UNIVERSITY OF CALIFORNIA,
IRVINE

Supporting Fault Tolerance in the Internet of Things

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical Engineering and Computer Science

by

Sen Zhou

Dissertation Committee:
Professor Kwei-Jay Lin, Chair
Professor Pai Chou
Professor Mohammad A. Al Faruque

2015

DEDICATION

To my parents, Lihui Lian and Mulin Zhou

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF TABLES

ACKNOWLEDGMENTS

I first would like to thank my advisor Prof. Kwei-Jay Lin for his instructions. He inspired me, helped me and supported me all the way through my PhD study.

Thanks to EECS department of University of California, Irvine for providing fellowship support and TA opportunities for my study; and to Intel-NTU Connected Context Computing Center at National Taiwan University for providing me with a chance to work on the Wukong project.

Many thanks to my committee members Prof. Pai Chou and Prof. Mohammad Al Faruque from University of California, Irvine; and to Prof. Jane Y.J. Hsu, Prof. Chi-Sheng Shih and Dr. Yu-Chung Wang from National Taiwan University and the Intel-NTU center for their valuable inputs to this research.

I would also like to thank my colleagues, Mark Panahi, Weiran Nie, Jing Zhang, Yi-Chin Chang, Zhenqiu Huang, Jenny Kim, Mira Kim, Vicki Hsiao, Congmiao Li and Ching-Chi Chuang for their help during my study.

CURRICULUM VITAE

Sen Zhou

EDUCATION

Doctor of Philosophy in Electrical Engineering and Computer Science **2015**
University of California *Irvine, California*

Master of Science in Electrical Engineering and Computer Science **2011**
University of California *Irvine, California*

Bachelor of Engineering in Computer Science and Technology **2009**
Tsinghua University *Beijing, China*

TEACHING EXPERIENCE

Teaching Assistant **2010–2014**
University of California, Irvine *Irvine, California*

PUBLICATIONS

1. Weiran Nie, Sen Zhou, Kwei-Jay Lin and Soo Dong Kim, **An On-Line Capacity-based Admission Control for Real-Time Service Processes**, IEEE Transactions on Computers, 63(9): 2134-2145, 2014

2. Sen Zhou and Kwei-Jay Lin, **Real-time service process admission control with schedule reorganization**, Journal of Service Oriented Computing and Applications, 7(1): 3-14, 2013.

3. Sen Zhou, Kwei-Jay Lin, Jun Na, Ching-Chi Chuang and Chi-Sheng Shih, **Supporting Service Adaptation in Fault Tolerant Internet of Things**, IEEE 8th Conference on Service-Oriented Computing and Applications (SOCA'15), 2015

4. Sen Zhou, Kwei-Jay Lin and Chi-Sheng Shih, **Device Clustering for Fault Monitoring in Internet of Things Systems**, IEEE World Forum on Internet of Things (WF IOT'15), 2015

5. Jun Na, Kwei-Jay Lin, Zhengqiu Huang and Sen Zou, **An Evolutionary Game Approach on IoT Service Selection for Balancing Device Energy Consumption**, IEEE 12th International Conference on e-Business Engineering (ICEBE'15),

2015

6. Zhenqiu Huang, Kwei-Jay Lin, Congmiao Li and Sen Zhou, **Communication Energy Aware Sensor Selection in IoT Systems**, IEEE International Conference on in Internet of Things(iThings'14), 2014

7. Weiran Nie, Sen Zhou and Kwei-Jay Lin, **Real-time Service Process Scheduling with Intermediate Deadline Overrun Management**, IEEE 5th Conference on Service-Oriented Computing and Applications (SOCA'12), 2012

8. Sen Zhou and Kwei-Jay Lin, **A Flexible Service Reservation Scheme for Real-Time SOA**, 2011 IEEE 8th International Conference on e-Business Engineering (ICEBE'11), 2011

ABSTRACT OF THE DISSERTATION

Supporting Fault Tolerance in the Internet of Things

By

Sen Zhou

DOCTOR OF PHILOSOPHY in Electrical Engineering and Computer Science

University of California, Irvine, 2015

Professor Kwei-Jay Lin, Chair

This thesis addresses the issue of fault tolerance in the Internet of Things(IoT). The goal of fault tolerance in IoT is to better adapt to changing environments and build up trustworthy redundancy. However, in real IoT deployment scenarios like smart homes or offices, heterogeneity and constant evolution of IoT systems pose a big challenge to building up redundancies and adapting to changing environment. Firstly, heterogeneous devices are deployed in the environment with limited duplications. This brings challenges to find redundant devices in the first place. Secondly, with a changing environment, there comes the need for devices, though deployed with different purposes and capabilities, to collaborate with one another and to be sharable among different applications with QoS requirements. This brings challenges to management of IoT applications and IoT devices. Thirdly, in order to achieve failure-resilience on heterogeneous devices, an evolving yet lightweight dynamic binding mechanism should be designed. This is the basis for supporting both previous points.

In this dissertation, we propose to address this above issues from a service-oriented point of view. Service-Oriented Architecture(SOA) provides IoT with a abstraction of integratable and manageable services. We have designed an IoT middleware to facilitate the cooperation of different devices to achieve this cross-modality fault tol-

erance. When a fault happens to a device, the middleware can reconfigure the system by using devices of other modalities to cover the fault. The three above problems are addressed in three different stages of service management: service discovery, service mapping, service execution.

For service discovery, this thesis presents a sensing device adaptation scheme for composing more available services. In IoT, sensors of different modalities may be used to enhance the system fault tolerance. We propose the concept of virtual services which use data from other sensor devices to replace an actual service on some faulty device. We do regression analysis to identify and generate virtual services using available sensors. Depending on the sensor correlation types, we can use with recursive least squares (RLS) or multivariate adaptive regression splines (MARS) for virtual service generation. These virtual services provide more choices of backup services without deployment of duplicate backup sensors.

For service mapping, we separate it into two steps: phase 1 pre-runtime mapping for functionality of the application and phase 2 run-time mapping for fault-tolerance. For pre-runtime mapping, we model it into a quadratic integer programming problem. Location policies are used to specify user preference during this mapping, and to limit the size of the QLP problem. For phase 2 mapping, with abundant provision of virtual services, we model it into a multiobjective optimization problem and use a multiobjective genetic algorithm, NSGA-ii, to solve it. With more sensor data from the network, virtual services are updated, and phase 2 mapping is triggered periodically in order to adapt to the changed environment.

For service execution, we set up hierarchical monitoring for monitoring service status. We investigate the issue of device clustering for fault monitoring in IoT systems. We model the new monitoring clustering problem as a multiple traveling salesman without depot problem. In order to detect device faults quickly, fault monitoring must be

conducted regularly and frequently. Therefore, it is desirable to reduce the communication cost for fault monitoring. We define the problem by extending the multiple traveling salesman problem (mTSP) in an integer programming (IP) formulation. We also present heuristic algorithms for constructing both monitoring clusters and also the monitoring route within each cluster. Simulation results show that our heuristic algorithms can deliver near optimal solutions on reducing the communication cost, with a low complexity.

Finally, we provide detailed design of the fault tolerance framework, which incorporate above stages and support from our fault recovery mechanism.

Chapter 1

Introduction

1.1 Service-Oriented Internet of Things

The Internet of Things (IoT) systems deploy sensors in the environment for data collection and device control purposes in order to build smart applications such as smart home, smart care, and smart industry[5]. Built on the foundation of wireless sensor network (WSN) [29], each IoT device can facilitate a set of sensors and actuators to connect with the physical environments. Emerging IoT systems often have heterogeneous sensors and can run multi-purpose applications, so that different applications may use a different subset of sensing devices according to their locations, capabilities and availability. Indigenous heterogeneity and constant evolution of IoT systems present new challenges on device management for functional flexibility and service quality [61]. Firstly, heterogeneous devices are deployed in the environment with limited duplications. This brings challenges to find redundant devices in the first place. Secondly, with a changing environment, there comes the need for devices, though deployed with different purposes and capabilities, to collaborate with one an-

other and to be sharable among different applications with quality of services(QoS) requirements. There will be tradeoffs between various kinds of QoS such as accuracy, energy, latency. These bring challenges to the management of IoT applications and IoT devices. Thirdly, in order to achieve failure-resilience on heterogeneous devices, an evolving yet lightweight dynamic binding mechanism should be used. This is the basis for supporting both previous points.

Designed for the integration and management of diverse services, service-oriented architecture (SOA) provides a promising paradigm for the IoT management problem [40][55][63][19][1][56]. Using the SOA paradigm, a light weight middleware can be constructed upon IoT devices to provide the abstraction of integratable and manageable IoT services. The developers and users of sensor devices are thus freed of the details on what and how sensors are used and connected. An important benefit of SOA is its composition and adaptation flexibility. As time changes, locations, users, service parameters or composition of an IoT workflow may also need to be changed. The potential for composing and adapting applications using IoT services makes SOA a very attractive paradigm for future IoT applications.

In our study, we define service-oriented IoT in three layers, i.e. Application, Service and Resource, as shown in Fig. 1.1.

- The application layer defines each application as a flow of *components*. Each component of an application needs to be mapped to actual services in order to be run. A deployment requirement specifies required service types of components and their invocation orders. The application layer also defines policies for applications. Policies help govern how an application should behave. There are applicatoin level and component level policies. Application level policies manage the application behaviour from an end to end execution point of view.

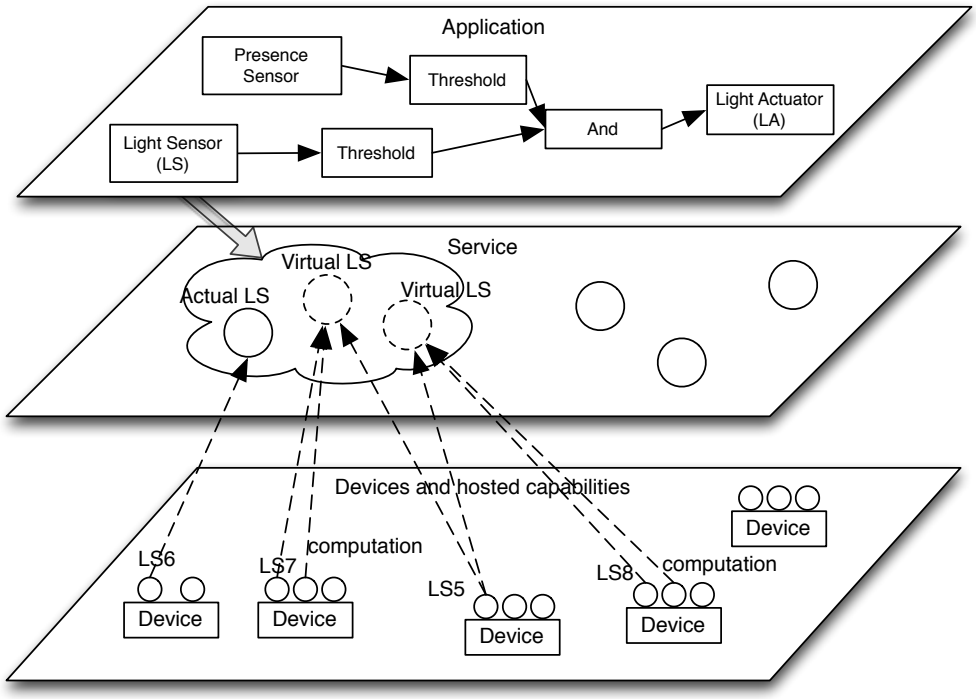


Figure 1.1: Three Layers in Service-Oriented IoT

Component level policies care about preferences from a component's local point of view. For example, different parameters such as energy could be given a high priority in an application policy as an important end-to-end measure of QoS[66][32]. At the same time, different component of the application, depending whether they have energy sources, can have different preferences on energy[48].

- The service layer provides an abstract interface, *sevices*, for the application layer to access *resources*(Discovery). While exposing some of the resource's properties to the application layer, it also adds in properties regarding to how to access the resources. This is useful when a resource could be shared and accessed simultaneously by more than one services. Examples of service layer properties include threshold for updating value change, valid data range etc.. As shown in Fig. 1.1, services may be provided by one resource, or may be

provided by the cooperation of multiple resources. One component from the application layer is able to be mapped to both kinds of services, as long as they can provide similar functionalities.

- The resource layer provides specifications for individual sensors, actuators or computing devices to upper layers. Each attribute in a specification is called a property. While some properties come from factory specifications, others are attained only when sensors are properly attuned and may change with time. Example of resource layer properties include sensor values, location, precision, response time etc..

1.2 Fault Tolerance in Service-Oriented IoT

Fault tolerance is important for many IoT applications since system failures may disrupt users' everyday activities or even endanger their lives. Different from those more fragile and homogeneous WSN sensors, in IoT systems, devices are usually individually deployed with different capabilities and may not be disposable. The lack of redundancy during deployment makes fault tolerance in IoT an important issue. How to provide fault tolerence using a limited set of devices is therefore a big challenge for IoT system design.

Service-oriented architecture is prone to all the faults related to distributed systems, during service publishing discovery, binding, and execution[13]. In a similar way, service-oriented IoT also may face faults from service generation, mapping, deployment and execution processes. In traditional SOA, ways like admission control and runtime migration are used for reducing and handling faults[69][49][68][50]. However, when we bring SOA to IoT, the lack of available services brings new issues. In service-oriented IoT, services are hosted by many different physical devices. It's not

common to have many copies of the same type of services. For example, two light sensors , though of the same type and are close to each other, may still have very different sensed values if there is a directional light source (like a flashlight or a mirror) nearby. This puts the flexibility and reliability of the service-oriented paradigm in danger in service-oriented IoT.

One common fault tolerance solution is to use replicated resources and deploy them in the same environment for services, as assumed in related works[19] [63][60]. For IoT, this requires a complete set of redundant resources, including sensors, devices, energy sources, and may not be practical due to cost reasons. To reduce the cost of a simple replication solution, we propose to combine data from sensing resources of different modalities, and use their combined readings to project the actual reading of actual sensors for backup purposes. This can help solve the availability issue when limited devices are present in the environment. (Chapter 3)

In addition to the availability issue, IoT devices, when compared to web services, are also frequently prone to many types of faults. Devices can be faulty due to hardware failure, power failure, or communication failure. It can affect multiple services on a single device simultaneously.

To achieve service fault tolerance, both fault detection and fault recovery need to be supported[46]. Service fault detection can be supported using either self-diagnosis or cooperative-diagnosis:

- In self-diagnosis, device nodes are supposed to send beacon messages to other nodes. A node can detect if one of its neighbors is at fault when it fails to receive any message from the neighbor for a predetermined time interval. This self-diagnosis method can be used to detect failures which are caused by the depletion of energy, malfunction in the device itself, or interference in the medium.

Self-diagnosis requires no consensus among multiple devices.

- Cooperative-diagnosis, on the other hand, is used more frequently for data fault, or for event tracking. Generally, both spatial correlation among different geographically close sensors, and temporal correlation from the same sensor could be exploited for this kind of detection[58]. Methods like least squared estimation (LSE), maximum likelihood (ML) could be used for spatial correlation, and methods like histograms, autoregressive integrated moving average (ARIMA) could be used for the latter. Cooperative diagnosis requires data from a group of multiple nodes, and a single decision maker, in order to decide the status of the monitored node. How to select the group is usually subject to different applications and different contexts.

We will show in Chapter 5 how we form clusters, and use a monitoring loop to detect single device or single service failures. The system also requires some kind of layered propagation mechanism for the propagation of news about faults. We introduce in Chapter 6 about how heads of different monitoring clusters can be used for propagation and triggering of faulty recovery process.

In order to incorporate the idea of virtual service and fault monitoring into SOA IoT, different processes of service discovery, mapping and execution needs to be taken.

Depending on the frequency and effort required in the process of fault detection and recovery, we can divide it into two categories.

Passive fault detection and recovery is the default fall-back plan for all services. It uses resource discovery for fault detection. Resource discovery is done periodically to catch the update from devices. Every resource discovery requires polling every device in the network. It makes discovery expensive in terms of energy and time cost. Therefore, the frequency of rediscovery is not that often due to the cost. Whenever

some executing services of an application fails, the failure will eventually be caught up. If it is a communication failure problem, it will be picked up by the next resource discovery; if it is from persistent wrong sensing data, it will be detected during the analysis of collected data. Both will trigger a system to update available services, remap, redeploy and restart affected applications, possibly using new devices and services.

Proactive fault detection and recovery will try to actively monitor and prepare backups for potentially faulty services. For many applications, especially for security ones, it is unacceptable to fail for a long time. Passive detection and recovery is slow in discovering faults. Beside that, it also faces the problem of interrupting not only the applications with faulty services, but also those sharing a common device with the affected applications during the restarting process. This interrupt may last several seconds per device for getting reprogramming from a central device, and it could easily accumulate to a long time if there are more devices. In order to lower the time to recovery, applications should prepare backups before fault happens.

Backups, along with monitoring services and backup management services, will be mapped and deployed together with active applications. This brings additional parameters to the service mapping problem which may contradict with the original QoS preference for mapping. Different than the traditional SOA service selection problem [63][67], we model this problem into a multiobjective problem and use the evolutionary method and different decision makers to help solve this conflict. (Chapter 4)

During execution, when a fault is detected by a monitor, the backup management will redirect the faulty service related communication to one of the backup services. This action only requires a rewrite of the message receiver ID in the device's ROM, and is a lot faster than the redeployment of a passive recovery. (Chapter 6)

1.3 Contributions

The contribution of the research reported in this dissertation can be summarized as follows:

- For sensors with different modalities and capabilities, this dissertation shows how to use regression methods to identify their compatibility and composability for fault tolerance. In order to incorporate this idea into SOA IoT, we introduce the concept of virtual services. (Chapter 3)

- This dissertation introduces the mapping problem in IoT and solves it as an integer linear programming problem. The geographical location-based tree model and location-related policies are used to bring down the complexity of the ILP formulation. In order to incorporate virtual services into mapping, an additional phase of fault-tolerant mapping is added for finding backup services. The fault-tolerant mapping problem with virtual services is then modeled as a multiobjective optimization problem and solved using the evolutionary algorithm.[70] (Chapter 4)

- This dissertation proposes a hierarchical structure of monitoring clusters in IoT system, which helps find efficient communication paths and reduce the communication interference among nodes in the same network. It models the monitor clustering as a new multiple travelling salesman with no depot problem. It gives the integer programming formulation to the problem of finding optimal clustering solution. It also proposes heuristic algorithms which achieve quick and close to optimal results. [71](Chapter 5)

- This dissertation proposes a two-phase system deployment methodology for building fault-tolerant IoT systems. The Wukong middleware design is extended

by incorporating fault monitoring, fault-tolerant mapping, and fault recovery. (Chapter 6)

1.4 Dissertation Organization

The dissertation is organized as follow. Chapter 2 surveys related works. Chapter 3 introduces the concept of virtual services and presents methods for virtual service generation. In Chapter 4, we present the concept of mapping and introduce location policies, which are used to help define and map services from geographical point of view. We also introduce how we incorporate virtual services into mapping. Chapter 5 shows the monitor clustering problem and solves it using both optimal and heuristic approaches. Chapter 6 incorporates the fault monitoring, virtual service generation, fault-tolerant mapping, and fault recovery into the WuKong system and introduces how they work through an example. Finally, Chapter 7 concludes the dissertation and points out possible future research directions for mapping and fault-tolerant mapping in IoT.

Chapter 2

Related Work

2.1 Modeling IoT Systems

In order to handle differences among devices, there are projects building middlewares for IoT systems. LooCI[33] is the closest to our project by building a reconfigurable component infrastructure. The LooCI component model supports interoperability and dynamic binding. However, it considers the component selection to be higher-level services outside of its core functionality. ADAE[14] can dynamically reconfigure data acquisition in the network according to a policy regarding the desired data quality. However, ADAE focuses more on the smaller data acquisition problem.

There have been several projects targeting on developing virtual machines for resource constrained devices, either using general purpose languages like Java(Darjeeling[12], Takatuka[7]), or specialized for WSN (Maté[45], Müller et al. citemuller07). These technologies allow IoT devices to be developed with a higher level language to handle heterogeneous networks. However, applications are still written as commands from the devices' perspective. There are also works approach the virtualization of

IoT systems from applications' point of view. In Agilla[23], applications consist of agents. Each agent can move autonomously around different devices. This allows applications to be flexibly distributed and deployed on multiple devices. However, the assembly-like instruction set in Agilla makes it hard to use. MagnetOS [47] proposes a novel programming model which allows the user to write the application as a single Java application, which is then automatically partitioned and deployed to minimize energy consumption. However, the system requires significant computing power on all devices.

2.2 Fault Tolerant Resource Management in IoT

In Service-Oriented IoT, heterogeneity of resources adds more issues to resource management. From the angle of sensors and devices, resource management is about how a network could be organized for fault monitoring and recovery. The most important parameters considered here are the metrics related to the wireless network, like energy, communication, duty cycle etc. From the angle of applications and services, it is about which services should be used in composing an application. End-to-end QoS such as response time, low error rate, or low cost may become important parameters in finding good solutions.

From the angle of sensors and devices, organizing resources for the fault tolerance purpose in wireless sensor network is one widely studied topic[11][43]. However, most of the works focus on networks composed of similar sensors, and used by one application. Clustering is a widely used way for handling the fault tolerance problem in wireless sensor network. For heterogeneous netoworks, the problem is studied as the resource allocation problem[42] or the role assignment problem[25]. Based on the purposes of different applications, different roles can be assigned to a set of nodes

in the network. For duty cycle scheduling, roles of ON and OFF can be assigned; for in-network aggregation, roles of SOURCE, SINK and AGGREGATOR can be assigned; for clustering, roles of CLUSTERHEAD, SLAVE, GATEWAY can be assigned; for fault tolerance, roles of ACTIVE, BACKUP can be assigned. [25] presents a role specification language and heuristic distributed algorithms for the role assignment according to such specifications. They show that generic role assignment is practically feasible for wireless sensor networks while previous works may focus on specific applications. In [42], the authors try to bring fault tolerance to their resource allocation problem. In order to achieve fault tolerance in tracking the location of objects, four kinds of sensors are used. The sensors are redundantly deployed, and multiple models using different sensors are set up, so that even if some sensors are at fault, the application still works fine. While both this work and our work use sensors of different modalities to cover one another, this work is different because it uses some application knowledge for setting up the multiple models. This is like they are setting up several applications simultaneously for the one purpose: one application may use RSSI-based distance discovery for tracking, one may use speedometer and accelerometer for tracking, one may use a mix of the sensors for tracking. We try to prepare our backup sensors only with sensor data and can do it automatically.

