers in the packets they exchange.

For processes that are fault detecting, no checkpoint object exists, so the process is simply remapped and restarted by the mapper service.

3.2.7 Security Engine

As in [34], the security in SNSP is based on *persona* and their access permissions. Personae control access to content and are able to set permissions dictating what can instantiate processes on the sensor network. A full-blown implementation of this security is left as future work for SNSP. There has been other work done on authentication, encryption and privacy in sensor networks, [37], [38], and [39] serve as excellent staring points for a secure system.

3.3 User defined services

The service description can be used to determine how a service should be replaced on failure and to check compatibility if a user wants to use or extend a service. The service description includes a high-level description of what the service does, and other properties that are useful in defining the service. The service API consists of several components; a brief description of these components is given below, followed by more detailed subsections.

Services have three stages of operation: initialization, execution, and termination. Each of these phases consists of a usage API and a high-level functional representation. When services are invoked, their initialization code is executed. During the execution stage applications and/or other services may query the executing service for content, or actuation of a resource. Queries and actuations are equivalent and handled through the same API. In addition to the three stages, services have structural, usage, and performance properties. Structural properties pertain to the service composition. Usage and performance properties are collected and stored in the repository service whenever the service is executing. These properties are expounded on and illustrated below.

3.3.1 Initialization

Initialization is an important part of setting up an application or preparing the system to record data. Initialization may contain several functions, e.g. turning resources on, calibrating sensors etc. Every function has an identifier and an API for calling the function. In addition to the functional API, there is also a description of what each function does (computationally), called the behavioral task. In the health monitoring example initialization might involve calibrating a particular sensor e.g. oxygenation or heart monitoring sensor to work with a particular patient.

3.3.2 Execution

The execution stage also consists of two main components: the service invocation API, and the service behavioral task description (computation description of each function in the API). The service invocation API refers to a set of queries that you can make to the service once it is running. It consists of a set of functions with typed arguments and return values. The basic functions through which a service module interacts with its environment is depicted in Figure 5 and an example service is given in Table 4 and Appendix I.

3.3.3 Termination

Similar to initialization, termination may consist of several functions, which are represented as an API and a behavioral task component. In the health-monitoring example, termination may happen when the patient is discharged from the hospital. It could involve recording all patient data in a permanent database and then erasing it from the sensor nodes.

3.3.4 Composition

These properties specify how a service is "put together". They are listed below:

- **Resources** This component specifies the type of sensor/actuator that is required and also other resources (ADC, bus etc.) that will be used during service execution. Further, computation and connectivity are also resources. Resources may be specified as a particular entity, or as an *equivalence class*.

Equivalence classes are explained in Section 4.2. In health monitoring, a resource may be the sensor required to monitor a particular patient's condition. A temperature sensor may be needed to monitor the patient's temperature. Temperature sensor is the equivalence group, and specific incarnations of it may be an oral temperature sensor or an inner-ear sensor etc.

- **Service Structure** A service can be either *simple* or *compound*. Simple services are self-contained and do not invoke any other services during execution. In contrast, compound services do invoke other services. If a service if compound, the service structure must also contain a list of the sub-services that are invoked during execution. These sub-services may be specified according to their equivalence class. A compound service's access restrictions must be a superset of all the access restrictions of its sub-services. For example, a seizure monitoring service may use inputs from the heart-rate sensor, the skin-moisture sensor, whereas a simple patient-temperature monitoring service may only use a temperature sensor. Note: these two services may themselves be part of a larger patient wellness application that is launched on patient arrival.
- **Service Scope** defines the scope, location and time, that a service can operate in. This is not the instantiation scope that is passed to the service as an invocation parameter. Rather, it is the scope that it is *possible* for the service to

operate at all. The service may be limited in location because certain services are provided by individual pieces of hardware. This hardware may be restricted in space and also in the time of usage. Also for certain services, it makes no sense to measure during certain times, e.g. nocturnal activity during the day or photovoltaic cell power generation in the night.

- **Service Content** Services generate results when they are executed. These results are classified as "content". Content may be stored in the network and used for later reference or consumed immediately. Caching may be used so the service does not have to execute each time it is invoked, rather, it may return already stored content. Further, a personae who instantiates a service may also impose privacy restrictions on the content (e.g. can it be shared with other personae or other services). For example, content may be a history of the patient's heart rate over 1 second intervals.
- **Security & Access** All services in the sensor network are controlled (and instantiated) by a persona or a group of personae. Access to service information (read/write) is determined by these personae. This component grants or restricts the access of certain personae to the service description. As an example, medical information may be accessed by the patient's direct family only, and not by other visitors. Further, the restriction is specified on a component-by-component level, it may differ across individual parts of the service description**.** Another important security constraint is instantiation

rights (execute). Personae can be denied access to instantiate a service in the network. Read/write and execute privileges are specified independently of each other. For compound services (calls other services to complete its result), the service's security must be at least as strict as its sub-classes. The security and access components are limited only to the service description. We assume that data encryption and other authentication is done by the service during runtime.

3.3.5 Performance Metrics

These are the performance metrics that the underlying network must have for the service to complete successfully. The performance of a service can be broken down into the following components. Each of these can be specified as a max or min value:

- Delay
- Synchronization (order on events & to reference)
- Accuracy
- Reliability (exactly one delivery, at most one delivery, best effort … etc)
- Throughput (network bandwidth)

If an application requests a performance level that a service cannot meet, or the service itself requests a performance level that the underlying network cannot meet, SNSP will simply send the application an error message.

3.3.6 Usage Measures

This section keeps statistics of the number of times a service is accessed. Also, statistics are kept just for each incarnation of the service definition, i.e. if two service definitions differ in any component, they must have different usage measures. The usage measures are of the following:

- **Access Count** indicates how many times this service definition is read, written, or executed. This is for unique accesses by different services/personae.
- **Alive Count** of how many alive/executing copies there are in the network. The repository makes a distinction between simply accessing the service definition, and accessing it to execute/instantiate it.
- **Validity** Time period specifying how long the current copy is valid for, e.g. 1 hour, 2 days. This time period specifies the refresh rate, entities (that are executing, or using the service definition) must check with the repository to see if the service has been updated. This requirement allows programmers to keep the code in the sensor network up to date and allow updates to propagate through the network in a timely manner. Note: when nothing changes, the "check-back" is only a control message exchange (check version #), so its overhead is relatively small.

A summary of the different parts that comprise a user-defined service is given below:

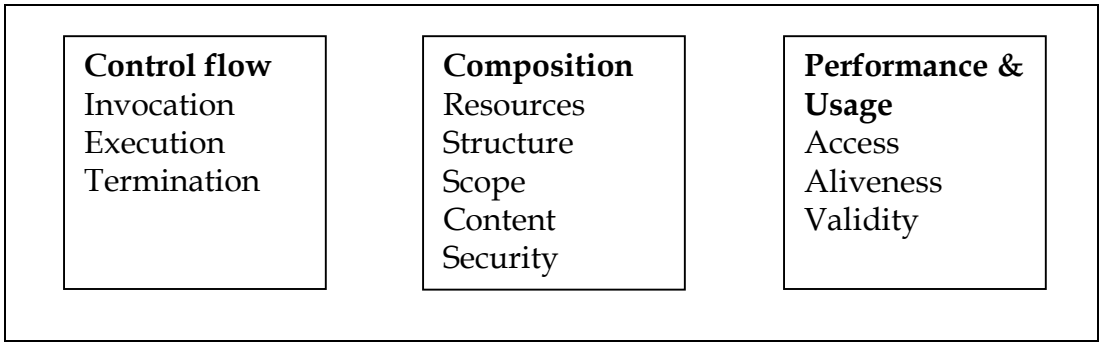

Figure 3: Components of a user-defined service.

4 Programming Language and User Interface

This chapter presents a programming model for SNSP. The programming language for SNSP is ANSI C, which it is very standard and well known. However, ANSI C is augmented with other descriptors (stored in separate files), which specify constraints and auxiliary services used by applications. The chapter concludes by presenting an Eclipse [40] integrated development environment (IDE) that can be used to write applications for SNSP. A sample application is presented in Appendix I and SNSP Eclipse IDE is presented in Appendix II.

Programming model

In SNSP's programming model, an application is a task that has inputs, outputs and a computational part. Any of these portions may be provided by other tasks/services. Given this structure, the application can conceptually be seen as a

hierarchical combination of tasks. The diagram below shows an example light-control application structure in which the application uses the person locating service (which gives the locations of people in the building), the sunlight metering service (which determines if sunlight is providing sufficient luminance), and the lightSwitch service (which actuates a given set of lights). In turn, the person locating service uses a combination of RFID reading services and services exposing data from motion sensors. The arrows between tasks in the figure signify communication between the different processes. Services can be considered a library of applications that conform to SNSP's service API presented in Chapter 3.

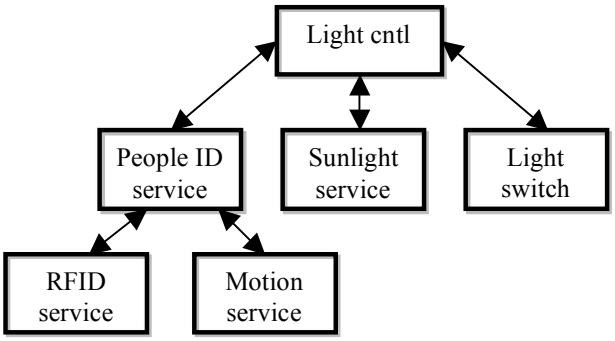

Figure 4: Light control service application structure.

4.1 Programming language: sensC

The programming language sensC is ANSI C [41] with a few extensions. C was chosen because of its widespread use in the embedded world. The language

consists of two types of files: the first type is standard C (.c and .h files with a few reserved keywords and functions) the second type is service specification files, which end with a *.serv* extension.

First, the .c and .h files define the functionality of the sensC program. A sensC program is divided into three sections: the *initialization*, *execution* and *termination* sections. The sections may also contain standard C functions. By default, these functions are public and may be called directly by other sensC modules in the sensor network. sensC supports timers and non-preemptive event queues for scheduling computation.

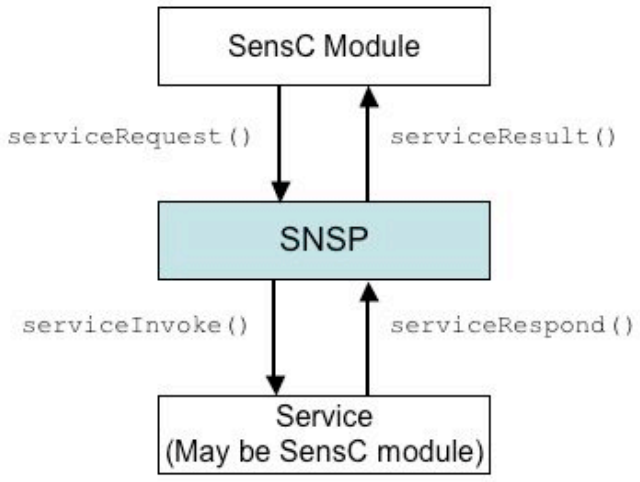

Figure 5: SNSP and sensC module.

Further, sensC modules interact with SNSP via a set of functions. Figure 5 illustrates the function calls between SNSP core, an application represented as a sensC module, and a third-party service that the application is using. The four functions are listed below:

- **`requestService()`** This function is called by a sensC module when it wants to use a service deployed on the sensor network. The first three parameters of the function are: (1) the requested service, which may be a user-defined service, or an essential service such as localization or the repository service, (2) scope, location and time to execute the service, and (3) the argument to the service. The argument to the service is a section, followed by the desired function within that section, for example `execution:invoke()`. Optional parameters are authentication, encryption, and request performance constraints, e.g. delay, reliability, accuracy etc.
- **`invokeService()`** This function is called by SNSP to notify a service that it has been requested. A sensC module that behaves like a service needs to implement this invokeService function. The parameters are the same as those of the requestService() function.
- **`serviceRespond()`** This function is called by a sensC service module when it responds, has results to return, to another module that requested the service. The required parameters are the request ID, the return scope, the return service, and a result value field. Note that the value field is a C struct, with a header that indicates the total payload length. The payload contains result entries, which contain the type and length of the result, as well as the actual result value. Optional parameters are authentication and encryption.

- **`serviceResult()`** This function is called by SNSP to return the result of a request to the original sensC module. The parameters are the same as serviceRespond().
- **`register()`** This function is periodically called by every sensC service module. It registers the service as belonging to the network and gives a complete service description (i.e. its interface as a list of functions with the formats of their arguments and results). See section **3.3** for more information on the service description.

When a user wants to write an application, they are presented with a blank template file containing the four service calls. The template file is further explained in section 4.3 as a part of SNSP IDE. Table 4 shows an example of a thermostat application. Some code has been left out to simplify the application.

In the example code the application initializes some default values on startup. SNSP (not shown in the user-code) will also query the repository service (CRS) to find out where the heater, temperature sensing and user-desired temperature sensing services are located. This information is stored in the availService array (the exact format of this is specified in Section 4.4.). Then, in the execution section, the application keeps requesting the temperature and a user temperature until it has received them, after which a control function is executed to turn the heater on or off.

```
boolean hasTemp
scope   tempScope
int     temperature;
boolean hasUserTemp;
scope   userTempScope
int     userTemperature;
scope   heatScope;
servDat availServices[SERVICES_USED];

Invocation:
  invoke(){
    hasUserTemp = hasTemp = false;
  }

Execution:
  executeControl (){
    if(!hasTemp && availServices[temp].exist){
     tempID = requestService(TEMP, tempScope);
     return;
    }
    if(!hasUserTemp && availServices[userTemp].exist){
     userTempID = requestService(USERTEMP, userTempScope);
     return;
    }
    if (availServices[heater].exist) {
        if(temperature < userTemperature)
           requestService(HEATER, heatScope, ON);
        else
           requestService(HEATER, heatScope, OFF);
        }
    }
  }
  terminate(){}
serviceResult(id, value){
  switch(id){
    case tempID:
      temperature = value->payload;
      break;
    case userTempID:
      userTemperature = value->payload;
      break;
  }
}
```

Table 4: Example code for a temperature control application.

4.2 .serv Requirement Specification

The .serv file consists of two segments. The first segment details information about the sensC module's runtime requirements. This will be used by SNSP to dynamically deploy the module on the sensor network. These requirements are:

- Data in-flow rate (shaped by leaky bucket)
- Data out-flow rate (shaped by a leaky bucket)
- Memory requirements such as dynamic memory and intermediate storage on a node
- Resource requirements (i.e hardware that must be co-located on the node)
- Fault tolerance requirements (see section 3.2.6 on fault detection and recovery)

The second segment contained in a .serv file specifies information about the other services or modules that the sensC module will invoke during execution. In order to make the sensC module more applicable to general sensor networks, services are specified via equivalence classes. Equivalence classes specify a set of properties that a service must have to be considered an equivalent candidate. In order to evaluate different equivalent services, the properties may be specified with an evaluation function that assigns a grade to the service depending on its exact value of its property. For example, there may be two temperature services in the network; one may have an accuracy of ±1°C and the other ±5°C. These

services are equivalent, but the first is more valuable to a module that has high accuracy requirements. Equivalence classes allow a service to be substituted for another when the original one is not present in a sensor network.

For large equivalent sets, a decision tree structure can be used to efficiently represent an equivalence class. The decision tree not only encodes all the properties required of the service, but also an order in which to evaluate them. Given a good choice of variable ordering, [42] has shown that decision trees are both a compact way to store evaluation functions and an efficient mechanism to check whether a service matches the criteria. In the current SNSP implementation equivalence classes are specified as priority lists.

Table 5 shows an example .serv file. The top half of the file specifies the constraints such as data in-flow-and-outflow rates. The second half of the file lists equivalent services. The line "Service:x" denotes the start of a new equivalent group. For service 2 there are two equivalent services: Cricket [43] and Motetrack [44] localization. Cricket is an acoustic localization service whereas Motetrack uses RF and is not as accurate. The accuracy of both localization services is known (represented in the functional API). The algorithm prefers Cricket to Motetrack if it is available in the network. Also, if QueryPeriod is set to 0, then it is the responsibility of the application to query the service during execution. If it is non-zero, SNSP will periodically query the service for results.

```
ModuleName:example

DataIn:10,5                 /* units are kb/s */
DataOut:8,2
Memory: 2                   /* units are kbytes */
Processing: 10
ResourceReq:ServiceName     /* Can also be left blank */
FaultTolerance:detection    /* recoverable, detection, nothing */

Service:0
Name:TEMPERATURE
Scope:KITCHEN
QueryPeriod:5               /* query once every 5 seconds */
Name:TEMPERATURE
Scope:LIVINGROOM
QueryPeriod:8

Service:1
Name:HUMIDITY
Scope:SP_SCOPE_ALL
QueryPeriod:10

Service:2
Name:CRICKET_LOCALIZATION
Scope: SP_SCOPE_ALL
QueryPeriod:0
Name:MOTETRACK_LOCALIZATION
Scope: SP_SCOPE_ALL
QueryPeriod:0
```

Table 5: Example of a .serv specification file.

4.3 Programming UI

SNSP is not just a distributed operating system for sensor networks. It is also a platform that allows users to write applications for sensor networks. This is known as an integrated development environment (IDE), which is a PC-side tool that allows users to create and compile SNSP applications. IDEs normally consist of a source code editor, and a compiler/interpreter.

Eclipse was chosen as the IDE for SNSP. [40] Eclipse started as an open-source Java tool to build Java IDE's, but now consists of an open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle.

The Eclipse platform is extensible via plug-ins. Plug-ins are "pluggable components," which conform to Eclipse's plug-in contract. These plugins work seamlessly with the core part of Eclipse during runtime. They may do something simple like adding a single button that displays a message to a user, or soething complex like adding support for compiling another language. There is a Tinyos plugin [45] that allows users to write and compile TinyOS code within the Eclipse IDE. SNSP Eclipse plugin is fully described in Appendix II.

4.3.1 Creating an SNSP Application

The first step in creating an SNSP application is defining all the services that will be used during the application's execution. The services as well as the scope are defined in the .serv file. See Section 4.2 for an example. Next, these specifications must be reflected in the C files. First, the services and locations used are defined in numerical format in the .h file. An example .h file is given in Table 6 below.

```
/*
 * Place the names of locations and services here
 * Note, the spelling & capitalization must be the same as that
 * used in the .serv file
 */
enum {

/* eg location:
 * KITCHEN = 1,
 * LIVINGROOM = 2,
 * DININGROOM = 3,
 */

/* eg Service names
 * TEMPERATURE = 9,
 * HVAC = 14,
 * COMFORT = 15,
 * DESIRED_TEMP = 16,
 * HVAC_CONTROL = 19,
 */

/* eg persona
 * OWNER = 22,
 */

  BRIDGE = 17,
  OFF = 2,
  ON = 1,
  SENSOR = 10,
  ACTUATOR = 20,
  CONTROL_TRIES = 3, //#times try to respond before timeout
  REPLY_TRIES = 3,
  ACTIVATE = 2,
  DEACTIVATE = 3,
};

typedef struct serviceUsage {
  uint8_t  name;
  scope_t  scope;
  int      platform;
  uint16_t ticksToQuery;
  int      invocationID;
  int      exists;
}
```

Table 6: .h file specifying the relevant services, scopes, and persona.

Table 6 also shows the serviceUsage structure, which is of particular interest. SNSP will create a serviceUsage array the size of all the services used. During execution, SNSP will periodically query the repository service to determine which of the equivalent services exist in the network. It will populate the serviceUsage array with information about the top equivalent service that it finds in the repository. If an application wants to query this service, it can find the service name and scope in the service usage array at the index specified in the .serv file.

When the application programmer starts a new SNSP project, the main functionality of the application gets placed in the C file that is created. The blank C file contains a number of functions that need to be filled in. Table 7 shows the blank C file. Most of the functions have comments describing the calling context and arguments. These functions are essentially those described in Section 4.1 by the sensC module interaction. An example of a completed C file and the resulting compiled tinyOS code is given in Appendix I. The example shows a thermostat application.

```
#include "table.h"

/*
 * Place all the variables to store results from services used
 * eg uint16_t sensorSample1
 * uint16_t sensorSample2
 */

/*
```

```
 * during execution these will be filled in and updated
 * with the name & scope of the equivalent service that
 * exists in the network - see bottom of file for
 * a definition of the struct
 */
serviceUsage services[NUM_SERVICES_USED];

/*
 * fill in, it will be called on initialization
 */
void invoke()
{
}

/*
 * fill this in - it will be called every second
 * Use it to process results, query more services, actuate etc.
 */
void executeControl()
{
}

/*
 * Fill this in if you want other components to use this service
 * it will be called on the service in response to a
 * requestService call and it should call serviceRespond
 * to pass the result back to the callee
 * @param fn function that will be called
 * @param args - arguments to the function
 * @param arg_len length of the arguments (not null terminated)
 * @param id -identifies the request, should be passed back to
 *           serviceRespond
 *
 *@return int return 0 if the service successfully invoked,
 *           1 otherwise
 */
int invokeService(uint8_t fn, char *args, uint16_t arg_len,
uint8_t id)
{
}

/*
 * Fill in to process the result of a service query/invoke
 * service
 * @param fn - function name that was called
 * @param payload - results
 * @param payload_len - length of results (they are not null
 *         terminated)
 * @param id - request id that was returned in requestService
```

```
 */
void serviceResult(uint8_t fn, char *payload, uint16_t
payload_len, uint8_t id)
{
}

/* called when the mapper is wrapping up the service
 * clean up any last minute state
 */
void terminate ()
{
}

/*------------------STUBS-----------------------*
 * Leave at the bottom of the file - will be filled in by SNSP */

/*
 * This is a stub will send a service  query
 * @param servNum is the equivalence class specified in
 *        the .serv file
 * @param servName is the class that was found in the network
 *        (that is equivalent from  - serviceUseage
 *        services[NUM_SERVICES_USED];)
 * @param fn is the function name that will be called
 * @param args is a void pointer to arguments to the service
 * @param argLen is the length of args (not null terminated)
 *
 * @return int returns the id that will be used in
 *         serviceResult to pass result back
 */
int requestService(uint8_t servNum, uint8_t servName, uint8_t fn,
char *args, uint16_t argLen)
{
}

/*
 * This is a stub
 * @param fn - function that was called
 * @param results - pointer to results
 * @param
 *
 * @return 0 if successful 1 if not
 */
int serviceRespond(uint8_t fn, char *results, uint16_t resLen,
uint8_t id)
{
}
```

Table 7: Blank template C file that the programmer needs to fill in.

