Supporting Mobile Distributed Services

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Ahmed Abdel Moamen

Permission to Use

Abstract

With sensors becoming increasingly ubiquitous, there is a tremendous potential for services which can take advantage of the data collected by these sensors, from the important – such as detecting medical emergencies and imminent natural disasters – to the mundane – such as waiting times experienced by diners at restaurants. This information can then be used to offer useful services. For example, a busy professional could find a restaurant to go to for a quick lunch based on information available from smartphones of people already there having lunch, waiting to be seated, or even heading there; a government could conduct a census in real-time, or "sense" public opinion. I refer to such services as *mobile distributed services*.

The barriers to offering mobile distributed services continue to be prohibitive for most: not only must these services be implemented, but they would also inevitably compete for resources on people's devices. This is in part because such services are poorly understood, and consequently, there is limited language support for programming them.

In this thesis, I address practical challenges related to three important problems in mobile distributed services. In addition, I present my efforts towards a formal model for representing mobile distributed services.

First, I address the challenge of enhancing the programmability of mobile distributed services. This thesis presents a set of core mechanisms underlying mobile distributed services. I interpret and implement these mechanisms for the domain of crowd-sourced services. A distributed runtime middleware, CSSWare, has been developed to simplify the burden of initiating and managing crowd-sourced services. CSSWare provides a set of domain-specific programming constructs for launching a new service. Service designers may launch novel services over CSSWare by simply plugging in small pieces of service specific code. This ease of programming promises to democratize the building of such services.

Second, I address the challenge of efficiently supporting the sensing needs of mobile distributed services, and more generally sensor-based applications. I developed ShareSens, an approach to opportunistically merge sensing requirements of independent applications. When multiple applications make sensing requests, instead of serving each request independently, ShareSens opportunistically merges the requests, achieving significant power and energy savings. Custom filters are then used to extract the data required by each application.

Third, I address the problem of programming the sensing requirements of mobile distributed services. In particular, ModeSens is presented to allow multi-modal sensing requirements of a service to be programmed separately from its function. Programmers can specify the modes in which a service can be, the sensing needs of each mode, and the sensed events which trigger mode transition. ModeSens then monitors for mode transition events, and dynamically adjusts the sensing frequencies to match the current mode's requirements. Separating the mode change logic from an application's functional logic leads to more modular code.

In addition, I present MobDisS (Mobile Distributed Services), an early model for representing mobile distributed services, allowing them to be carefully studied. Services can be built by composing simpler services. I present the syntax and operational semantics of MobDisS.

Although this work can be evaluated along multiple dimensions, my primary goal is to enhance programmability of mobile distributed services. This is illustrated by providing the actual code required for creating two realistic services using CSSWare. Each service demonstrates different facets of the middleware, ranging from the use of different sensors to the use of different facilities provided by CSSWare. Furthermore, experimental results are presented to demonstrate scalability, performance and data-contributor side energy efficiency of CSSWare and ShareSens. Finally, a set of experimental evaluation is carried out to measure the performance and energy costs of using ModeSens.

Acknowledgements

First and foremost I want to thank my advisor, Prof. Nadeem Jamali. I appreciate his contributions of time, ideas, and funding to make my PhD experience productive and stimulating. The joy and enthusiasm he has for his research were contagious and motivational for me, even during difficult times throughout the pursuit of my PhD. I am also thankful for the excellent example he has provided as a successful professor, which has encouraged me to pursue a career in academia.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Derek Eager, Prof. Carlos A. Varela, and Prof. Aryan S. Mehr, not only for their insightful comments and encouragement, but also for the hard questions which motivated me to strengthen my research from various perspectives.

Last but not the least, I would like to thank my wife Omnya and my daughter Mariam for all their love and encouragement. Special thanks to my parents who raised me with a love of Science and supported me in all my pursuits.

This thesis is dedicated to my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the growing ubiquity of personal computational devices, such as smartphones and wearable devices, has come the ubiquity of sensors on these devices, as well as the potential for triggering actions virtually anywhere. This opens up an opportunity to offer a variety of services across a wide variety of domains including health-care, entertainment, environmental monitoring and transportation. These services rely on the state of the context in which devices are located, such as a person or a group of people carrying the devices, their geographical location, etc. I broadly refer to these as *mobile distributed services*.