SenseWeb[40] is a project which focuses on the improvement on the QoS of multimedia data collection from different sensors for different applications. sMAP[19]tries to profile physical devices and make it fit into the RESTful paradigm. [63] looks into the service composition problem considering the overall cost of the application. However, it treats IoT services as if they were web services which could be easily replaced by one another, while the reality is not. However, these works build IoT middleware without the fault tolerance support.

Only a few projects have considered the fault-tolerant support for multi-modality of

sensors in IoT from the angle of applications and services. [52] tries to find backups by using the error rate of each device. This work assumes each node knows its own probability of errors through a moving average filter . However, errors may be caused by either hardware/software or medium/communication reasons. This may make the chance for a device to go erroneous different from time to time, thus making one stable expression of the error rate to cover all possible errors a challenging problem to solve. [60] looks into the problem of fault recovery in IoT using backups. It assumes the installation of multiple replicated services are already in the system. Another work on fault tolerant health monitoring is reported in [27] which also uses redundant devices and implements an enhanced gateway for fault tolerance. However the reality is that replicated services are very limited in IoT. Even though they are of the same modality, sensors deployed at different places could provide totally different services. Our work tries to avoid using redundant IoT resources to achieve fault tolerance.

2.3 Data Fusion in IoT Systems

Multisensor data fusion is widely used for increasing authenticity and availability of data. However, there are limited works on using data fusion for increasing the availability of IoT services, or in automatic fusion[41].

Our goal of increasing the availability of IoT services and data fusion's goal of increasing data availability results in different preferences in choosing proper algorithms. The correlation among sensors of different modality may result in a common noise observed multiple times on different sensors. This results in a biased observation, and is undesirable for the purpose of increasing data availability in times of noise. However, when we use probabilistic methods for finding potential sensor backups, we are exploiting the correlation among different sensor data for finding unknown

relationships among different sensors, so the correlation is what we are looking for in increasing the IoT service availability.

Though the purpose is different, many data fusion works use different methods for projecting missing or vague data from sensors, similar state estimation methods could be used for data estimation. Some common probabilistic methods that could be used for state estimation in data fusion includes Bayes estimators like maximum likelihood, Kalman filter related methods[51][39] and Monte Carlo simulation-based techniques like particle filter[18]. Monte Carlo simulation-based techniques generally have high computation complexities due to the large population of particles in the simulation. Considering there could be many potential pairs of sensors for testing of correlations, Bayes estimators are chosen for both linear and non-linear estimators in Chapter 3 for the reason of complexity.

2.4 Fault-Tolerant Clustering in IoT Systems

Clustering is widely used in sensor network to deal with the inherently lack of structure in mesh networks, for fault-tolerance as well as other purposes like energy efficiency, communication efficiency[11]. One widely used model to solve fault-tolerant clustering problem is the dominate set clustering [62][43][65]. This technique tries to find a dominate set within the network thus every node in the network is within k hops to the nodes in the dominating set. One famous work based on this model is LEACH[30]. LEACH proposes a distributed algorithm in which a cluster head is elected based on their energy level among neighbors, and nodes join different clusters according to their communication cost to those cluster heads. Cluster heads could be rotated as energy changes in order to help with load balance and fault tolerance of the network. All these works care more about clustering, but not the organization of nodes within

a cluster, thus a star topology is assumed, resulting in extra burdens in the cluster head during run time.

Most of the works we surveyed comply with a distributed pattern. The benefit of distributed algorithms mainly resides on that it could better adapt to the mobility of nodes in wireless ad hoc networks, and a better scalability[43][6]. However, for smart home or office scenarios, only limited mobility for devices is required. Once we deploy sensor nodes, we rarely move them. Location tracking may have moving objects, but in most cases, the sensors detecting human badges or moving objects are set at fixed locations. The benefit of a centralized algorithm is that it can get more information and be more powerful. Among the limited centralized algorithms, G. Gupta et al.[28] tries to achieve fault tolerance on clustering by assigning nodes to some existing gateway nodes through heuristic algorithm. This aims more at partitioning the devices. E.I. Oyman et al.[53] uses k-means to set the sinks in multiple-sink networks considering energy. They try to find the minimum number of sinks while maximizing the network lifetime. This work aims more at finding the sinks or CHs of a cluster. We formulate a centralized clustering problem, which is to solve both the CH location problem and the node assignment problem at the same time.

2.5 Overview of the Wukong Middleware

Our work is based on the Wukong middleware[56]. Wukong is designed to perform automatic sensor identification, node configuration, application upgrade, and system re-configuration. It allows system developers to specify the application behavior at a higher level, instead of telling each sensor node what to do.

The overall flow of Wukong project is shown in Fig. 2.1. There are three parts in the figure.

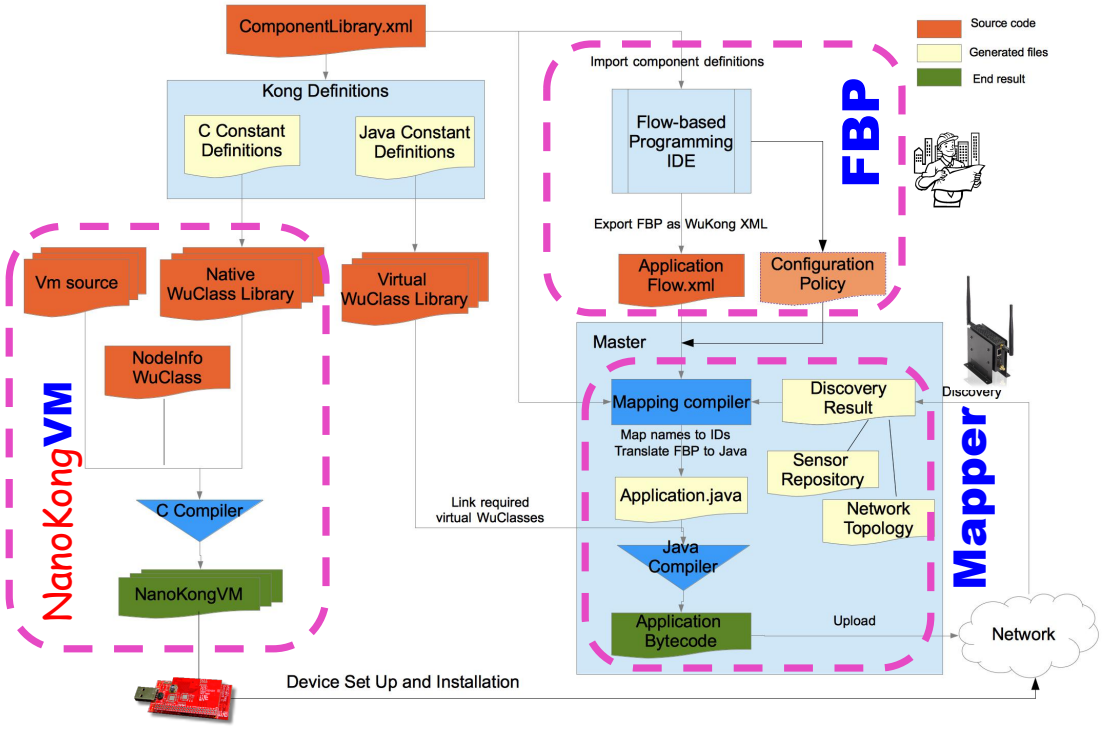


Figure 2.1: Wukong System Flow

- **Flow-Based Programming (FBP)**: On the top right of Fig. 2.1 is a box named flow-based programming IDE. This is the component for users to develop applications in Wukong. Flow-based programming models IoT applications as a flow of information among components. FBP users will select *components* from a library to construct an IoT application. Each component describes a requirement which could be fulfilled by a physical sensor/actuator or some software running on a device. Every component exposes their external interface through one or more *properties*. *Links* are used to connect different properties of components. FBP allows the user to focus on abstract information flow of the

program while the middleware handles the details of communication between components.

One example of a simple FBP program is shown in Fig. 2.2. Fig. 2.2 defines a simple heater control application which turns on the heater when the temperature in an area is below a certain threshold set by the thermostat controller. In this example, the whole application is divided into 4 components, a temperature sensor, a thermostat controller, a threshold and a heater. The temperature sensor will send its latest sensed temperature value to the threshold at a default sampling rate. The thermostat controller will also send the desired temperature to the threshold. Once the threshold finds out that the actual temperature from temperature sensor is below the desired temperature, it will output "True" to the heater to turn it on; Otherwise, "False" will be sent to turn the heater off. The result of this flow-based application is saved in a .xml file shown in Fig. 2.3.

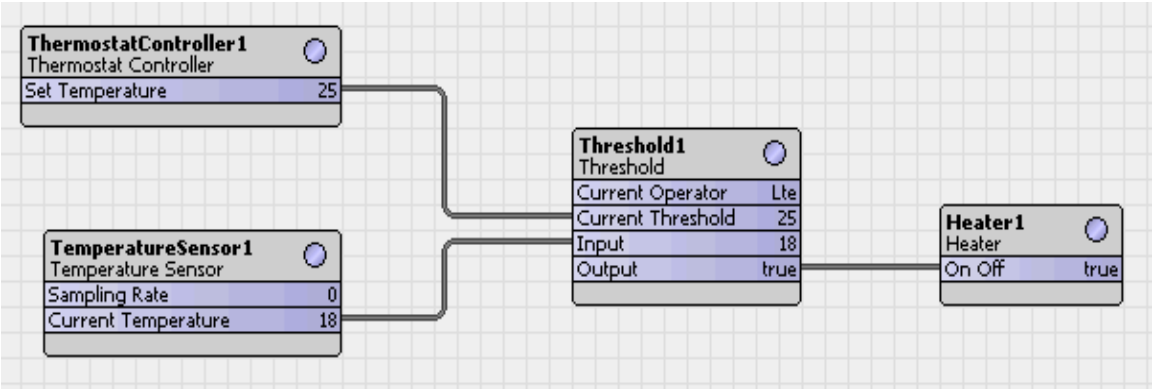


Figure 2.2: A simple heater control IoT application design using FBP.

- **Wukong Master**: Wukong master is shown at the lower right of Fig. 2.1. After an application is designed, it is taken into a centralized control device called Wukong master for deployment. Beside the application flow from FBP GUI, there are also *policies* for instructing the mapping process. Policies can be defined for the entire application, or just for one component. If it is for an

```
<application name="Heater Control">
    <component type="ThermostatController1" instanceId="30" x="245" y="119" w="130" h="100">
        <link fromProperty="current_value" toInstanceId="32" toProperty="selector"/>
        <signalProperty set_temperature="25" />

    </component>
    <component type="TemperatureSensor1" instanceId="31" x="198" y="258" w="120" h="100">
        <link fromProperty="current_temperature" toInstanceId="32" toProperty="input"/>
        <signalProperty sampleing_rate="0" />
    </component>
    <component type="Threshold1" instanceId="32" x="430" y="172" w="110" h="190">
        <link fromProperty="output" toInstanceId="33" toProperty="on_off"/>
        <signalProperty current_operator="LTE"/>
    </component>
    <component type="Heater1" instanceId="33" x="867" y="203" w="120" h="100">
    </component>
</application>
```

Figure 2.3: Heater control application in XML format

entire application, a policy can be used to define the preferable behavior and goals (e.g. energy saving, fault-tolerance) for mapping process; if it is for a single component, a policy can be used to define the preferable QoS(e.g. device location, service accuracy) for the component. In master, according to the FBP and policies, each component is mapped to and then deployed on a real device.

- **NanoKong Virtual Machine (NanoKongVM)**: The left part of Fig. 2.1 shows the process of compiling and deploying the Nanokong VM on sensor nodes. The NanoKong virtual machine is an embedded Java virtual machine which runs in C code but support a simplified version of Java code. It also knows about each device's capability, and supports remote reprogramming on the device. There exists a file called *ComponentLibrary.xml* which describes all possible kinds of components the master supports. Each kind of components here is called a **WuClass**. All WuClasses a device supports are saved in the device's local WuClass library.

  Depending on the implementation language, WuClasses can be divided into two categories: those implemented in C(*Native WuClasses*) and those implemented in Java(*Virtual WuClasses*). Virtual WuClasses runs on the virtual machine, so they can be deployed onto devices remotely and conveniently without touch-

ing the NanoKongVM. Native WuClasses is quicker in terms of the execution speed, but they need to be compiled together with NanoKongVM and are thus not flexible. The differences make native WuClasses more suitable for resources requiring the hardware access function, while make virtual WuClasses more suitable for processing functions which may change over time, especially application specific ones. The comparison between virtual and native WuClasses is shown in Table. 2.1. There is also a NodeInfo WuClass shown in Fig. 2.1. It is a special native WuClass which saves general information about the device, such as device ID, network address and location.

Similar to the class and object definitions in object-oriented programming, there is also a **WuObject** defined for NanoKongVM execution. While WuClasses defines interfaces of components in the VM, WuObjects are used like handlers to load and execute WuClasses. WuObjects are scheduled in the NanoKongVM in a round robin fashion. Every round, the scheduler in the VM will go through each of the WuObjects and check if the WuObject has jobs for execution. If there are, the jobs are executed; otherwise, the turn is skipped for the WuObject. A job could be reading from a certain property, computation, or writing to a property. For property writing, because data usually change much slower than property checking frequency for IoT, only changes need to be written in NanoKongVM to conserve communication.

Table 2.1: Types of WuClasses

Name	Language	Main Usage
Native WuClasses	C	Hardware access, fast processing functions
Virtual WuClasses	Java	processing functions

Chapter 3

Improving the Availability of Services in IoT

Because of limitation on hardware deployment, resources are not always available for providing services for IoT applications. The lack of replaceable services reduces the flexibility of service-oriented IoT. In order to tackle this problem, relationship among different resources need to be exploited to make the most use of existing deployed hardware.

For example, passive infrared sensors (PIR) are often used to detect human presence. However, human presence can also be inferred by other types of sensors such as graphical monitors, infrared sensors (IR), and sound sensors. Integrating several sensors of different modalities can be used to provide the virtual service for a PIR.

3.1 Concept of Virtual Services

We expand the concept of IoT services from simple *actual* services to include *virtual* services. In our definition, an actual service is directly implemented on a device with native resources, but a virtual service may use non-native resources to provide the functionality compatible with the actual service. An actual service usually uses only one resource in the resource layer, but a virtual service may rely on one or more sensing resources.

The relationship among components, virtual services and resources is shown in Fig. 3.1.

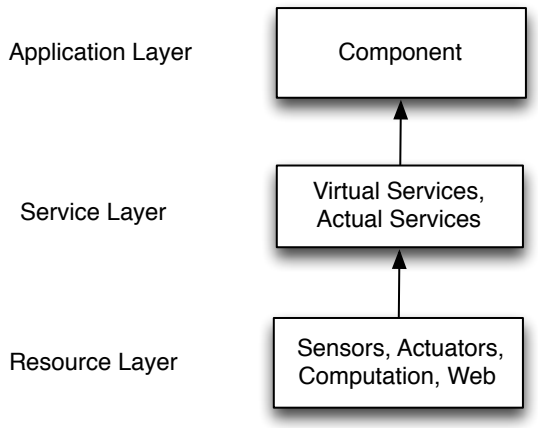


Figure 3.1: Concepts in Service-Oriented IoT

We now investigate how to identify virtual services through historical data collection. One simple idea is to include all sensor resources within the vicinity of a sensor as candidates for identifying its virtual services. This takes advantage of the geographical proximity to indicate potential data correlation. However, the significance of correlations among data may come from causal relationships from the deployed application itself. Such correlations should be used with caution.

For example, Fig. 3.2 shows a light control FBP application which turns on the light when both light sensor and presence sensor in an area are above some thresholds. A

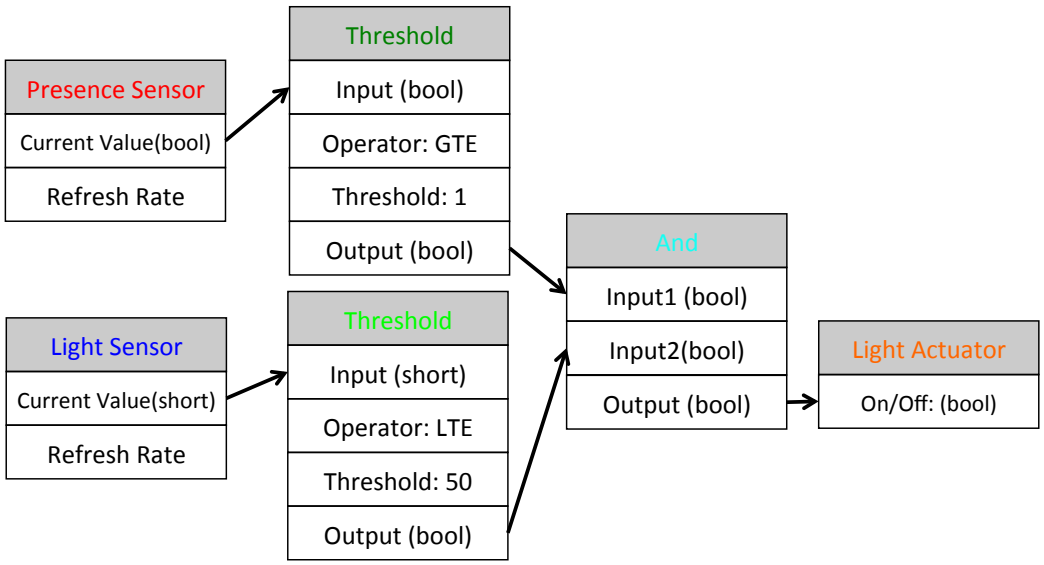


Figure 3.2: FBP using presence and light sensors to control light

detected human presence will trigger the light actuator in a dark room, which will then affect the light sensor. Sensor data collected from this room may indicate a relationship between the presence and light sensors. This may suggest the possibility of using light sensors as a backup for the presence sensor. However, the light level change is due to the application operation, and cannot replace the presence sensor. To avoid such situations, manually defined relationships among different sensor modalities would be very useful.

3.2 Vitual Service Generation

We identify virtual services through regression. Methods of regression can be divided into two categories, linear models and non-linear models. Both models have their merits. In general, linear models are faster and require relatively less data, non-linear models are more flexible yet more complex and thus require more data to converge. Depending on the situations, both models can be useful.

3.2.1 Linear Models

A linear model is very common for correlation among real world sensor data. For example, there is usually a linear relationship among light sensors deployed at various locations in a room with sensing values linearly related to their distances to the light source. Suppose there are $M-1$ sensors that are linearly related. We can express the relationship among these data through the following linear expression:

$$y_i = \sum_{k=1}^{M-1} x_{k,i} * h_{k,i} + x_{M,i} \tag{3.1}$$

In Eq. 3.1, $x_{k,i}$ is the correlation factor for sensor k at time instance i. $x_{M,i}$ is the correlation factor for the constant term. For ease of expression, we can model it as a pseudo sensor M with $h_{M,i} = 1$ at all time i. Therefore the equation can be expressed as a simple matrix multiplication:

$$y_i = H_i^T X_i \tag{3.2}$$

where $H_i = [h_{1,i}, h_{2,i}, ..., h_{M,i}]$, $X_i = [x_{1,i}, x_{2,i}, ..., x_{M,i}]$. Assuming correlation factors X_i have little change in the small time window before time i, our goal is to identify X_i using the values in $\mathbf{Y}$ and all relevant sensors $\mathbf{h_k}$ up to time i.

We propose to use the Recursive Least Squares (RLS) algorithm [59] for finding X's estimated expected value, $\bar{X}$, and its variance Φ.

$$\Phi = E((X - \bar{X})(X - \bar{X})^T) \tag{3.3}$$

RLS is a commonly used algorithm for estimation. It recursively finds the correlation

factors that minimize the weighted sum of error squares relating to the input data. As a recursive algorithm, it only requires the newest data and the last estimation of $\bar{X}$ and Φ to generate the next estimate. For a cold start when there is no historical data, we can set $\bar{X_0} = [1, 1, ..., 1]$, $\Phi_0 = \infty * I$. After that, $\bar{X}$ and Φ are updated at each time interval according to the following RLS formula:

$$\bar{X_i} = \bar{X}_{i-1} + K_i e_i \tag{3.4}$$

where e_i is the error between the observed value and the estimated value at time i. K_i is a gain vector calculated using Φ_{i-1}, $\bar{X}_{i-1}$, new data H_i and a forgetting factor λ:

$$K_i = \frac{P_{i-1}\bar{X}_{i-1}}{\lambda + H_i^T P_{i-1} H_i} \tag{3.5}$$

$$P_i = \Phi_i^{-1} = \lambda^{-1}(P_{i-1} - K_i \bar{X_i}^T P_{i-1}) \tag{3.6}$$

One benefit of RLS is that it doesn't require much memory for processing and can be done very fast. Although it actually uses all data for regression, only the newest data is required every time for updating the result. Another benefit of RLS is that we can use a forgetting parameter λ to discount older historical data. As scenarios progress with time, older data may not be as useful as newer data. The forgetting parameter λ between 0 and 1 can be used: 1 means old data are of the same importance as new data, and 0 means old data is not useful at all for estimating new correlation factors. A practical λ value ranges from 0.95 to 1 depending on the scenario and data interval.

3.2.2 Non-Linear Models

In many systems, the relationships among sensors are not linear. This is especially common when we use virtual services across modalities. For example, using a sound sensor to predict the presence requires the translation from a numerical value of sound wave amplitude to a binary presence result. This conversion is more of a categorical relationship than a linear one, making non-linear models better suited.

Multivariate adaptive regression splines (MARS) [26] is one of the well-studied non-linear models. The benefit of MARS is that it can efficiently approximate the relationships between active sensors and potential backup sensors in a piece-wise regression. As a non-parametric regression approach, MARS divides data into subregions, and use different model functions for regression in each subregion. MARS model shares a similar form to linear regressions, as shown in the equation below. However, the b in the equation is different from the raw data h in linear progression.

$$y_i = sum_{k=1}^{M} x_k * b_{k,i} \tag{3.7}$$

In MARS, b is either a constant or a product of hinge functions. It is through hinge functions the original data can be divided into subregions. A hinge function can be expressed as either $b*_{k,i} = max(0, b_{k,i} - c)$ or $b*_{k,i} = max(0, c - b_{k,i})$, where c is a constant called a knot, representing the borders of each subregion. To protect against over fitting, MARS also prunes unnecessary hinge functions. Because of its good performance in simulating categorical relationships, we use MARS as a non-linear model in our study. In many systems, the relationships among sensors are non-linear. This is especially true when we use virtual services across different modalities. For example, using a sound sensor to predict human presence requires the translation from a numerical value of sound amplitude to a binary presence result. This conversion is

more of a categorical relationship than a linear one, making non-linear models better suited.