4.4 Coding Efficiency Gain

The programming language and UI were developed to make writing sensor network applications easier for programmers. One way to measure the gain in efficiency is to compare the number of code lines saved. In the example given in Appendix I, the C code is 146 lines long. The resulting TinyOS code is 711 lines for the HVAC_CONTROLM.nc module (which contains the application code) plus an additional 2109 lines for the supporting SNSP TinysOS code. That is a savings of 1932%. As the program gets more complex, the application's C part will get longer and the supporting SNSP code will be amortized over many modules. However, there is still a significant savings of 2674 lines of code.

5 File Allocation

The underlying challenge in providing this service is to optimize content placement for availability and to minimize communication (both access and update costs). This problem is known as the file allocation problem (FAP) and has been extensively studied for traditional distributed databases. The FAP may be based on static allocation, which means that the files are allocated once ahead of time, or dynamic allocation. Further, the file access patterns may be deterministic, probabilistic or unknown. All versions of the FAP are NP complete.

This research focuses on dynamic file allocation with unknown file access patterns. Further, the work places more importance on reducing the communication cost to access a file, and aims to increase file availability. The chapter starts by outlining the formal file allocation problem. Next related work and the process of locating files are addressed, followed by the presentation of

three heuristic algorithms: A deterministic central replication algorithm, a distributed probabilistic replication algorithm, and a learning-based algorithm. These algorithms are compared via simulation to a control case with no replication.

5.1 Problem Formulation

A version of the static FAP, similar to that in [35] is formulated below.

Given:

- n nodes
- m files
- l_j is the length of the j^{th} file for $0 < j \leq m$
- b_i is the available memory of the i^{th} node for $0 < i \leq n$
- c_j is the storage cost per unit at the j^{th} node
- CA_i is the capacity of the i^{th} link

And traffic vectors where the incoming traffic arrives with a poisson distribution:

- u_{ij} is the query traffic originating from node i for file j
- v_{ij} is the update traffic originating at node i for file j
- u'_{ij} is the return traffic to node i from queries for file j
- v'_{ij} is the return traffic to node i from updates of file j

Chu [35] formulated the file allocation problem as a zero-one integer-programming problem. The variable X_{ij} is the indicator function that the j^{th} file is stored on the i^{th} node.

$$X_{ij} = \begin{cases} 1 & \text{if } j\text{th file is stored on the } i\text{th node} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Storage cost is given by:

$$Z_{store} = c^T X l \tag{2}$$

Where $c = (c_1, c_2, \ldots, c_n)^T$ and $l = (l_1, l_2, \ldots, l_m)^T$

If file j is stored at q_j different nodes in the network, where

$$q_j = \sum_{i=1}^{n} X_{ij} \tag{3}$$

The total traffic generated in the network is now:

$$Z_{comm} = \sum_{i=1}^{n} \sum_{j=1}^{m} \left[\left(u_{ij} + u'_{ij} \right) \left(1 - X_{ij} \right) + \left(v_{ij} + v'_{ij} \right) \left(q_j - X_{ij} \right) \right] \tag{4}$$

The objective of the file allocation problem is to minimize the storage cost (eqn 2) and the total communication cost (eqn 4) of transporting data through the network.

For SNSP, storage capacity and availability constraints can be formulated as follows:

$$C_{cap} = \sum_{j} X_{ij} L_j \leq b_i \quad \text{for } 1 \leq i \leq n \tag{5}$$

Analytical expressions for availability are hard to obtain, so an approximation, given in [46] is used. This approximation is explained below.

First, we define r_{ik} to be the probability that two nodes i, and k successfully communicate. In order to derive the analytical expression, we assume r_{jk} is independent r_{mn} for all m not equal to j and all n not equal to k. This can be calculated for each pair of nodes if the routing and node availability is known. This probability includes the probability that node k is available. (In the simulation the availability to access a file at location k from location j is not explicitly calculated; only the availability of the node k is taken into account).The availability a_{ij} of file j accessed by node i is

$$a_{ij} = \left[1 - \prod_{k=1}^{n} \left(1 - r_{ik} X_{kj} \right) \right] \tag{6}$$

Let W_{ij} be the total traffic demand at node i for file j. $W_{ij} = u_{ij} + u'_{ij} + v_{ij} + v'_{ij}$. The weights indicate the relative popularity of files. The traffic-weighted availability for each file is given by:

$$C_{avail}^{j} = \sum_{i=1}^{n} \frac{W_{ij} a_{ij}}{\sum_{i=1}^{n} W_{ij}} \tag{7}$$

The optimization problem given a target *t*, availability constraint *a* and variable λ, is: an allocation F(X) that minimizes

$$F(X) = Z_{store} + \lambda Z_{comm} \quad \text{(eqn 2, 4)}$$

subject to

$$C_{cap} \quad (eqn\ 5)$$

$$C^{j}_{avail} > a \quad \text{for} \quad 0 < j \leq m \quad \text{(eqn 7)}$$

5.2 Related Work

Content replication algorithms (also known as file allocation problems) were studied extensively with the rise of networked computer systems in the late 60's and 70's. The optimal solution is NP complete. Techniques for file allocation include branch and bound, randomization, predictive Markov techniques, genetic algorithms and other heuristics. [47] gives a good overview of these difference techniques. Further, the solution techniques employed depend on the assumptions that are made and the different constraints that are considered.

As shown in Table 8, in one variant of the content replication problem the input pattern is not known in advance but the algorithm must react to file requests as they arrive. This class of problem is known as online problems. Typically, competitive algorithms are used to solve/analyze online problems. Competitive algorithms have the property that their performance on any sequence of requests is within a constant factor of the performance 'of any other algorithm (including the optimal) on the same sequence. See [48] for more details. This work will

compare algorithms that do not know the access pattern in advance to each other.

Input	Pre-determined Probabilistic Unkown
Solution type	Static (files are allocated once) – optimal or heuristic Dynamic (files migrate during process) – optimal or heuristic
Structure	Single File (assume files are independent) Multiple Files (for high-throughput case, files may content with each other w.r.t. queueing delay) Files as program data
Metrics	Minimize execution cost Minimize communication cost Maximize availability Maximize throughput Minimize access delay Minimize file transfer times

Table 8: FAP Classification

In the sensor network setting there has not been task allocation work per se, but there has been work on the related problem of data storage and retrieval. Solutions include modeling the sensor network as a database [13], adapting hash tables from content addressable networks with geographically distributed hash tables [49]. This is used to find and store data and sensor network fusion, where a sensor needs to figure out where to send its data when queried [50]. This work has focused on making data retrieval more reliable and cheaper, but it has been

mostly focused on collecting data through a central point, and has not allowed for data replication in the network.

5.3 Locating Files

Existing content replication algorithms do not address locating the content. In the worst case, nodes need to flood the network to locate information. In order to avoid flooding, an indexing system is used. Indices are placed at warehouse nodes, which have higher availability and storage space than their peers. Warehouse nodes are selected via a clustering algorithm in a d-hop neighborhood. This work uses the clustering scheme presented in [51], which selects as cluster heads nodes within D hops having the highest id. This work uses a combination of the node's availability and storage as criteria for cluster head selection. The cost of maintaining the content indices is added as overhead cost. Note, the cost of accessing a warehouse is on average be D/2, the average distance a node is from its cluster head. Thus it is beneficial to reduce D to lower the cost. However, as D is decreased, the total number of cluster heads increases, which increases the cost to write a new location to all warehouses.

5.4 Algorithms to Evaluate

5.4.1 Deterministic, Central Replication Algorithm

This algorithm was adapted from [52]. The algorithm is O(*logn*) competitive. For each data item, the algorithm maintains a list of L read requests and also a tree structure T of pointers to all replicated data. d is the size of the data. The algorithm executes as follows:

- On read request r, the algorithm finds sphere S with radius k that contains d reads
- If no copy of a file exists within a sphere of radius λk of the r, the file is replicated to the node with the highest availability in the sphere, and it is added to T
- Read requests from nodes in S are deleted from L
- After d writes, all copies in T are deleted and the one on the node with the highest availability is kept

5.4.2 Distributed Algorithm

This algorithm was adapted from [53] to improve the availability of the data. It is O(*log2n/log2d*) competitive (where d is the size of the data and n is the number of nodes). A tree T_p is kept for replicas of item p:

- On a read request from r, insert the node with the highest availability in r's one-hop neighborhood in T_p with probability $1/d$

- On a write request from w, with probability 1/(sqrt(3)*d), delete all nodes in T_p, except for the one with the highest availability and add w to T

5.4.3 Adaptive Observation-Based Algorithm

This algorithm uses an observation period during which nodes observe the number of read/write requests. Replication decisions are made after this period.

Initial step:

- Observe 100 accesses of a data item p, keep record of their location and whether it was a read/write.

Reallocation:

- After r reads
- Create k = floor(r/20) replicas of the data item
- Use k-means to divide the data requesters into k clusters and store one data item on the node within each cluster with highest availability.

Now that there are replicas in the network, writes can either come directly from a node (meaning that the replica was the closest copy to be written) or they can happen when data is written through for consistency.

Refinement steps (on each node that has a copy of p):

- Observe 100 accesses of the data item.
- Accesses are either reads r, direct writes w_d, or update writes w_u.

- If $w_u > r + w_d$ remove the replica from the node
- If $r > w_d$ create k = floor($r/20$) replicas and use k-means to distribute the replicas onto a node with the highest availability in each cluster.

5.5 Simulation Setup

I used the discrete event simulator Omnet++ [54] with the Mobility Framework (MF) [55] extension. MF was developed at TU-Berlin and provides a good model of the WSN physical and MAC layers. The simulation consists of 1000 nodes randomly placed in a 2000m x 2000m square. The nodes have a radio range of approximately 150m. The network's heterogeneity is also varied, with a *heterogeneous network* consisting of nodes with 3 amounts of storage: 10k, 50k, and 200k. Nodes with a large amount of storage are available 99.9% of the time, nodes with a medium amount of storage have a 95% availability and the smallest nodes have a 80% availability. In the *homogenous network*, nodes have 2 different amounts of storage, 50k and 100k, and they are available at rates 97% and 94% respectively. Availability refers to the percentage of the time a node will be awake and respond to incoming packets. All nodes are unavailable for an exponentially distributed time with the same mean, so a higher availability means lower chance of becoming unavailable. The relative times are calculated so that nodes are probabilistically awake for the percentage specified by their availability.

When a node is available, it wakes up every 0.25 seconds, and decides with probability 0.1 to read or write a randomly selected piece of data. Each simulation consisted on average of 39,000 queries. All data items are the same size (100), but they have different read to write ratios of 1:1, 3:1, 15:1 and 100:1. Cluster head election occurred at the start of the simulation and the cluster heads did not change during the simulation.

For the lower layers, a standard fading channel was used with an Aloha MAC [56] (both of these are provided by the mobility framework). At the network layer, geographic routing and addressing is used.

The following assumptions are made:

- Nodes know the id's of data items they want to access
- ID's are much smaller in length than the data item
- Data items are indivisible and the original copy cannot be deleted or migrated to another node
- There are no concurrent read/writes (relatively few events in WSN)
- Nodes have a finite amount of storage
- Node failures/deaths are modeled as an independent binomial process, independent of other nodes.
- Nodes also fail temporarily (which corresponds to temporary downtime e.g. sleeping to conserve power).

- Nodes know their own availability, i.e. the proportion of time that they are in functioning condition.

5.6 Results

The first experiments examined the tradeoff between number of cluster heads, their availability, and the number of hops, D. Results in Table 9 show a summary of the number of cluster heads vs number of hops, D. The results were similar across both topologies. All cluster heads were the highest availability nodes, even with D of 2. Further, there is clearly a tradeoff between the number of hops, D, a cluster head is away from a node and the number of cluster heads to keep synchronized when a data item's location changes. Due to the relatively small change in number of clusters from 3 to 8 hops, D was chosen to be 3 for the remainder of the simulations.

D	# Cluster Heads	Availability
2	10	99.9
3	6	99.9
4	5	99.9
8	3	99.9

Table 9: Number of cluster heads vs number of hops, *D*.

Figure 6 shows the total cost (the number of hops data was transported) per data access for each of the schemes. The solid portion of the bar represents the direct cost of the operation. For example, for a write operation that represents the cost to transport the data from the writing node to the closest replica in the network.

The top portion of the bar is overhead to keep replicas consistent. Considering the data access cost, the adaptive algorithm performs the best, followed by the distributed algorithm, the deterministic algorithm and last the control algorithm.

However, considering total cost, the distributed algorithm outperforms the adaptive algorithm. This is because the distributed algorithm saves on replication cost by replicating to the nodes (or nodes very close to them) that make the queries. While the adaptive algorithm has a better placement of replicas (because its access cost is lower), it incurs additional cost because it locates replicas independent of queries. The overhead of the deterministic algorithm makes it more costly than the control algorithm.

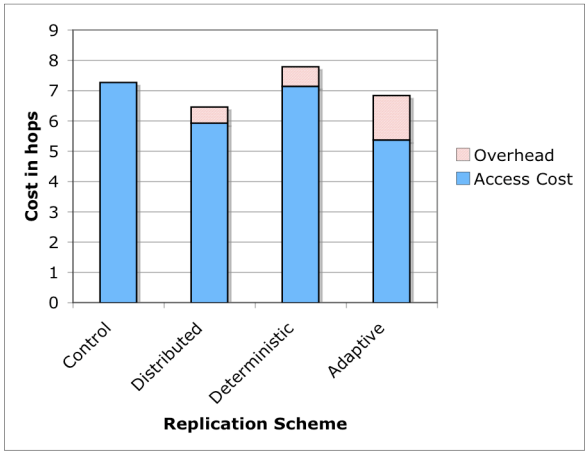

Figure 6: Comparison of data access cost and replication overhead.

Figure 7 shows the cost breakdown for the distributed and adaptive schemes by topology and data read/write ratio. Topology 1 is the heterogeneous topology and topology 2 is the homogeneous topology. There is not much difference

between the two topologies. On average, across all read/write ratio entries for the same scheme, the two topologies differ by, 0.3 hops or roughly 4%.

The read/write ratio has a bigger impact on cost. In both schemes the same pattern is visible; the overhead is much larger for data with lower read/write ratios. This overhead shrinks to almost nothing for data with a read/write ratio of 100 in the distributed algorithm. Intuitively, this is because the placement of the data does not incur an overhead, and even though there are many replicas, writes are so infrequent that they add little to the total cost. Thus, these schemes provide more benefit for data that is read much more frequently than it is written.

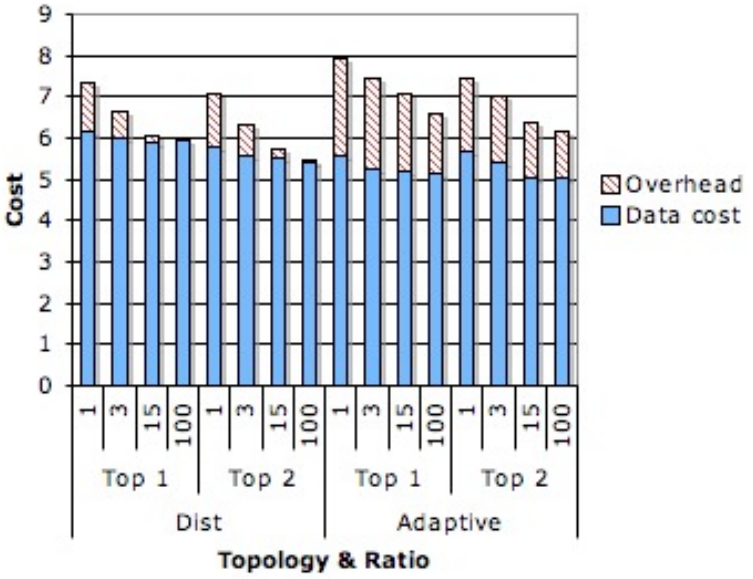

Figure 7: Comparison of data access cost vs topology and data read/write ratio.

The previous graphs showed only the cost of moving data around the network. Figure 8 shows the control message overhead. The distributed algorithm has the lowest control message overhead followed by the deterministic and the adaptive algorithms. The data read/write ratio does not have a large impact on overhead cost, because the overhead is dominated by the cost of finding an item; there is not much difference between finding a location to read from or one to write to. Moreover, this overhead is not as significant as the data cost shown in Figures 6 and 7. If data ids are a 10th the size of the data payload, then these numbers should be divided by 10 to scale them to those in Figure 6. On average, the distributed algorithm would cost 0.27 hops more, the deterministic algorithm would cost 0.34 more and the adaptive algorithm would cost 0.4 hops more. The control overhead becomes even more significant if the ratio between the data payload size and the data id size becomes larger.

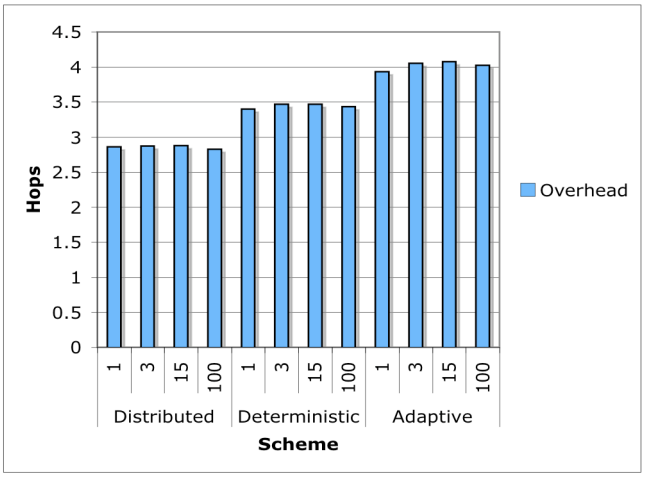

Figure 8: Comparison of control message overhead.

In addition to the cost of the replication scheme, data availability is also an important concern in SNSP. Figure 9 shows the percentage availability of data for all schemes and topologies (the 1, 2 on the x axis indicates topology). Topology does have an impact on availability; the homogeneous topology has a higher availability than the heterogeneous topology. The distributed replication improved reliability the most with a 33% improvement for topology 1 and a 51% improvement for topology 2.

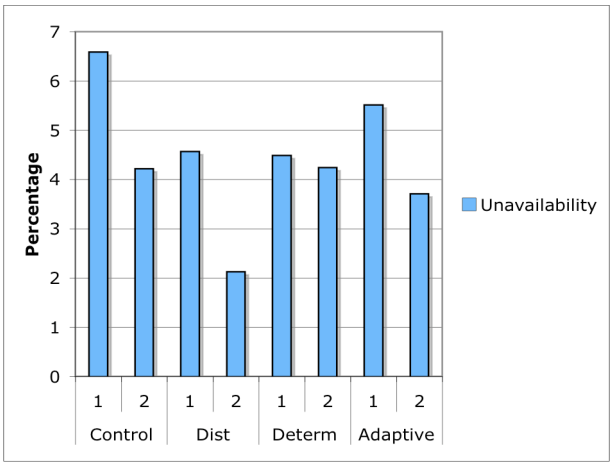

Figure 9: Percentage of unavailable data for different schemes and topologies.

5.7 Discussion

This chapter presented a formal problem definition of content replication and management. The problem was then investigated in the context of sensor networks. Specifically, as part of SNSP, a distributed operating system, which dynamically maps data and programs onto the sensor network. Out of the three algorithms that were empirically compared, the distributed replication scheme

performed the best with the lowest total cost and the highest availability. The adaptive algorithm developed for SNSP performed the best when considering only data access cost, indicating that it had better placement than the other algorithms. However, it also had higher control message overhead.

The simulations showed that all replications schemes provided more benefit for data with more frequent reads than writes. Finally, the simulations showed that although the replications schemes improved data availability, the sensor network topology has an impact on data availability. The homogenous network had roughly 2% more availability than the heterogeneous network. From a practical perspective, the distributed algorithm performed the best, it is very simple to implement and has low overhead.

6 Task Allocation

This chapter formulates the task allocation problem for SNSP. The task mapper takes SNSP task description (including a list of sub-tasks that can be further partitioned or assigned to individual sensor nodes) as input. In a sensor network the mapping must happen in a decentralized way. The problem is formally stated in this chapter, followed by related work. The optimal solution is NP complete, and therefore the goal of this chapter is to evaluate heuristic file allocation solutions for SNSP. In order to facilitate this, three algorithms are outlined in this chapter and then compared via simulation. The algorithms can also implement incremental mapping of processes. Incremental mapping is when a task (that is part of a process) is already running in the network, it is shared between the two processes, instead of mapping two copies of the task on the network. The results compare both incremental and non-incremental approach.

The algorithms are distributed. The solution can be structured so that each task has an independent mapping agent or each resource has a mapping agent. The task-mapping agent evaluates the best mapping and then takes care of obtaining the resource for the task and any contention that may result. The resource agent receives bids to obtain its resource and performs allocations to the highest bidders. The algorithms chosen are aimed to reduce the communication overhead between tasks and to increase information availability.

6.1 Problem Formulation

The mapper's function is to allocate resources to processes so that they can execute on the sensor network.

6.1.1 Assumptions

- Nodes know the IDs of data items they want to access
- Data id's may be arbitrary in length, so long as they are prefix-free
- Nodes have a finite amount of storage available for data items
- Node failures/deaths are modeled as an independent binomial process, whereby each node fails independently of others.
- Nodes know their own availability, i.e. the proportion of time that they are in functioning condition.

In the mapping problem, we assume that we are given the following information:

Given n nodes and m processes

for $0 < j \leq m$ and $0 < i \leq n$

- h_j is the RAM requirement of the jth process
- g_i is the computation requirement of the jth process
- o_i is the hardware requirement (e.g. sensors) of the jth process
- t_j is the location requirement (e.g. kitchen, house) for the jth process
- e_i is the available RAM of the ith node
- q_i is the available CPU of the ith node
- c_i is the cost to execute a process at the ith node
- u_i is the hardware on the ith node
- p_i is the location of the ith node
- s_{ik} is the bandwidth between nodes i and k for $0<k\leq n$
- CA_{kj} is the communication cost between tasks k and j if they are executed on different processors for $0 < k \leq m$

The problem can be formulated as a zero-one integer programming problem where:

$$X_{ij} = \begin{cases} 1 & \text{if } j\text{th process is executed on node } i \\ 0 & \text{otherwise,} \end{cases} \tag{8}$$

and one copy of the process is executing

$$\sum_{i=1}^{n} X_{ij} = 1. \tag{9}$$

Note, multiple copies of a process could be executing on the network if different applications are using the same service and it is not convenient to share the output of the service. However, each application has its own allocation matrix X, and if two applications share the output of the same service, the mapper keeps note of that separately.