Consider a restaurant recommendation service which samples data collected about experiences of clients at a number of restaurants in a neighborhood and ranks them according to the service experienced by these clients. The source of the data could be sensor feeds on clients' smartphones, used to guess whether they are waiting, seated, enjoying their meals, paying or leaving. Consider other services, such as one for recommending hospital emergency services to people. Such services have a pattern of communication in which contextual data offered by a number of contributors becomes the basis for a service. Another class of applications with a similar pattern of communication is social media applications, such as Twitter, where crowds contribute to collective messages by contributing short free-form messages, which are then available to others in a digestible form.

However, offering such services continue to be challenging because of the lack of precise understanding, specification, and analysis of such services. Furthermore, they would also inevitably compete for resources on people's devices. Although there is a growing body of work devoted to this area (see [1–5]), the focus has been on narrow application areas or specific concerns, making it difficult to utilize them for a wider class of services.

1.1 Approach

This thesis presents both practical and foundational contributions to address some key challenges in supporting mobile distributed systems. The practical side of this work builds mechanisms to support programming of mobile and distributed sensing-based systems, prototypes them, and experimentally evaluates them. Particularly, this thesis addresses three different challenges in supporting such systems: (1) enhancement of the programmability of crowd-sourced services (Section 1.1.1), (2) energy-efficient mobile sensing (Section 1.1.2), and (3) multi-modal sensing (Section 1.1.3). The foundational work is aimed at understanding principles underlying such services and informing design and implementation decision about them. I present an early model, MobDisS (Mobile Distributed Services), for representing mobile distributed services, allowing them to be rigorously studied (Section 1.1.4).

I use the Actor model [6] for concurrency as a reference model for my research. Actors are autonomous concurrently executing primitive agents (i.e., active objects) which communicate using asynchronous messages.

1.1.1 CSSWare: An Actor-Based Middleware for Crowd-Sourced Services

The programming required for offering a new mobile distributed service can be significant if done from scratch. However, there is an opportunity created by the similarity in the patterns of communication required for such services, especially for crowd-sourced services where contextual data offered by a number of contributors becomes the basis for the service. This pattern of communication was originally defined in [7] as multi-origin communication. This is the type of communication where a group of senders contributes to a group message, without any of them necessarily *taking the lead*. Contrast this with a single-origin (multi-sender) communication [7], which is initiated by a single party which solicits interest from other parties to join together in sending a particular message. An example of the latter would be a workplace petition drafted by an individual and presented to others to sign. In multi-origin (implicitly also multi-sender) communication, there is no single party which needs to take the lead. In other words, multiple parties may autonomously launch messages which are then aggregated in order to create a group message.

It turns out that unlike single-origin multi-sender messages, multi-origin messages require a setup in advance. Consider a public square where a number of citizens spontaneously begin to gather to party or protest. In this context, the physical space of the square serves as part of a setup which allows mutual observation, an opportunity to join in or leave, to endorse, reject or refine the collective message or experience over time. The closest electronic equivalent of such a physical space would be social media services such as Twitter, which allow people to observe others' tweets in an aggregate form (which is quite natural in physical space, but requires filtering and counting mechanisms in electronic space), endorse them by adopting hashtags, improve upon the message, and so on. In general, for a crowd (or mass) – conceived communication to happen, there is a need for a mechanism to be in place to coordinate the generation of the group message by soliciting (or more generally instigating) individual messages, receiving them, and then aggregating them into a group message. The solicitation lays out the rules to be followed for selection of the potential senders, receiving their messages and aggregating them. For example, imagine a multi-stage communication with the first solicitation being to invite nominations for topics to have the message on, followed by a vote to select the topic, followed by a solicitation of messages, followed by a final vote to agree on an aggregate message. The communication could be one-time, periodic, or continual. There may or may not be a time-out for responding to the solicitation. All these aspects would be laid out in the original solicitation.

Multi-origin coordination mechanisms can be provided on a platform over which such class of services could be implemented relatively easily. I interpret and implement these mechanisms for the domain of crowd-sourced services. This thesis presents my efforts in realizing that potential by implementing CSSWare, a middleware for crowd-sourced services. CSSWare provides a set of domain-specific mechanisms to support initiating and managing crowd-sourced services. Having CSSWare as a platform, all that a service designer needs to do to launch a new service is to identify a constituency of potential contributors, and to provide a few lines of service-specific code for specifying the nature of contributions and for aggregating them when they arrive. I present source code for two realistic crowd-sourced services to illustrate the ease with which new services can be specified and launched. Finally, I present a set of experimental results demonstrating scalability, performance and data-contributor side energy efficiency of the approach.