Multivariate adaptive regression splines (MARS) [26] is one of the well-known non-linear models. The benefit of MARS is that it can efficiently approximate the relationships between active sensors and potential backup sensors in a piece-wise regression. As a non-parametric regression approach, MARS divides data into subregions, and uses different model functions for regression in each subregion. The MARS model shares a similar form to linear regression, as shown below.

$$y_i = \sum_{j=1}^{N} x_j * b_{j,i} \tag{3.8}$$

However, the b value in Eq. 3.8 is different from the raw data h in linear progression. In MARS, b is either a constant or a product of hinge functions. A hinge function can be expressed as either $b_{j_1,i} = max(0, h_{k,i} - c)$ or $b^1_{j_2,i} = max(0, c - h_{k,i})$, where c is a constant, called a *knot*, representing the border for each subregion. Using hinge functions we can divide the data from a sensor into subregions based on their values, and have different correlation factors for different subregions. This enable MARS to catch non-linear categorical relationships. To protect against overfitting, MARS also prunes unnecessary hinge functions.

Many relationships among sensor values may be categorical. For example, in order to compose a binary virtual sensor using numerical actual sensors, numerical sensor values need to be truncated into at least two categories. Because MARS has a good performance in simulating categorical relationships, we use MARS as a non-linear model in our study.

3.2.3 Practical Issues

In general, a linear model runs faster and requires relatively less data, and a non-linear model is more flexible yet more complex and thus requires more data to converge. Depending on the system characteristics, both models can be useful. In Sec. 3.3, we will show actual examples and compare their performances.

One practical issue, however, is the correlation among data may come from some causal actions due to the application itself. Such correlations should be excluded carefully. For example, in Fig. 3.2, a detected human presence will turn on the light in a dark room, which in turn affects the readings of the light sensor. Therefore, the sensor data history from the room may indicate a strong correlation between the presence sensor and the light sensor and suggest the possibility of using light sensors as a backup for the presence sensor. However, the light level change is in fact the result of the application's response to the PIR sensing, and should not be used as a presence sensor. To avoid such confusions, pre-defined causal relationships among sensors should be identified for excluding their replacement for each other in the virtual service generation.

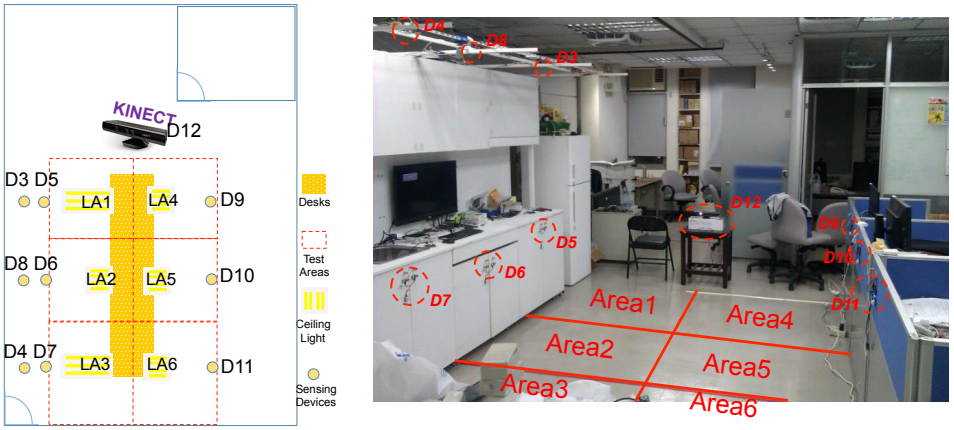
3.3 Experimental Study

To verify the effectiveness of building virtual services, we build and deploy the application in Fig. 3.2 in an office area, collect sensor data, analyze the data using RLS and MARS.

3.3.1 Experiment Setup

We deploy the light control application from Fig. 3.2 for our experiment. As shown in Fig. 3.3(a), a meeting room scenario is used with a row of desks in the center of the room, and 6 seats along the desk. There are 6 ceiling lights in the room corresponding to the 6 seats. We deploy the application so that when a person is in an area, the light for the area is turned on.

10 devices are deployed in the testing area (Fig. 3.3(b)). Devices $D3, D8, D4$ are deployed at higher positions on the left part of the room, devices $D5, D6, D7$ at lower positions, and devices $D9, D10, D11$ on the right side. There is also device $D12$, a Kinect to monitor the whole area. The Kinect is able to identify the area where a person is and is used as a presence sensor.

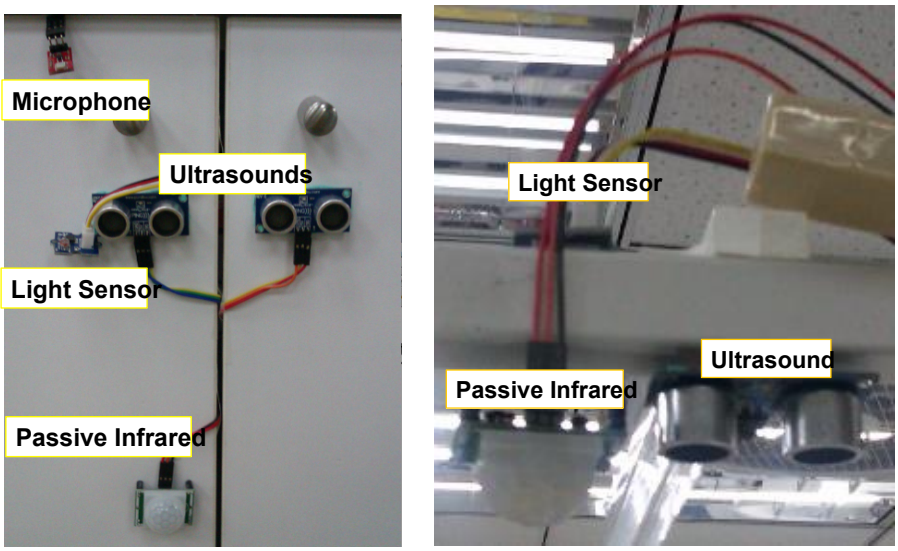


(a) Simplified Floor Plan (b) Actual Device Deployment

Figure 3.3: Scenario Layout

As can be seen in Fig. 3.4, every device at lower positions is equipped with a light sensor(LS), a passive infrared sensor(PIR), a sound sensor(MICRO), and a pair of ultrasound sensors(DIST). Every device at higher positions is deployed with a light sensor, a PIR and an ultrasound sensor. We'll name resources after their device ID. A light sensor on device $D4$, for example, is called $LS4$. Ultrasound sensors are used

to detect distances, and will be abbreviated as $DIST$.



(a) Sensors at Lower Devices($D5, D9$ etc.) (b) Sensors at Higher Devices($D3$ etc.)

Figure 3.4: Sensor Deployment on Devices

For initial deployment, we study an instance of Fig. 3.2 application deployed in area 2. Kinect ($D12$) is selected as the presence sensor, and $LS6$ from area 2 is chosen as the light sensor. They send data to trigger the light in area 2 $LA2$. After the initial deployment, data from all devices are collected for analysis. A person will randomly move and stand in the test area, holding a video player to simulate talking. This process lasts for 40 minutes for data collection. We record the status of every sensor in every second of the 40 minutes. The data are analyzed using RLS and MARS. For RLS, we set the forgetting factor λ to be 1, which means there is no need for forgetting.

3.3.2 Generating Light Sensing Virtual Services

Given the system has 8 backup LS resources for the original $LS6$, there can be a total of 255 combinations to create virtual services. In order to show the effectiveness of

both regression methods, we repeat the experiments for $LS5$ as a comparison. We show the results in Table 3.1 and Table 3.2 respectively. We apply a 5-fold cross validation (CV) on a selection of virtual services in both cases. CV can help better show the compatibility of RLS and MARS models for prediction, limiting problems like overfitting. However, since CV requires all data to be saved in memory for calculation, it will harm the performance of online calculation, especially for the RLS calculation. Therefore, for virtual service generation, we use the simpler correlation coefficients.

Table 3.1: Correlation and Cross Validation(CV) for simulating LS6

Sensors	RLS	MARS	RLS CV	MARS CV	Sensors	RLS	MARS	RLS CV	MARS CV
LS3	0.6366	0.7654	0.5543	0.6828	LS5, LS7	0.9312	0.8957	0.9019	0.8764
LS4	0.4578	0.8997	0.5571	0.8801	LS3, LS4	0.6562	0.8911	0.7200	0.8795
LS5	0.9059	0.8490	0.8319	0.7862	LS5, LS8	0.9510	0.9578	0.8922	0.8935
LS7	0.6723	0.6838	0.8125	0.8274	LS8, LS10	0.9021	0.9592	0.8892	0.9135
LS8	0.5037	0.9145	0.6397	0.8712	LS3,4,9	0.9173	0.9558	0.8837	0.9278
LS9	0.6186	0.8435	0.4717	0.7540	LS5,8,10	0.9670	0.9736	0.9191	0.9309
LS10	0.7355	0.9245	0.5810	0.8317	LS3,5,7,8,10	0.9657	0.9615	0.9216	0.9330
LS11	0.7393	0.9160	0.6129	0.8495	All 8 LSs	0.9656	0.9596	0.9189	0.9298

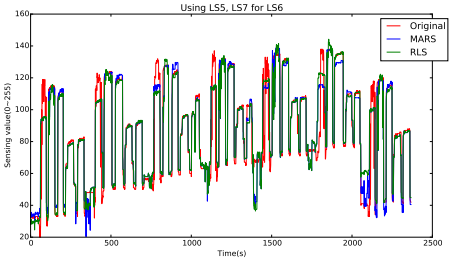
Table 3.2: Correlation and Cross Validation(CV) for simulating LS5

Sensors	RLS	MARS	RLS CV	MARS CV	Sensors	RLS	MARS	RLS CV	MARS CV
LS3	0.5538	0.5648	0.5904	0.5443	LS6, LS7	0.9130	0.8820	0.8273	0.8647
LS4	0.1811	0.8362	0.2754	0.8375	LS3, LS4	0.5208	0.8462	0.6171	0.7335
LS6	0.9036	0.9275	0.8317	0.8412	LS6, LS8	0.9169	0.9502	0.8399	0.8833
LS7	0.4696	0.5005	0.6603	0.7457	LS8, LS10	0.7902	0.9123	0.7910	0.8832
LS8	0.2442	0.8868	0.4112	0.8688	LS3,4,9	0.8919	0.9489	0.8672	0.9065
LS9	0.7464	0.9297	0.5953	0.8139	LS6,8,10	0.9203	0.9444	0.8483	0.8851
LS10	0.7875	0.8698	0.6509	0.7784	LS3,5,7,8,10	0.9336	0.9289	0.8715	0.8624
LS11	0.7307	0.8793	0.6216	0.8031	All 8 LSs	0.9326	0.9495	0.8809	0.8209

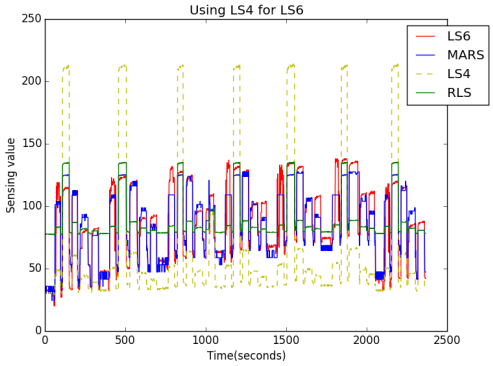
In Table 3.1, $LS6$ is deployed at center of the room, backup sensors are relatively more helpful in this case. MARS usually shows a slight better performance than RLS, but not always. RLS has a good performance largely because LS's are numerical sensors and we are using the same modality of sensors for generating LS virtual services. This means the relationships could actually fit into linear models. Though less complex,

RLS still outperforms MARS in some situations (e.g. using $LS5$ in Table 3.1). This may be due to that MARS has more coefficients to be set, and MARS has the process of pruning unnecessary hedge functions while RLS doesn't. As a comparison, in Table 3.3, because $LS5$ is deployed at one corner of the room, backup sensors are relatively less helpful. We can see both the correlation coefficients and CV of virtual services, though follow a similar pattern to $LS6$ cases, are lower compared to $LS6$ cases.

In Table 3.1, $LS5$, $(LS5 + 7)$, $(LS5 + 8 + 10)$, $(LS3 + 5 + 7 + 8 + 10)$ are the best virtual services with 1, 2, 3, and 5 sensors respectively under RLS cross validation. The results are all quite good, ranging from 0.83 to 0.92. Using $LS5 + 7$ can achieve 0.90, 0.88 under RLS and MARS respectively, as shown in Fig. 3.5(a). From the table, we can also see that starting from 2 sensor virtual services, adding more sensors won't help much to improve the correlation coefficient. One interesting thing is for virtual services using $LS4$ or $LS8$, MARS has a much better results than RLS, suggesting the relationships between $LS4, LS8$ and $LS5, LS6$ are more categorical than linear. Fig. 3.5(b) shows the results of using $LS4$ for predicting $LS6$. It suggests that this may be caused by $LS4$ and $LS8$ are deployed too close to light sources $LA3$ and $LA2$ respectively. Those light sources therefore have far more significant impacts on those two sensors compared to other light sources. This leads it to be two different cases when these close light sources are on or off. Another interesting case is $(LS3 + 4 + 9)$. These sensors are farther from $LS6$ than other sensors, and perform badly by themselves individually. But by working together as a virtual service, these sensors achieve a result quite close to the best ones. This suggests the inclusion of virtual services indeed brings real benefits for backup plans.



(a) Generating virtual service for LS6 using LS5 and LS7



(b) Generating virtual service for LS6 using LS4

Figure 3.5: Sensor Deployment on Devices

3.3.3 Composing Presence Sensing Virtual Services

The presence sensor has one important difference from the light sensor. Since the presence sensor is to produce a binary output, the output of its virtual services should be a binary value as well. We need to use a threshold to decide if the output value is 1 or 0. A simple fixed threshold of 0.5 is used for binarization in our experiments.

Table. 3.3 shows the correlation results in the first two columns, and cross validation (CV) results in the last two columns. In this study, MARS is much better than linear regressions in most cases. This is because the underlying relationships between collected data and the desired presence result is not a linear one.

Table 3.3: Correlation and Cross Validation(CV) for PIR

Sensors	RLS	MARS	RLS CV	MARS CV
MICRO6	0.3547	0.4889	0.2504	0.4898
all MICROs	0.3547	0.4692	0.2607	0.4823
DIST6	0.9547	0.9547	0.6098	0.7859
all DISTs	0.9547	0.9661	0.6872	0.7862
MICRO6,DIST6	0.5425	0.9431	0.5020	0.7845
all PIRs	0.2994	0.3297	0.1427	0.1606

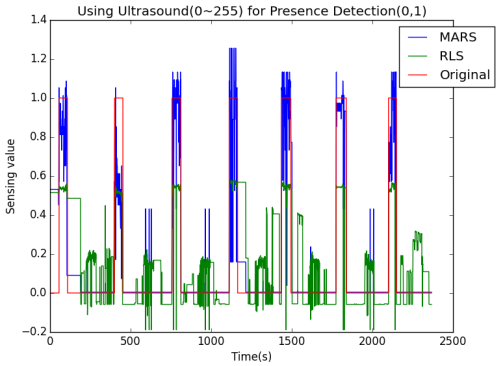
One interesting result is that with all PIR sensors, although they are widely used for

presence and motion detection in many real world systems, we can only achieve 0.16 for the MARS CV result. This is because PIR has a big detection range of several meters but the study area is a small rectangular area. So the intended detection range is not a good fit. In comparison, the ultrasound $DIST$ sensor can easily scale to rectangular areas and $DIST6$ from area 2 alone achieves a high cross validation value of 0.79 under MARS. Unfortunately, since an ultrasound sensor only has a narrow line of detection, more DISTs won't help much to improve the result. The comparison between the $DIST6$ virtual service results and the original Kinect results before binary conversion is shown in Fig. 3.6(a). We also show the 0.5 line used for binarization in this graph. We can see that RLS often produces a very borderline result (values only slightly above 0.5) compared to the detection by the presence sensor. But the binarization step makes it a good virtual service.

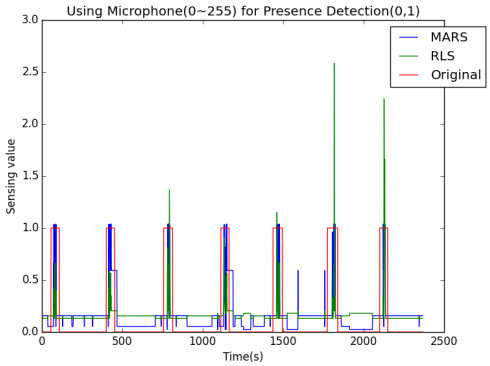
The results from cross validation reveal that there is a drop from row 3 (DIST6) RLS to row 5 (MICRO6, DIST6) RLS even when more sensors are used. It is because the additional MICRO6 data provided often lower the RLS sensor values to be slightly below 0.5, and cause the binary output to switch from 1 to 0. This big difference suggests that the correlation result before and after binarization may be quite different, and we cannot use the correlation results before binarization to reflect the goodness of simulated binary sensors.

Another interesting result can be found for microphones. Using $MICRO6$ for human presence in area 2 before binary conversion is shown in Fig. 3.6(b). Although it has only a cross validation result of 0.49 using MARS according to Table. 3.3, the results in Fig. 3.6 actually show that $MICRO6$ has very low false positive results, (0.02, 0) respectively for (RLS, MARS). Looking further into the data, we find that the microphone sensing remains zero most of the time even though people are walking in the area. If we have microphones with a better sensibility, they can make good

substitutions for the presence detection.



(a) Using ultrasound sensors before binarization



(b) Using microphone before binarization

Figure 3.6: Detecting presence using ultrasound sensor or microphone

In general, when considering numerical sensors for virtual services, especially when sensors have the same modality, it is more likely there is a linear relationship among those sensed values. RLS may be a better choice since it doesn't need old data for update and requires less time and memory for execution. It can also adopt a time discounting factor which makes it better for adapting to some changing environment. However for most cases, we may not know whether the relationship among sensors is linear. In fact, whenever data conversion is required from numerical sensing values to binary results, the relationship is not linear. Under those cases, we should use MARS or other non-linear algorithms for detecting the relationships.

Chapter 4

Mapping in IoT

In IoT, if we map a service onto a device, two types of QoS parameters will be generated: device parameters and inter-device ones. Device parameters are the ones defining capabilities of the device. Depending on which is more limited, device context information like device energy, device location, memory, disk space and prices could all be considered as the device cost. Inter-device parameters are the ones related to sending and receiving messages to other devices. This cost as well could be consisted by many things. Examples include sending energy consumption, receiving energy consumption, bandwidth consumption, etc.

4.1 Using Communication Cost for Inter-Device Cost

For the inter-device cost, we'd like to consider the link communication cost as the main QoS. The motivation for approaching the mapping problem through inter-device communication comes from our experience in deploying IoT applications in Wukong.

When we conduct experiments in an indoor lab, it always looks fine if we deploy a relative small IoT application with 3 or 4 Arduino device boards. However, if we deploy several applications at the same time, each application with its own set of boards, or if we deploy a large application, sometimes, the execution of the application simply would stuck. By further looking into the problem, we found that if the device is densely deployed and if communication messages are sent too frequently in applications, there would be severe interference in communication. We have deployed two applications in the same room. One application is a light control application which uses three devices(light controller, threshold, light actuator) and a linear structure. The other application is a data collection application which uses a totally different set of devices and has them sending their data to a common gateway device. The frequency of the sending is 1s. All the communication is done through Z-Wave with CSMA/CA exponential backoffs. An ack is required for every successful sending of message. We switched the controller and recorded how much time it needs until the light actuator is turned on or off. Fig. 4.1 shows the result. With no device sending data to gateway, the light control application takes about 1.16s with a standard deviation of nearly 0 on 10 runs. However, if we have 5 devices sending their data, it takes on average 4.37s with a 3.76s standard deviation.

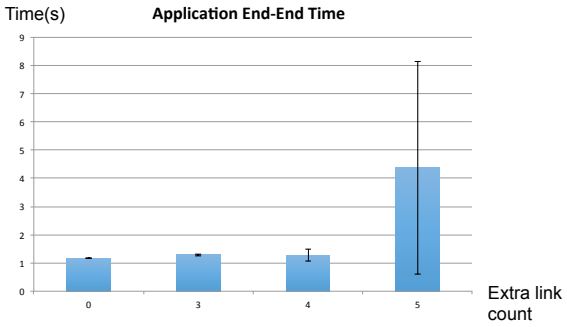


Figure 4.1: More communication in the space will inversely affect execution time of the same application

Thus, we'd like to consider the communication cost as the parameter to be optimized.

Minimizing the communication cost not only reduces the time spent on sending and receiving messages, it could also help reduce the number of devices used for the application. Since in Wukong, both system control messages and data messages have a small payload of no more than 20 bytes, differences in the message length will not make a big difference in communication. For reason of simplicity, we also fix the sending signal strength for different devices. Therefore, only the frequency of messages and the number of hops between components will be used as an indicator of the communication cost along a communication link. We have:

$$linkCost = f * h \tag{4.1}$$

In Eq. 4.1,$linkCost$ is the cost along a certain link in an IoT application, f is the frequency of message sending along the link, and h is the number of hops between two ends of the link.

The number of hops h is a parameter determined by the network topology and devices mapped. In Wukong, the network topology is created during discovery/rediscovery of devices, so it is easy for master to calculate/update how many hops it requires to send messages from devices to devices.