The goal of the mapper is to find an allocation that minimizes the execution cost (9) and the total communication cost (10). The execution cost is:

$$Z_{exe} = c^T X \bullet \vec{1} \tag{10}$$

Where $c = (c_1, c_2, \ldots, c_n)^T$

The communication cost is:

$$Z_{comm} = \sum_{(i,j)\in e_1} \sum_{k=1}^{n} CA_{ij} X_{ik} (1 - X_{jk}) \tag{11}$$

where e_1 is the set of edges in the task graph.

The constraints of the mapping problem will be unique to each application. While the delay, reliability, processing, bandwidth, and memory constraints are

given by a less or greater than operator. The hardware and location constraints may have more complex matching functions. For example, a location may be contained within another, or hardware may be a superset of what is required etc.

The mapping problem can then be formulated as follows:

Given operators α and β, that take as input location or hardware data and constraints and return 1 if the data or hardware constraints respectively are met, an availability vector *a,* and parameter λ find an allocation X that minimizes:

$$F(X) = Z_{exe} + \lambda Z_{comm} \tag{12}$$

subject to

$$\left[1 - \prod_{k=1}^{m}\left(1 - Y_{kj} \sum_{p,i \in N} r_{pi} X_{pj}(1 - X_{ik})\right)\right] \geq a_j \tag{13}$$
$$\text{for } 0 < j \leq m$$

where $Y_{kj} = \begin{cases} 1 & \text{if processes } j \text{ and } k \text{ communicate} \\ 0 & \text{otherwise} \end{cases}$

and r_{pi} is the probability that nodes p,i communicate successfully.

$$s_{jk} \geq \sum_{i=1}^{m} \sum_{p=i}^{m} CA_{ip} X_{ji}(1 - X_{kp}) \quad \text{for } 0 < j \leq m \text{ and } 0 < k \leq j \tag{14}$$

$$\sum_{i=1}^{n} X_{ij} e_i \geq h_j \quad \text{for } 0 < j \leq m \tag{15}$$

$$\sum_{i=1}^{n} X_{ij} q_i \geq g_j \quad \text{for } 0 < j \leq m \tag{16}$$

$$1 = \alpha\left(\sum_{i=1}^{n} X_{ij} p_i, t_j\right) \quad \text{for } 0 < j \leq m \tag{17}$$

$$1 = \beta\left(\sum_{i=1}^{n} X_{ij} u_i, o_j\right) \quad \text{for } 0 < j \leq m \tag{18}$$

Equation 13 represents the availability constraint. Equation 14 represents the bandwidth constraint between all processes allocated on nodes *j,k*. Equations 15 and 16 represent memory and cpu requirements respectively, while 17 and 18 represent the subtasks' location and hardware requirements.

As mentioned above, the optimal mapping problem is NP complete. Future work will investigate different heuristics for mapping. Further, this is a static formulation of the problem whereas in a real system the number and type of applications and services executing on the network is dynamic. Other future work includes evaluating mapping cost and performance trade-offs for partial vs. complete remapping when either the underlying sensor network or the applications change. In addition, because duty-cycling to save power is such an important part of sensor network operation, the computation constraint will contain a duty-cycle field as well.

This research focuses on assigning tasks to processors. It does not deal with scheduling multiple tasks on a single processor once they have been assigned. There are however, many different optimal algorithms for doing this. For example, generalized processor sharing (GPS) [57] with earliest deadline first. GPS is well suited here, because in the task description it already specifies a rate of processing required for each task.

6.2 Related work

Mapping has been extensively studied. First in the context of mapping multiple processes onto a single processor and then later in the context of mapping processes onto distributed networked systems. There are many different metrics for which task allocation can be optimized, e.g. execution time, task communication, system reliability or load balancing. This thesis considers task communication and system reliability as metrics to evaluate the task allocation algorithms.

The typical formulation is as follows: Given a set of partially ordered communicating tasks T and a set of interconnected processors P, each with p_i processing resources, define a mapping of processes to processors that minimizes cost. The typical cost model is minimizing execution time on the processors (processes take different amounts of time to run on the processors depending on their load and cpu power). Alternatively, the communication cost may be

minimized (number of messages related to where the communicating tasks are allocated), or reliability may be maximized (by replicating processes or allocating them to more reliable processors). For a graph formulation, the set of tasks are given as a task graph $T = (V_1, E_1)$ where the edges represent the amount of communication required between tasks. The set of processors are given as a processor graph $P = (V_2, E_2)$ where the edges represent the bandwidth of the links between processors. A valid allocation then finds a weak homomorphism from T to P, that is, T is weakly homomorphic to P if there exists a mapping such that if an edge $(a, b) \rightarrow E_1$, then edge $(M(a),M(b)) \rightarrow E_2$.

Optimal task allocation is NP-complete for any of these metrics. Thus, heuristic algorithms have been extensively studied since the early 70's [58]. These algorithms have largely been designed to operate in traditional networked environments, with high bandwidth connections and reliable nodes. Typical solutions utilize heuristics such as dynamic programming, genetic search algorithms, graph embedding techniques, and micro-economic approaches. [59] gives a detailed overview of various mapping solutions.

This research draws from the graph search method to construct a greedy mapping solution as well as taking the TASK algorithm presented in [60]. For the last algorithm, a micro-economic resource mapping approach is combined with a genetic search algorithm. Existing micro-economic algorithms outline bidding functions and strategies for which the market will reach a fair equilibrium.

However, they do not specifying how much value the bidder should actually place on one resource compared to another. The genetic search algorithm solves the problem of determining values for resources by searching through the solution space and coming up with fitness values for a set of different allocations. A side result of the genetic algorithm is it searches through many different allocations, thus if the bidder cannot obtain the best choice resource because there is contention, it has other options. The three chosen algorithms are presented next.

6.3 Allocation Algorithms

6.3.1 Greedy Spanning Tree

This algorithm finds a feasible mapping by greedy allocation. Tasks are allocated independently, and there is no contention mechanism. That is, if task A is taking up a resource that task B values more, the resource is not reallocated to task B. Every task that is allocated has a single agent that allocates the entire task. The algorithm takes a task graph as input, and allocates submodules of the task similar to how a spanning tree is built in Prim's algorithm [60]. Prim's algorithm works as follows: Given a graph $G = (V, E)$, take the edge with the minimum cost and put that edge in the spanning tree, place the two nodes that it connects in V'. Repeat the minimum cost edge selection until all vertices in V are in V'. The

greedy algorithm chooses edges in the same way, however, it does not remove vertices from the task graph, as all edges in T must be mapped to P.

The greedy task allocation algorithm will choose the first two tasks t_1, t_2, in the task graph T with the highest inter-communication rate. It will then allocate these tasks as close to each other as possible in the processor graph P to minimize communication cost. The mapping must be feasible, that is, all other constraints must be met. The link between t_1 and t_2 is then removed from the task graph T. Next, the algorithm will choose two tasks linked with the highest remaining communication cost, and allocate these. If one, or both, of the modules are already allocated to processors, the algorithm simply allocates the remaining task or finds the minimum communication path in P. As a slight modification, when the algorithm is allocating two tasks t_1 and t_2, the communication cost between t_1, t_2 and other previously allocated nodes will be taken into account.

The allocation at each step is now reduced to allocating at most 2 tasks. However, if all possibilities are enumerated, this could still lead to $0(n^2e_2)$, where n is the number of vertices in P, and e_2 is the number of edges in P. This is because there are at worst $n(n\text{-}1)$ locations for the two tasks, and once a task is assigned to a location the best communication path between the two nodes must be established. This allocation of two tasks may also take place at worst m-1 times. The allocation will instead follow a set of heuristics even for two nodes. First, the algorithm will evaluate the cost if the two nodes are allocated on the

same processor. The two tasks will be allocated as follows: scan all processors in P to determine if the two tasks can be co-located. If the tasks communicate with other tasks that are already allocated, choose the processor that minimizes the communication cost to these nodes. If the two tasks cannot be allocated on the same processor, do a gradient-based local search. In the gradient-based local search, assign the two tasks randomly to two processors. Take turns examining each task. When a task is examined, randomly choose another processor and evaluate whether swapping the task to this processor would improve the cost. If it does, move the task, if it does not, leave the task where it is, and look at the second task. If the allocation has not improved more than δ in the last k iterations of swapping, the allocation is done.

Given: processor graph $P=(v_2, e_2)$, a process graph $T=(v_1, e_1)$ where v_1 is a list of tasks and e_1 is a list of edges, each edge connects two tasks.

```
sortEdgesByCost(e1)
while there are edges in e1
   edge =  e1.pop_front()
   task 1 :=  getFirstTask(edge)
   task 2 = getSecondTask(edge)
    if (MapOnTheSameNode(task1, task2, edge.bandwidth)) {
        continue;
    }
    mapTask(task 1)
    mapTask(task 2)
    iter := 0
```

```
   do
        t := task1 or task 2 in order
        improve = swapProcessors (t)
        if (improve < delta)   /* calculate # of iterations without improvement */
             iter++
        if iter > k
             goto end
   end do loop
end – when all edges are scheduled
```

Table 10: Pseudo code for the greedy algorithm.

6.3.2 TASK Algorithm: Local Search for Graph Assignment

This algorithm is taken from [61]. The algorithm constructs a list of the highest cost nodes and then evaluates if moving them to new positions will improve the task allocation cost. [61] defines several terms:

- *Entry node* where data is created (enters the task graph)
- *Exit node* where data is consumed (exits the task graph)
- Weight of a node $w(n_i)$ is the cost to execute task n_i
- $tlevel(n_i)$ the largest sum of communication and computation costs at the top level of a node n_i (from an entry node to n_i) excluding its own weight $w(n_i)$
- $blevel(n_i)$ the largest sum of communication and computation costs at the bottom level of node n_i, (from n_i to an exit node)
- *CP* The critical path is the longest path in the task graph, that is, the path with the highest communication & processing requirements

- $L(n_i) = tlevel(n_i) + blevel(n_i)$
  $L_{CP} = \max\{L(n_i)\}$

If the graph has been scheduled

- $pe(n_i)$ is the processor that task n_i is scheduled on
- $p(n_i)$ is the predecessor node that has been scheduled immediately before node n_i on $pe(n_i)$. If no other node has been scheduled on processor $pe(n_i)$ then $p(n_i)$ is set to 0
- $s(n_i)$ is the successor node that has been scheduled immediately after n_i on $pe(n_i)$

The TASK algorithm starts from an initial, feasible schedule. It guarantees for each step thereafter that the allocation either improves or stays at least as good. The initial allocation marks every edge in the task graph T unvisited. In addition, if two tasks are scheduled on the same processor, a pseudo edge of weight 0 is inserted between the two tasks, this creates a modified graph T'. There is also a variable called *nextk* for each processor k, and it points to the next task scheduled on processor k that has not yet been visited by the local search algorithm. Initially nextk points to the first node scheduled on k. A node is ready to be visited by the search algorithm if all its parents in T' have been inspected.

The Task algorithm is described in pseudo code given below.

```
Given: an initial schedule of nodes in the task graph T=(v1, e1) onto processors in
the processor graph P=(v2, e2)
Find LCP and each node ni ∈ LCP
while there are nodes to be scheduled:
  ni := a node in LCP that is ready
  find Lt(ni) where
   Lt(ni) = min{k ∈ v2}Lk(ni) = tlevel(ni)+blevel(ni)
   (node i is feasibly scheduled on processor k)
   if (t == pe(ni))
        do nothing
   else
        move node ni from processor pe(ni) to t
        modify pseudo edges in T'
        propate tlevel of ni to its children
    end
    mark node ni as scheduled
end – when all nodes are scheduled
```

Table 11: Pseudo-code for the TASK algorithm.

The complexity of the algorithm is $0(e + mn)$, where e is the number of edges in T, m is the number of nodes, and n is the number of processors.

The initial allocation will be a greedy allocation that starts by allocating all entry nodes first. Then non-entry nodes whose' parents have been allocated, are allocated. Each allocated is made so as to minimize the communication cost previously allocated nodes. When a single node is allocated the best allocation can be found by exhaustive search. The communication cost to already allocated nodes is found via the iterative deepening depth-first search algorithm [62].

The basic TASK algorithm does not deal with multiple processes or contention. This paper proposes a modification to TASK, where the individual processes are combined to form a larger graph for TASK to map. When a process is allocated, first execute TASK on the individual process. Consider this as an initial allocation of all sub-tasks on the sensor network and mark all edges and nodes unvisited. Then perform TASK on the super-set of processes. Note, insert pseudo edges between sub-tasks allocated on the same processors, even if the two sub-tasks do not belong to the same process.

6.3.3 Genetic Search Algorithm Combined with a Bidding Market Protocol

The third algorithm applies a genetic algorithm solution presented in [63]. The genetic algorithm finds the best heuristic mapping for a single process. The results of the genetic algorithm are combined with a market protocol algorithm. The market protocol manages contention between processes by allowing multiple processes to bid for a resource in a decentralized fashion.

Each process has an agent that executes the genetic search algorithm on its behalf. The genetic portion of the algorithm is not decentralized. This is not a large drawback for the scheme, because the genetic algorithm is only centralized for an *individual process,* and it is still decentralized with respect to other processes that are being mapped simultaneously. The task allocation (bidding) is decentralized. The genetic search algorithm probabilistically searches and

evaluates a set of solutions. Thus, it not only provides the best heuristic solution, but also searches through others that may be nearly as good. When a process cannot obtain a resource from its best choice by bidding, the process will try its next best alternative, which is given by the genetic algorithm.

The genetic algorithm works as follows: Choose a population size N_p and a number of generations N_g. Then build the first chromosome according to task priority. A chromosome, in this context, is just an ordered list of sub-tasks. The task priority is given by the sum of communication costs in and out of the task. The first chromosome is then randomly perturbed until a population size of N_p is reached. This is the first generation. To create the next generation, apply the mapping heuristic to generate a solution for each chromosome in the population. Save the mapping solution and cost. Calculate the cost and fitness of each chromosome, and apply crossover and mutation to the fittest chromosomes to form a new population. This occurs for N_g generations. When the algorithm terminates, the k best solutions can be obtained by taking the mapped solution for the k fittest chromosomes.

The mapping heuristic in [63] considers only computation cost. Thus, the mapping heuristic used in this paper is adapted to consider communication cost. The mapping heuristic, given a chromosome x, is as follows: Allocate nodes in the priority order of the chromosome x. Each node is allocated to minimize the communication cost to those already allocated. As described for the Greedy

algorithm, the best allocation can be found by exhaustive search, where each processor is tried, and the communication cost (cheapest path to already allocate nodes) is found via the iterative deepening depth-first search algorithm.

The fitness of each x chromosome is given:

$$f(x) = \frac{(MaxCost - \cos t(x))^t}{\sum_{j=1}^{N_p} (MaxCost - \cos t(j))^t}$$

where *MaxCost* is the communication and computation cost of the worst chromosome's mapping, and $cost(x)$ is the cost of chromosome x's mapping. t is a fitness scaling parameter that balances convergence and diversity. In this case t is set to 3.

After each step, chromosomes for the next generation are created via crossover and mutation. In [63] a one-point crossover is applied when the priority values immediately to the right of a randomly selected cross-site are swapped between two mating chromosomes. Mutation occurs when the priority of the site is randomly perturbed. Chromosomes are selected for crossover or mutation probabilistically according to their fitness value. The probability of a chromosome being selected is $\frac{f(x)}{\sum_{j=1}^{N_p} f(j)}$. This means that chromosomes with higher fitness values contribute more to the next generation. Crossover and mutation are done N_c and N_m times such that $N_c + N_m = N_p$. In this case $N_c = 2 \cdot N_m$

which means that 2/3 of replications are crossovers and 1/3 are mutations. After nodes have selected their k best solutions, they bid on those solutions. The solution they bid on contains a bundle of resources. Thus, combinatorial auctions must be held to obtain the bundles of goods. A combinatorial auction allows bidders to submit a single bid for a bundle of items. Winner selection in combinatorial auctions is also NP-hard. The solutions to combinatorial problems are typically centralized and require bidders to submit bids for all combinations of items [64]. However, [65] proposes an iterative solution to the combinatorial auction, where the auctioneer will iteratively raise the price of a contended good. Iterative auctions are more vulnerable to bidder manipulation and collusion. However, for the purposes of this work, agents will not price anticipate or delay their bidding.

At each iteration r, the price λ^r_p of a contended good p is adjusted so that:

$$\lambda_p^{r+1} = \max\{0,\ \lambda_p^r + s\sum_{i=1}^{M} d_{ip} - 1\}$$

Where $d_{ip} = 1$ if process i bids on resource p and s is a step-size parameter. The agents who bid on behalf of a process determine their bids by taking the normalized difference in cost between the k best solutions. If an agent is bidding on solution i out of k, the bid value is $\sum_{j=i}^{k-1} \frac{\cos t(j+1) - \cos t(j)}{\cos t(j)} + 1$. During the iterative bidding process, agents may raise their bidding price by s at each step

(up to a maximum of $d \cdot s$) if the contended resource appears in d of a process's k best solutions.

6.4 Simulation Setup

The Omnet++ [54] network simulator was used for this work. For the lower layers, a standard fading channel was used with an Aloha MAC (both of these are provided by the mobility framework). At the network layer, geographic routing and addressing was used.

The simulation consists of 400 nodes randomly placed in a 1000m x 800m rectangle. The nodes have a radio range of approximately 150m. The network is heterogeneous, featuring nodes with 3 amounts of availability: 99.9%, 95% and 80% availability. Processing power, dynamic memory and bandwidth are randomly assigned to nodes. In this model, bandwidth is assigned to individual nodes, and can be used either to send or receive. When two nodes are communicating at rate k the remaining bandwidth decreases by k. This is just a high-level model and not meant to represent underlying medium access routines (TMDA, CDMA etc.). The nodes are also assigned to one of ten locations based on their coordinates within the square.

When a node is available, it wakes up every unit of time and decides with probability 0.01 to instantiate a new process with a random lifetime. The time step, process lifetime and probability were adjusted so that on average there

were 20 tasks mapped on the network at a time. The simulations generated a total of 300 tasks to be mapped on the network.

Tasks are also randomly generated. First the number of tasks is randomly selected from a uniform distribution of 4 to 14 tasks. Next, each task is randomly assigned memory, processing and availability constraints. Each task is assigned an availability that is uniformly selected between 50-90%. These tasks may not be assigned on nodes with availability less than their requirements. Then, the task's connectivity is decided. On creation, a task communicates with any other previously existing task with probability 0.1. If tasks do communicate, the required bandwidth is randomly selected from a uniform distribution. Once this random assignment is complete, the tasks are divided into two groups: connected, and disconnected; and edges or random bandwidth are inserted until the disconnected group is empty.

For the second algorithm, there is a periodic adjustment of all tasks every 30 time units. For the third algorithm, genetic search and bidding, agents receive bids asynchronously; an agent will wait at least 2 time units before allocating resources to a bidder. Once a resource has been allocated, it cannot be reallocated for another 10 time units. However, if an agent receives a higher bid for the resource after 10 time units, it is free to reallocate it to the next highest bidder.

The algorithms were also modified to map processes incrementally. The mapper first scans the network to see if any sub-task in the process is already running on

the network (as part of another process). If a task is found, it is marked as allocated. These allocated tasks are "fixed points" in the allocation algorithms. Their only constraint is to then allocate the other tasks that communicate with the fixed tasks.

In order to evaluate the performance of the algorithms, they were compared to the optimal algorithm. The optimal algorithm was implemented as an exhaustive search and took prohibitively long to simulate. Thus, in order to compare performance, the optimal algorithm was simulated on a smaller network of 80 nodes with only 4 locations. For the optimal comparison, only three processes were created and mapped, one with 4 tasks, 9 tasks and 14 tasks respectively.

Last, a regular topology is used as a control. In a regular topology it is possible to easily calculate the optimal allocation; all 800 nodes are placed on a grid, there are still 10 rectangular locations or regions, and the nodes all have the same computation, communication, and memory capabilities. The regular topology illustrates the differences between the algorithms and how close they come to the optimal. The two task descriptions, shown in Figure 10, are used. Figure 10 shows the location constraints of the tasks as well as the communication rate between subtasks (labeled on the edges). Tasks A, B, and C are restricted to location 1 while tasks D and E are restricted to location 2. The two tasks in Figure 10 differ only in the communication cost of two links. This was chosen to illustrate its impact on the mapping algorithms. Moreover, location 1 and 2 are

adjacent to each other and the node capabilities are sized so that three tasks can fit onto a single node. The optimal solution is then to allocate tasks A, B, and C onto a single node that is within 1 hop from nodes in location 2, and to allocate tasks D and E on a node in location 2 that is one hop away from the node on which Tasks A, B, and C are allocated.

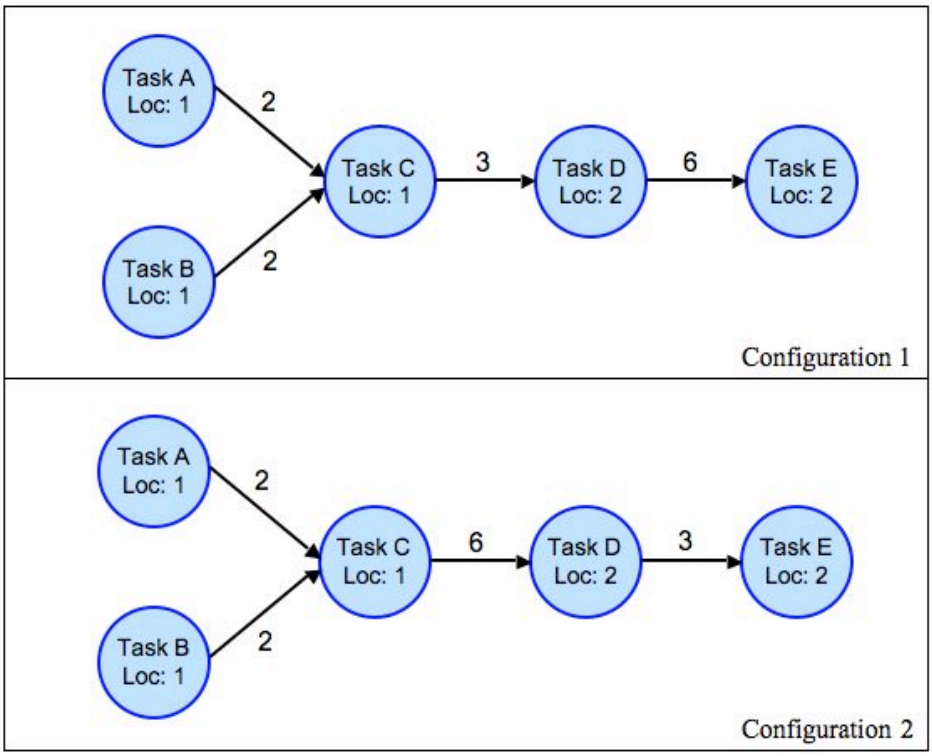

Figure 10: Two task descriptions.