1.1.2 ShareSens: An Approach for Energy-Efficient Mobile Sensing

Smartphones and a growing number of wearables have a variety of different sensors built into them. These sensors can be divided into three main categories: motion sensors (e.g., accelerometer, gravity, gyroscope, GPS, and rotation vector), position sensors (e.g., orientation, geomagnetic field, and proximity), and environment sensors (e.g., light, pressure, humidity, and temperature). Sensors typically have different configuration methods requiring specification of various parameters or settings such as sampling rate, trigger conditions, identifiers, and calibration.

Different sensors place different levels of power demand on a device. For example, GPS and orientation sensors are significantly more power-expensive than accelerometers. A continuous sensing workload in particular – which requires sensors to be sampled for a period of time – can quickly drain the battery of a mobile device [8]. Such workloads are required for a variety of independent applications, such as those focused on personal health and fitness, but their impact on power consumption would be most pronounced if they were for contributing data to a significant number of services [9]. One widely used approach to reducing the energy consumed by continuous sensing is to decrease the sensor's sampling rate, the number of raw samples collected by the sensor per unit time [10]. Although this can certainly extend battery life, it can also lower sampling accuracy to a level that is below what would be ideal for the application. My aim is to opportunistically optimize the power consumption of sensors while meeting user-specified accuracy requirements.

Consider a scenario where two continuous sensing apps are running simultaneously on the same mobile device, requiring periodic samples from a sensor, but at different sampling rates. The first is an activity detection app, which needs accelerometer samples at the rate of $10Hz$, while the other is a background service for location-detection, which needs accelerometer samples at the rate of $20Hz$. Even though the $20Hz$ stream obviously *contains* a $10Hz$, existing sensor APIs on mobile platforms would set up two separate sampling requests, amounting to a cumulative sampling rate for the sensor of $30Hz$.

I develop ShareSens, an approach which opportunistically economizes on collection of sensor data. When multiple applications make sensing requests, instead of serving each request independently, ShareSens opportunistically merges the requests, achieving significant power and energy savings. Custom filters are then used to extract the data required by each application. Applications – running over ShareSens – can request sensor feeds of fixed or flexible sampling rates. An optimizing sampling scheduler then schedules the sampling of sensors based on the sensing requirements received from applications running concurrently. The scheduler opportunistically optimizes the effective sampling rate of each sensor, exploiting opportunities for different apps to share sensor samples when possible. ShareSens also opportunistically offers higher sampling rates than minimally required to applications which indicate – through flexible rate requests – that they can use them. I present the design and implementation of ShareSens, as well as experimental results on the power savings that can be achieved by using it.

1.1.3 ModeSens: An Approach for Multi-modal Sensing Support

As the number of services on mobile devices grows, so does their energy demand. An important opportunity for conserving energy for such services lies in optimizing sensing for the evolving context that a device is in. Consider, for example, a health-monitoring service which monitors a device owner's heartbeat at a low frequency; however, if an unexpected pattern is detected, it switches the mode to begin monitoring several sensors at a high frequency. This is called *multi-modal sensing*.

I am interested in modeling context switching, in which it is represented as a Finite State Machine (FSM) model for context-aware mobile systems. Using the proposed model, I aim to model the dynamicity of the context-aware mobile systems that adjust their own behavior according to changes in the context.

I develop ModeSens, a mechanism which allows multi-modal sensing requirements of a service to be programmed separately from its function. Programmers can specify the sensing modes in which a service can be, the sensing needs of each mode, and the sensed events which trigger mode transition. ModeSens then monitors for mode transition events, and dynamically adjusts the sensing frequencies to match the current mode's requirements. Separating the mode change logic from an application's functional logic leads to more modular code. I present a set of experimental results demonstrating the performance and energy costs of using ModeSens.

1.1.4 MobDisS: A Formal Model for Representing Mobile Distributed Services

I present my efforts towards a formal model, MobDisS (Mobile Distributed Services), for representing mobile distributed services, which allows fundamental principles of mobile distributed services to be carefully studied. MobDisS identifies core mechanisms underlying mobile distributed services. Services can be built by composing simpler services. I precisely describe the syntax and operational semantics of MobDisS. The simplest form of a service is defined by having one contributor sending a service feed to one client; more complex services are specified by composing services.