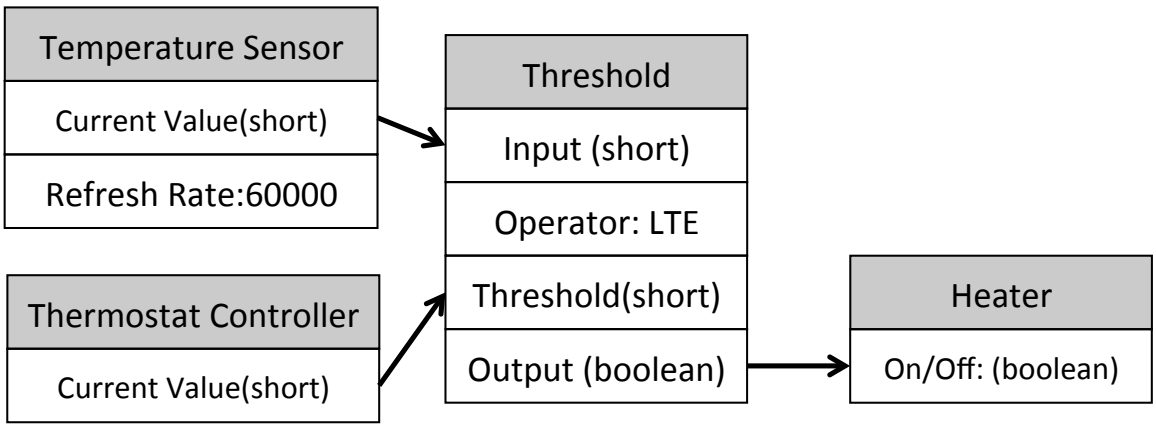


Figure 4.2: A simple heater control application designed through FBP.

Frequencies of messages through links f, on the other hand, are irrelevant to how applications are mapped and they can be deduced by using the frequency of the head components. A head component is a component in an application that doesn't have any inward links. There can be two kinds of head components. If it is a component for some actuator(e.g. a switch or a timer), the frequency of communication is the same as the expected frequency of triggered actuation. If it is a component for a sensor, the frequency of communication would depend on the sensor output data type. For a numerical sensor, the data reading frequency would be used as the communication frequency; however for a binary sensor, the data reading frequency divided by 2 should be used as the communication frequency. This is because only changes are sent by Wukong for output, and for binary sensors, the values are expected to be changed half of the times. Output link frequencies for other components in the application would be decided by their inputs. Normally, the frequency of a component being executed is the same as the component getting a new input, which is the summation of frequencies from all inputs. Again, the output frequency is different based on the output type in this case. If a component has a numerical output, the message frequency would be the same as the execution frequency; if a component has a binary output, the message frequency is only half of the execution frequency.

Fig. 4.2 shows an example of a heater control application. In the example, the "Temperature Sensor" is a numerical sensor with the sample rate set to 60s(1Hz). So the link from "Temperature Sensor" to "Threshold" sends messages at a frequency of 1Hz. The "Thermostat Controller" is an actuator triggered by human which changes about twice per day(0.001Hz). So the link from "Thermostat Controller" to "Threshold" sends messages at 0.001Hz. The "Threshold" is an component with binary output. So the link from "Threshold" to "Heater" sends messages at half of the input frequencies which is 0.5005Hz.

4.2 Using Location Policy for Device Cost

Component policies are used by mapper to help define the device cost. Whenever a policy is given by a user for one component, a score can be produced according to this policy. Based on the scores from different policies, services from different devices are evaluated. Depending on the resource scarcity, there could be many kinds of important component policies. Energy, device location, sensing accuracy, availability, and memory usage can all make good QoS in ranking the services. Among these, we select device location as our QoS parameter.

Geographical location information is an important dimension for the context of devices. Different than many other QoS parameters, location is also a unique QoS information for IoT systems. Most of the times, users of IoT applications are not the ones who install devices and sensors, and they are not clear about the capabilities and properties of each individual device. However, they may be able to describe the locations of desired services. For example, for a user of the heater application in Fig. 4.2, he may not know which temperature sensors are available. However, he knows he wants to use the temperature sensor in **the same room** as the thermostat controller and the heater.

Thus it would be helpful to have location models and policies to support this kind of queries. Such location policies could be the fundamental ones in deciding the usability of different devices and their services. One component mapped to a service at a wrong location may greatly affect the correctness of the whole application. Thus, in this section, we'll define location policies which are used to describe the best locations for mapping components.

4.2.1 Geographical Modeling

The purpose of location policies is to support location-aware queries for services. We intend to support the most common ones, range queries and k-nearest neighbor (closest, farthest) queries for location policies. They are "the major focus of attention" in literatures concerning queries[34].

The first step towards location queries is to adopt a location model to express the spatial relationships among different devices the services are hosted on, and the environment. Generally, there are two kinds of approaches for spatial modeling, the geometric-based approaches and the symbolic-based approaches[4]. The geometric-based approaches mainly use coordinates to express spatial relationships. Examples of geometric-based approaches include quadtree[57], grid-based expressions[22], expression using irregular spaces like through triangulations[21]. The geometric-based approaches can provide precise locations, direction and distance information. However, it suffers from the lack of flexibility in dynamic environments, which is important for tracking and localization in many IoT applications. Some extra effort are also required for achieving queries in geometric-based models, since the models are not set up directly for query purposes. The symbolic-based approaches try to summarize spatial relationships into some abstract ones such as "contain", "near", "on top of". Examples of symbolic-based approaches can virtualize the space according to sensor ranges[10], voronoi paths[16], and space separations (e.g. rooms and floors)[31][8]. This simplification makes symbolic-based approaches less accurate when compared to geographic approaches. Symbolic-based approaches also lack geometric connectedness information among places. However, symbolic approaches, by extracting and maintaining only partial geometrical information, have a good level of efficiency for queries and good flexibility for dynamic environment.

In order to take advantage of both the convenience of symbolic hierarchical location expression and the explicity of coordinate based location, we adopt and extend the hybrid location model[37] . We can explain the model through an example floor plan shown in Fig. 4.3. This floor plan can be expressed in a tree structure shown in Fig. 4.4(a). In order to make use of the coarse grained hierarchical expression, multiple coordinate systems coexist in the tree. For a coarser level like "2F", there is only the symbolic model, while for finer levels like "204" and below, both symbolic (containment relationships) and geometric model (sizes and coordinates of spaces) coexist. The tree structure is like an index for devices, and is updated periodically from location queries to all devices inside of it.

In order to overcome the lack of connectedness problem in the hybrid model, we also incoporate the accessibility graph[36] into the hybrid tree to bring in more space context information. *Entrances* , defined as distances from one point in one coordinate system to one point in another coordinate system, are defined to connect different coordinate systems in the tree. For example, in Fig. 4.4(b), there is an entrance from coordinate (0,0) of Room 204 to coordinate (1,3) of South Corridor. The distance is 0 because they are actually the same location. If all rooms and places are defined under the same coordinate system, it falls back to the geometric location model which is more complex but has more information.

One coordinate system can be used by one floor, one room, or even just one region of a room. In Fig. 4.4(b), each room has its own coordinate system. When a room could be further divided into subregions, the subregions inherit the coordinate system of their parent regions. For a valid location tree, every leaf node in it must be in one coordinate system.

There are *devices*("LS3", "LS4" in the example) and *landmarks*("Desk" in the example) defined under coordinate systems in the location tree. Devices may hold several

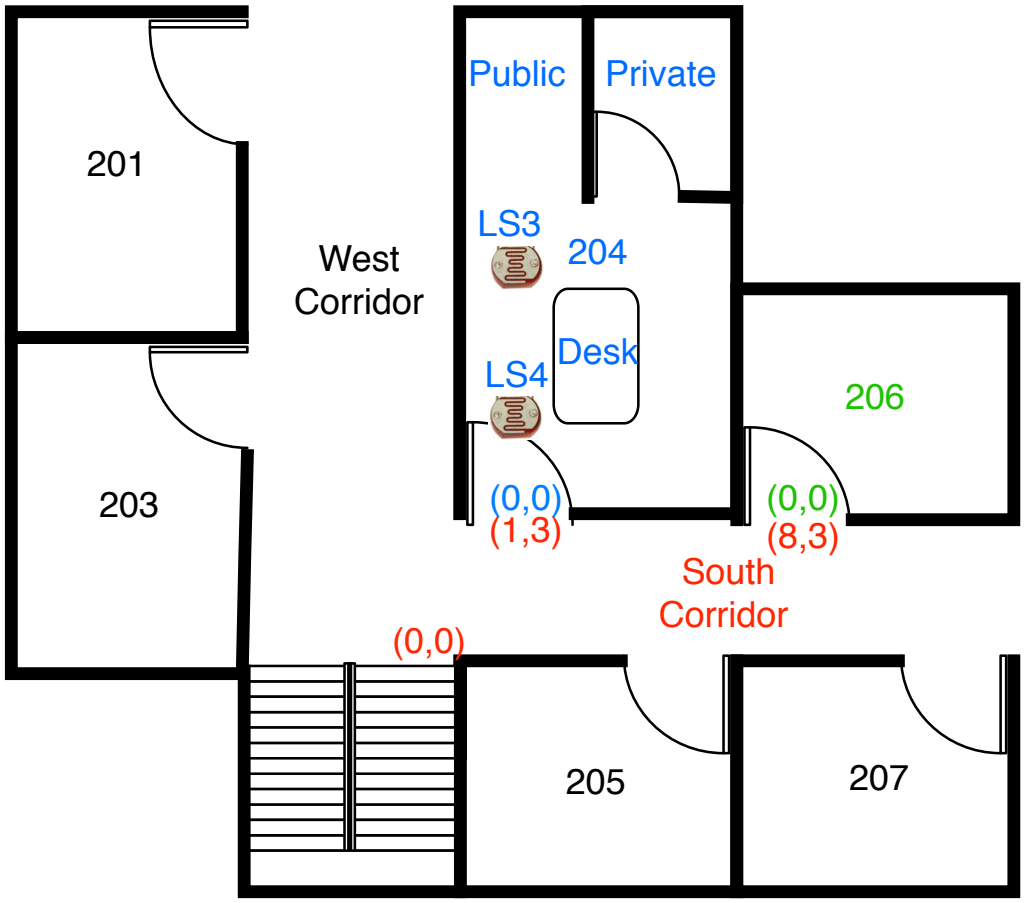
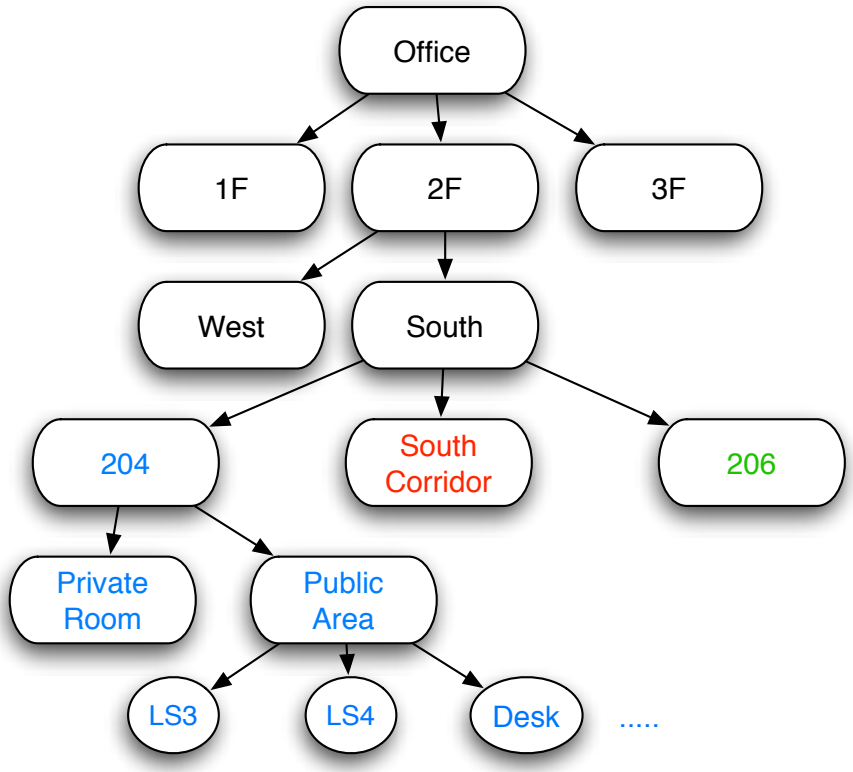


Figure 4.3: Example Floor Plan

actual services. An actual service uses the location of its device for the service location, while a virtual service uses the location of its output service holder as its location. This is because virtual services always have only one output device, and it is this device that defines where the virtual service affects other services.

Locations can be expressed in a URL-like format for the hybrid model. For example in Fig. 4.4(b), the location of $LS3$ can be expressed as:

$$"/Office/2F/South/204@(0,8)"$$



(a) Example location tree

(b) Example accessibility graph

Figure 4.4: Example showing a floorplan and its corresponding location tree and accessibility graph. Different coordinate systems are shown in different colors.

The strings before "@" is used for expressing the hierarchical location in the tree, and the one after "@" is used for expressing the local coordinate. We assume device and sensor locations can be assigned manually by some device installers, or could be acquired by some localization methods (e.g. anchor-based triangulation[54]). Locations are stored in ROMs of these devices. On device discovery, the master will query all registered devices; the devices who are still alive will respond with their locations along with other device information. These devices will then be inserted into the

location tree. If a device is discovered at a new location, this location will be created in the location tree as well. For example in Fig. 4.4(b), if a new device is discovered with a location indicating $/Office/2F/South/205@(0,1)$, then a new node "205" will be inserted under the "South" node in the location tree. It is through the device location information that an initial location tree can be set up.

This initial location tree can already perform queries, but in order to make use of landmarks, entrances, human installers also need to add more information after the initial set up. Beside the landmarks and the entrance information, properties about spaces can also be set to describe the location tree nodes. These detailed properties include height, width and length. For example in Fig. 4.4(b), properties could be set to "204", "Private Room", "LS3", etc. Basically any node under a coordinate system can be given some properties. This helps to support more accurate location policies.

As a benefit of the hybrid model, only important spaces with devices need to provide the geometric model for building the tree. Thus geometric information could be expressed by different granularities of precision, saving the effort in covering the entire place. When needs arise, it's also easier to adjust the model by adding more spaces.

4.2.2 Distance Calculation

Distance in location tree is defined as the attainability between different points in space. It is achieved by calculating the Euclidean distance within one coordinate system, plus the shortest path in the accessibility graph among different coordinate systems. For example, the distance from LS3 of room 204 to coordinate (0,1) of 206 would the distance from LS3 to the entrance of 204, to the corresponding entrance of South Corridor is, to the 206 related entrance in South Corridor, to the 206 entrance,

and finally to (0,1) coordinate. This distance definition will skirt around walls which divide spaces. Therefore, it makes a better representation of distance when a user sets the policy and when sensors sense.

4.2.3 Location Policy Design

Table 4.1: Location Policy Functions

Name	Input	Action
InRange	coordinate, distance	unqualified device filtered
CloserTo	coordinate	distance to the device
FartherTo	coordinate	max(CloserTo) - distance to the device
Outside	coordinate and shape	unqualified device filtered
Inside	coordinate and shape	unqualified device filtered
Above	coordinate and shape	unqualified device filtered
Below	coordinate and shape	unqualified device filtered

Table. 4.1 shows a list of location policy functions we currently support. We adopt a simplified rectangular model for describing objects, so only height, length and width are required. Among all the policies, "InRange", "Inside", "Outside", "Above", "Below" belong to range queries, while "CloserTo" and "FartherTo" belong to nearest neighbor queries. Range queries will filter unqualified devices. Nearest neighbor queries will try to calculate the cost for different devices. For "CloserTo", the cost is achieved by calculating the distances to a device. For "FartherTo", the cost is the maximum of "CloserTo" subtracted by the distance to a device. A lower cost indicates a better fitness to location policy.

All individual location policy functions are combined together using disjunctive normal form(DNF) for expressing users' preferences. Costs will be evaluated upon every qualified candidate service using the DNF for decision making.

An example of expression of location policies using DNF is shown below:

$$"/Office/2F/South/204$$
$$\#InRange(Desk, 2m)|CloserTo(Desk)\&InRange(Desk, 5m)"$$

There are two parts separated by the '#' sign in this location policy string. The first part is used to express the containment relationships within the hybrid location model. It also defines the range of search for devices in the location tree. The second half connect all policies through the DNF. In DNF, the priorities of operators are:

NOT($\neg$) > AND(&) > OR(|).

So the example DNF is requesting to find a device within 2 meters to the desk or a closer device within 5 meters to the desk . An importance weight of 0–10 can be added to each policy by the user (through input as an extra parameter in the policy function). This weight is used to reflect how much the user values each policy setting. Different conjunctive clauses in DNF will perform their own calculations: firstly, all filtering actions are performed for each conjunctive clauses; then, the weighted sums of scaled individual policy costs are calculated as the cost of the entire conjunctive clause. The cost for the ith conjunctive clause $cost_i$ is:

$$cost_i = \frac{1}{K} \sum_k \frac{s_{k,i} - \min_j s_{k,j}}{\max_j s_{k,j} - \min_j s_{k,j}} * w_k$$

In the formula, there are K policies used in the conjunctive clause. Each policy has an importance of w_k, and will give out a score $s_{k,i}$ for the ith candidate. For one candidate service, the minimum of all the conjunctive clause costs will be taken as

the final location cost.

$$cost = min_i(cost_i)$$

If we don't add in other costs, this cost is also the device cost for mapping. A candidate with a lower device cost will have a higher chance during mapping.

The following is the details of BackusNaur Form (BNF) definition for our location policies and location definition. The BNF form defines and can be used to parse the location policy. Scores are calculated along the parsing. After the parsing, all candidate services to a component will get a score. The service with the highest score will be more likely to be chosen as a component.

⟨*locationPolicy*⟩ ::= ⟨*path*⟩ | ⟨*path*⟩ '#' ⟨*opOr*⟩

⟨*locationDef*⟩ ::= ⟨*path*⟩⟨*coordinate*⟩

⟨*coordinate*⟩ ::= ' @(' ⟨*real*⟩ ',' ⟨*real*⟩ ')'

⟨*opOr*⟩ ::= ⟨*opAnd*⟩ '|' ⟨*opOr*⟩ | ⟨*opAnd*⟩

⟨*opAnd*⟩ ::= ⟨*opNegate*⟩ '&' ⟨*opAnd*⟩ | ⟨*opNegate*⟩

⟨*opNegate*⟩ ::= '∼' ⟨*opNegate*⟩ | ⟨*function*⟩

⟨*function*⟩ ::= ⟨*functionName*⟩ '(' ⟨*paramList*⟩ ')'

⟨*paramList*⟩ ::= ⟨*param*⟩ | ⟨*param*⟩ ',' ⟨*paramList*⟩

⟨*param*⟩ ::= ⟨*word*⟩ | ⟨*real*⟩

⟨*word*⟩ ::= ⟨*word*⟩ '[a-zA-Z0-9_]'

⟨*real*⟩ ::= ⟨*number*⟩ | ⟨*number*⟩ '.' ⟨*number*⟩

⟨*number*⟩ ::= [0-9] | [0-9]⟨*number*⟩

4.3 Mapping Problem Formulation

For mapper, because device location costs and inter-device communication costs are different metrics, it's not very meaningful to combine them together. Since location costs are generated for individual services while inter-device communication cost are generated for the entire application flow, we'd like to divide the mapper into two phases: filter and mapping. In the first phase, services on different devices are ranked by the location policy, only the service candidates with lower costs go into the second phase. Then in the second phase, the candidates with lower device costs are considered for the real mapping process using the inter-device cost.

We now formulate our problem. Let's say we have a system with a set of m devices:
$D = \{d_1, d_2, ..., d_i, ..., d_m\}$
An application with n components:
$S = \{s_1, s_2, ..., s_j, ..., s_n\}$
needs to be deployed in the system. Frequencies of communications among components are expressed through f_{j_1,j_2}. We know about which services are supported on each device from the device discovery, and let's use a variable y to denote this:

$$avail_{i,j} = \begin{cases} 1, & \text{if component} s_j \text{can be deployed on device} d_i \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

We also know the number of hops from device d_{i_1} to d_{i_2} is h_{i_1,i_2} . We want to have a solution so that every component in S is mapped onto a feasible device from D while the total communication cost is minimized. This problem , being similar to the general mapping problem, is NP-Complete [42]. It could actually be mapped to an **integer programming (IP)** problem.

Let's use $x_{i,j}$ to denote if s_j is mapped onto d_i.

$$x_{i,j} = \begin{cases} 1, & \text{if} s_j \text{is deployed on device} d_i \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

Thus the mapping problem can be formulated like the following:

$$\min \quad \sum_{j_1=1,\ldots n} \sum_{i_1=1,\ldots m} \sum_{j_2=1,\ldots n} \sum_{i_2=1,\ldots m} x_{i_1,j_1} * f_{j_1,j_2} * h_{i_1,i_2} * x_{i_2,j_2} \tag{4.4}$$

$$s.t. \quad \sum_{i=1,2\ldots,n} x_{i,j} = 1, \quad \forall j \tag{4.5}$$

$$x_{i,j} \leq avail_{i,j}, \quad \forall i,j \tag{4.6}$$

$$x_{i,j} \in \{0,1\}, \quad \forall i,j \tag{4.7}$$

Eq. 4.5 ensures every component is mapped to one device. Eq. 4.6 maps components only to devices that support them. Eq. 4.7 ensures it is an integer programming. This formulation makes the mapping problem a pseudo-quadratic problem. To solve it, we need to further change it into its linear form. So let's add in a set of new variables:

$$y_{i_1,i_2,j_1,j_2} = x_{i_1,j_1} * x_{i_2,j_2} \tag{4.8}$$

Then our problem could be rewritten as:

$$\min \quad \sum_{j_1=1,\ldots n} \sum_{i_1=1,\ldots m} \sum_{j_2=1,\ldots n} \sum_{i_2=1,\ldots m} y_{i_1,i_2,j_1,j_2} * f_{j_1,j_2} * h_{i_1,i_2} \tag{4.9}$$

$$s.t. \quad \sum_{i=1,2\ldots,n} x_{i,j} = 1, \quad \forall j \tag{4.10}$$

$$x_{i,j} \leq avail_{i,j}, \quad \forall i,j \tag{4.11}$$

$$x_{i,j} \in \{0,1\}, \quad \forall i,j \tag{4.12}$$

$$y_{i_1,i_2,j_1,j_2} \leq x_{i_1,j_1}, \quad \forall i_1,i_2,j_1,j_2 \tag{4.13}$$

$$y_{i_1,i_2,j_1,j_2} \leq x_{i_2,j_2}, \quad \forall i_1,i_2,j_1,j_2 \tag{4.14}$$

$$1 + y_{i_1,i_2,j_1,j_2} - x_{i_1,j_1} - x_{i_1,j_1} \geq 0, \quad \forall i_1,i_2,j_1,j_2 \tag{4.15}$$

$$y_{i_1,i_2,j_1,j_2} \in \{0,1\}, \quad \forall i_1,i_2,j_1,j_2 \tag{4.16}$$

Eq.4.10– 4.12 are the ones inherited from the quadratic form. Eq.4.13– 4.15 make sure y is the product of two xs. From the problem formulation, we can see the problem search space is $2^{m^2n^2}$, where m is the number of devices and n is the length of the application.