The key to mapping these tasks efficiently is mapping tasks C and D on the border between location 1 and 2. The three algorithms map tasks differently. Take the greedy algorithm, for configuration 1, it will map tasks D and E first

because they have the largest communication link connecting them. These tasks can be mapped on any node in location 2 and are not guaranteed to be mapped near location 1. Therefore when task C is mapped it may incur additional cost to transport data to wherever E and D are. However, in configuration 2, the link between C and D is the largest and therefore tasks C and D are mapped first. The algorithm will place them one hop apart and the remaining tasks will be placed on the same nodes. For the second configuration the greedy algorithm will get the same result as the optimal algorithm.

The TASK and genetic search algorithms map each task independently. For the TASK algorithm there is only one initial mapping. In configuration 1 task D is mapped first; and in configuration 2 task C is mapped first. The performance of the algorithm relies on the chance that a node close to the border of regions 1 and 2 is chosen in for the initial task mapping. For the initial task that is mapped, all nodes in a region appear equal because there are no communication constraints. This is is also similar for the genetic algorithm, but it switches task mapping order and tries more combinations. Thus, it has a larger chance of picking a node close to the border for tasks C and D.

6.5 Results

The first results show the cost of mapping processes shown in Figure 10 onto the regular network. The genetic and bidding algorithm achieved the optimal

solution for configuration 1, and the greedy algorithm achieved the optimal solution for configuration 2.

	Optimal	Greedy	Task	Genetic + Bid
Configuration 1	3	12	9	3
Configuration 2	6	6	11	8

Table 12: Results of mapping two processes onto the regular grid network.

The next set of results examines the performance of the allocation algorithms, comparing the resulting cost of the mapped processes. Figure 11 shows the solution cost for each algorithm, including the optimal for the smaller simulation. In this case, genetic and bidding algorithm's cost is about 30% lower than the greedy and TASK algorithms for the tasks that are mapped. The genetic search and bidding algorithm performs within 5% of the optimal algorithm. On the other hand, the greedy and the TASK algorithm have similar performance for the smaller tasks' mapping. However, the TASK algorithm performs slightly better on the larger task's mapping. Figure 12 shows the average mapped cost across all tasks for the different algorithms. The cost has been normalized so that the greedy cost is equal to one. The results in Figure 12 show a much smaller performance increase of the genetic and bidding algorithm over the greedy/TASK algorithms. Results show that the genetic algorithm produced only 6% better mapped cost than the greedy algorithm. The difference between the TASK and greedy algorithms are about the same; the task algorithm outperformed the greedy algorithm by a small margin.

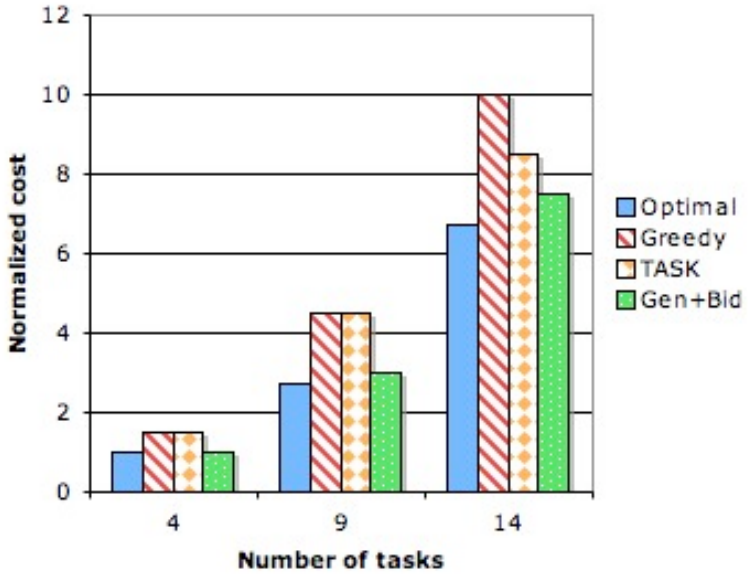

Figure 11: Mapped costs for 3 processes vs. different algorithms including optimal exhaustive search.

One explanation for the difference in performance of the genetic and bidding algorithm is when only three tasks were mapped, the tasks always obtained their first choice. However, in the larger experiment, on average 20 tasks were mapped onto the network simultaneously. Therefore, in the bidding stage, processes could not always obtain the resources that comprised their first choices, and had to settle for a less optimal mapping. Figure 13 shows the percentage of times that nodes obtained their nth choices. 71% of times nodes were able to obtain their first choice. However, in 9% of cases nodes could only obtain their fourth choices.

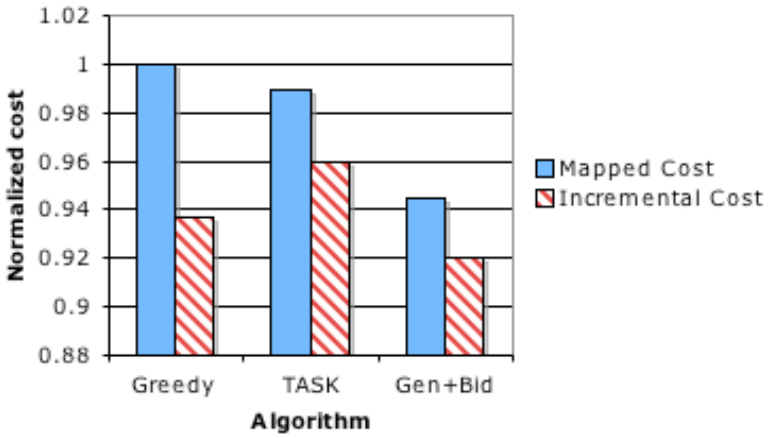

Figure 12: Average mapped cost for tasks vs algorithm type.

Further, Figure 12 shows the average mapped cost for the incremental version of the algorithms compared to the non-incremental cost. Contrary to intuition, the incremental mapped cost is lower than when each task is allocated independently of what is already on the network. The reason is that the mapping is resource constrained and sharing resources frees up more resources for subsequent mappings. These subsequent mappings are then more optimal.

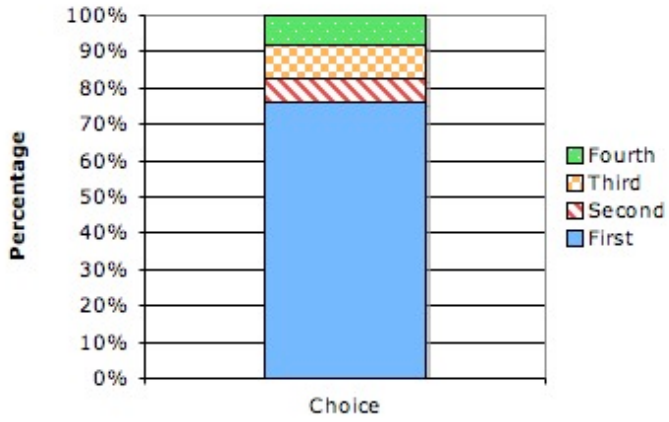

Figure 13: Tasks' choice obtained in they genetic search + Bidding algorithm.

Figure 14 also shows the mapped cost for different algorithms, however, the results are also categorized by process size, the number of tasks that a process contains. Results are normalized to the cost of the greedy algorithm for 4 tasks. For all algorithms, the cost increases monotonically with the number of tasks. For the greedy algorithm, the cost of mapping increases 4 times as the number of tasks go from 4 to 9 and then doubles when the number of tasks increase from 9 to 14. Again the genetic and bidding algorithm outperforms the other two algorithms. However, the difference is less for small tasks because there are fewer degrees of freedom, so the additional searching and mapping yields fewer results

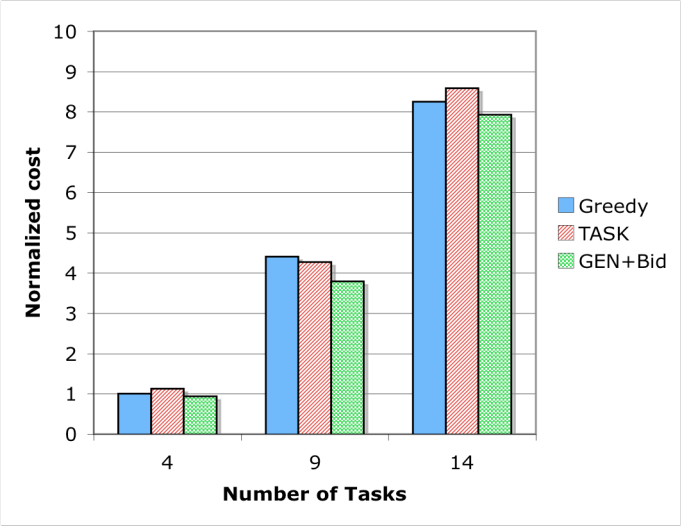

Figure 14: Average mapped cost for application sizes vs algorithm type.

The TASK algorithm actually performs worse than the greedy algorithm for the extreme processes – those with a small or large number of tasks, but performs better for the average processes. This is most likely because it maps the node with the highest bandwidth by itself first. That node has high-bandwidth links to other nodes and if that node is mapped to a processor that cannot accommodate the other nodes the high bandwidth works against it. The greedy algorithm prevents this problem by mapping high bandwidth links (i.e. two tasks at a time) first, while the genetic search searches through multiple mapping orders to avoid this problem.

The performance difference can also be seen by looking at the number of tasks that are mapped on the same processor. Figure 15 shows a histogram of tasks mapped together for processes containing 9 tasks. The histograms show on average how many times processors contained a single task, 2 tasks and 3+ tasks.

Note, on average, 4 tasks were mapped to a processor less than 0.01 times per mapping, and 5 tasks were never mapped to a single processor.

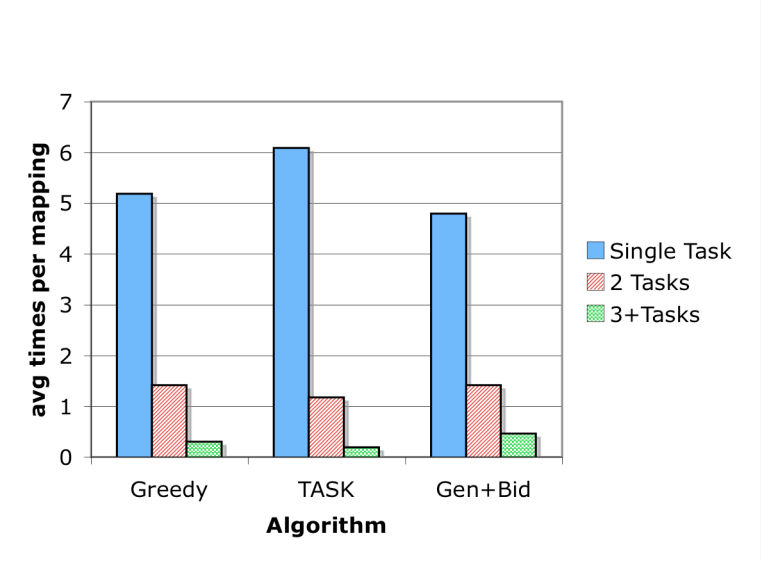

Figure 15: 3 Histograms of the number of times tasks were mapped *n* to a processor.

From Figure 15 it can be seen that it is more common to have tasks mapped alone, out of 9 tasks, an average of 5 are mapped alone across all algorithms. However, the genetic algorithm and bidding performs better than the other two because it has the lowest single task per processor and the highest 3+ tasks per processor. On the other hand, the greedy algorithm has a higher rate of grouping tasks together than the TASK algorithm. The greedy algorithm tries to map tasks together more aggressively, however, the performance of the two algorithms was close, so it stands to reason that the greedy algorithm could not find a good mapping for the tasks it could not group together.

The next set of results show the complexity of the mapping algorithms. The complexity has been normalized so that the complexity of the non-incremental greedy algorithm is one. The complexity was calculated by summing the number of processors that the algorithm considered when placing each task. Figure 16 shows the mapping complexity for each algorithm. As expected, the incremental algorithms have lower computation complexity than their corresponding non-incremental versions. This is because some tasks are fixed and the problem effectively becomes smaller. The TASK algorithm incurred additional computation cost during its adjustment phase every 30 time units.

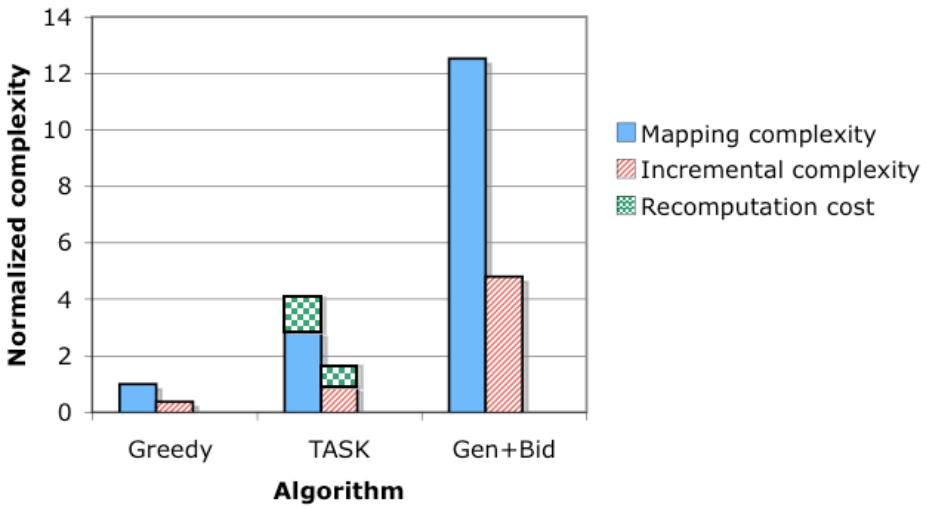

Figure 16: Mapping complexity vs algorithm type.

The greedy algorithm is the least complex, the TASK algorithm is about four times more complex and the genetic algorithm is 12 times more complex because it evaluates 9 mappings for each process (3 generations and a population size of

3). These ratios are the same for both the incremental and non-incremental algorithms respectively. The mapping complexity does not cost the sensor network energy in terms of communication cost, however, the 12 times overhead for the genetic algorithm only yields a 6% improvement in mapped cost.

Figure 17 shows the initial mapping complexity for the three algorithms for different task sizes. The re-computation cost is not included per process because it is done for all tasks together. The computation complexity grows super-linearly; the complexity difference to map a process with 14 tasks is about 10 times more than to map one with 4 tasks.

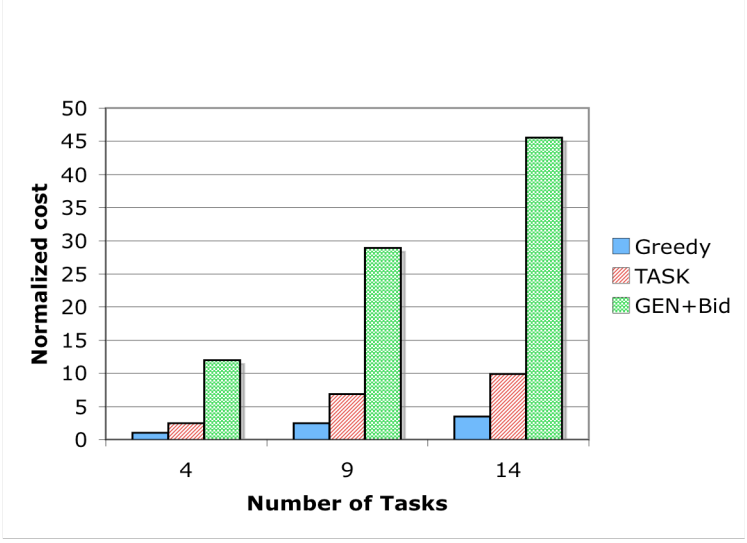

Figure 17: Mapping complexity for different task sizes vs algorithm type.

Unlike the greedy algorithm, the TASK and genetic search algorithms incurred other overhead during the course of the simulation. The TASK algorithm did

adjustments every 30 time units. The genetic and bidding algorithm incurred additional costs due to the fact that processes' agents had to bid on resources and had to re-bid if they did not obtain their first choice. Agents could also lose a resource that had been allocated to them, in which case they would have to re-bid for other resources. Table 13 shows the number of processes that were moved from their original allocation during execution, the cost of moving the process, and the cost of any additional bids to obtain the new resources.

Algorithm	Proc moved	Cost to move	Cost of add bids
Task	30	20	0
Gen+Bid	10	10	102

Table 13: Additional communication costs incurred during simulation.

The cost to move the process was normalized to the cost that it would take on average to map a process. For the TASK algorithm, 30 processes were moved at a cost of normally allocating 20 processes. The cost to move a process is lower, because during the adjustment only a few tasks of a process are moved. For the genetic and bidding algorithm, the cost to move the process is equal to the cost of mapping a process from scratch. In terms of the total overhead added by moving tasks, bidding adds a 10% overhead (considering that mapping 200 processes would normally incur 200 in normalized cost) and TASK adds a 20% overhead.

The genetic and bidding algorithm adds additional overhead even if no tasks are moved, because processes do not always obtain their first choice. The process needs to resubmit bids for its second through fourth choices. The last column in

Table 13 shows the overhead for all additional bidding. This overhead is significant at 51% of the cost to map a task. Last, in the bidding and genetic algorithm, the process is not mapped immediately. On average it took 2.6 units of time for a process to be mapped.

6.6 Discussion

This chapter presented a formal problem definition for the file allocation problem. Three algorithms were presented and empirically compared. The algorithms mapped each task independently, or followed an incremental approach where sub-tasks were shared across processes. The incremental approach yielded a lower mapped cost for all algorithms because it freed up more resources in the network. The incremental approach also had lower computation complexity. Out of the three algorithms the genetic search and bidding algorithm performs the best. However, it only yielded a 6% performance increase while adding a 61% communication overhead and a 12x computation complexity increase. In addition, the bidding algorithm also introduces a delay before the process is mapped onto the network. Due to the excessive overhead of the bidding algorithm, it may not be appropriate in the sensor network context where low complexity is very important.

The TASK algorithm performed only marginally better than the greedy algorithm (one percent) while still adding 20% communication overhead and 4x

complexity. Thus, for sensor networks the greedy algorithm was the best in terms of a performance vs. overhead tradeoff.

7 SNSP TinyOS Implementation

This section outlines the implementation that was done on two mote platforms. The final goal of the implementation was to build a proof-of concept SNSP platform and take measurements to evaluate the performance of the content replication service and task allocation service. The remainder of this chapter is organized as follows: Sections 7.1 gives an introduction to the demo and the process of building it. Sections 7.2 through 7.4 give a more detailed overview of the network hardware, applications, persona and the gui. Section 7.6 presents measurements that were taken from the testbed and Section 7.7 concludes with a discussion.

7.1 Creating the Implementation Scenario

There are two important aspects of SNSP that the implementation must demonstrate: First the multi-platform aspect of SNSP showing applications that

are agnostic to the hardware platform and can communicate cross-platform. The second is the dynamic mapping and content replication outlined in chapters 5 and 6. As such the implementation occurred in two phases. The first phase, a complete application with the requisite sensors and actuators, was implemented. It consisted of both Mica2 [66] and Telosb nodes [67]. Initially, all content was stored on the laptop that formed the bridge between the Telosb and Mica2 networks. The implementation had no dynamic mapping. The initial applications on the testbed were fire alarm control and HVAC control (thermostat) with demand response [68]. Demand response (DR) is fully explained in Section 7.3.1. These applications also incorporated persona. Figure 20 shows a picture of the initial hardware used, while Figure 22 shows the application GUI, which demonstrates the active applications and the persona on the testbed. Details about the hardware and the GUI are discussed in Sections 7.2 and 7.5.

Phase 1 was then augmented with the content management and replication algorithm, described in Section 5.3.2, and the greedy dynamic mapping scheme, described in Section 6.3.1. In order to make the mapped applications more interesting, several virtual sensors and actuators were added to the testbed. The node pretends it has a sensor or actuator attached to it and registers this information with the content repository service so it can be used by other applications. This allowed the testbed to support a wider variety of applications for mapping.

7.2 Testbed Setup

7.2.1 Hardware

The testbed consisted of 40 nodes, 31 Telosb nodes and 9 Mica2 nodes. The Mica2 nodes are older generation nodes made by Crossbow. Figure 18 shows an image of the Mica2 mote. The Mica2 mote is typically powered by 2 AA batteries, and can tolerate a voltage range of 2.7-3.3V. There is also an external power connector. It has a 51 Hirose connector that connects to an expansion board (also made by crossbow). This expansion board allows one to connect sensors and actuators to the analog to digital (ADC) converter channels of the mote. The Mica2 has a 10-bit ADC that has 8 channels with 0-3V input.