1.2 Contributions

The contributions of this work are as follow:

- Design, implementation and evaluation of CSSWare, an Actor-based middleware for initiating and managing crowd-sourced services.
- Design, implementation and evaluation of ShareSens, an approach for efficiently supporting the sensing needs of sensor-based applications.
- Design, implementation and evaluation of ModeSens, a mechanism for programming mode transition concerns of multi-modal sensing applications separately from their functional concerns.
- Compositional definition of the MobDisS model, and its operational semantics.

1.3 Outline

The rest of this thesis is organized as follows: Chapter 2 discusses related work. Chapter 3 describes a set of domain-specific mechanisms for supporting crowd-sourced services using multi-origin communication, as well as the design, implementation, and evaluation of CSSWare. Chapter 4 presents the design, implementation and evaluation of ShareSens. Chapter 5 presents ModeSens, an approach to modeling and programming multi-modal sensing requirements of mobile applications. MobDisS and its operational semantics are presented in Chapter 6. Finally, Chapter 7 concludes this thesis.

Chapter 2
Related Work

My research work to support mobile distributed services has three main aspects – programmability, mobile sensing and formal modeling – with a particular focus on mobile crowd-sourced services within programmability. This chapter discusses related works in programming frameworks and middlewares, mobile sensning, multi-modal sensing and formal models.

2.1 Programming of Mobile Crowd-Sourced Services

There have been a number of projects – both in academia and industry – involving crowd-sourced services. The term crowd-sourced can refer to two types of services: participatory sensing services and crowdsensing services. Participatory sensing involves the explicit participation of human beings in possession of mobile devices, whereas crowdsensing relies on sensor feeds automatically flowing from devices to servers.

I first present some representative examples of both these types of crowd-sourced services, I then present some existing frameworks for enabling it, and finally discuss the existing models for composing such services.

2.1.1 Examples of Crowd-Sourced Services

Some of the best examples of participatory sensing services can be found in services aimed at assisting automobile drivers. Waze [11] is one of the largest community-orientated mobile travel applications with users volunteering information about their driving experience in real time, by reporting on congestions, delays, and gasoline prices. These reports then become the basis for information displayed on other drivers' maps (on their mobile devices), to help them make routing decisions.

Similarly, TrafficPulse [12] combines sensor data from mobile devices with real-time traveler reports from frequent travelers, and then offers this information to other drivers in an aggregate form.

Crowd-sourcing has also been found to be useful in efforts to coordinate rescue efforts following major disasters, such as the Haitian earthquake in 2010 [13]. Information aggregated from social media (e.g., blogs, emails, tweets, and Facebook status updates) was used to overcome challenges created by both the inadequacy of maps and the change in the landscape because of the devastation.

CrowdHelp [14] uses smartphones to collect direct feedback from mobile users about their medical condition, in combination with data coming from sensors in smartphones. This information is used to enable a swift response to emergencies. For example, when CrowdHelp is used for emergency reporting, mobile users submit information relevant to an event (such as the number of injured people and their state) to a central server. This information is collected and sent to the nearest health care facility capable of treating the injured.

Among crowdsensing services, the real-time traffic information displayed on Google Maps is arguably the most widely used one. The service relies on location data voluntarily made available by users of Google's services, which is then aggregated and then visualized on Google's Maps to show traffic flow. Since Google's acquisition of Waze in 2012, Waze's participatory sensing service has now been combined with Google's crowdsensing service for providing real-time traffic flow information.

Crowdsensing has also been used by Uga et al. [15] in an earthquake warning system, which uses data from accelerometers present in many modern mobile devices to detect seismic vibrations. Devices send reports of likely seismic activity to a server which then aggregates the reports received to send out warnings.

2.1.2 Frameworks for Mobile Crowd-Sourced Services

My work is more closely related to research focused on supporting crowd-sourced applications. Existing efforts have taken different approaches to supporting such applications, focusing on concerns from programmability (e.g., Medusa [1] and AutoMan [2]), to privacy (e.g., AnonySense [3]), to participatory crowd-sensing (e.g., CDAS [4]), and efficient sensor data collection (e.g., MECA [5]). I discuss these frameworks below.