The solver to the IP problem is implemented in Python. To test its speed, a simulation is done on a PC with an 2.3GHz Intel core i5 and 4GB memory. During the simulation setup, devices are deployed in a square matrix. Every device is only able to communicate with devices at its 4 neighboring positions. The routing hop count is set up based on shortest path between devices.

For each run, every kind of services is deployed randomly on 20% of the devices. A linear application with varying sizes will be deployed onto the matrix of devices. For simplicity of the simulation, communication frequencies among components of the application are set to be the same. We calculate the average time of 10 runs for each setting of parameters and show the result running time in Fig. 4.5. This result

shows the running time of the IP solver increases exponentially to both the number of devices and the size of the application.

The simulation stops at a deployment with 64 devices and 108 related services because it is about a similar size to the large deployments in a real world smart home. For example, PlaceLab[35] at MIT, as an experimental environment with dense sensors, deploys 125 sensor devices in a 1000 sq. ft. apartment. However, because these 125 devices each hold only one sensing services and there are 7 kinds of services in the system, the number of devices really considered for mapping problems in the system is much smaller. We can see from Fig. 4.5, mapping an application with 9 components onto the 64 device deployment takes about 8 minutes. This is an acceptable speed for an offline algorithm with infrequent use. Thus we can use this solver directly in the mapper.

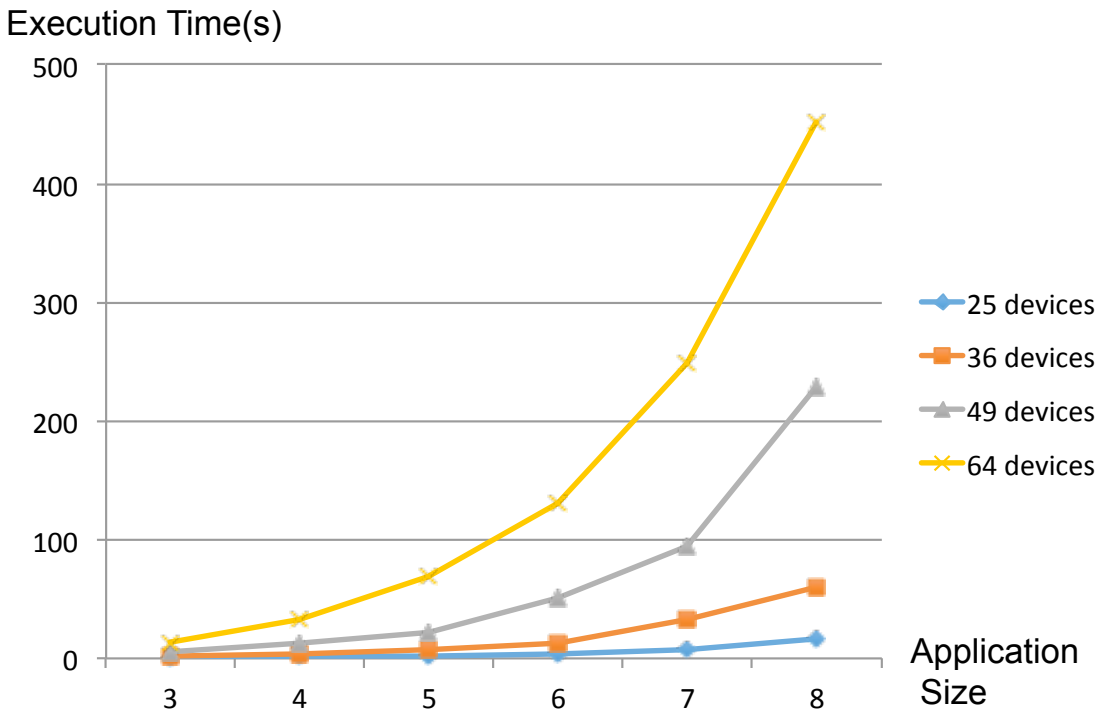


Figure 4.5: Running time for mapper using an IP solver.

4.4 Mapping for Fault-Tolerant Backups

The mapper in Section 4.3 finds appropriate services for each component. Without considering fault tolerance, the mapping process is modeled as a service selection problem considering end-to-end quality of services. However for the mapping of backup services, there is no guarantee which service may be broken first and what the overall application flow will be like after some faults happen. So it is really difficult to plan backups according to end-to-end QoS's as presented in the original mapping process. Additionally, the generation of virtual services requires data from active service mapping result. Thus, mappings for fault-tolerant backups with the help of virtual services should happen after the mapping of actual services for components is done. In other words, there should be two phases for fault-tolerant mapping. In the first phase, components only use actual services for mapping. Virtual service generation begins after the first phase. Redundant services from the first phase mapping are used to generate the first batch of virtual services. These virtual services are composed of only one device and have default fidelity. With more data collected, these virtual services may have various qualities, more virtual service may also be generated. When there are enough qualified candidates for backup mapping, a second phase mapping will happen.

For phase 2 mapping, inclusion of backups and virtual services brings about new parameters and goals to consider. The fidelity of a virtual service is crucial to the correctness of the application. Meanwhile, the number of devices used for virtual services is also an important decision factor. Therefore, we model the fault-tolerant mapping problem as a *multiobjective optimization* problem with two goals.

The first goal is to maximize the combined fidelity of all backup services. Because any badly mapped component will result in a significant performance degradation of

the whole application, an application composed of average performance components is preferable to the one with mixed good and bad performance components. Thus, the fidelities of backups from different active services in an application are multiplied together to get the score of the application. If an component requires two backups, the average of the two backups is put into the multiplication.

The second goal is to minimize the number of devices used for candidate services under the premise of meeting fault-tolerance requirements. To minimize the number of devices, we prefer backup plans of different components to share devices as much as possible. This has two benefits. Firstly, this means a smaller number of devices inflicted by fault monitoring. Fault monitoring requires sending messages periodically among devices which is a big burden for the network. Reducing devices used for backups means less system messages occupying the wireless communication bandwidth. Secondly, encouraging service instances of different components to share devices also implies there may be more intra-device communications and less inter-device communications. Communications among components on the same device are fast and light weight, and don't require bandwidth. This will reduce both the communication cost and running time of the application.

In order to handle device failures, during mapping, chosen backup services of the same component must share no common device. Therefore, on device or actual service failure, there is always a backup ready to fill in. There are also other restrictions. Components may choose to have more than one backups. Also, some important components may require only services with certain scores to qualify. Our formulation of fault-tolerant mapping problem brings together all these restrictions.

To formalize the fault tolerant mapping problem, suppose we have a set of N components $\{C_1, C_2, ..., C_i, ..., C_N\}$ for the application. For each component C_i, there is a set of candidate services: $\{s_{i,1}, s_{i,2}, ..., s_{i,j}, ..., s_{i,K_i}\}$. Every component C_i requires

b_i backups. Every service $s_{i,j}$ has a score of $p_{i,j}$ and requires the cooperation of a set of $D_{i,j}$ devices . The minimum score for component i is p_{min_i}. We'd like to find out which backup services will maximize the overall application fidelity while minimize the number of devices used.

We'll use a binary selection variable $x_{i,j}$ to express if a service is selected during mapping. Please note i starts from 1, but j is from 0.

$$x_{i,j} = \begin{cases} 1, \text{if} s_{i,j} \text{is selected} \\ 0, \text{otherwise} \end{cases} \tag{4.17}$$

The multiobjective optimal problem is expressed as:

$$\max \quad \prod_i \frac{1}{b_i} \sum_j p_{i,j} x_{i,j} \tag{4.18}$$

$$\min \quad |\cup_{i,j} \{D_{i,j} | x_{i,j} = 1\}| \tag{4.19}$$

$$\text{s.t.} \quad \sum_j x_{i,j} = b_i + 1, \forall j \tag{4.20}$$

$$x_{i,j} \leq p_{i,j} - p_{min_i} + 1, \forall i, j \tag{4.21}$$

$$D_{i,j_1} \cap D_{i,j_2} = \emptyset, \forall i, j_1 \neq j_2, \text{where} x_{i,j_1} = x_{i,j_2} = 1 \tag{4.22}$$

In the definition, the first goal is to maximize the product of all selected backup services, and the second goal tries to minimize the total devices used. Eq. 4.20 defines the number of backups. Eq. 4.21 enforces the minimal score. Eq. 4.22 means no two backup of the same component share the same devices.

Sometimes due to the lack of resources, some device may be required by all possible candidate services of the same component. The above multiobjective optimal problem won't be able to find a solution under this circumstance. This means there is no

possible solution to guarantee fault recovery on one device failure. However, we may still want a solution that at least provides fault tolerance for some of the devices. To adapt this, condition Eq. 4.22 can be relaxed to only requiring services to vary by some of the devices. Then Eq. 4.22 could be written as :

$$D_{i,j_1} \cup D_{i,j_2} \neq D_{i,j_2}, \forall i, j_1 \neq j_2, \text{where} x_{i,j_1} = x_{i,j_2} = 1 \tag{4.23}$$

$$D_{i,j_1} \cup D_{i,j_2} \neq D_{i,j_1}, \forall i, j_1 \neq j_2, \text{where} x_{i,j_1} = x_{i,j_2} = 1 \tag{4.24}$$

This change will make it easier to have feasible solutions. However, Eq. 4.23 and Eq. 4.24 won't be able to guarantee there is always a backup ready when the shared backup device fails.

4.5 Solving Fault-Tolerant Mapping Problem

The multiobjective optimization problem is not an integer linear programming problem since the fidelity has the multiplication term. So we have to rely on some heuristic algorithms for solving it. Since there are two objectives, there may exist several best solutions. Utility functions could be used to solve the problem, but utility functions only give us one fixed solution while there exist multiple ways in balancing the tradeoffs between the score(gain) and the device number(cost). Moreover, it is hard to decide weights a priori for utility functions. Thus evolutionary methods better suite our interest. Evolutionary methods can yield a Pareto frontier, which is a set of Pareto optimal solutions, rather than only one solution. This results in a better flexibility. Among evolutionary methods, we adopt non-dominated sorting genetic algorithm II (NSGA-ii)[20], a fast elitist multiobjective genetic algorithm. It has a low computational complexity, good spread of solutions, and a convergence close to the real Pareto frontier.

In NSGA-ii, each active service makes up a gene in the genome. All the available backup service candidates for this active service are modeled to be bases for that particular gene. The rationale of this setting is that backups with a higher fidelity and a lower device count tend to make good genes and should be preserved through generations.

After a Pareto frontier is generated by NSGA-ii, another decision is required in deciding the final solution. While users can choose to make their own decisions, some a posteriori utility functions can be defined to be the decision maker (DM). There are already many good methods proposed for deciding the preferred solution [72]. But because our solution space is small due to the limited possibilities of device count, we'll only use two simple DM for the final decision.

The first decision maker is the *Least Devices First*. This method will sort the final Pareto optimal results according to their devices and choose the ones with the least devices. The argument behind this is that as long as the minimum fidelities are met, the system may care more about the increased cost for maintaining fault tolerance.

The second decision maker is the *Efficiency First*. This method tries to make a decision based on how much marginal fidelity gain could be earned for each extra device used. The marginal fidelity gain is the difference between the application fidelity and unacceptable fidelity threshold. For virtual services corresponding to actual service i, suppose the minimum fidelity is $pmin_i$, then the unacceptable fidelity threshold for the whole application would be $\prod_i pmin_i$. For each solution in Pareto front, suppose the result of its first objective about fidelity is p and its second objective regarding device count is d. Then its efficiency can be defined as:

$$\text{Efficiency} = \frac{p - \prod_i pmin_i}{d} \tag{4.25}$$

A higher efficiency means this solution uses its devices better in improving the fidelity from the minimum required. These two methods are provided as policies for fault tolerance before deployment of application.

Chapter 5

Fault Detection for IoT

In previous chapters, we have discussed about how we make use of service-oriented architecture to make backups for fault tolerant IoT. However, in order to support recovery through backups for fault tolerance, an efficient fault monitoring mechanism is necessary.

5.1 Motivation

As a motivation example, consider a smart home that supports many IoT applications, from human location monitoring to automatic light controls. Sensors and actuators are deployed on devices with wireless communication capabilities such as Zigbee, Zwave, and Wifi. Each component of an application must be deployed on some hardware devices and run as IoT services. Communications are then established to facilitate the application flow.

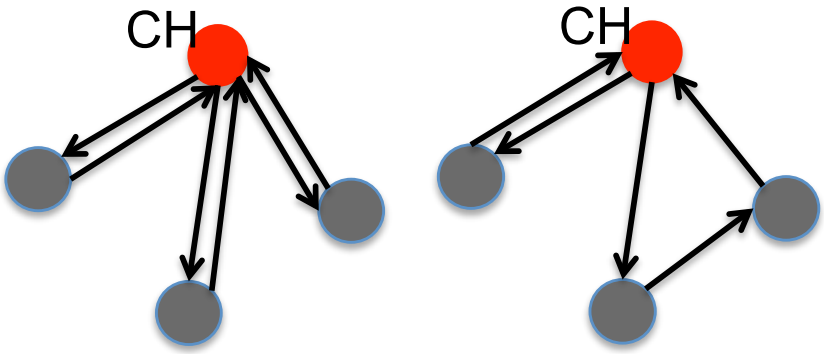
The goal of monitoring in IoT is for locating faults or anomaly on devices, and identifying affected services so that these services can be replaced during fault recovery.

The most straightforward way for fault monitoring is to use a central controller. This controller would send polls to periodically to ask devices to report status. If any abnormal behavior is found, or if someone doesn't reply, there may be a fault on the device. This central controller will then consider which services may be at fault based on what services are deployed on that device.

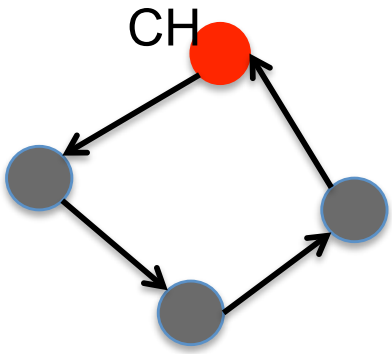
This simple mechanism has the scalability issue. Firstly, as more devices join the network, the controller has to take care of more fault tolerance combinations. Secondly, with the increasing number of devices, the time for sending and receiving polls also increases. Lastly, all monitoring related messages going to the controller will create bottlenecks at the controller device and devices around it. This will not only waste the communication resource and energy, but also affect the performance of active applications running on these devices.

Therefore, instead of only one controller, we propose to divide the IoT devices in a system into several clusters for monitoring. Monitoring happens only internaly in clusters. Each cluster is managed by a coordinator, a *Cluster Head* (CH) , to enhance efficiency. A CH is used to keep the backup information for each device in the cluster. When a fault happens, some device in the cluster will detect the fault and will report to the CH. The CH will then replace the faulty service with backup ones. The problem here is how we can detect the fault.

In IoT, instead of the simple time-triggered communication pattern, event-triggered communication is widely used in many IoT systems to save the communication cost. Event-triggered communication means a message is sent only when a significant change in the sensed value is detected. However, this also means that if no message is received, a receiving device cannot tell whether nothing has happened or the sender has failed. For fault detection using event-triggered communication, periodic beacons are often used to let beacon receivers know that senders are still healthy.



(a) Star Topology (b) Mixed Topology



(c) Loop Topology

Figure 5.1: Loop topology has less communication cost than star and mixed topologies.

To save the communication cost and to divide the load of a CH, we propose to use a loop topology for sending beacon messages. We compare the differences between the loop topology and the commonly used star topology in Fig. 5.1. The star topology would expect the CH to send and receive all beacon or ack messages. However for a loop topology, the work is divided by all devices in the cluster. As long as the beacon can be passed down through the loop, there is also no need for reporting back. This greatly reduces the number of packets used for monitoring in a loop topology cluster than other types of clusters. Considering the CH may need more than one hops to reach other devices in a cluster, the loop topology saves more hops than the star topology.

However, one main challenge of the loop topology is that any fault in it may interrupt the beacon message propagation. To handle this, two timers are used to measure the beacon period for each device: one for controlling the pace of beacon message sending, one for message receiving time out. Depending on different application requirements, different devices could have different sending and receiving periods. The CH is needed to be the one in charge of initializing the timers. One node will send a beacon message only when its sending timer times out. With the CH staggering the sending timers of the nodes in a cluster, this will also prevent monitoring beacon message burst within a cluster. With the help of receiving timers, even if no beacon message is received from a previous node, the next beacon message will be be sent. This helps make loop structure more robust. For CH, if it finds a node constantly to be faulty, it will check what services are affected on the faulty node, make requests to backup nodes for replacement of these faulty services. After all replacements are in place, it safely removes the faulty node from the monitoring loop by letting the faulty node's parent directly monitors its child. Whether or not all these changes are successful, the CH reports to the master about the situation and updates, and the master updates the status of devices accordingly.

Another challenge of the loop topology is that a big cluster may result in hardship in management and scalability concerns, while a small cluster may limit the capability of CH to coordinate the message sending among nodes. Thus, for scalability and load-balancing reasons, we decide to have a lower bound (LB) and an upper bound (UB) on the number of nodes in a cluster. The lower bound depends on the fault tolerance requirement of all applications in the system. If there could be two node faults happening simultaneously, the size of a cluster should be at least 3, and there should be 2 backups for CH. The upper bound is chosen based on the maximum number of nodes in a space(eg. a room). Placing nodes from different spaces into one cluster will bring extra monitoring cost and is undesirable. Setting an upper bound

can also help to greatly limit the search space for the monitoring solution.

5.1.1 Definition of the Monitoring Clustering Problem

We would like to minimize the monitoring communication cost for the clustering problem. Since monitoring message size matters little in the transmitting time in this case and there may be many frequency levels of monitoring, we use the hop count multiplied by the message frequency as the communication cost.

We assume an IoT scenario with richly deployed devices forming a mesh network where all nodes could reach any other device in the network through one or more hops. A comprehensive routing table can be drawn from the routing information collected from all devices and we know the number of hops between different nodes. The requirement for monitoring frequencies are given by applications on different devices, they may be the same for most of the times for a practical system. However, the actual monitoring frequency calculation should take into consideration active application communication among the devices. For example, if the required monitoring frequency of a device is 5s per message (12 messages per minute), and a specific route from the device is already preoccupied by a sensor detecting and sending data every 10s(6 messages per min). Considering the underlying communication protocol using the piggyback mechanism to send both messages in one shot, the extra need for the monitoring effort will be 6 messages per min for the route, so the level of frequency is changed. Thus, we know the monitoring communication cost between every pair of devices. Using these costs as edge weights and devices as nodes, a fully connected graph can be drawn.

The *Monitoring Clustering Problem* (MCP) is defined using this constructed graph G. Let V be all nodes in the network, $n = |V|$. $G = (E, V)$ be a fully connected

directed graph. There is an $e \in E$ for any possible pair of $v_1 \in V$ and $v_2 \in V$, and denoted as (v_1, v_2). There is a weight w_{v_i,v_j} indicating the cost for each (v_i, v_j). MCP is to divide V into K sets $\{s_1, ..., s_k, ..., s_K\}$ as monitoring clusters and select E', a subset of E, as monitoring routes, so that:

- $\forall v_i \in V$, there is one and only one s_k, that $v_i \in s_k$.
- $\forall s_k \in S$, the size of it, $|s_k|$, is between the lower bound LB and the upper bound UB, or $LB \leq |s_k| \leq UB$.
- $\forall v_i \in V$, there is only one incoming and one outcoming edge in E'
- There doesn't exist a $(v_i, v_j) \in E'$ so that $v_i \in s_{k_1}, v_j \in s_{k_2}$ and $k_1 \neq k_2$
- The total weight of edges in E', $\sum_{(v_i,v_j)\in E'} w_{i,j}$, is minimized

5.1.2 Integer Programming Formulation

We find the MCP problem similar to the multiple travelling salesman problem (mTSP) [9]. However, mTSP requires fixed starting depots, and in our clustering problem, how many CHs are there and which nodes are CHs are not pre-determined. Thus we have to extend the mTSP problem to a problem of mTSP with no depot.

For the integer programming(IP) formulation, let's first define a binary selection variable $X = \{x_{i,j,k}\}$. $x_{i,j,k}$ is 1 if (v_i, v_j) is selected in a cluster with v_k as the CH (denoted as s_k) and 0 otherwise. i, j, k range from 0 to $n - 1$.

$$x_{i,j,k} = \begin{cases} 1, \text{if } (v_i, v_j) \in s_k \\ 0, \text{otherwise} \end{cases} \tag{5.1}$$

To simplify the expression, we introduce two sets of auxiliary variables. y_i is another binary variable in the problem used to express whether a node is selected as a CH, where a cluster starts to grow. It depends on $x_{i,j,k}$ and is optional.

$$y_i = \sum_j x_{i,j,i}, \forall i \tag{5.2}$$

u_i is a non-negative integer variable for expressing the monitoring order of node v_i in a cluster. If v_i is the CH of a cluster, u_i is assigned to 1, it is then increased by one for every node after it in the monitoring loop. With these three sets of variables, the IP problem formulation for MCP is:

$$\min \quad \sum_k \sum_i \sum_j w_{i,j} * x_{i,j,k} \tag{5.3}$$

$$\text{s.t.} \quad \sum_k \sum_j x_{i,j,k} = 1, \quad \forall i \in E' \tag{5.4}$$

$$\sum_k \sum_i x_{i,j,k} = 1, \quad \forall j \in E' \tag{5.5}$$

$$\sum_i x_{i,j,k} = \sum_i x_{j,i,k}, \quad \forall j \in E', \forall k \in E' \tag{5.6}$$

$$(LB-2) \sum_{v_j \in V} x_{i,j,j} \leq u_i + y_i - 1, \quad \forall i \in E' \tag{5.7}$$

$$u_i + UB \sum_{v_j \in V, j \neq i} x_{j,i,i} \leq UB - 1, \quad \forall i \in E' \tag{5.8}$$

$$\begin{aligned} u_i - u_j + (UB+1) \sum_k (x_{i,j,k} + x_{j,i,j} - y_j) + 1 \\ + (UB-1) \sum_k x_{j,i,k} \leq UB, \quad \forall i, j \in E', \end{aligned} \tag{5.9}$$

In the IP formulation, constraints (5.4) and (5.5) ensure every node is monitoring exactly one other node and is monitored by exactly one node. Constraint (5.6) ensures the two edges of the same node belong to the same cluster. Constraints (5.7) and (5.8) use u_i to take care of lower bound and upper bound of the size of clusters.