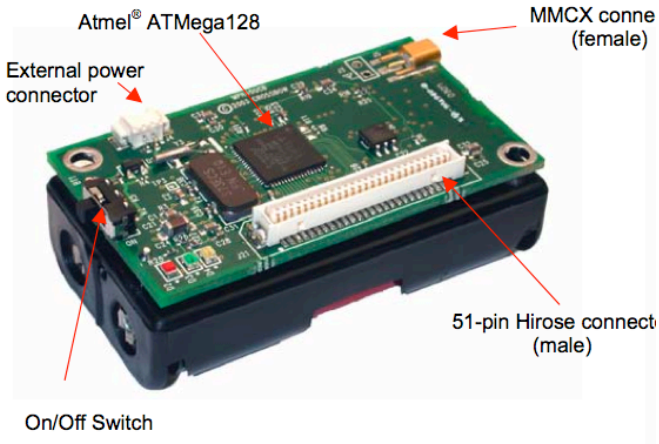

Figure 18: Annotated Mica2 mote (taken from [66])

The Mica2 uses the Chipcon CC1000 (see [69] for data sheets), FSK modulated radio and can come in three models according to their RF frequency band: the MPR400 (915 MHz), MPR410 (433 MHz), and MPR420 (315 MHz). The 915MHz

model was used in the testbed. The microcontroller is an Atmega128L micro-controller. In order to program the Mica2, an additional board is required. For the testbed, the MIB510 Serial Interface Board [70] was used, which allows programming over the parallel port, as well as communication with the mote over the serial port during operation. The Mica2 has 128k bytes of program memory, 512Kb of external flash storage for data, and 4Kb of RAM.

Telosb, the second platform that is used in the testbed is made by Moteiv [67]. Figure 19 shows a detailed view both the back and front sides of a Telosb node. The Telosb nodes use a 250kbps 2.4GHz IEEE 802.15.4 Chipcon Wireless Transceiver (see [69] for features and usage). The node may be powered by two AA batteries in the operating range of 2.1 to 3.6V DC. The Telosb node has a USB port for programming or communication. Tmote Sky uses a USB controller from FTDI to communicate with the host computer. In order to communicate with the mote, the FTDI drivers [71] must be installed on the host. FTDI provides drivers for Windows, Linux, BSD, Macintosh, and Windows CE. The Telosb node also receives power from the USB port.

The microcontroller is a Texas Instruments MSP430 F1611 microcontroller featuring 10kB of RAM, 48kB of flash, and 128B of information storage. The node has 1024kB of external flash to store data, 8 external ADC ports and 8 internal ADC ports. The ADC internal ports may be used to read the internal thermistor or monitor the battery voltage. Tmote Sky has two expansion connectors, a 10-

pin IDC header and a 6-pin IDC header, for connecting peripherals. The connectors provide digital input and output signals as well as and analog inputs.

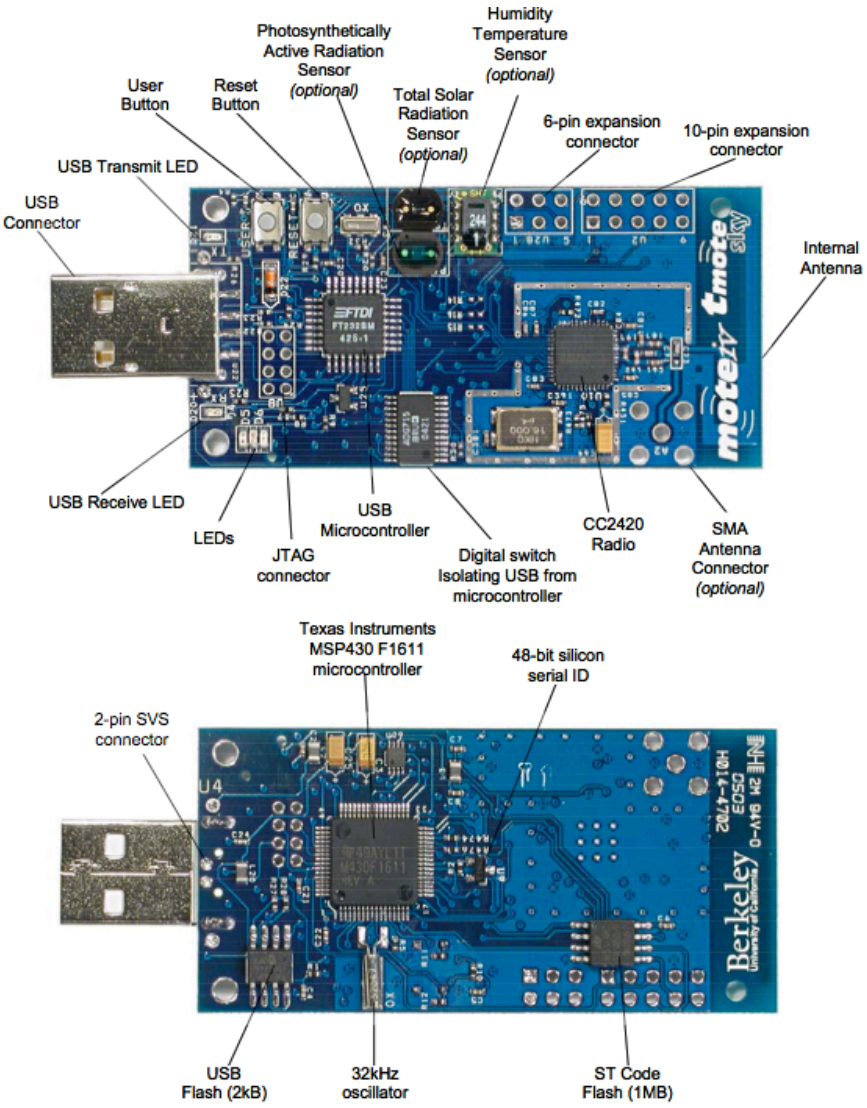

Figure 19: TelosB mote (taken from [67])

Figure 20 shows the components used in the first implementation of the testbed. The laptops are used as display, the TelosB and Mica2 nodes are either visible by themselves, or in enclosures connected to peripherals. The smart thermostat

(black box with two knobs in Figure 20) contains two Mica2 nodes that are connected to the rotating potentiometers via GPIO pins. The fan is turned on and off by another Mica2 node connected to a digital switch in the blue box.

Figure 21 shows the expanded testbed of 30 Telosb motes on which the application mapping was tested. This time the nodes are all wired together with USB that provides power to them. All communication is wireless. It should also be noted that the nodes are all within one communication hop from each other.

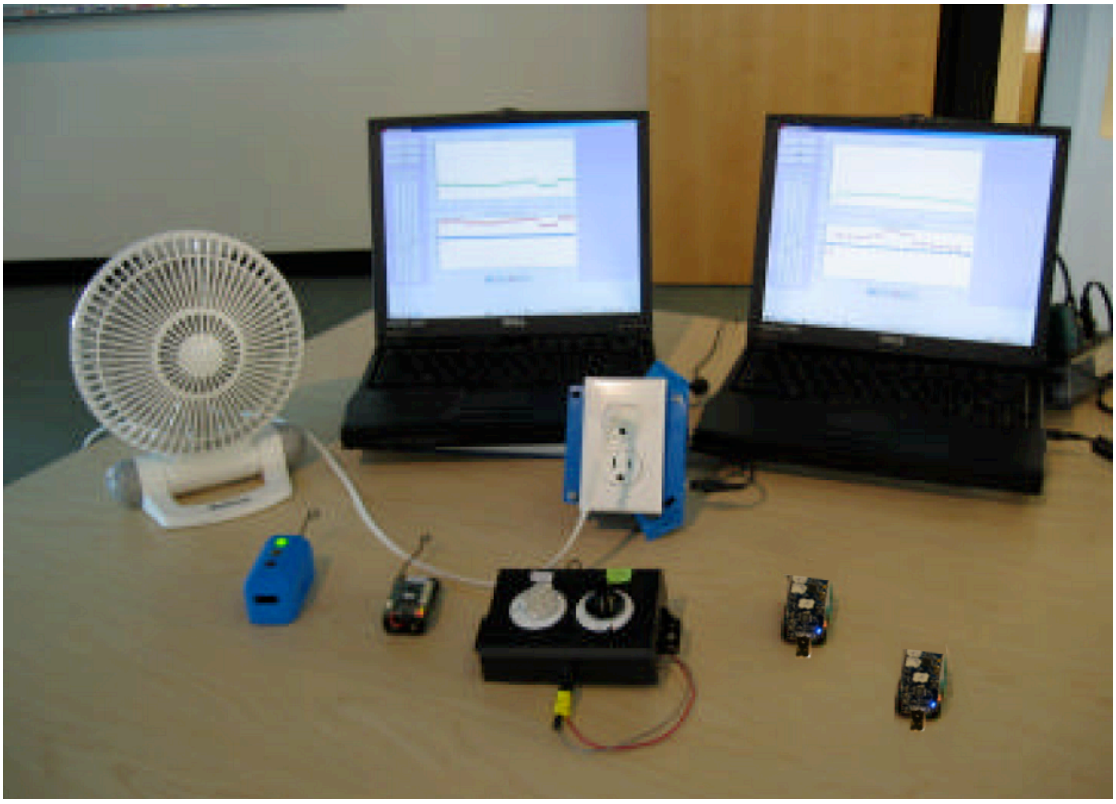

Figure 20: Testbed with HVAC control and DR that bridges Mica2 and TelosB nodes.

Figure 21: 31-Node testbed, powered via USB & batteries.

7.2.2 Location and Connectivity

While the nodes are all close to each other in the lab, the testbed is situated in a virtual house with 8 rooms. The layout of the house is reflected in the GUI, which is shown in Figure 22. The connectivity was artificially restricted so that only adjacent rooms could communicate with each other. This resulted in a maximum of four hops across the house. All addressing is done via location, based on the semantic locations that are outlined by the application.

7.2.3 Sensors and actuators

Several real and virtual sensors were used in the testbed. First, the Telosb's built-in temperature sensor is used to measure temperature. Sensirion AG

manufactures the sensor, more information can be found in the SHT1x datasheet available at [72]. Second, the comfort and desired temperature sensors, which comprise the smart thermostat, are rotating potentiometers that are attached to an ADC pin on the Mica2 nodes. Virtual sensors include: indoor motion sensor (achievable via light beams), door/window position sensors (achievable via magnetic contacts), a smoke detector, and a price signal detector.

For the actuators, the HVAC can be switched on and off by a Mica2 mote. An electro mechanical switch connected to a GPIO pin controls the current to the fan. The switch takes 12V input and, in order to allow the 3V GPIO pin to switch it on and off, a transitor switch is placed in front of it. The second actuator is a board with three large LED's, used to indicate the price electricity. These are simply connected to the node's GPIO pins. The third actuator is a fire alarm, which consists of a buzzer connected to a node's GPIO pins.

7.2.4 Content Replication & Capacity

As mentioned above, the probabilistic replication scheme was implemented. This scheme will replicate the content with probability 1/p on any read. On a write, all replicas are deleted with probability $1/\sqrt{3}p$. In the implementation, the content replication service has a single cluster head that keeps track of where the data is replicated. This cluster head is located in the dining room of the house and is reachable by all nodes. Every node can store up to 10 content items locally. The content in the network is made up of the service descriptions that are

accessed when tasks are mapped onto the network. There is a total of 60 content items (determined by the services in the network).

7.2.5 Task Allocation

In the implementation, nodes in the network decide when to map applications. When a node decides to map an application onto the network, it must first query the content management service to find out where the other services are in the network. When the node receives the information it runs the greedy algorithm to decide where to place the tasks in the network.

7.3 Applications

There are 6 main applications that nodes can instantiate. A main application in this case means that nothing else is using it as a service. The main applications are: HVAC control (standard), HVAC control with DR, Home security, tracking children, localization, and fire system control. The interaction of HVAC control with DR is explained in Section 7.2.1, and Motetrack (a practical implementation of localization [44]) is explained in Section 7.2.2. There are 4 other complex services that these applications use. A complex service is one that uses other services. The four complex services are: Perimeter security (uses door/window sensors in different locations in the house), Internal security (uses some door/window sensors and internal motion detector), DR display (controls

aggregate of individual DR displays), and temperature aggregation (provides average temperature from a number of rooms).

7.3.1 Demand Response and HVAC Control

In electricity grids, demand response (DR) refers to mechanisms to manage the demand from customers in response to supply conditions. The goal of DR is to smooth out the energy usage curve so that resources are not underutilized during low times, and so that the peak usage does not spike so high that smaller, less efficient plants need to be brought online to fill excess demand. Today, typically only commercial and industrial users participate in DR, and user critical peak shedding is usually achieved by calling customers a day ahead of time, this is known as the day-ahead market. [73] explains the types of programs offered to large customers.

However, the real potential of demand response is to bring it to all customers, including residential customers. A 2006 Carnegie Mellon study [74] looked at the importance of demand response for consumers for the Pennsylvania-New Jersey-Maryland Regional Transmission authority. Results showed that even small shifts in peak demand would have a large effect on costs for additional peak capacity: a 1% shift in peak demand would result in savings of 3.9%, billions of dollars at the system level. An approximately 10% reduction in peak demand would result in systems savings up to $28 billion (this is just for Pennsylvania, New Jersey and Maryland).

Sensor network technology is promising to instrument the home so that real-time pricing may be offered to residential customers. The particular flavor of demand-response implemented on the testbed is one where the residence receives a price signal every 15 minutes. The user indicates a comfort level of how much they are willing to pay to be comfortable. Based on the comfort level, the sensor network will adjust the heating and cooling in the house, in addition to turning on/off other appliances. In the testbed, the sensor network controls a fan, and will also turn on red, yellow, or green lights on appliances to indicate the price of electricity to users. The threshold at which red, which means expensive, is displayed depends on the comfort-level chosen by the user.

7.3.2 Motetrack Localization

Motetrack [44] is an RF localization algorithm developed by Harvard. There are two types of nodes in Motetrack, beacons and the target node. The target node must first be trained with the beacons before localization occurs. During the training phase, the target beacon is moved around the area where localization is to occur. At certain locations, it is given the coordinates and then collects a signature of all the beacons' rf transmissions at the given coordinates. The target node in effect builds up a database of signatures at the different locations. After the training phase the node is ready to enter the localization phase. During this phase it tries to match the rf signature that it is currently receiving with ones in its database. The estimated position is a mixture of the locations.

The beacon nodes hop on a number of selected channels N and transmit a beacon every 100ms. The target node also hops on these channels with a frequency of $N \cdot$ 100ms, where N is the number of channels. This ensures that the target node receives a message from every beacon on the target frequency. However, it also ensures that the nodes cannot participate in a normal network for other communication.

In order to adapt motetrack to work with SNSP, the nodes have to return to the default channel to participate in the network. The beacon nodes are modified to stay on the default channel until they need to transmit a beacon. Beacon nodes then go to the specified channel, transmit the beacon and return to the default channel. They also provide a radio interface to SNSP. SNSP can use the radio interface to determine if the radio is busy (off the default channel) or not.

The target node needs to hop to all the frequencies that the beacons are being transmitted on for a larger amount of time to receive all beacons. Because the radio's frequency switching is rather slow (>10ms), the target node instead does 50% duty cycling. This means that it stays on the default channel for $N \cdot$100ms, and then hops to the first channel and receives beacons on it for $N \cdot$100ms. It continues alternating between the default channel and the next beacon frequency so that it cycles through all N beacon frequency channels. The target node also exposes the radio interface to SNSP.

The modified motetrack along with SNSP was deployed at Telecom Italia's lab in Berkeley. Measurements indicated an accuracy of ±2m in the office. With further smoothing of measurements (when in a room the estimates will be in that room 90% of the time and 10% will be just outside) the accuracy could be improved. However, motetrack's performance was good enough to identify most of the time whether the target was in one of the 7 offices, the conference room, the kitchen, or the cubicle area.

7.4 Persona

The implementation included persona: there is a homeowner and a fire department. In the implementation, a persona may have permissions and properties. The third persona aspect, preferences, is not included in the implementation. When a persona is present in a space, it registers with the content management and repository service. Currently the persona sets its permissions in the network. That is, it notifies both the content repository service and the corresponding service in the network of any access restrictions. When the content management and repository service is queried, it will return only the results that a persona may access, i.e. results of services that have not been restricted by another persona. Similarly, a service will only respond to a query from a persona that has authorization to query. All services start out without any authorization restrictions.

7.5 Testbed User Interface

The User Interface is meant to visually demonstrate the activity in the network. The UI consists of a GUI that shows all the services registered with the content repository as well as the services that are active in the network. Figure 22 shows a screenshot of the GUI. The top area displays the names and locations of network services. The bottom portion shows the layout of the house. Services pop up at their locations when they activate. Services are displayed as boxes and personae are displayed as triangles. The boxes are color coordinated to show permissions. In Figure 22 the box in bedroom 1 is red, indicating that it may be accessed by only the fire department. In the first demonstration, all services appear in the lower portion of the GUI (on the house map) and complex services show a list of other services that they are using. In order to demonstrate dynamic mapping, the GUI was modified so that only the complex services appear when they are activated and disappear again as they are unmapped.

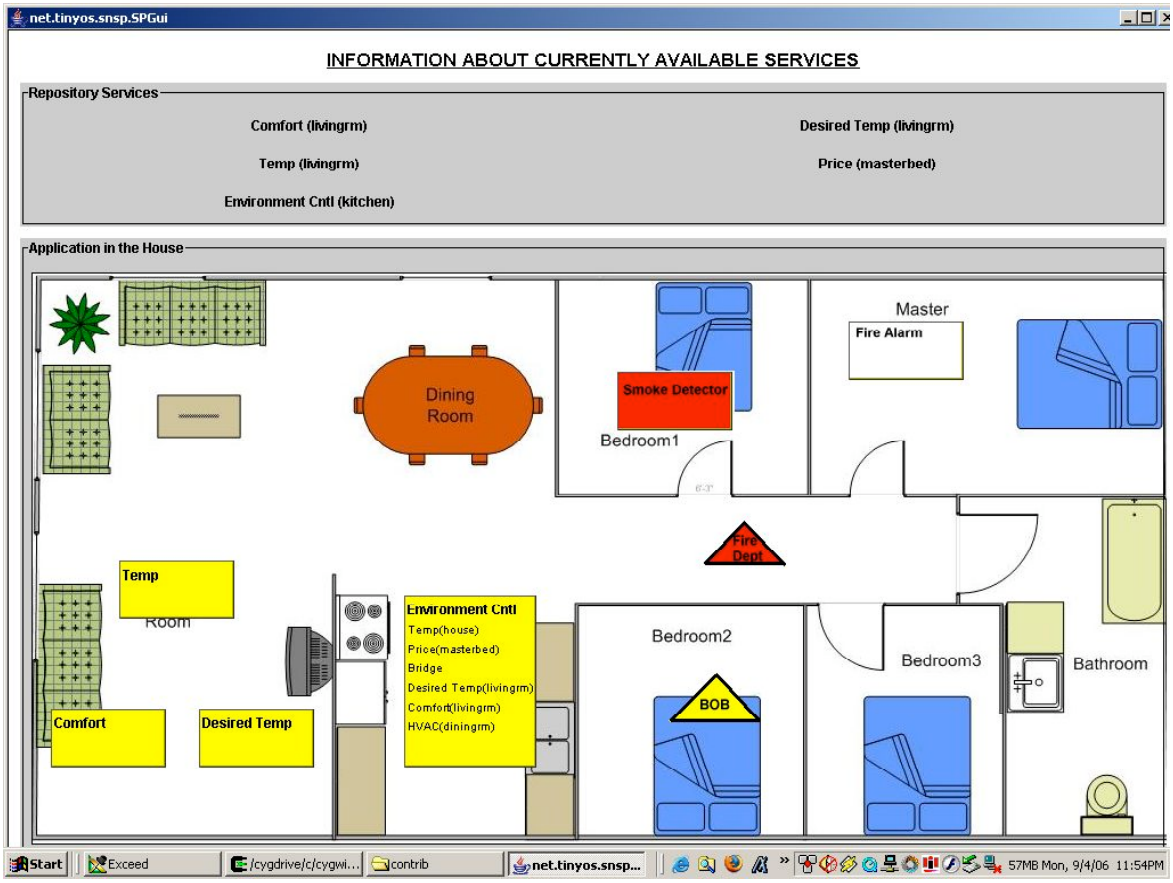

Figure 22: GUI showing the house, repository services, and applications that are mapped on the network.

7.6 Experiment

7.6.1 Setup

The testbed for the content management and task allocation experiment consists of 31 TelosB nodes. Each node has an additional TinyOS module that wakes up at 10 second intervals and then decides with probability 0.01 to instantiate one of the applications. The module is provided with a list of 10 applications which it

can instantiate and it chooses one randomly. The applications are composed of a list of other services that communicate with each other. The module then queries to repository to find out where the requisite/equivalent services are. As its local middleware copy receives responses from the content management service it decides with probability 0.1 to cache a local copy. The node may store up to 10 items locally. The module then sends activation messages to the services that it has decided to use in the network.

The module also decides what the lifetime of the application is. The average application lifetime is 40 seconds. These time constants were chosen so that the experiment would map a reasonable number of applications in a short time. Once an application has exceeded its lifetime, the original module that mapped the application deactivates it, and each sub-service comprising the application.

Further, the experiments were conducted with two network configurations. In the first configuration, the node capacity was assigned so that on average 2 tasks (a maximum of three tasks on nodes with larger capacity) could be mapped onto the same node. For the second configuration, an average of 3 tasks, and a maximum of four, could be mapped onto the same node. Note that the sub-tasks in the application all had the same CPU requirement.

As mentioned above, there is 60 total content items in the network. The experiment ran for 85 minutes and in that time 517 applications were successfully mapped onto the test network. Results regarding the performance of

the content management and task allocations algorithms are collected every 8 minutes.

7.6.2 Results

The first set of results show the performance of the file allocation algorithm. Table 14 shows the number of replicas in the network over time. The number of replicas increases linearly as nodes access data remotely. At the end of the experiment there are 186 replicas, which means that the network is roughly 2/3 full. As the number of replicas increase, nodes should find more information in their local caches.

Time (min)	Replicas
8	32
16	54
24	72
32	97
40	119
48	135
56	149
64	169
72	178
80	186

Table 14: Number of replicas as a function of time

The lower line in Figure 23 shows the proportion (between 0-1) of queries for which results were found in the node's local cache. This proportion increases from 0.07 at the start to 0.44 at the end of the experiment. The cost of retrieving content depends on how many results are found locally. The higher graph in Figure 23 shows the average cost, in number of hops, to retrieve each content

item (note, when content is found locally it has a cost of 0). The cost to find results decreases from 1.7 hops to just under 0.9 hops.

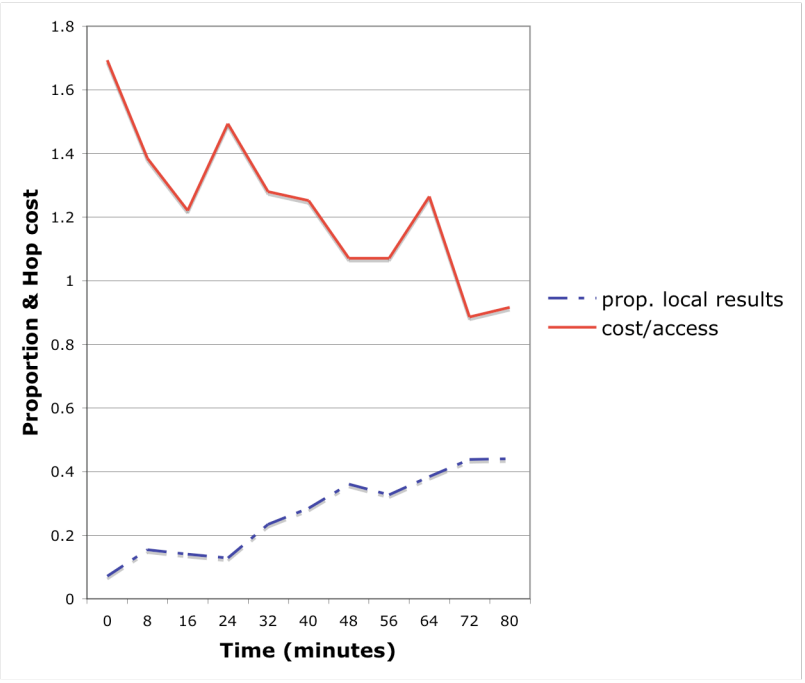

Figure 23: Cost per content access and the proportion of content found replicated on the actual node.

The time to map a process also depends on how many results are found locally. This is because a remote result is found by first querying the cluster head, and then the storage location in the content repository. Both of these steps are time consuming. Figure 24 shows the average amount of time it takes to map a process in milliseconds. This time includes finding and activating the relevant services. There is a decrease in the amount of time required to map a process as the number of replicas increases. The decrease in mapping time is steeper at the

start, and then levels off. This is because there is a fixed amount of time to invoke applications that remains unchanged by content management and replication.

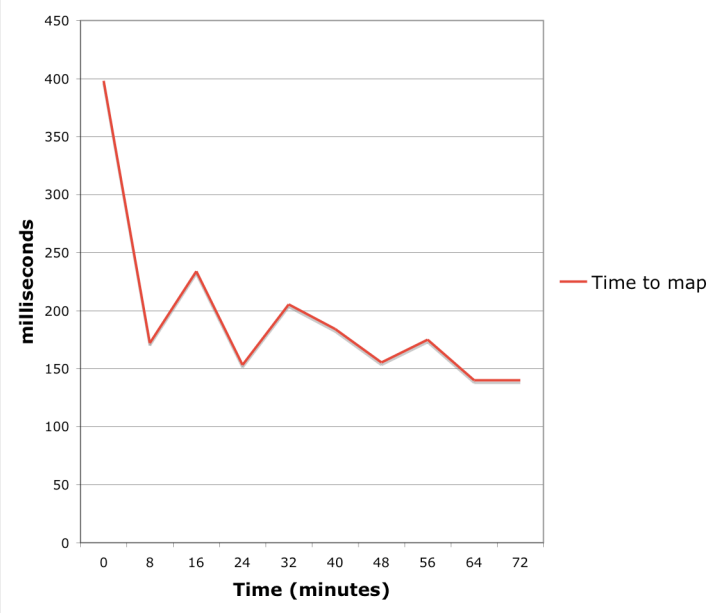

Figure 24: Time to map a process in milliseconds.

The next set of results was collected somewhat accidentally. During the course of one run, a microwave was turned on and off as students heated their lunches. The microwave causes RF interference with the 2.4 GHz ZigBee radios. Figure 25 shows the effects of the interference on the time to map an application. The microwave was turned about 50 minutes into the simulation. The time to map increases drastically from 300ms to 700-900ms. This is because packets are lost due to interference and nodes must retransmit the packets.

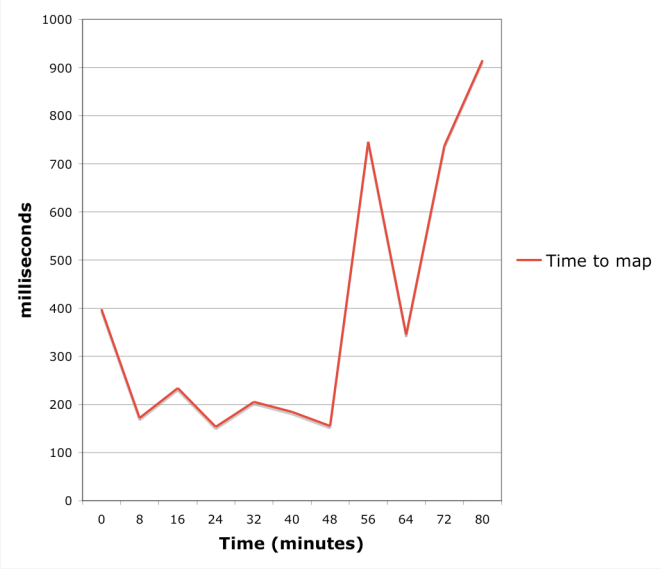

Figure 25: Time to map with rf interference.

The task allocation cost is only compared across the two elements, higher and lower CPU parameters, of the simulation. The task allocation performance cannot be compared to the simulations conducted in Chapter 6. This is because there are too many parameters in the test network that does not match those of the simulations in Chapter 6, e.g. locations, number of nodes per location, application structure with location constraints etc.

In the case of lower CPU availability per node, the mapping cost was 9% higher. Figure 26 shows two histograms of how many times four, three, two and one task respectively were mapped onto a single node. We see that for the lower CPU case, many more tasks were mapped individually (average of 3). The number of times 2 and 3 tasks were mapped together is about the same. However, where

the higher CPU configuration wins, on average 0.23 times 4 sub-services were mapped together, resulting in lower cost.

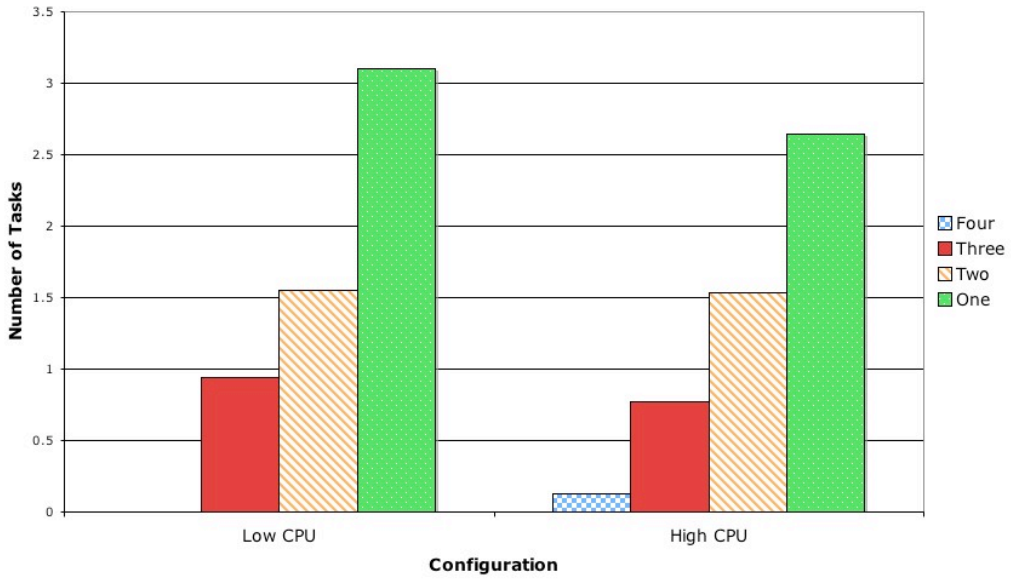

Figure 26: Histograms of the number of times tasks were mapped *n* to a node for the two network configurations.

7.7 Discussion

This section presented a proof-of-concept implementation. Results showed that the content management and replication algorithm helped reduce cost incurred to access content and the time of task allocation. Further, real-world results demonstrated that RF interference, in the form of a microwave, increases the mapping time substantially.

8 Conclusion

8.1 Summary

This dissertation presented SNSP, a distributed service-based operating system for sensor networks. SNSP is a full-fledged operating system with memory management, location transparency, and resource allocation. SNSP enables a user to create modular code through services. Services separate the content of the service from its implementation, and provide a clean set of functional abstractions that can be re-used. Further, SNSP dynamically maps applications onto the network at runtime to minimize communication cost between modules in the application and to fulfill application constraints such as availability, location and sensor/actuator requirements.

SNSP's architecture is comprised of two types of services: OS-level services and user-level services. OS-level services are 1) content management and replication,

2) task allocation, 3) resource discovery & repository, 4) resource utilization monitoring, 5) application migration, 6) fault detection and recovery, 7) security. They form the core of SNSP. User-level services are modular applications written by users that conform to the interface specified in Chapter 4. These services are application-layer functionality that can be reused across sensor networks.

This dissertation also presented and compared two sets of algorithms to achieve content management & replication and task allocation respectively. These algorithms were evaluated through simulation. A proof-of-concept implementation was also done on a sensor network testbed.

To summarize, the contributions of the thesis are:

- Identified and designed the core set of services for a distributed operating system
- Devised a programming model that allows users to create applications that are very flexible and can be run across different platforms.
- Created an integrated development environment (IDE) using Eclipse that allows programmers create and compile their SNSP sensC and .serv code into tinyOS code.
- Developed a novel file allocation algorithm that outperformed the two algorithms selected for comparison in terms of cost to access data. However, the algorithm's replication cost (overhead) was higher, resulting in an overall higher cost.

- Developed two file allocation algorithms: 1) a greedy file allocation algorithm based on Prim's minimum spanning algorithm, 2) a hybrid genetic search and combinatorial auction algorithm. These two algorithms were compared to the optimal allocation (exhaustive search) as well as a third existing algorithm. The hybrid algorithm outperformed the other two algorithms and came close to the optimal allocation. However, its cost was more than 10 times that of the greedy algorithm for only a 6% benefit.
- Proof of concept implementation of SNSP on a sensor network testbed. File allocation time and data access cost measurements from the testbed demonstrated the effectiveness in file replication to reduce both time and cost of allocating tasks.

8.2 Future Perspectives

This dissertation has provided the basic SNSP framework and thoroughly investigated two of its basic services. Although the other services were outlined, a thorough treatment of them would provide a variety of continuing research directions. For example, the implementation of security in SNSP is not specified in this dissertation. A great deal of research can still be done on two aspects of the security 1) its distributed nature and 2) low-power and low-complexity requirements. A low-overhead mechanism for fault detection and recovery may provide another research direction.

Further, the implementation was done on TinyOS, which does not support dynamically loadable code modules. Therefore, code was not transported/interpreted in the network, but simply activated. For a full SNSP implementation, further research is needed into devising a very compact code representation to reduce the overhead when a task is mapped.

Bibliography

[1] IEEE 1451 http://ieee1451.nist.gov/

[2] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, I. Stoica "A Unifying Link Abstraction for Wireless Sensor Networks" in *Proc. of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Nov 2-4, 2005.

[3] J. van Greunen and Jan Rabaey, "Content Management and Replication in the SNSP: a Distributed Service-based OS for Sensor Networks", CCNC 2008, Las Vegas USA.

[4] A. Boulis, C.-C. Han, and M. B. Srivastava, "Design and Implementation of a Framework for Efficient and Programmable Sensor Networks", *ACM MobiSys*, San Francisco, 2003

[5] W. Weber, J. Rabaey, E. Aarts, "Ambient Intelligence", *Springer Verlag,*2005

[6] M. Demirbas and A. Arora and M. Gouda "A pursuer-evader game for sensor networks" *Sixth Symposium on SelfStabilizing Systems (SSS'03)*, pp. 1-16, 2003.

[7] S. D. Gribble, G. S. Manku, D. S. Roselli, E. A. Brewer, T. J. Gibson and E. L. Miller, "Self-Similarity in File Systems", in *Measurement and Modeling of Computer Systems*, pp. 141-150, 1998

[8] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler "TinyOS: An Operating System

for Wireless Sensor Networks" *Ambient Intelligence* edited by W Weber, J Rabaey, and E Aarts 2005.

[9] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. "The nesC Language: A Holistic Approach to Networked Embedded Systems", in *Proc. of Programming Language Design and Implementation (PLDI)*, June 2003

[10] Beutel nodes

[11] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks, vol. 10, no. 4*, pp. 563-579, August 2005.

[12] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler and M. Srivastava, "SOS: A dynamic operating system for sensor networks," in *Proc. of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*, 2005.

[13] S. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong. "TinyDB: An Acqusitional Query Processing System for Sensor Networks," *ACM TODS*, 2005

[14] Y. Yao and J. Gehrke, "The Cougar Approach to In-Network Query Processing in Sensor Networks", *SIGMOD*, Vol. 31 , Issue 3 , pp. 8-19 2002

[15] Srisathapornphat, C. Jaikaeo, and C. C. Shen, "Sensor Information Networking Architecture", in *Proc International Workshops on Parallel Processing*, pp.23-30, 2000.

[16] Karl Aberer, M. Hauswirth, A. Salehi, "The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks," Technical report, Ecole Polytechnique Federale de Lausanne (EPFL) 2006

[17] E. Yoneki and J. Bacon, "Pronto: MobileGateway with Publish-Subscribe Paradigm over Wireless Network," *ACM/IFIP/USENIX International Middleware Conference,* June 2003.

[18] E. Souto. G. Guimaraes, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz and J. Kelner, "A messaging-oriented middleware for sensor networks," *Personal and Ubiquitous Computing*, Vol. 10 Issue 1, pp. 37-44, 2006

[19] S. Li, S. H. Son, and J. A. Stankovic, "Event Detection Services Using Data Service Middleware in Distributed Sensor Networks," In *IPSN* 2003, Palo Alto, USA, April 2003

[20] W. Heinzelman, A. Murphy, H. Carvalho and M. Perillo,"Middleware to Support Sensor Network Applications," *IEEE Network Magazine Special Issue.* Jan. 2004

[21] M. Kochhal, L. Schwiebert, S. Gupta and Changli Jiao, "QoS-Aware Core Migration for Efficient Multicast in Mobile Ad hoc Networks", *Wayne State University, WSU-CSC-NEWS/04-TR02*, July 2004.

[22] S. Reddy, T. Schmid, N. Yau, G. Chen, D. Estrin, M. Hansen, M. B. Srivastava, "ESP Framework: A Middleware Architecture For Heterogeneous Sensing Systems," UCLA, TR-UCLA-NESL-200612-06, December 2006.

[23] C.-L. Fok, G.-C. Roman, C. Lu. "Mobile Agent Middleware for Sensor Networks: An Application Case Study" *In Proc of the 4th International Conference on Information Processing in Sensor Networks (IPSN'05)*, pp. 382-387, April 25-27, 2005.

[24] P. Levis and D. Culler, "Mate: A tiny virtual machine for sensor networks", *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, 2002

[25] P. Hanáček, "Parallel Simulation Using the Linda Language", 5th Moravo-Silesian International Symposium on Modelling and Simulation of Systems, pp. 263--267, 1993.

[26] T. He, S. Krishnamurthy, L. Luo, T. Yan, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui and B. Krogh, "VigilNet: An Integrated Sensor Network System for Energy-Efficient Surveillance," *ACM Transactions on Sensor Networks*, 2006.

[27] M. Broxton, J. Lifton, J. Paradiso, "Localizing a sensor network via collaborative processing of global stimuli", in *Proc of the Second European Workshop on Wireless Sensor Networks,* pp 321-332, 2005.

[28] J. Horey, J.-C. Tournier, A. B. Maccabe "Kaizen: improving sensor network operating systems" , in *Proc of the 4th International Conference on Embedded Networked Sensor Systems*, pp. 413-414, 2006

[29] S. Dulman and P. Havinga, "Operating System Fundamentals for the EYES Distributed Sensor Network" *Progress 2002*, Utrecht, the Netherlands, October 2002

[30] B. Hurler, H. Hof, and M. Zitterbart, "A General Architecture for Wireless Sensor Networks: First Steps" in *Proc of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04) – Vol. 7*, pp. 442-444, 2004.

[31] X. Sun, "SCAN: a small-world structured p2p overlay for multi-dimensional queries", in *Proc. of the 16th International Conference on World Wide Web*, pp. 1191-1192, 2007

[32] Asad Awan, Ahmed Sameh, Ananth Grama, "The Omni Macroprogramming Environment for Sensor Networks", In *The International Conference on Computational Science,* Reading, UK, May 2006.

[33] H. Liu, T. Roeder, K. Walsh, R. Barr, E. G. Sirer, "Design and Implementation of a Single System Image Operating System for Ad-hoc Networks". In *Proc. of The International Conference on Mobile Systems, Applications, and Services (Mobisys)*, Seattle, Washington, June, 2005

[34] C. R. Baker, Y. Markovsky, J. Van Gruenen, A. Wolisz, J. Rabaey, and J. Wawrzynek, "ZUMA: A Platform for Smart-Home Environments", In *Proc of IET Intelligent Environments*, Athens, Greece, July 2006

[35] W.W. Chu. "Optimal File Allocation in a Multiple Computer System", *IEEE Transactions of Computers, 18(10)*, October 1969.

[36] Y. C. Chow, and W. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system" *IEEE Trans. Cornput*. C-28, 5, pp. 334-361, May 1979

[37] Z. Benenson and N. Gedicke and O. Raivio, "Realizing robust user authentication in sensor networks" In *Real-World Wireless Sensor Networks (REALWSN),* Stockholm, June 2005

[38] L. B. Oliveira, D. Aranha, E. Morais, F. Daguano, J. Lo'pez, and R. Dahab, "TinyTate: Identity-Based Encryption for Sensor Networks" in *Cryptology ePrint Archive*, Report 2007/020, 2007

[39] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and D. Tygar, "SPINS: Security Protocols for Sensor Networks", in *Wireless Networks Journal (WINE)*, pp. 521--534, September 2002

[40] Available for download: http://www.eclipse.org/

[41] Standard: JTC1/SC22/WG14 http://www.open-std.org/jtc1/sc22/wg14/

[42] S. B. Akers. "Binary Decision Diagrams," *IEEE Transactions on Computers,* C-27(6):509–516, June 1978

[43] N. B. Priyantha, A. Chakraborty, H. Balakrishnan, "The Cricket Location-Support system", in *Proc. 6th ACM MOBICOM*, Boston, MA, August 2000.

[44] K. Lorincz and M. Welsh, "MoteTrack: A Robust, Decentralized Approach to RF-Based Location Tracking," In *Proc. of the International Workshop on Location and Context-Awareness (LoCA 2005) at Pervasive 2005,* May 2005.

[45] Available for download: http://www.dcg.ethz.ch/~rschuler/

[46] R. Tewari and N. R. Adam, "Distributed File Allocation with Consistency Constraints", *International Conference on Distributed Computing Systems,* pp. 408-415, 1992.

[47] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review", in P*roc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995

[48] S. Albers, "Competitive online algorithms", BRICS Lecture Series LS-96-2, BRICS, Department of Computer Science, University of Aarhus, September 1996

[49] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, "GHT: A Geographic Hash Table for Data-Centric Storage in SensorNets", In *Proc. of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, Atlanta, Georgia, September 2002.

[50] F. Zhao and L. Guibas. Wireless Sensor Networks: An Information Processing Approach. Elsevier/Morgan-Kaufmann, 2004

[51] A. Amis, R. Prakash, T. Vuong, D. Huynh, "Max-Min D-Cluster Formation in Wireless Ad Hoc Networks", *Infocom*, p 32-41, 2000.

[52] B. Awerbuch, Y. Bartal, and A. Fiat. "Competitive Distributed File Allocation", In *Proc of the 25th Ann. AMC Symp. On Theory of Computing*, pp 164-173, 1993

[53] Y. Bartal, A. Fiat, Y. Rabani. "Competitive Algorithms for Distributed Data Management." *24th ACM STOC*, 1992

[54] A. Varga, "The OMNeT++ discrete event simulation system," in *European Simulation Multiconference*, 2001.

[55] W. Drytkiewicz, S. Sroka, V. Handziski, A. Köpke, H. Karl, "A Mobility Framework for OMNeT++", *3rd Intl OMNeT++ Workshop*, at Budapest University of Technology and Economics, Hungary, 2003

[56] R. T. B. Ma, V. Misra, and D. Rubenstein, "Modeling and Analysis of Generalized Slotted-Aloha MAC Protocols in Cooperative, Competitive and Adversarial Environments", in Proc. of the 26th IEEE International Conference on Distributed Computing Systems, pp. 62, 2006

[57] A. K. Parekh , R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case", *IEEE/ACM Transactions on Networking (TON)*, v.1 n.3, p.344-357, June 1993

[58] R. K. Arora and S. P. Rana, "On module assignment in two-processor distributed systems". *Information Processing Letters* 9, 3, pp. 113-117, 1979

[59] C. C. Price, Task allocation in distributed systems: A survey of practical strategies, in *Proc. of the ACM Conference*, pp. 176-181, 1982

[60] R. C. Prim, "Shortest connection networks and some generalizations", In *Bell System Technical Journal,* v 36, pp. 1389–1401, 1957

[61] Wu, M., Shu, W., and Gu, J. "Local Search for DAG Scheduling and Task Assignment." In *Proceedings of the international Conference on Parallel Processing*, Washington, DC, 1997.

[62] Richard E. Korf. "Depth-first iterative-deepening: An optimal admissible tree search." In *Artificial Intelligence*, v27(1) pp 97-109, 1985.

[63] I. Ahmad and M. K.Dhodhi, "Task assignment using a problem-space genetic algorithm." In *Concurrency: Pract. Exper.* Vol 7,5 pp 411-428, 1995.

[64] T. Smith, T. Sandholm, and R. G. Simmons, "Constructing and Clearing Combinatorial Auctions Using Preference Elicitation", in *Proc. Nat. Conf on Artificial Intelligence (AAAI) Workshop on Preferences in AI and CP*, 2002

[65] D. C. Parkes and L. H. Ungar. "Iterative combinatorial auctions: Theory and practice". In *Seventeenth National Conference on Artificial Intelligence*, pages 74-81, 2000.