Constraint (5.9) takes care of the ordering of monitoring. It says if $(v_i, v_j) \in E'$, the difference between u_i and u_j is 1 except the case if v_j is the CH.

This formulation is valid only when $UB \geq LB \geq 2$, and there is a combinatorial solution of using clusters with varying size from LB to UB to cover all nodes that require monitoring. A simple invalid setting example is for setting $UB = 4$, $LB = 3$, with there are 5 nodes to be monitored. There is no possible solution for this setting.

This optimal IP formulation is very complex to solve. Since mTSP is an NP-hard problem and our extension greatly increases the complexity of the computation, so mTSP with no depot is an NP-hard problem as well. Given n nodes to be monitored, we will have n^3 selection variable x and n selection variable u, which means a total of at most $2^{n^3}UB^n$ possible combinations to search. We will compare the optimal solution with mTSP in Sec. 5.3.

5.2 Heuristic Solution for Clustering

Since finding the optimal MCP solution is very complex; a heuristic solution can help greatly for reducing the complexity. We divide the problem into two steps:

1. Find initial nodes as CHs for clusters, then add other nodes into these clusters one by one.

2. Decide the monitoring sequence in each cluster.

We design greedy algorithms for both steps and show simulation results in Sec. 5.3.

5.2.1 Greedy Solution for Identifying Cluster Heads

For cluster heads identification, the common idea is to use those nodes which have lower costs to other nodes to be the heads of potential clusters. The rationale is that a well-connected CH can find more close nodes to grow the cluster, which can possibly reduce the communication cost during monitoring. However, in IoT, devices from different locations may cooperate, it's likely to have nodes far away from one another for monitoring. Some of these nodes may have a high cost to all other nodes and are relatively less connected. Such nodes are not tended well by this greedy solution and may lead to a bad overall result.

We therefore decide to start finding CHs from those least connected nodes. We use their most well-connected neighbors as the cluster heads. Such cluster heads will be closer to both relatively isolated nodes and many easily connected nodes, reaching a balance to grow clusters.

For any target node, we define its k-hop neighbors to be the set of nodes which can reach the target node with k or less unit of communication cost. This concept is used in Algorithm 5.1 to find the least connected nodes.

In Algorithm 5.1, for all nodes that are not in any cluster yet, Line 5 finds the least connected node. Line 7 finds its most connected neighbor and tries to use it as the CH of a cluster. Lines 14-15 add all its open neighbors to the cluster. If the result cluster is still smaller than the lower bound, Lines 17-19 move nodes from existing clusters to the newly created cluster. After this, if the newly created cluster has enough members, Lines 20-25 accepts the new cluster. If a cluster with a proper size cannot be formed within k hops, Line 3 increases k and tries again. The overall complexity of the algorithm is at $O(n^4)$. With the help of a good data structure maintaining the nodes and sorting result, it could be further reduced. This reduces

Algorithm 5.1 Finding Least Connected Points(FLCP)

Input: Set of nodes $S = \{nodes\}$; lower bound LB and upper bound UB of cluster size

Output: Set of clusters C, CH_i is the head of cluster $c_i \in C$

1: Hop count $k \leftarrow 0$, $D \leftarrow S$
2: **while** $D \neq \emptyset$ **do**
3: $k \leftarrow k + 1$
4: Find k-hop neighbors in D for all nodes $n \in S$
5: **while** $D \neq \emptyset$ **do**
6: $lcn \leftarrow node \in D$ with the least neighbor
7: Get sp from $neighbor(lcn)$ with the most neighbor
8: **if** sp is already a CH and $|cluster(sp)| < UB$ **then**
9: Add lcn to $cluster(sp)$
10: **else if** sp is already a CH and $|cluster(sp)| \geq UB$ **then**
11: Go to Line 7 to try again
12: **else**
13: Create new cluster $cluster(sp)$ with sp as CH
14: **end if**
15: Fill in$cluster(sp)$ with nodes with least neighbors in $neighbor(sp) \cap D$
16: $nSet \leftarrow neighbor(sp) \cap (S - D)$
17: **while** $|cluster(sp)| < LB$ and $|anyothercluster| > LB$ **do**
18: move $nd \in neighbor(sp) \cap S - D$from original cluster to $cluster(sp)$
19: **end while**
20: **if** $|cluster(sp)| < LB$ **then**
21: Return nodes in $cluster(sp)$ to D or original clusters
22: Go to Line 7 to try again
23: **else**
24: Add $cluster(sp)$ to C
25: **end if**
26: **end while**
27: **end while**

the original optimal solution from NP-hard to polynomial complexity.

An example for finding the clustering solution using Algorithm 5.1 is shown in Fig. 5.2. For simplicity of showing, we assume the same monitoring frequency for all nodes. Each node is able to reach one node away horizontally or vertically, and one node away diagonally. Yellow nodes are the least connected nodes, and red nodes are the cluster headers selected for them. Black nodes are the other nodes added to clusters using Algorithm 5.1. Our heuristic algorithm happens to find the optimal result in this example.

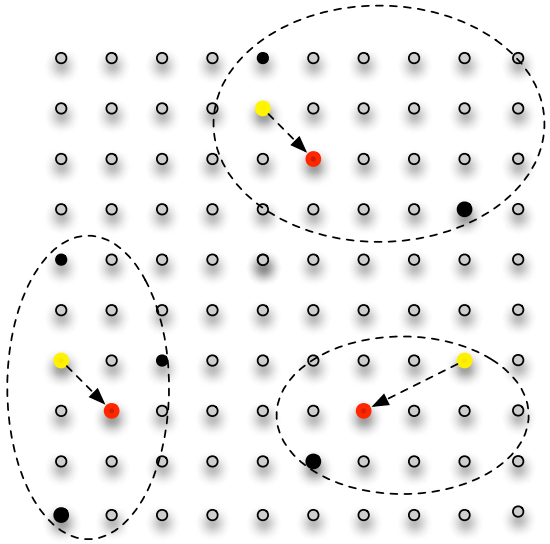


Figure 5.2: Example Deployment using Algorithm 5.1

5.2.2 Finding the monitoring sequence in each cluster

After deciding the monitoring clusters, we still need to find the monitoring sequence so that the total communication distance is minimized. This is the classical travelling salesman problem (TSP), a well-known NP-hard combinatory problem.

Optimal Solution

We now model TSP in its IP form following [64]. Just like in mTSP, we use binary selection variables:

$$x_{i,j} = \begin{cases} 1, \text{if}(v_i, v_j)\text{is selected} \\ 0, \text{otherwise} \end{cases} \tag{5.10}$$

and the auxiliary u, which is a non-negative integer variable expressing the order in the monitoring sequence in a cluster with the CH's u assigned to 1. Suppose there are n nodes in a cluster, the formulation to decide the routing is as follows.

$$\min \quad \sum_{i=0}^{n} \sum_{j=0, j\neq i}^{n} w_{i,j} * x_{i,j} \tag{5.11}$$

$$\text{s.t.} \quad \sum_{j, j\neq i} x_{i,j} = 1, i = 0, 1, ..., n \tag{5.12}$$

$$\sum_{i, i\neq j} x_{i,j} = 1, j = 0, 1, ..., n \tag{5.13}$$

$$u_i - u_j + n x_{i,j} \leq (n-1), 1 \leq i \neq j \leq n \tag{5.14}$$

Constraints (5.12) and (5.13) guarantee messages from only one other node arriving or departing from the node for monitoring. Constraint (5.14) ensures all nodes in a cluster is connected by only one route. There are at most $\lfloor n/LB \rfloor$ clusters to be considered. For each cluster of m nodes, the number of combinations in this problem is $2^{m^2} UB^m$. One special note is that if constraint (5.14) is removed, this can be used as a shortcut to optimally solve the monitor clustering problem with $LB = 2$ and $UB = n$.

Heuristic Solution

Heuristic solutions for TSP have been well-studied [15]. Among them, the nearest-neighbor (NN) algorithm is one of the least complex solutions. From a starting node, NN will try to add a nearest neighbor as the next node to be visited. We adapt the NN algorithm to our problem in order to achieve a balance between the computation time and the result quality.

In our node clustering scenario, since many nodes may have the same monitoring frequency and hop count, there may be many ties during comparison. In this case, the nearest neighbors which are farther from other nodes may have less opportunities and should be considered first. Based on this rationale, we give scores to nodes as follows:

$$score_i = \sum_k m^{-k} * (m - |k^{th} \text{ order neighbor of } i|) \tag{5.15}$$

In Eq. (5.15), m is the total number of nodes in the cluster, $|k^{th}$ order neighbor of $i|$ is the count of nodes who can reach the target node in k hops but not in $(k-1)$ hops. A higher score indicates a node has closer neighbors, but the number of such neighbors is smaller than that of other such nodes.

As per D. Johnson et al.[38], the nearest neighbor algorithm on average yields a path 25% longer than the shortest one. This result should also bound our algorithm.

The example of showing the monitoring sequence using Algorithm 5.2(LCNN) is shown in Fig. 5.3. For this very example, the result is the same as the optimal TSP result.

Algorithm 5.2 Least Connected Nearest Neighbor (LCNN)

Input: set of nodes S in a cluster; sp is the CH
Output: A list ML as the monitoring sequence

1: $k \leftarrow 1$
2: $ML \leftarrow [sp]$
3: $current \leftarrow sp$
4: remove sp from S
5: **while** $|S| \neq 0$ **do**
6: sort nodes in S by scores (Eq. (5.15)) from high to low
7: **for all** $nd \in S$ **do**
8: **if** nd is k-hop neighbor of $current$ **then**
9: $current \leftarrow nd$
10: remove nd from S
11: append nd to ML
12: $k \leftarrow 1$
13: Go to Line 5
14: **end if**
15: **end for**
16: $k \leftarrow k + 1$
17: **end while**

5.3 Simulation

For simulation, we compare among different combinations of algorithms. To further evaluate the performance of Algorithm 5.1 (FLCP), we take the idea from traditional WSN clustering method like LEACH [30], which tries to find the most powerful node as the cluster head directly, and based on it design another algorithm Finding Most Connected Points (FMCP), which follows the same logic as Algorithm 5.1 except for the part of finding the cluster heads. FMCP will find the most connected nodes at each range as the cluster heads. We will compare among combinations of algorithms: the mTSP without depot IP solver (optimal solution), FLCP cluster headers + mTSP [9], FMCP cluster headers+ mTSP, FLCP + TSP, FMCP+ TSP, FLCP +LCNN, FMCP +LCNN.

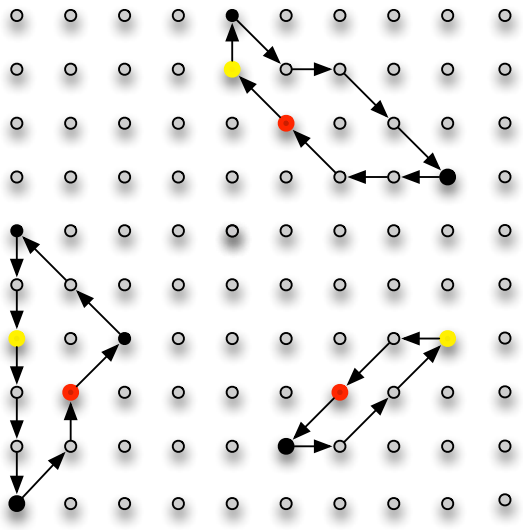
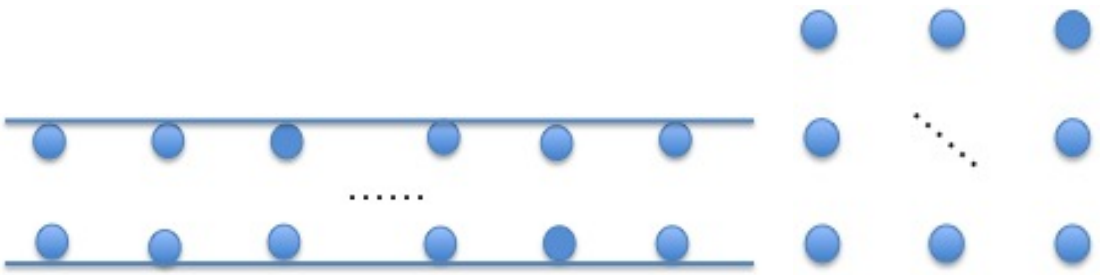


Figure 5.3: Example monitoring sequence using Algorithm 5.2 based on Fig. 5.2

5.3.1 Simulation Setup

We use two scenarios Fig. 5.4. The first scenario is a 300m long, 5m wide hall way with a total of 100 devices on both sides of the hallway. The distance between neighboring sensors is 6m. In the second scenario, there is a 100mx100m office with 100 devices deployed uniformly. The distance between neighboring sensors is 10m. For both scenarios, we assume all devices can communicate directly with their neighbors within 30m. These two scenarios are selected because they achieve similar results to the random device placement while having a lower randomness. An application that require monitoring may be deployed across all nodes. For each run, we randomly select several nodes out of the 100 nodes for monitoring. For simplicity, the monitoring frequencies are the same and the hop count is the only factor for the communication cost. Each measuring is the average of 20 runs. The main difference between the two scenarios is that the hallway requires more message relaying (hopping) than the office scenario. With the assumption of a unified system-wide monitoring frequency, we use hop counts as a simplified communication cost.



(a) Sensor Deployment in Hallway (b) in Office

Figure 5.4: Different Sensor Layout

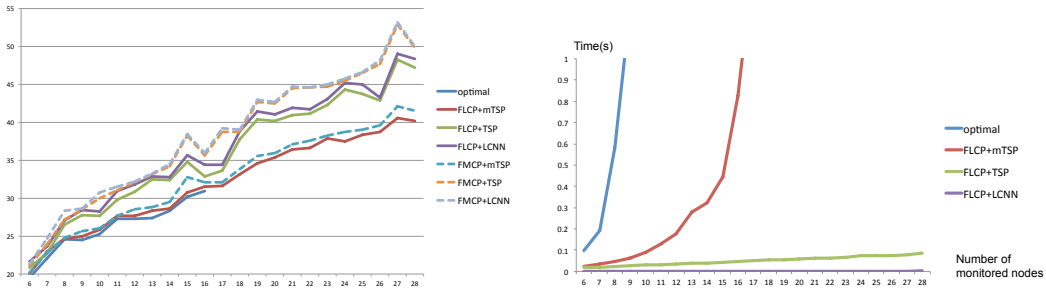
For both scenarios, we set nodes to be deployed in different densities, give different application sizes, and compare the average running times and hop counts from different algorithms. We set the cluster size to be from 3 to 7 which is a practical size. This will make the jobs for ordering within a cluster easier.

5.3.2 Impact from application size

In Fig. 5.5(a), we show the total monitoring message communication cost given different application sizes. For the optimal solution, we only show results up to 16 nodes because of the long computation time. In order to show the performance of heuristic algorithms in the first step of selecting clusters, we compare among the results of optimal, FLCP ones and FMCP ones. If heuristic algorithms are only used for selecting cluster heads in the first step, with the help of mTSP in the second step, both FLCP and FMCP achieve a near optimal result. If heuristic algorithms are also used for deciding the members in each cluster, when TSP or LCNN is used for the second step, we can see in both cases, FLCP is better than FMCP. Though their distances to optimal case increases when application size increases, both FLCP and FMCP achieve results close to optimal. In order to show the performance of heuristic algorithm LCNN in the second step, we compare the results to the optimal TSP ones. It shows LCNN can also achieve near optimal results in both FLCP and FMCP cases.

This may be due to the size of every cluster is small.

Fig. 5.5(b) shows the computation time of different algorithms. The optimal IP solution is using the branch and bound algorithm. The results of FMCP are nearly the same as its FLCP counterparts and are not shown. The IP solvers are implemented using GLPK and running using only one core of a Intel 2.3GHz i5 processor. The running times of the optimal solutions vary a lot depending on the effectiveness of branch and bound used by GLPK. But as a conclusion, it is impractical to rely on optimal solution for problems sizes larger than about 14 due to complexity.



(a) Communication counts for different application sizes

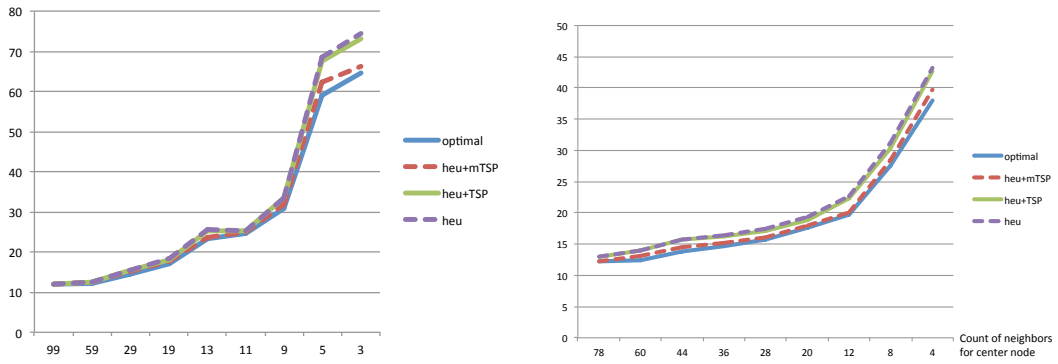
(b) Computation time of algorithms

Figure 5.5: Results of all algorithms.

5.3.3 Impact from Device Deployment Density

The device deployment density determines how many potential neighbors a device may have in the network topology. Deploying devices close to each other generally results in sensors having more potential neighbors for communication, thus more likely having less hops in clustering.

The simulation result for devices deployed under different densities is shown in Fig. 5.6. The communication range is fixed at 30 meters, so the number of reachable neighbors decreases as the density is reduced. We can see that heuristic algorithms can achieve



(a) Sensor Deployment in Hallway (b) Sensor Deployment in Office

Figure 5.6: Different Sensor Layout

near optimal numbers of hops when devices are densely deployed. The numbers of hops required for monitoring decrease when the distance among devices increases and the numbers of neighbors for each device decrease. In the worst case when every device is only able to reach its nearest neighbor device, heuristic algorithms show on average respectively 15% and 13.5% more hops than the optimal solution. We consider this to be quite acceptable.

Generally speaking, simulation results show that our heuristic algorithms can greatly reduce the complexity of finding a solution while the result communication cost is not far from the optimal solution. Device clustering for fault monitoring is just the first step for building fault-tolerant IoT. After a fault is detected, we need to recover from the fault by finding a replacement device to continue the IoT.

Chapter 6

Fault-Tolerant IoT System Implementation

6.1 System Architecture

With expanded mapper and the goal of fault-tolerance, the Wukong master from Fig. 2.1 can be refined as shown in Fig. 6.1. The master discovers the availability of IoT resources and their locations in a target environment through device discovery. The result is saved in the *Device Repository*. In order to organize the devices according to their geographical context, a location tree is set up based on discovered devices and their locations. Based on what resources are available on the devices, a service generator will create services in service repository and make them available for mapper.

The mapper, with the input of services and an application flow, has four jobs to do. Firstly, location policies and other policies, which define the concerns of FBP users about how components should be mapped, are set from the web GUI. They need to

be interpreted as parameters for the mapper. For the case of location policies, using the BNF defined in Sec. 4.2.3, a syntax tree needs to be setup for every location policy setting of a component. For convenience of the user, minor syntax errors, like missed prefix or delimiter, are also fixed during this process. Location policies will be checked during interpreting, if it is not consistent with the location tree, warnings will be issued. After interpreting is done, all devices in the device repository need to be filtered using policy settings. The smaller number of candidates can greatly reduce the overhead of ranking and binding. For location policy, this step happens twice The first time is when containment relationship is parsed, all services from other places will be filtered. The second time is when individual location policy functions are parsed, some functions will do a filter job. The third step of mapper is ranking and reranking, based on policies, scores are calculated for each remaining candidate of a component. For location policy, this happens when policy functions are being evaluated. Depending on different policies, filtering and ranking may happen alternately. The fourth step is binding. Usually the services with the highest scores are bound to devices. After binding, link settings, parameter configurations and generated Java byte codes are sent to devices.

The steps described above are shown in Steps 1 to 4(Discovery, Actual Service Identification, Mapping, Application Deployment) of Fig. 6.2. Because how virtual services are generated depends on which service a component is mapped to, it happens after application is already deployed and the whole system workflow can be described into two phases. The first phase generates a mapping result for the components in the application without backups, it consists of Steps 1$\sim$4. Then in the second phase, based on the initial mapping result, virtual services are generated. While more virtual services are generated, the mapper will find enough backup services for executing services and deploy them. While the first phase of process corresponds to a cold start application deployment, the second phase happens after an application has already

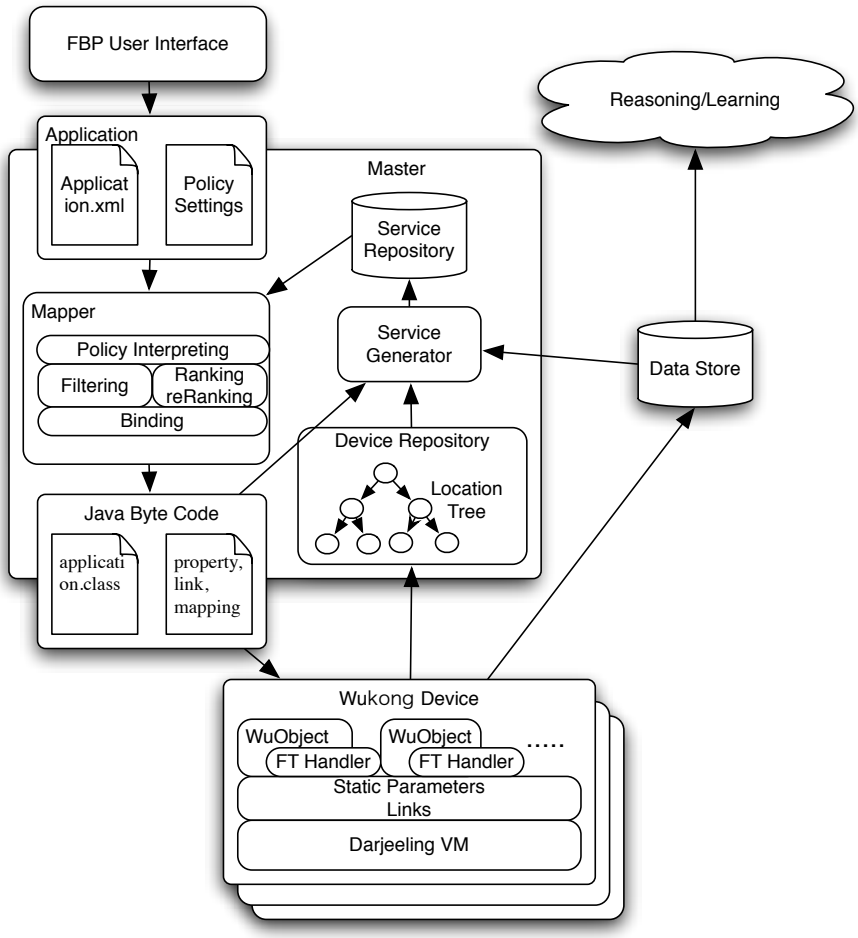


Figure 6.1: System Architecture

been deployed, and can be executed periodically in case better virtual services are generated.

The second phase is triggered if any component in the deployed application requires backups. Since data from sensors will be sent for further learning of environment context, a deployed application will always send data to a component different than the master called *data store* (Step 5). In order to handle the different clocks of devices, data are labeled with receiving time when data store receives it. If a component requires a backup in the application, the *service generator* will start to generate virtual services for the mapped service of this component (Step 6). It will start by subscrib-

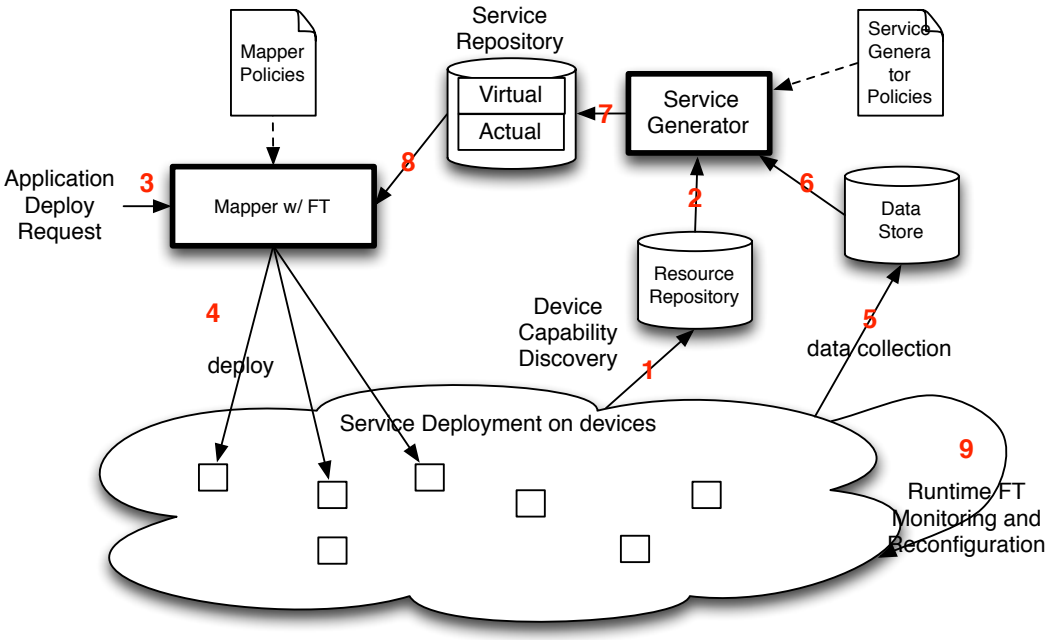


Figure 6.2: System Deployment Workflow

ing sensing data histories from all geographically close sensors from data store. It then applies regressions upon different combinations of subscribed data. This process, depending on the algorithms used, may work continuously or periodically as long as new data comes. The service generator will then learn and generate/update models of relationships among resources. In order to make use of a virtual service, the result regression model will be deployed on a device as a computation service. Theoretically, any device that has the reprogramming capability can hold this computation service. However, in order to reduce the number of devices used by a virtual service and to save communication cost, this computation service will be placed on the device which is already used by the virtual service and has the shortest topological distance to all other devices of the same virtual service. This device that holds the computation service is also the device that "holds" the virtual service, since it produces the output flow to other services. The virtual service information will be saved alongside actual

services in service repository (Step 7). The mapper will check the service repository periodically for updated service status. Once it found there are more available and better virtual services, it will start to trigger a mapping for backup services (Step 8). This mapping for component backups cares more about the fidelities of services and cost of fault-tolerance. Thus, it can be different than the Step 3 mapping which finds active services components. A virtual service will be really deployed on devices only if the virtual service gets selected as a backup by the mapper. Communication links will be reconfigured and computation service will be downloaded for the deployment. After deployment, there will be beacons passing around the devices which are mapped to fault-tolerant components. If there is anything wrong with any active or backup service, reconfiguration will be triggered to use backup services for the broken components (Step 9).

In general, Step 1 and 2 will provide the resources for actual services, while Step 5 and 6 will provide the data for generating virtual services. The service generator is in charge of generating both actual and virtual services. Then, the mapper, given the services generated by the service generator, will be able to map the application, or the application's backup services.

6.2 Fault Recovery Mechanism

One big part of runtime Step 9 is fault monitoring and recovery mechanism. With monitor loops introduced in Chapter 5, now we focus on introducing the fault recovery mechanism. In order to support a fast recovery, all mapped backup services are deployed into the network before fault happens. In order to keep track of them, one simple way is to save all the backup information on CHs of monitoring loop since they know about which active services are failing. However, this means the CH needs to

know every backup plan for services in its monitoring loop. It also means whenever backup plans are updated, the CH devices, even though they may hold no service required by new backup plans, need to be updated as well. These two points cause extra communication and computation overheads for CH, which already bears more burdens due to maintaining monitoring loop.

Our solution is to distribute the recovery job and let services maintain the backup plan information for themselves. As shown in Fig. 6.1, a special WuObject (service) called *FT Handler* is deployed with every WuObject that requires the fault tolerance support. This FT Handler is used to remember all backup services. It also has the privilege to use system messages to change links and replace the current active service with backups for the same component. In time of need, a FT Handler will be asked to redirect links from and to a faulty service to the next mapped backup service. At the same time, a CH of the monitoring loop is only responsible for contacting FT Handlers for replacement. For every service under its monitoring, the CH will ask its FT Handler for emergency contact information, only one of the fault handler should be remembered. This FT Handler, also called emphprimary FT Handler for the CH, is initially assigned to the next backup to take action if the one under CH is failing. Different CHs may have different primary FT Handlers for the same component. If all services of one component fall under one monitoring loop, the CH as well may have different primary FT Handlers for each one of them.

The process of fault handling is described in 4 steps in Fig. 6.3. The left part of the figure shows a linear application ABC, with B having 1 active (b1) and 2 backup services (b2, b3). Each of these B services comes with a FT Handler for B. The lower right of the graph shows the monitoring cluster responsible for monitoring b1, with the red node in the loop as the CH of the cluster. In Step 1, the node holding b1 becomes faulty. This causes service b1 to become faulty as well (Step 1'). This faulty

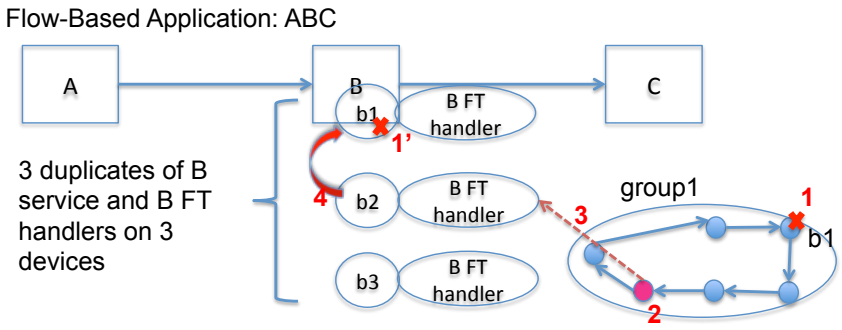


Figure 6.3: Fault Recovery Mechanism in 4 steps

device will be reported back to the CH (Step 2). By looking up b1's primary FT Handler, the CH decides to send the information to B FT Handler with b2 (Step 3). B FT Handler with b2 looks up in itself for backup plans, finding its b2 is the next one to take b1's job, it then will redirect all the links to and from b1 (A to B, and B to C in this case) to go through b2 (Step 4). Updating messages will be sent to the master and other B FT Handlers (b3 FT Handler in this case) before and after the change, and receivers will update their information accordingly. In the example, if b3's primary FT Handler was b1, the updated information will also trigger it to update its primary FT Handler with its own CH.

6.3 Link Changing Mechanism Design

FT Handlers have the ability to change sender and receiver of existing links. In order to maintain the robustness of the process, link changing mechanism should be designed carefully.

When a FT Handler receives a notice of fault service from a CH, there are three possible situations:

- The service is faulty. Backup service should be used to replace the faulty service.

All incoming and outgoing links should be rerouted through the backup service. This is the most common situation. The goal of fault recovery and link changing mechanism is primarily designed for this situation.

- The CH itself has problem communicating with the service due to problems in the medium. However, the FT Handler still can reach the supposed "faulty" service. This means the fault notice is a false alarm to the FT Handler. In this situation, because CH is unable to monitor the desired faulty service, FT Handler is still expected to handle the fault for CH.

- Both CH and FT Handler has problem reaching the "faulty" service, but this "faulty" service is acting normally in its application. This kind of error may happen when a certain medium is interrupted, or if different wireless protocols (e.g. Z-Wave, Zigbee, Wi-Fi) are mixed together for the "faulty" device. In the latter situation, the failure of a certain protocol may result in the failure of communication among the "faulty" service to only part of the other devices. This situation means even though the "faulty" service may still be alive and consistently send messages to receiving services, FT Handler should have a mechanism to block such messages from interrupting reconfigured application flow on the receiving services.

In order to deal with all three situations, locks are used for expressing transitional state during link change. In an application flow, locks are generated by FT Handlers and are passed to downstream mapped services to indicate upstream data is not reliable for now. The use of locks grants more global knowledge to local decision making services and actuators. Once a lock is seen, services will see certain upstream property data change as unreliable. In our current setting, actuators will not act based on unreliable data.

A lock is composed with 4 parts: lock ID, link ID, sending service ID, receiving service ID. The generator FT Handler's ID is used as the lock ID. This ID is used for deciding who is responsible for setting and deleting the lock. The IDs of sender, receiver and link are used for identifying which services and link the lock is about.

Locks are propagated through piggybacking with every property setting messages and link changing messages. There is a lock table in ROM (e.g. EEPROM) of every device. Every time a sending service wants to propagate properties or change links, the locks related to this sending service in the lock table are piggybacked with the sent messages. Whenever a service receives a message with locks, it will update the lock table in the device filling it up with the received locks, and clearing from the table existing locks which are not in the received message.

We show an example of lock propagation in Fig. 6.4 for the light control application in Fig. 3.2. The Light Sensor FT Handler (LSFT) is going to replace Light Sensor 1 with Light Sensor 2 in this graph. Thus, it needs to change the links on both the Light Sensor 1's side and the Threshold2's side (the red arrows in Fig. 6.4). The LSFT will send a change link command with piggybacked locks to those two services, and Light Sensor 1 and Threshold become the first services to have LSFT lock. The change of LSFT lock will be further propagated from Threshold2 to And service, and then from And service to Light Actuator. At this time, even though Presence Sensor and Threshold1 still don't have the LSFT lock and may send property update normally, the update will not result in the Light Actuator change. This ensures the safety of the Light Actuator during link changing process. Right after sending to Light Sensor 1 and Threshold 2, LSFT will send link change messages without LSFT lock to Light Sensor 2(The blue dotted arrow in Fig. 6.4). This will clear the LSFT lock in them. When the lock change gets propagated, the locks in And service and Light Actuator get cleared as well.

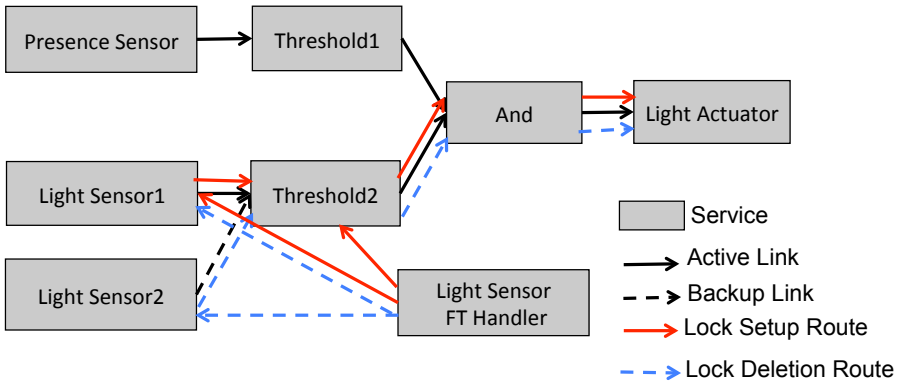


Figure 6.4: Locks are set by FT Handler link changing messages and propagated through piggybacking on application property setting messages.

With the introduction of locks, we now show how FT Handlers change links in Algorithm 6.1. In this algorithm, outgoing links are handled firstly in order to lock actuators at an earlier time. If there are multiple links connecting to the faulty service, every link will has its unique lock. For every link, there are 4 messages to be sent. There are two attempts to change link tables in the faulty device, the first is to stop the faulty services from sending messages, and the second is to clear the lock. However, if the faulty service is already dead, both message sendings may fail.

In order to show a simple example on how Algorithm 6.1 works, we create the deployment of a simpler version of light control application in Fig. 6.5(a). We show global service IDs, device local link tables and link IDs in this graph. Because FTLS1 and FTLS2 are actually duplicates of the same FT Handler, they have the same service ID.

When LS1 is found faulty, FTLS2, the primary FT Handler of LS1, will start sending messages to use LS2 instead LS1. The sequence of message sending is shown in Fig. 6.5(b). We also show in the graph how FTLS2 update its information with its duplicates (FTLS1 in the graph). It propagates the newest status before it starts the whole relinking process and after it is done with the relinking process.

Algorithm 6.1 Link Change Algorithm in FT Handler

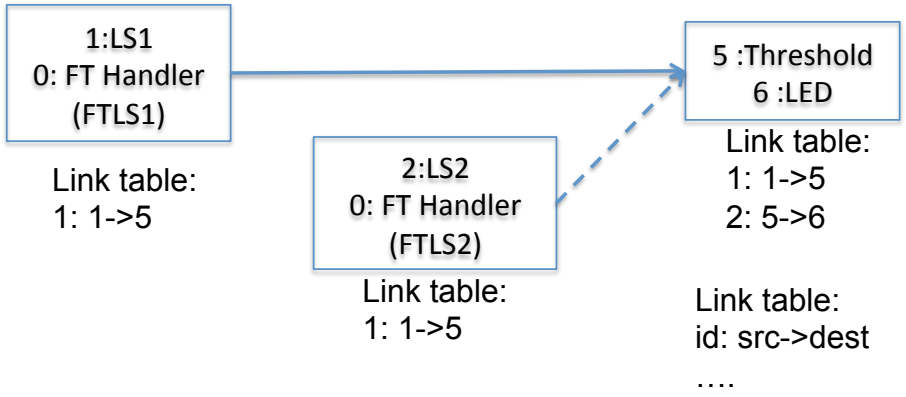
Input: Faulty service s_f, backup service s_b, incoming links L_i, outgoing links L_o

1: **for** $(s_f, dest) \in L_o$ **do**
2: ask s_f to change link to $(s_b, dest)$ and set lock for $(s_f, dest)$
3: ask $dest$ to change link to $(s_b, dest)$ and set lock for $(s_f, dest)$
4: **end for**
5: **for** $(src, s_f) \in L_i$ **do**
6: ask s_f to change link to (src, s_b) and set lock for (src, s_b)
7: ask s_b to change link to (src, s_b) and set lock for (src, s_b)
8: **end for**
9: **for** $(s_f, dest) \in L_o$ **do**
10: ask s_b to change link to $(s_b, dest)$ and clear lock for $(s_f, dest)$
11: ask s_f to change link to $(s_b, dest)$ and clear lock for $(s_f, dest)$
12: **end for**
13: **for** $(src, s_f) \in L_i$ **do**
14: ask src to change link to (src, s_b) and clear lock for (src, s_b)
15: ask s_f to change link to (src, s_b) and clear lock for (src, s_b)
16: **end for**

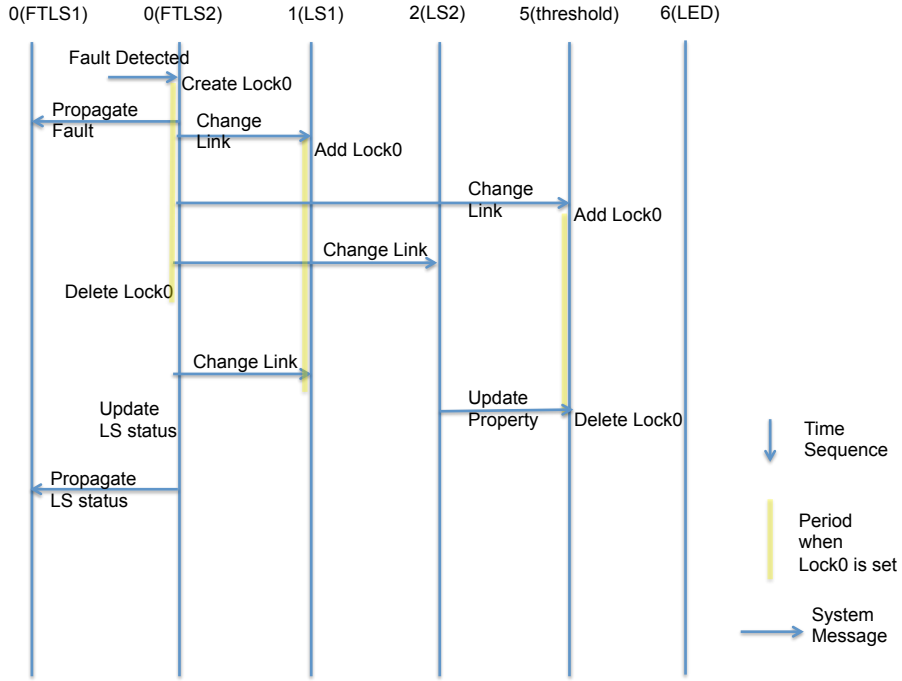
6.4 Fault Tolerance Policies

Policies can be specified by users for each component's fault tolerance requirement. Both service generator and mapper need fault tolerant related policies to define whether they need to have fault tolerant mechanism and how services are generated. For mapper, there is only one policy defined for each component: the number of backups. As explained in the system workflow, this policy defines whether fault-tolerant mechanism should be applied to the services which are mapped to the component. If so, how many backups should be deployed at the same time. Most of the times, the more backups needed for an application, the more devices should be put under monitoring, and thus, the more system workload there is for supporting fault tolerance.

For service generators, policies are used to define what a good virtual service looks like. There are two dimensions for defining the quality of a virtual service: fidelity



(a) An Example Simple Light Controller Application showing FT Handlers and Links. Devices are shown as rectangles in this graph.



(b) Time sequence graph showing how FT Handler switches link from LS1 to LS2

Figure 6.5: Mechanism of FT Handler

and cost. Because correlations between the regression result and the actual service data can be used to reflect the similarity between the two data streams, we use the correlations to express the fidelities of virtual services. For cost, because the service generator runs in backend servers, its cost does not have a significant impact on the runtime execution of fault-tolerance. Meanwhile, monitoring is run periodically and is a constant runtime burden for both devices and the wireless network. So we'd

like to use the monitor cost for FT related services. In order to avoid making the monitor cost rely too much on some specific monitoring methods or protocols, we use the number of devices in each service for monitor cost. Thus, we'll have fidelity and device count as the two properties of a service. A service with a higher fidelity and a lower device count is preferred for fault tolerance purposes.

With the above definition, we can see that actual services, with the highest 100% fidelity and lowest 1 device count, will always be the most preferred service available. For virtual services, certain policies need to be given for define what is a good virtual service. For service generator, we can set maximum device count and minimum fidelity as policies for each actual service. The maximum device count could be set to any integer above 1, and the minimum fidelity can be set to any number between 0 and 1. Any unqualified virtual services will be filtered out.

6.5 Deployment Example

In order to demonstrate how the fault-tolerant system works, we use virtual services generated from data in Chapter 3. We show how to deploy the light control application of Fig. 3.2 in our test environment (Fig. 3.3). Colors of different devices are used to represent how components from Fig. 3.2 are mapped.

Before mapping in phase 1, policies for the master need to be set. We set the FT policy to prepare 2 backups for the light sensor component and 1 backup for the presence component. The service generator is set to use the minimum fidelity of 0.85 and the maximum device count of 4 for both Kinect backups and light sensor backups. During the first phase mapping, $LS6$ and the Kinect are selected for light sensor and presence sensor components, all other components (computation, light actuator) are

deployed on $LA4$.

After 30 minutes of normal execution, virtual services are generated and updated by the service generator. The light sensor component uses RLS and its models are updated whenever a new sensing data arrives. The presence sensor component uses MARS and its models are updated periodically. Due to the minimum fidelity and the maximum device count constraints, 134 of the 256 possible RLS virtual light sensors are generated. For presence sensors, results after binarization are used for the fidelity calculation. Since there are a total of 22 PIR, MICRO and DIST sensors, any sensor with a very weak relevance to the target (fidelity < 0.2) are not used for virtual service considerations. This removes 11 sensors and greatly reduces the time for virtual service generation. At the end, 391 virtual services, with help from $DIST6$ or $MICRO6$, meet the requirement.

Then phase 2 mapping starts. We implement NSGA-ii using Jenes[2], a library for genetic algorithms in Java. The algorithm is executed for 100 generations with an initial population size of 200. We set the crossover probability to 0.8 and the mutation probability to 0.02. Fig. 6.6 shows the Pareto optimal mapping results from the NSGA-ii algorithm. The result shows that we can achieve 0.857 application fidelity from only 3 devices while the application fidelity threshold is 0.723. We can also see the increase in fidelity by using more devices is marginal, and both Least Devices First and Best Efficiency First from Sec.4.5 will choose this solution with 4 devices. In this solution, $DIST6$ alone is selected for Kinect backup with a fidelity of 0.95, $(LS5, LS7)$ with 0.93 fidelity and $(LS5, LS8)$ with 0.95 fidelity are selected for $LS6$ backup.