[66] http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf

[67] http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf

[68] International Energy Agency, "The Power to Choose Demand Response in Liberalised Electricity Markets", *OECD Publishing*, Dec 2003.

[69] http://www.chipcon.com

[70] http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf

[71] http://www.ftdichip.com/

[72] http://www.sensirion.com

[73] Hopper, N. and C. Goldman, "Not All Large Customers Are Made Alike: Disaggregating Response to Default-Service Day-Ahead Market Pricing", In *Proc. of the 2006 ACEEE Summer Study on Energy Efficiency in Buildings. LBNL-59629*. August 2006

[74] K. Spees and L. Lave "Impacts of Responsive Load in PJM: Load Shifting and Real Time Pricing", *Carnegie Mellon, CEIC-07-02*, 2006

Appendix I

HVAC_CONTROL Application

HVAC_CONTROL.serv

```
ModuleName:HVAC_CONTROL
DataIn:10                       /* units are kb/s */
DataOut:2
Memory: 2                       /* units are kbytes */
Processing:
ResourceReq:          /* Can also be left blank */
FaultTolerance:detection       /* values are recoverable, detection, nothing */

Service:0
Name:TEMPERATURE
Scope:KITCHEN
QueryPeriod:5                 /* query once every 5 seconds */
Name:TEMPERATURE
Scope:LIVINGROOM
QueryPeriod:8

Service:1
Name:COMFORT
Scope:SP_SCOPE_ALL
QueryPeriod:10

Service:2
Name:DESIRED_TEMP
Scope:SP_SCOPE_ALL
QueryPeriod:10

Service:3
Name:PRICE
Scope:SP_SCOPE_ALL
```

```
QueryPeriod:10

Service:4                          /* this one will be queried by the app */
Name:HVAC
Scope:SP_SCOPE_ALL
QueryPeriod:0
```

HVAC_CONTROL.h

```
/*
 * Place the names of locations and services here
 * Note, the spelling & capitalization must be the same as that used in the
 * .serv file
 */
enum {

  KITCHEN = 1,
  LIVINGROOM = 2,
  DININGROOM = 3,
  BEDROOM1 = 4,
  BEDROOM2 = 5,
  BEDROOM3 = 6,
  MASTERBEDROOM = 7,
  BATHROOM = 8,

  TEMPERATURE = 9,
  PRICE = 13,
  HVAC = 14,
  COMFORT = 15,
  DESIRED_TEMP = 16,
  DISPLAY = 18,
  HVAC_CONTROL = 19,
  TEMP_MARGIN = 2,
  OFFSET_INCREMENT = 60,

/* eg persona
 * OWNER = 22,
 */

 /* some predefined values */
  TIMEOUT = 2,
  BRIDGE = 17,
  OFF = 2,
  ON = 1,
  SENSOR = 10,
  ACTUATOR = 20,
  CONTROL_TRIES = 3, //how many times you try to respond before timeout
  REPLY_TRIES = 3,
  ACTIVATE = 2,
  DEACTIVATE = 3,
  GUI = 20,

};

typedef struct serviceUsage {
   nx_uint8_t name;
   sp_scope_t scope;
```

```
   uint8_t invID;
   uint16_t ticksSinceRecv;
   uint8_t network;
   uint8_t type;
   uint8_t ack;
   uint8_t isThere;
   uint16_t timeout;
} serviceUsage;

int invokeService(uint8_t fn, char *args, uint16_t arg_len, uint8_t id);
void serviceResult(uint8_t fn, char *payload, uint16_t payload_len, uint8_t
id);
void terminate ();
void invoke();
int requestService(uint8_t servNum, uint8_t servName, uint8_t fn, char *args,
uint16_t argLen);
int serviceRespond(uint8_t fn, char *results, uint16_t resLen, uint8_t id);
```

HVAC_CONTROL.c

```
#include "table.h"

/*
 * Place all the variables to store results from services used
 * eg uint16_t sensorSample1
 * uint16_t sensorSample2
 */

uint8_t  m_active;
uint16_t m_price;
uint16_t m_desired_comfort;
uint16_t m_desired_temp;
uint16_t m_temp;
uint16_t m_switch_state;

/*
 * during execution these will be filled in and updated with the name & scope
of the
 * equivalent service that exist in the network - see bottom of file for
 * a definition of the struct
 */
serviceUsage services[NUM_SERVICES_USED];

/*
 * fill in, it will be called on initialization
 */
void invoke()
{
       //defaults
       m_active = 0;
       m_price = 10;
       m_desired_comfort = 2;
       m_desired_temp = 70;
       m_temp = 75;
       m_switch_state = OFF;
}
```

```
/*
 * fill this in - it will be called every second
 * Use it to process results, query more services, actuate etc.
 */
void executeControl()
{
    uint16_t temp_setpoint;
    uint8_t servName, fn, servNum;
    uint16_t argLen;
    char *args[10];
    argLen = 0;

    if (!m_active) {
       return;
    }

    temp_setpoint = ((m_desired_temp * 100) + ((5-m_desired_comfort) * m_price
* OFFSET_INCREMENT)) / 100;
    servName = services[4].name;
    fn = services[4].name;

    if ((m_temp >= (temp_setpoint + TEMP_MARGIN))){
       args[0] = ON;
    } else if ((m_temp < (temp_setpoint - TEMP_MARGIN)) ){
       args[0] = OFF;
    }

    argLen = 1;
    servNum = 4;
    requestService(servNum, servName, fn, args, argLen);
}

/*
 * Fill this in if you want other components to use this service
 * it will be called on the service in response to a requestService call
 * and it should call serviceRespond to pass the result back to the callee
 * @param fn function that will be called
 * @param args - arguments to the function
 * @param arg_len length of the arguments (not null terminated)
 * @param id -identifies the request, should be passed back to serviceRespond
 *
 *@return int return 0 if the service successfully invoked, 1 otherwise
 */
int invokeService(uint8_t fn, char *args, uint16_t arg_len, uint8_t id)
{
    if (fn == ACTIVATE) {
       m_active = 1;
    } else if (fn == DEACTIVATE) {
       m_active = 0;
    }
    return 0;
}

/*
 * Fill in to process the result of a service query/invoke service
 * @param fn - function name that was called
 * @param payload - results
 * @param payload_len - length of results (they are not null terminated)
 * @param id - request id that was returned in requestService
```

```
 */
void serviceResult(uint8_t fn, char *payload, uint16_t payload_len, uint8_t id)
{
    /* since we specify query periods, the crs will automatically query these
for us */
    if (id == services[0].invID) {
       m_temp = payload[0];
    } else if (id == services[1].invID) {
       m_desired_comfort = payload[0];
    } else if (id == services[2].invID) {
       m_desired_temp = payload[0];
    } else if (id == services[3].invID) {
       m_price = payload[0];
    }
}

/* called when the mapper is wrapping up the service
 * clean up any last minute state
 */
void terminate ()
{
    return;
}

/*------------------STUBS-----------------------*
 * Leave at the bottom of the file - will be filled in by SNSP */

/*
 * This is a stub will send a service  query
 * @param servNum is the equivalence class specified in the .serv file
 * @param servName is the class that was found in the network
 *        (that is equivalent from  - serviceUseage
services[NUM_SERVICES_USED];)
 * @param scope is the scope the service is in
 * @param args is a void pointer to arguments to the service
 * @param argLen is the length of args (not null terminated)
 *
 * @return int returns the id that will be used in serviceResult to pass result
back
 */
int requestService(uint8_t servNum, uint8_t servName, uint8_t fn, char *args,
uint16_t argLen)
{
}

/*
 * This is a stub
 * @param results - pointer to results
 * @param
 *
 * @return 0 if successful 1 if not
 */
int serviceRespond(uint8_t fn, char *results, uint16_t resLen, uint8_t id)
{
}
```

Resulting TinyOS Code

Module

```
generic module HVAC_CONTROLM(uint8_t m_serviceName,  uint16_t
NUM_SERVICES_USED, uint16_t NUM_SERVICES_HANDLED,uint16_t NUM_SERVICES_EQUIV) {
       provides {
              interface App;
              interface StdControl;
       } uses {

              interface SPInvocationAccess;
              interface SNSP;
              interface Leds;
              interface Timer;
       }
}

implementation {

uint16_t control_tries;
//specific variables
int16_t timerval = 500;



sp_scope_t m_scope;   //stores the location of this node
uint8_t m_network;    //stores network type of this node ie TELOS/MICA etc
uint8_t m_funName;
uint8_t m_scopeInvID;
uint8_t m_servInfoInvID;
uint8_t m_timeSinceUpdate;
uint8_t m_activated;
uint8_t m_activatedScope;

bool m_replyLock;
bool m_sendLock;

sp_container_handle_t m_register;
sp_container_handle_t m_handle;

//service call variables
nx_uint8_t m_name;
sp_scope_t m_destScope;
uint8_t m_arg1;
uint8_t m_arg2;
uint8_t *m_arg3;
uint16_t m_argLen;
nx_uint8_t m_resName;
nx_uint8_t m_resDestScope;
nx_uint16_t m_res1;

serviceUsage equivServices[NUM_SERVICES_USED][NUM_SERVICES_EQUIV];
pendingQuery servicePendingQueries[NUM_SERVICES_HANDLED];
```

```
void task queryScope();
void task registerService();
void task sendServiceInfo();
void task queryCRS();
void task executeControl();
void task sendQueries();
uint8_t serviceCall();

int invokeService(uint8_t fn, char *args, uint16_t arg_len, uint8_t id);
void serviceResult(uint8_t fn, char *payload, uint16_t payload_len, uint8_t
id);
void terminate ();
void invoke();
int requestService(uint8_t servNum, uint8_t servName, uint8_t fn, char *args,
uint16_t argLen);
int serviceRespond(uint8_t fn, char *results, uint16_t resLen, uint8_t id);

int8_t  m_active;
uint16_t m_price;
uint16_t m_desired_comfort;
uint16_t m_desired_temp;
uint16_t m_temp;
uint16_t m_switch_state;

serviceUsage services[NUM_SERVICES_USED];


void invoke()
{
       //defaults
       m_active = 0;
       m_price = 10;
       m_desired_comfort = 2;
       m_desired_temp = 70;
       m_temp = 75;
       m_switch_state = OFF;
}

void task executeControl()
{
    uint16_t temp_setpoint;
    uint8_t servName, fn, servNum;
    uint16_t argLen;
    char *args[10];
    argLen = 0;

    if (!m_active) {
       return;
    }

    temp_setpoint = ((m_desired_temp * 100) + ((5-m_desired_comfort) * m_price
* OFFSET_INCREMENT)) / 100;
    servName = services[4].name;
    fn = services[4].name;

    if ((m_temp >= (temp_setpoint + TEMP_MARGIN))){
       args[0] = ON;
    } else if ((m_temp < (temp_setpoint - TEMP_MARGIN)) ){
       args[0] = OFF;
```

```
    }

    argLen = 1;
    servNum = 4;
    requestService(servNum, servName, fn, args, argLen);
}

int invokeService(uint8_t fn, char *args, uint16_t arg_len, uint8_t id)
{
    if (fn == ACTIVATE) {
       m_active = 1;
    } else if (fn == DEACTIVATE) {
       m_active = 0;
    }
    return 0;
}

void serviceResult(uint8_t fn, char *payload, uint16_t payload_len, uint8_t id)
{
    if (id == services[0].invID) {
       m_temp = payload[0];
    } else if (id == services[1].invID) {
       m_desired_comfort = payload[0];
    } else if (id == services[2].invID) {
       m_desired_temp = payload[0];
    } else if (id == services[3].invID) {
       m_price = payload[0];
    }
}

void terminate ()
{
    return;
}




 /* ----------------------------Init & Locks---------------------------*/

 command result_t StdControl.init()
 {
       uint16_t i = 0;
       uint16_t j = 0;
       m_scope = SP_EMPTY;
       m_sendLock = 0;
       m_replyLock = 0;
       m_activated = 0;
       m_activatedScope = 0;

       for(i=0;i < NUM_SERVICES_USED; i++){
              services[i].isThere = 0;
              services[i].name = 0;
              for(j=0;j < NUM_SERVICES_EQUIV; j++){
                     equivServices[i][j].isThere = 0;
                     equivServices[i][j].name = 0;
              }
       }
```

```
       for(i=0;i < NUM_SERVICES_HANDLED; i++){
              servicePendingQueries[i].name = 0;
       }

      equivServices[0][0].name=TEMPERATURE;
      equivServices[0][0].scope=KITCHEN;
      equivServices[0][0].timeout=5;
      equivServices[0][1].name=TEMPERATURE;
      equivServices[0][1].scope=LIVINGROOM;
      equivServices[0][1].timeout=8;
      equivServices[1][0].name=COMFORT;
      equivServices[1][0].scope=SP_SCOPE_ALL;
      equivServices[1][0].timeout=10;
      equivServices[2][0].name=DESIRED_TEMP;
      equivServices[2][0].scope=SP_SCOPE_ALL;
      equivServices[2][0].timeout=10;
      equivServices[3][0].name=PRICE;
      equivServices[3][0].scope=SP_SCOPE_ALL;
      equivServices[3][0].timeout=10;
      equivServices[4][0].name=HVAC;
      equivServices[4][0].scope=SP_SCOPE_ALL;
      equivServices[4][0].timeout=0;

       m_activated = 0;
       m_timeSinceUpdate = TIMEOUT;
        return SUCCESS;
 }

 command result_t StdControl.start()
 {
     call Timer.start(TIMER_REPEAT, timerval);
     return SUCCESS;
 }

command result_t StdControl.stop()
 {
     call Timer.stop();
     return SUCCESS;
 }

 bool replyLock()

  {
    if (m_replyLock)
      return FALSE;
    else {
      m_replyLock = TRUE;
      return TRUE;
    }
  }

  bool releaseReplyLock()
  {
    bool oldLock = m_replyLock;
    m_replyLock = FALSE;
    return oldLock;
  }

  bool sendLock()
```

```
  {
    if (m_sendLock)
      return FALSE;
    else {
      m_sendLock = TRUE;
      return TRUE;
    }
  }

  bool releaseSendLock()
  {
    bool oldLock = m_sendLock;
    m_sendLock = FALSE;
    return oldLock;
  }

 /*---------------------Register, Query & serv Info update ----------------*/
int requestService(uint8_t servNum, uint8_t servName, uint8_t funName, char
*args, uint16_t argLen)
 {
    uint8_t invID;
    if(sendLock()){
        m_funName = funName;
       if(services[servName].network != m_network){
           m_name = BRIDGE;
           m_destScope = SP_SCOPE_ALL;
           m_arg1 = services[servNum].name;
           m_arg2 = services[servNum].scope;
           m_arg3 = args;

       } else {
           m_name = services[servNum].name;
           m_destScope = services[servNum].scope;
           m_arg3 = args;
           m_arg1 = m_arg2 = 0;
       }

       m_argLen = argLen;
       invID = serviceCall();
       /* look up which service this corresponds to and replace invID */
       services[servNum].invID = invID;
       return invID;
    } else {
       return 0;
    }

 }


uint8_t serviceCall(){
      sp_container_handle_t handle;
      uint8_t invID;
      call SPInvocationAccess.newContainer(&handle, MSG_TYPE_INVOCATION,
                                           m_name, m_serviceName, m_scope);
      call SPInvocationAccess.setDestScope(handle, m_destScope);
      call SPInvocationAccess.setPersonaScope(handle, m_activatedScope);
      call SPInvocationAccess.setPersona(handle, m_activated);
      call SPInvocationAccess.newFunction(&m_funName, handle);
```

```
      if(m_arg1 != 0){
            call SPInvocationAccess.newArgument(&m_name, SP_UINT8, 1,
                              (uint8_t*)&m_arg1, handle);
      }
      if(m_arg2 != 0){
            call SPInvocationAccess.newArgument(&m_name, SP_UINT8, 1,
                              (uint8_t*)&m_arg2, handle);
      }
      if(m_arg3 != 0){
            call SPInvocationAccess.newArgument(&m_name, SP_CHAR_ARRAY,
m_argLen,
                                          m_arg3, handle);
       }

      call SPInvocationAccess.getInvocationID(handle, &invID);
      call SNSP.issueRequest(handle);
      releaseSendLock();
      return invID;
 }

void task queryScope()
 {
      nx_uint8_t name;
      sp_scope_t temp;
      temp = SP_SCOPE_ALL;
      call SPInvocationAccess.newContainer(&m_register, MSG_TYPE_REGISTRATION,
CRS, m_serviceName, m_scope);
      call SPInvocationAccess.setDestScope(m_register, temp);
      name = SP_SCOPE_ALL;
      call SPInvocationAccess.newFunction(&name, m_register);
      call SPInvocationAccess.getInvocationID(m_register, &m_scopeInvID);
      call SNSP.issueRequest(m_register);
  }

void task registerService()
  {
      sp_scope_t temp;
      temp = SP_SCOPE_ALL;
      //register (register handle does not get destroyed)
      call SPInvocationAccess.newContainer(&m_register, MSG_TYPE_REGISTRATION,
CRS, m_serviceName, m_scope);
      call SPInvocationAccess.setDestScope(m_register, temp);
      call SNSP.spRegister(m_register);

  }


  void task sendServiceInfo(){
      nx_uint8_t name;

      uint16_t I;
      sp_container_handle_t handle;
      nx_uint8_t bridge;
      sp_scope_t temp;
      bridge = BRIDGE;
      temp = SP_SCOPE_ALL;
      i = 0;
      call SPInvocationAccess.newContainer(&handle, MSG_TYPE_INFO,
                                          GUI, m_serviceName, m_scope);
```

```
      call SPInvocationAccess.setDestScope(handle, temp);
      name = SP_SCOPE_ALL;
      call SPInvocationAccess.newServiceInfo(&name,&m_activated, handle);
      call SPInvocationAccess.newServiceInfo(&i,&i, handle);

      if(m_activated){
      for(i=0;i < NUM_SERVICES_USED; i++){
         if(services[i].network == m_network){
             call SPInvocationAccess.newServiceInfo(&(services[i].scope),
                                &(services[i].name),handle);
         }
      }

      call SPInvocationAccess.newServiceInfo(&temp, &bridge, handle);
      for(i=0;i < NUM_SERVICES_USED; i++){
         if(services[i].network != m_network && services[i].network != 0){
             call SPInvocationAccess.newServiceInfo(&(services[i].scope),
                                &(services[i].name), handle);
         }
      }
      }
      call SNSP.issueRequest(handle);
  }


 /*-------------------------- Timer Actions ------------------------*/
event result_t Timer.fired(){
      //initialize
      if(m_scope == SP_EMPTY){
         post queryScope();
      } else {

          //periodically keep CRS registration & service info alive
          if(m_timeSinceUpdate >= TIMEOUT) m_timeSinceUpdate = 0;
          if(m_timeSinceUpdate == 0) post registerService();
          if(m_timeSinceUpdate == 1) post queryCRS();
          if(m_timeSinceUpdate == 2) post sendServiceInfo();
          m_timeSinceUpdate++;
          if(m_activated){
              if(m_timeSinceUpdate > 2 && m_timeSinceUpdate & 1) {
              post sendQueries();
              post executeControl();
           }
          }
      }
      return SUCCESS;
 }

void task queryCRS(){
      sp_scope_t temp;
      sp_container_handle_t handle;



      nx_uint8_t name;
      temp = SP_SCOPE_ALL;
      call SPInvocationAccess.newContainer(&handle, MSG_TYPE_INVOCATION, CRS,
m_serviceName, m_scope);
      call SPInvocationAccess.setDestScope(handle, temp);
```

```
      name = ALL;
      call SPInvocationAccess.newFunction(&name, handle);
      call SPInvocationAccess.getInvocationID(handle, &m_servInfoInvID);
      call SNSP.issueRequest(handle);
      return;
  }

  void task sendQueries(){
       uint16_t i = 0;
       for(;i< NUM_SERVICES_USED; i++) {
             if(services[i].type == SENSOR
                && (services[i].timeout != 0 && services[i].ticksSinceRecv >
services[i].timeout)
                && services[i].network != 0){ //we have heard from them

                //assume that bridge exists
                if(sendLock()){
                    if(services[i].network != m_network){
                        m_name = BRIDGE;
                        m_destScope = SP_SCOPE_ALL;
                        m_arg1 = services[i].name;
                        m_arg2 = services[i].scope;
                        m_arg3 = 0;

                    } else {
                        m_name = services[i].name;
                        m_destScope = services[i].scope;
                        m_arg1 = m_arg2;
                        m_arg3 = 0;
                    }
                    serviceCall();
                }
             }
             if(services[i].network != 0) {
                 services[i].ticksSinceRecv++;
             }
       }
  }

/*--------------------- (SENSE/ACTUATE) and result ------------------------*/
command sp_result_t App.invocateRequest(uint16_t requestID,
sp_container_handle_t handle){

       nx_uint8_t name;
       nx_uint8_t type;
       nx_uint16_t len;
       uint8_t invID;
       sp_scope_t scope;
       nx_uint8_t funName;
       nx_uint8_t arg[128];
       uint16_t i = 0;
       if(call SPInvocationAccess.getInvocationID(handle, &invID) != SP_SUCCESS
||
          call SPInvocationAccess.getOriginatingService(handle, &name) !=
SP_SUCCESS ||
          call SPInvocationAccess.getScope(handle, &scope) != SP_SUCCESS ||

          call SPInvocationAccess.getMsgType(handle, &type) != SP_SUCCESS){
```

```
          return SP_FAIL;

       }

       for(i = 0; i < NUM_SERVICES_USED; i++){
          //found an empty slot
          if(servicePendingQueries[i].name == 0){
             break;
          }
       }

       if(i == NUM_SERVICES_HANDLED) return SP_FAIL;
       if(funName == ACTIVATE){
                 m_activated = name;
           m_activatedScope = scope;
           invoke();
       }

       if(funName == DEACTIVATE){
           terminate();
       }

       servicePendingQueries[i].invID = invID;
       servicePendingQueries[i].ticksSinceReply = 2; //will reply next
       servicePendingQueries[i].name = name;
       servicePendingQueries[i].scope = scope;
       servicePendingQueries[i].funName = funName;
       servicePendingQueries[i].replyFrequency = 2;

        if (call SPInvocationAccess.getArgument(&name, handle, 0,
                                  &type, arg, &len)!= SP_SUCCESS){
          servicePendingQueries[i].name = 0;
          return SP_FAIL;
       }

       call SPInvocationAccess.freeContainer(handle);
       return invokeService(funName, (char *)arg, len, invID);
   }

int serviceRespond(uint8_t fn, char *results, uint16_t resLen, uint8_t id)
{
    uint16_t i;
    sp_container_handle_t handle = 0;
    if (!replyLock()) {
         return 1;
    }

    for(i = 0;i < NUM_SERVICES_HANDLED; i++){
        if(servicePendingQueries[i].invID == id &&
servicePendingQueries[i].name != 0){


           if(call SPInvocationAccess.newContainer(&handle, MSG_TYPE_REPLY |
MSG_TYPE_INVOCATION,
                  servicePendingQueries[i].name, m_serviceName, m_scope) !=
SP_SUCCESS ||
              call SPInvocationAccess.setDestScope(handle,
servicePendingQueries[i].scope)
              != SP_SUCCESS ||
```

```
               call SPInvocationAccess.setInvocationID(handle,
servicePendingQueries[i].invID)
               != SP_SUCCESS ||
               call
SPInvocationAccess.newFunction(&(servicePendingQueries[i].funName), handle)
               != SP_SUCCESS ||
              call
SPInvocationAccess.newResult(&(servicePendingQueries[i].funName),
SP_CHAR_ARRAY,
                     resLen, results, handle) != SP_SUCCESS || signal
App.requestResponse(0, handle) != SP_SUCCESS){
                     if(handle != 0) call
SPInvocationAccess.freeContainer(handle);
                     return 1;
             }
             //reply just once in the control function
             releaseReplyLock();
            servicePendingQueries[i].name = 0;
            return 0;
       }
    }
}

event sp_result_t SNSP.requestResult(uint16_t requestID, sp_container_handle_t
handle)

  {
       uint8_t invID;
       nx_uint8_t name;
       uint8_t val;
       uint8_t longVal[128];
       uint8_t type;
       uint8_t len;
       uint16_t i = 0;
       uint16_t num = 0;
       uint8_t numByte = 0;
       uint16_t k, j, tempj, tempk;
       tempj = tempk = 0;
       call SPInvocationAccess.getInvocationID(handle, &invID);
       if(call SPInvocationAccess.getOriginatingService(handle, &name) !=
SP_SUCCESS)
                                                 return SP_FAIL;
       if(invID == m_scopeInvID && name == CRS){
                name = SP_SCOPE;
                if(call SPInvocationAccess.getResult(&name, handle, 0, &type,
                              (uint8_t*)&val, &len) != SP_SUCCESS)
           {
                 dbg(DBG_TEMP, "PROB in getting query result\n");
           }

           m_scope = val;
           name = NETWORK;
          if(call SPInvocationAccess.getResult(&name, handle, 0, &type,
                              (uint8_t*)&val, &len) != SP_SUCCESS)
           {
                dbg(DBG_TEMP, "Prob in getting query results \n");

           }
```

```
            m_network = val;

                 m_scopeInvID = 0xff;
               }   else if(name == CRS && invID == m_servInfoInvID){
                   name = ALL;
                   if(call SPInvocationAccess.getNumberResults(&name, &numByte,
handle)

                      != SP_SUCCESS){
                      dbg(DBG_TEMP, "PROB in getting query result\n");
                   }
                   num = numByte & 0x00ff;

                   for(i = 0; i< NUM_SERVICES_USED; i++){
                       services[i].network = 0;
                   }
                   services[4].network = 1;
                   i = 0;
                   while(i < num){
                    if(call SPInvocationAccess.getResult(&name, handle, i, &type,
                                  &val, &len) != SP_SUCCESS)
                   {
                    goto end;
                   }
                   for(k=0;k < NUM_SERVICES_USED; k++){
                    for(j=0;j < NUM_SERVICES_EQUIV; j++){
                       if (equivServices[k][j].name == val) {
                          tempk = k;
                          tempj = j;
                          break;

                       }
                    }
                  }

                   if(call SPInvocationAccess.getResult(&name, handle, i+1, &type,
                                  &val, &len) != SP_SUCCESS)
                   {
                    goto end;
                   }

                   equivServices[tempk][tempj].network = val;
                   equivServices[tempk][tempj].isThere = 1;

                   /* copy the first choice over */
                   for(j=0;j < NUM_SERVICES_EQUIV; j++){

                     if (equivServices[tempk][j].isThere) {
                       services[tempk].name = equivServices[tempk][j].name;
                           services[tempk].scope = equivServices[tempk][j].scope;
                           services[tempk].invID = equivServices[tempk][j].invID;
                           services[tempk].ticksSinceRecv =
equivServices[tempk][j].ticksSinceRecv;
                           services[tempk].network =
equivServices[tempk][j].network;
                           services[tempk].type = equivServices[tempk][j].type;
                           services[tempk].ack = equivServices[tempk][j].ack;
                           services[tempk].isThere =
equivServices[tempk][j].isThere;
```

```
                        services[tempk].timeout =
equivServices[tempk][j].timeout;
                    break;
                  }
              }
            }
            i +=2;
       } else {

               if(name == BRIDGE){
                    call SPInvocationAccess.getFunctionName(&name, 0, handle);
                }
                if(call SPInvocationAccess.getResult(&name, handle, 0, &type,
                         longVal, &len) != SP_SUCCESS)
                {
                  goto end;
                }

               serviceResult(name, longVal, len, invID);
       }
end:
       call SPInvocationAccess.freeContainer(handle);
       return SP_SUCCESS;
  }

  command sp_result_t App.status(){ return SP_SUCCESS;}

  command sp_result_t App.registrationInfo(sp_container_handle_t handle){
        return SP_SUCCESS;
  }

}
```