This solution results in the deployment shown in Fig. 6.7(a). In this figure, rectangles are devices and circles are resources. Devices are connected through Z-Wave protocol[3] or Wi-Fi. In total, 6 devices are used for the entire application, with

backups running on $D5$, $D6$, $D7$, $D8$, all of them close to area 2. A solid arrow line means an active connection between the two resources, while a dotted one means a potential connection between devices on both ends. This potential connection may be activated when a fault happens. Three dotted arrow lines show the connection from three backups: $DIST6$ for Kinect, $LS5+7$ and $LS5+8$ for $LS6$. $LS5+7$ and $LS5+8$ run the computing resources and produce outputs for virtual service $(LS5, LS7)$ and $(LS5, LS8)$ respectively. The arrows within a virtual service are solid because we try to use the results of virtual services for some data fault detection. These arrows would be dotted if it is just for backup deployment.

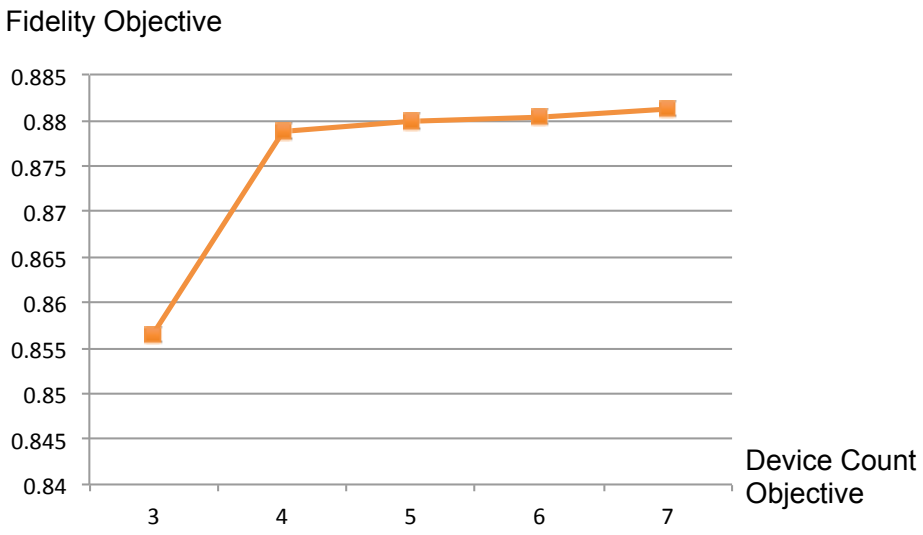
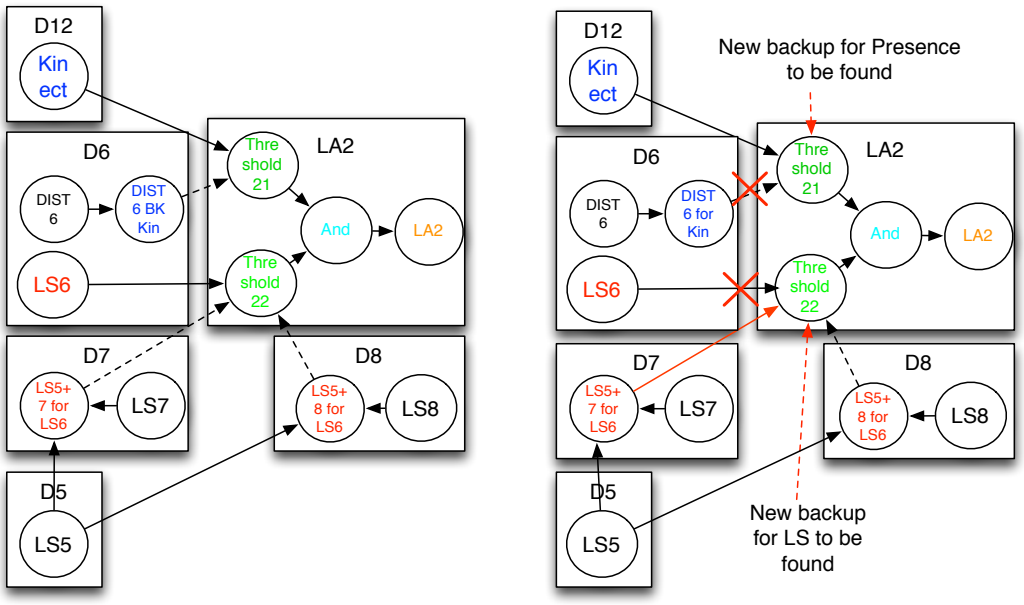


Figure 6.6: Pareto optimal results maximizing overall fidelity and minimizing device counts

After these virtual services are deployed, if a fault in $D6$ is detected, the fault recovery then produces a configuration as shown in Fig. 6.7(b). For the $LS6$ failure, the link from $LS6$ to $Threshold22$ is disabled, and the link from $LS5+7$ to $Threshold22$ is activated. For the $DIST6$ failure, since it is just a backup service, no runtime action is needed. However, this lack of backup service may trigger a remapping to find new

backups. Overall, only two links need to be changed instead of adding 2 extra devices. Failures of $D12$, $D7$ and $D8$ can be handled similarly.

To check the effectiveness of the application using backups, we examine those virtual sensor readings for an additional 10 minutes after the 30 minutes execution and training. We set the threshold for light value in the application to be 89, which is the medium value of $LS6$ data, to make sure the threshold is not too high or low for the data. The result shows a virtual light sensor can produce the same threshold result as the actual one for 92.57% of the time; virtual presence sensor can get the same result as Kinect for 98.82% of the time. Overall, the application is able to make a correct judgment for 93.92% of the time. The false positive rate and false negative rate are 5.64% and 6.56% respectively. All these results show that virtual services, under the goal of fidelity, can indeed perform very well in backing up actual sensors for fault tolerance.



(a) Fault-Tolerant Example Application Deployment after Mapping

(b) When D6 fails

Figure 6.7: Deployment Example

Chapter 7

Conclusions and Future Work

Fault tolerance is an important problem for IoT. According to IETF statement of concepts in IoT, the purpose of fault-tolerance in IoT is to "maintain robust and trust worthy dynamic ubiquitous networking requires redundancy in several levels and ability to automatically adapt to changed conditions depending on the required quality of service"[44]. In this dissertation, in order to better adapt to changing environments and build up trustworthy redundancy, we have designed the IoT middleware to facilitate the cooperation of different devices to achieve this cross-modality fault tolerance. When a fault happens to a device, the middleware can identify a system reconfiguration using devices of other modalities to cover the fault.

We provide redundancy for fault tolerance purpose from a service-oriented point of view, through three stages of services: service discovery, service mapping, service execution.

- For service discovery, we propose an innovative approach that uses IoT devices of different modalities as fault tolerant backups for each other. Although every sensor is designed for collecting data in a certain modality (such as light, tem-

perature or pressure), we find that some real world activities can be detected by sensors at different locations and of different types effectively. We use both linear and non-linear progression methods to integrate readings from different sensing devices to form virtual services. We also find RLS to be more suitable for virtual services composed of numerical sensors of the same modality, while MARS, with a cost of complexity, is good for more general cases. These virtual services provide backup services without deployment of duplicate backup sensors.

- For service mapping, we separate it into two steps: phase 1 pre-runtime mapping and phase 2 run-time fault-tolerant mapping. For pre-runtime mapping, we model it into a quadratic integer programming problem. Location policies are used to specify user preference during this mapping, and to limit the size of the QLP problem. For phase 2 mapping, with abundant provision of virtual services, we model it into a multiobjective optimization problem and use a multiobjective genetic algorithm to solve it. With more sensor data from the network, virtual services are updated, and phase 2 mapping is triggered periodically in order to adapt to changed environment.

- For service execution, we set up hierarchical monitoring for monitoring service status. In order to save system monitoring cost, a more decentralized loop structure is used as compared to the centralized star monitoring topology. We model the new monitoring clustering problem as a multiple traveling salesman without depot problem. This extends the traditional multiple travelling salesman problem (mTSP), which has fixed depots, to a more general form. We create the integer programming formulation of the mTSP with no depot problem. Because of the extremely high complexity of this problem, we present heuristic algorithms which divide the problem into two steps: locating cluster heads, and

creating fault-tolerant monitoring loop from each cluster head. Through simulation, we show the heuristic algorithms can greatly reduce the complexity of finding solution while the result communication cost is not far from the optimal solution.

We also show how services can be replaced through FT Handlers in the system during run-time. In order to fast and automatically adapt to device faults, backup services are deployed and virtually linked beforehand. When a fault happens, only these virtual links need to be triggered to activate backup services.

Our study shows the feasibility and efficiency of utilizing the heterogeneity of IoT devices for fault tolerance purpose. Yet it is only a starting point for the effort, by further mining and utilization of context information in IoT, the existing work can be expanded from the following directions:

- Inclusion of more environmental context information and the concept of scenarios to make virtual services more adaptive. IoT environments may be classified into different scenarios expressed as different context parameters. Different scenarios may require different virtual services. Using the microphone presence virtual service as an example, it only works when the room environment has a small number of people and is relatively quiet. When rooms are noisy, new virtual services need to be generated and the microphone virtual services may not work. If the quiet and noisy situation alternates frequently, the cost of retraining the model would be costly and time consuming. Inclusion of more environmental context information, like the count of people in the room, or information from body area network(BAN), and using the information as indicators of different scenarios, may help avoid unnecessary re-training of models.

- Detailed modeling for different modalities of sensors to make mapping smarter. Currently, during mapping and remapping, all location and fault tolerance policies are set by human installers who know about the characteristics of different sensors. More details about sensor capabilities like sensing range, sensing angle and whether line of sight can also be set by installers. There are already works like semantic sensing network ontology[17] providing descriptions of such sensor capabilities. If we incorporate such descriptions into our model, an extra layer of virtualization could be added. With the help of the extra layer, a human installer only needs to describe about his goal for a component (e.g. which location a sensor need to take care of), and location policies can be set automatically according to the goal. Search of resources for virtual sensors can also be bounded by this same goal.

- Inclusion of time dimension to the location model and location policies to make the mapping smarter. Currently, we don't handle the problem of moving objects. We assume moving devices can locate themselves through methods like triangulation, and locations of moving objects are updated by repeatedly discovery of devices. This method is not practical for fast moving ones. By generation of possible traveling routes in our current location model and inclusion of time dimension into it, location tracking and estimation can be achieved through filters[59][24]. Location policies can be upgraded to include time dimensions as well. Together with detailed modeling of different sensor modalities, the mapper can be pushed to a higher virtualization level: Instead of current application of "Turn on the office light if office presence sensor detect something", application description like "Turn on a corresponding light if someone will arrive at a certain location" can be supported.

With the popularity of the Internet of Things, more sensors of different modalities will

be deployed into our environment, and abundant sensing data can be acquired. We believe the aggregating data from different modalities of sensors will not only benefit fault tolerance in IoT, but also has potential to influence research on the sensor device placement, scenario reasoning and other aspects.

Bibliography

[1] Internet of things - architecture. `http://www.iot-a.eu/public/public-documents/documents-1/1/1/D2.5/at_download/file`. Accessed: 2015-11-17.

[2] Jenes - genetic algorithms for java. `http://sourceforge.net/projects/jenes`. Accessed: 2015-10-30.

[3] Understanding z-wave networks, nodes and devices. `http://www.vesternet.com/resources/technology-indepth/understanding-z-wave-networks`. Accessed: 2015-06-30.

[4] I. Afyouni, R. Cyril, and C. Christophe. Spatial models for context-aware indoor navigation systems: A survey. *Journal of Spatial Information Science*, 1(4):85–123, 2012.

[5] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393 – 422, 2002.

[6] M. Y. Ameer Ahmed Abbasi. A survey on clustering algorithms for wireless sensor networks. *Computer Communications*, 30(2-3):28262841, 2007.

[7] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom. Introducing takatuka: A java virtualmachine for motes. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 399–400, New York, NY, USA, 2008. ACM.

[8] C. Becker and F. Dürr. On location models for ubiquitous computing. *Personal and Ubiquitous Computing*, 9(1):20–31, 2005.

[9] T. Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209 – 219, 2006.

[10] M. Bhatt, F. Dylla, and J. Hois. Spatio-terminological inference for the design of ambient environments. In *Spatial Information Theory*, pages 371–391. Springer, 2009.

[11] O. Boyinbode, H. Le, and M. Takizawa. A survey on clustering algorithms for wireless sensor networks. *International Journal of Space-Based and Situated Computing*, 1(2-3):130–136, 2011.

[12] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM.

[13] S. Brüning, S. Weissleder, and M. Malek. A fault taxonomy for service-oriented architecture. 2007.

[14] M. Chang and P. Bonnet. Meeting ecologists' requirements with adaptive data acquisition. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 141–154, New York, NY, USA, 2010. ACM.

[15] C. Chauhan, R. Gupta, and K. Pathak. Survey of methods of solving tsp along with its implementation using dynamic programming approach. *International Journal of Computer Applications*, 52(4):12–19, August 2012.

[16] H. Choset and J. Burdick. Sensor-based exploration: The hierarchical generalized voronoi graph. *The International Journal of Robotics Research*, 19(2):96–125, 2000.

[17] M. Compton, P. Barnaghi, L. Bermudez, R. GarcíA-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, et al. The ssn ontology of the w3c semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012.

[18] D. Crisan and A. Doucet. A survey of convergence results on particle filtering methods for practitioners. *Signal Processing, IEEE Transactions on*, 50(3):736–746, Mar 2002.

[19] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler. smap: a simple measurement and actuation profile for physical information. *in: Proc. 8th ACM Conf. on Embedded Networked Sensor Systems*, 2010.

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[21] D. Demyen and M. Buro. Efficient triangulation-based pathfinding. In *Aaai*, volume 6, pages 942–947, 2006.

[22] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

[23] C.-L. Fok, G. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 653–662, June 2005.

[24] D. Fox, J. Hightower, L. Liao, D. Schulz, and G. Borriello. Bayesian filtering for location estimation. *IEEE pervasive computing*, (3):24–33, 2003.

[25] C. Frank and K. Römer. Algorithms for generic role assignment in wireless sensor networks. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 230–242. ACM, 2005.

[26] J. H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):pp. 1–67, 1991.

[27] T. N. Gia, A.-M. Rahmani, T. Westerlund, P. Liljeberg, and H. Tenhunen. Fault tolerant and scalable iot-based architecture for health monitoring. In *Sensors Applications Symposium (SAS), 2015 IEEE*, pages 1–6, April 2015.

[28] G. Gupta and M. Younis. Fault-tolerant clustering of wireless sensor networks. In *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE*, volume 3, pages 1579–1584 vol.3, March 2003.

[29] J. Heidemann and R. Govindan. Embedded sensor networks. In *Handbook of Networked and Embedded Control Systems*, pages 721–738. Springer, 2005.

[30] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *System sciences, 2000. Proceedings of the 33rd annual Hawaii international conference on*. IEEE, 2000.

[31] H. Hu and D.-L. Lee. Semantic location modeling for location navigation in mobile environment. In *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, pages 52–61. IEEE, 2004.

[32] Z. Huang, K.-J. Lin, C. Li, and S. Zhou. Communication energy aware sensor selection in iot systems. In *Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing(CPSCom), IEEE*, pages 235–242, Sept 2014.

[33] D. Hughes, K. Thoelen, J. Maerien, N. Matthys, J. Del Cid, W. Horre, C. Huygens, S. Michiels, and W. Joosen. Looci: The loosely-coupled component infrastructure. In *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, pages 236–243, Aug 2012.

[34] S. Ilarri, E. Mena, and A. Illarramendi. Location-dependent query processing: Where we are and where we are heading. *ACM Comput. Surv.*, 42(3):12:1–12:73, Mar. 2010.

[35] S. S. Intille, K. Larson, E. M. Tapia, J. S. Beaudin, P. Kaushik, J. Nawyn, and R. Rockinson. Using a live-in laboratory for ubiquitous computing research. In *Pervasive Computing*, pages 349–365. Springer, 2006.

[36] C. S. Jensen, H. Lu, and B. Yang. Graph model based indoor tracking. In *Mobile Data Management: Systems, Services and Middleware, 2009. MDM '09. Tenth International Conference on*, pages 122–131, May 2009.

[37] C. Jiang and P. Steenkiste. A hybrid location model with a computable location identifier for ubiquitous computing. In *Proceedings of the 4th International Conference on Ubiquitous Computing*, UbiComp '02, pages 246–263, London, UK, UK, 2002. Springer-Verlag.

[38] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. 1995.

[39] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering*, 82(1):35–45, 1960.

[40] A. Kansal, S. Nath, J. Liu, and F. Zhao. Senseweb: An infrastructure for shared sensing. *IEEE Multimedia*, 14(4):8–13, 2007.

[41] B. Khaleghi, A. Khamis, F. O. Karray, and S. N. Razavi. Multisensor data fusion: A review of the state-of-the-art. *Information Fusion*, 14(1):28 – 44, 2013.

[42] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentell. Fault tolerance techniques for wireless ad hoc sensor networks. In *Sensors, 2002. Proceedings of IEEE*, volume 2, pages 1491–1496 vol.2, 2002.

[43] F. Kuhn, T. Moscibroda, and R. Wattenhofer. Fault-tolerant clustering in ad hoc and sensor networks. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 68–68. IEEE, 2006.

[44] G. M. Lee, J. Park, N. Kong, and N. Crespi. The internet of things - concept and problem statement. `https://tools.ietf.org/html/draft-lee-iot-problem-statement-00`. Accessed: 2015-10-30.

[45] P. Levis and D. Culler. MatÉ: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 85–95, New York, NY, USA, 2002. ACM.

[46] H. Liu, A. Nayak, and I. Stojmenović. Fault-tolerant algorithms/protocols in wireless sensor networks. In *Guide to Wireless Sensor Networks*, pages 261–291. Springer, 2009.

[47] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *In MobiSys*, pages 149–162. ACM Press, 2005.

[48] J. Na, K.-J. Lin, Z. Huang, and S. Zhou. An evolutionary game approach on iot service selection for balancing device energy consumption. In *e-Business Engineering (ICEBE), 2015 IEEE 12th International Conference on*, Oct 2015.

[49] W. Nie, S. Zhou, and K.-J. Lin. Real-time service process scheduling with intermediate deadline overrun management. In *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, pages 1–8, Dec 2012.

[50] W. Nie, S. Zhou, K.-J. Lin, and S. D. Kim. An on-line capacity-based admission control for real-time service processes. *Computers, IEEE Transactions on*, 63(9):2134–2145, Sept 2014.

[51] R. Olfati-Saber. Distributed kalman filtering for sensor networks. In *Decision and Control, 2007 46th IEEE Conference on*, pages 5492–5498. IEEE, 2007.

[52] E. Ould-Ahmed-Vall, B. H. Ferri, and G. F. Riley. Distributed fault-tolerance for event detection using heterogeneous wireless sensor networks. *IEEE Transactions on Mobile Computing*, 11(12):1994–2007, 2012.

[53] E. Oyman and C. Ersoy. Multiple sink network design problem in large scale wireless sensor networks. *Communications, 2004 IEEE International Conference on*, pages 3663 – 3667 Vol.6, 2004.

[54] N. Patwari, A. O. Hero III, M. Perkins, N. S. Correal, and R. J. O'dea. Relative location estimation in wireless sensor networks. *Signal Processing, IEEE Transactions on*, 51(8):2137–2148, 2003.

[55] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services: Design and implementation of interoperable and evolvable sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 253–266, New York, NY, USA, 2008. ACM.

[56] N. Reijers, K.-J. Lin, Y.-C. Wang, C.-S. Shih, and J. Y. Hsu. Design of an intelligent middleware for flexible sensor configuration in M2M systems. In *SENSORNETS*, pages 41–46, 2013.

[57] I. Satoh. A location model for smart environments. *Pervasive and Mobile Computing*, 3(2):158–179, 2007.

[58] A. B. Sharma, L. Golubchik, and R. Govindan. Sensor faults: Detection methods and prevalence in real-world datasets. *ACM Transactions on Sensor Networks*, 6(3):1864–1869, 2010.

[59] D. Simon. *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons, 2006.

[60] P. Su, C.-S. Shih, J.-J. Hsu, K.-J. Lin, and Y.-C. Wang. Decentralized fault tolerance mechanism for intelligent iot/m2m middleware. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 45–50, March 2014.

[61] R. Sugihara and R. K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):8:1–8:29, Apr. 2008.

[62] J. Wang, Y. Yonamine, E. Kodama, and T. Takata. A distributed approach to constructing k-hop connected dominating set in ad hoc networks. *2013 International Conference on Parallel and Distributed Systems*, 0:357–364, 2013.

[63] X. Wang, J. Wang, Z. Zheng, Y. Xu, and M. Yang. Service composition in service-oriented wireless sensor networks with persistent queries. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–5. IEEE, 2009.

[64] R. T. Wong. Combinatorial optimization: Algorithms and complexity (christos h. papadimitriou and kenneth steiglitz). *SIAM Review*, (3):424–425, 1983.

[65] J. Wu and H. Li. On calculating connected dominating set for efficient routing in ad hoc wireless networks. In *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, DIALM '99, pages 7–14, New York, NY, USA, 1999. ACM.

[66] S.-Y. Yu, C.-S. Shih, J.-J. Hsu, Z. Huang, and K.-J. Lin. Qos oriented sensor selection in iot system. In *Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing(CPSCom), IEEE*, pages 201–206, Sept 2014.

[67] T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)*, 1(1):6, 2007.

[68] S. Zhou and K.-J. Lin. A flexible service reservation scheme for real-time soa. In *e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on*, pages 215–222, Oct 2011.

[69] S. Zhou and K.-J. Lin. Real-time service process admission control with schedule reorganization. *Service Oriented Computing and Applications*, 7(1):3–14, 2013.

[70] S. Zhou, K.-J. Lin, J. Na, C.-C. Chuang, and C.-S. Shih. Supporting service adaptation in fault tolerant internet of things. In *Service-Oriented Computing and Applications (SOCA), 2015 8th IEEE International Conference on*, 2015.

[71] S. Zhou, K.-J. Lin, and C.-S. Shih. Device clustering for fault monitoring in internet of things systems. In *IEEE World Forum on Internet of Things(WF IOT'15)*, 2015.

[72] E. Zio and R. Bazzo. A comparison of methods for selecting preferred solutions in multiobjective decision making. In C. Kahraman, editor, *Computational Intelligence Systems in Industrial Engineering: With Recent Theory and Applications*. Atlantis Publishing Corporation, 2014.