Configuration

```
includes SNSP;
includes App;

configuration HVAC_CONTROL {
}

implementation {
components Main, LedsC, TimerC, new SNSPC(SP_SCOPE_ALL, TELOS) as mySNSP,  new
HVAC_CONTROLM(HVAC_CONTROL, 5, 4, 2), SPInvocationAccessM, RandomLFSR, ADCC;

 Main.StdControl -> SPInvocationAccessM;
 Main.StdControl -> mySNSP;
 Main.StdControl -> TimerC;
 Main.StdControl -> HVAC_CONTROLM;

 mySNSP.App[0]   -> HVAC_CONTROLM;

 HVAC_CONTROLM.SNSP                -> mySNSP.SNSP[0];
```

```
 HVAC_CONTROLM.Timer                -> TimerC.Timer[unique("Timer")];
 HVAC_CONTROLM.SPInvocationAccess   -> SPInvocationAccessM.SPInvocationAccess;

HVAC_CONTROLM.Leds                 -> LedsC;

SPInvocationAccessM.Random               -> RandomLFSR;
}
```

Appendix II

Eclipse Plugin

SNSP Eclipse plugin allows one to create a new project of type SNSP. The new project must have a name, which will be the application or service name. When a new SNSP project is created, four files are attached to the project. The first file is a README that explains how to write an SNSP application. The second file is a .c file containing stubs that must be completed. The third file is a .h file that contains a mapping of semantic names to numbers (names for locations and services). The fourth file is the .serv file. These files are explained in detail in Section 4.3.1 below and the .serv file is explained in section 4.2 above. The application writer can edit these files and fill in the functionality of the application. Once they are done, there are two compile options, one compiles their regular c code to check for syntax errors. The other compiles the c code they have written into TinyOS code and places it in a tinyos/ sub-directory. This code can then be further cross-compiled for a mote platform and loaded onto a mote using the TinyOS Eclipse plugin.

SNSP eclipse module comes in a tar file format. The tar file is downloaded with an unpack script. The user must run the unpack script at the command line. The unpack script takes one argument, which is the name of the service that the user wishes to create. The unpack script creates another tar file with the service name in the same directory. This file must be imported into the eclipse workspace.

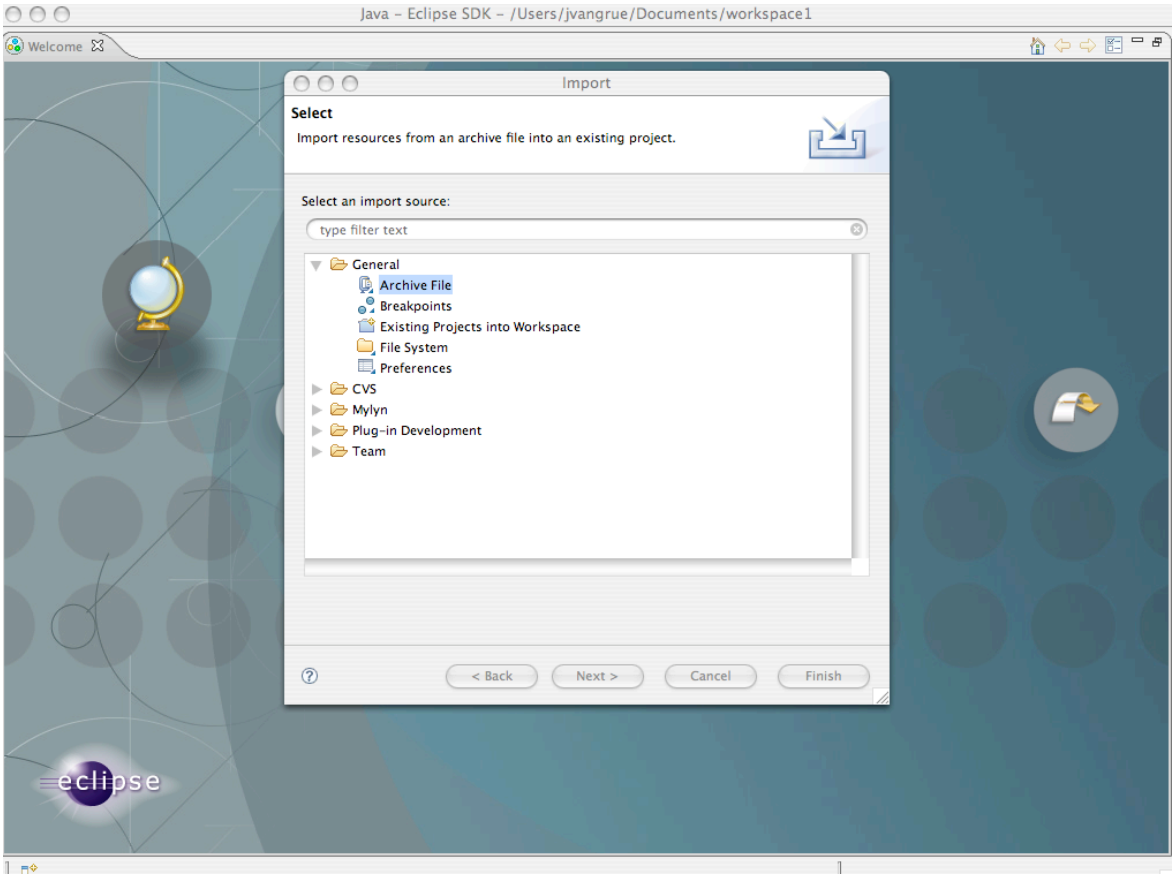

Figure 27: Importing an SNSP project into Eclipse.

Figure 27 demonstrates what eclipse looks like when it first opens, and when the file->new item has been selected from the menu. The user must then select to import "Existing Projects into Workspace" and a file browser will appear from

which they can choose the tar file created by the unpack script. Figure 28 shows how to switch between compilation targets. In order to switch, the user must right-click on the project in the right task bar, and the window in Figure 28 will appear. There is then two choices: compile, or the given project name. Compile will compile the c code in the .c and .h files. The project name option will build the TinyOS code.

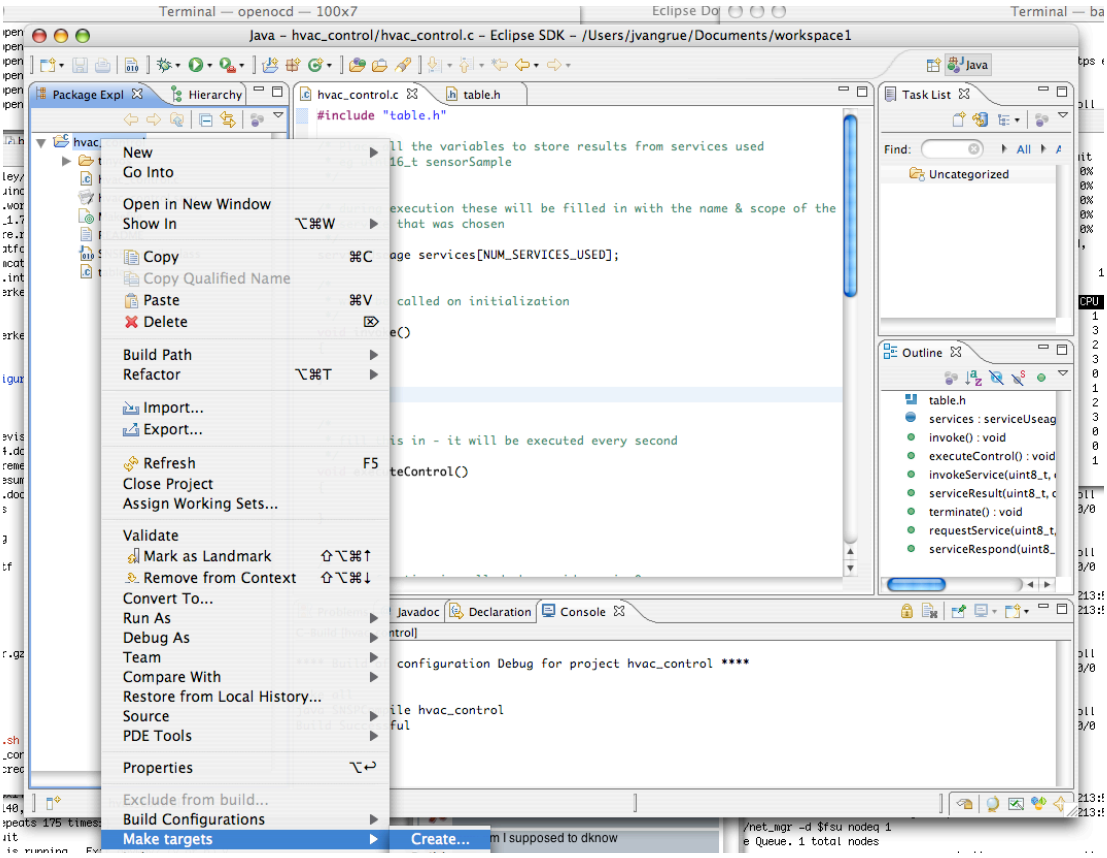

Figure 28: Choosing a target to compile.

Figure 29 shows the result of a C compilation that had an error. The bottom window has details about what the error was. The eclipse editor can also be seen

in Figure 29 as well as the project browsing toolbar on the right that lists all open files within the project.

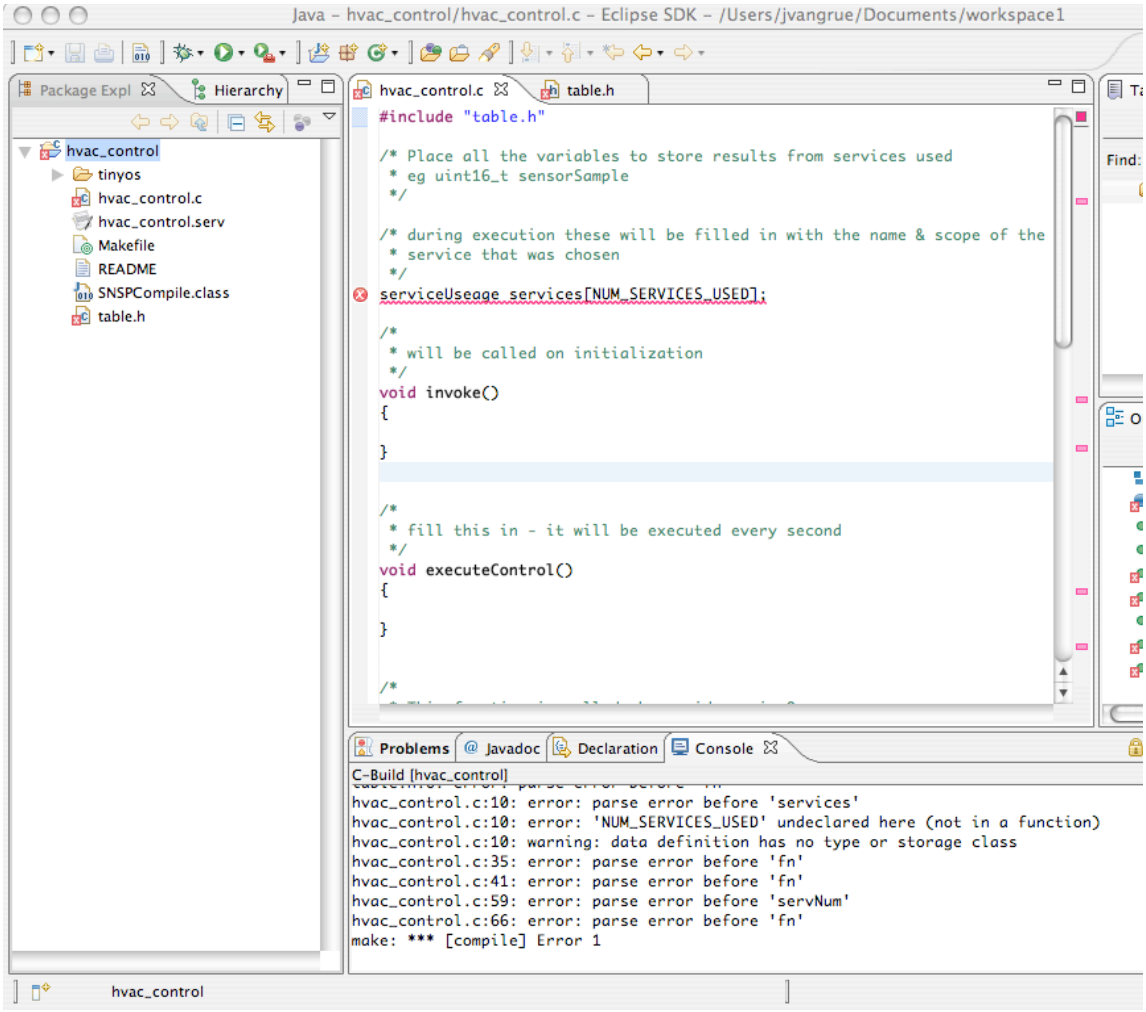

Figure 29: An open SNSP project, with an open file and a C compilation error.

Figure 30 shows the result of compiling the project into TinyOS code with eclipse. In the right-hand toolbar the TinyOS files are visible in the tinyOS folder. The bottom toolbar also displays a message of either success or an error message if the build failed.

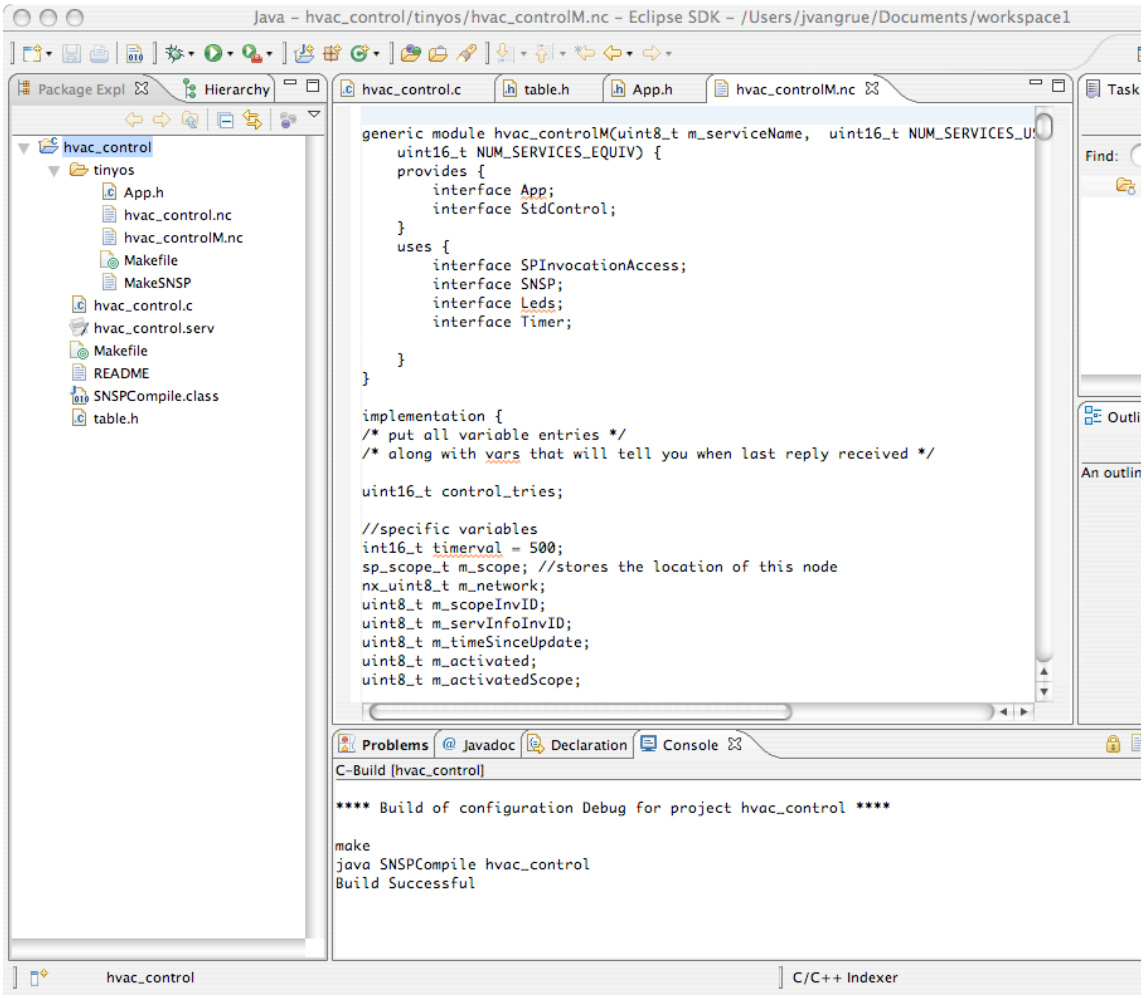

Figure 30: Compiling SNSP code with Eclipse.