Medusa [1] is a programming framework for crowd-sourced applications. A task (such as video documentation or citizen journalism) is launched by a requester, and *workers* are solicited through Amazon's Mechanical Turk (AMT) service.[1] These workers – volunteering smartphone users – then provide raw or processed data to be used as part of a social or technical experiment. An XML-based programming language, MedScript, is used to specify the required task as a series of several stages, from the initial recruitment of volunteer workers, to the workers' (say, for a video documentation task) recording videos on their smartphones, summarizing them, and then sending them back. The stages can involve actions selectable from a library of executables, which are downloaded to mobile devices from a cloud server. Because Medusa requires that tasks pick from a limited set of activities, it suffers from limited programmability and generality, and is not applicable to a large class of crowd-sourced services.

AutoMan [2] is a programming platform for enabling a function-call-like mechanism for requesting human beings to carry out tasks involving vision, motion, natural language understanding, etc. The platform supports scheduling, pricing and quality control. A programmer's interface to AutoMan is a set of function calls, implemented as an embedded domain-specific language for the Scala programming language. The platform also uses Amazon's Mechanical Turk for hiring workers to perform these short-term human-based assignments. Although the way AutoMan supports interactions with people is interesting, its scope is limited

[1] Mechanical Turk is an online human hiring system which acts as an intermediary between employers (requesters) and employees (workers or turkers) to perform short-term computational tasks. https://www.mturk.com/mturk/

to that function. For instance, a similar approach could be used along with my middleware for applications requiring deliberate human input.

AnonySense [3] is another framework for collecting and processing sensor data, which pays particular attention to privacy concerns. AnonySense allows a requester to launch one of a selected group of applications with their parameters. The application then distributes sensing tasks across anonymous participating mobile devices (referred to as carriers), and finally aggregates the reports received from the carriers. Achieving anonymity relies on separating sensor data from identifying features (such as homes or workplaces in GPS traces) to obscure individual identities. Similarly to Medusa, AnonySense has limitations in programmability and generality because of its limited focus on a collection of sensor data and in-network processing.

CDAS [4] is an example of participatory crowd-sensing frameworks. It enables deployment of various crowd-sensing applications which require human involvement for simple verification tasks to deliver high accuracy services. Similar to CDAS, MOSDEN [16] is a collaborative mobile sensing framework that operates on smartphones to capture and share sensed data between multiple distributed applications and users.

The MECA (Mobile Edge Capture and Analysis) middleware for social sensing applications [5] focuses on efficient data collection from mobile devices. It uses a multi-layer architecture to take advantage of similarities in the data required for different applications to lower the demand on devices on which data is being collected. MECA's focus is limited to a narrow class of applications, and does not address wider programmability challenges. Furthermore, MECA – like other similar frameworks – uses the smartphone as a dumb data generator, offloading all processing to the server layer. This increases communication cost and does not allow applications to take advantage of data collected while the mobile device is not connected.

In summary, existing frameworks for crowd-sourced applications focus on narrow application areas or specific concerns, making it difficult to utilize them for a wider class of services. Also, none of them support concurrent execution of multiple services from within one service platform, which precludes taking advantage of opportunities to optimize for shared sensing requirements.

2.2 Mobile Sensing

An important concern in continuous mobile sensing is its impact on the power consumption of smartphones and wearables. I first present some representative continuous mobile sensing applications, and then discuss existing platforms which support them.

2.2.1 Continuous Mobile Sensing Applications

Many modern smartphones and wearables are equipped with a number of sensors, creating opportunities for applications which can use feeds from these sensors. Of particular interest to my research work is continuous mobile sensing, which involves continual sampling of sensors over a period of time. A number of applications rely on continuous mobile sensing across a variety of domains, from healthcare (e.g., [14]) to social networks (e.g., [5]) to environmental monitoring (e.g., [13]) to transportation (e.g., [11, 12, 17]) and human activity recognition (e.g., [18, 19]).

Activity Monitoring

Human activity recognition has opened the door for new types of applications in the field of personal healthcare such as fitness monitoring, elder-care support, and chronic care [19]. Activity recognition systems typically have a sensing module that continuously collects relevant information about the body activities (e.g., walking, sitting, lying down, etc.).

Activity recognition systems can be broadly classified into video and physical sensor-based activity recognition [20]. Video sensor-based activity recognition involves continuous capture of human activities through cameras [21]; however, the extraction of features from the captured images requires complex computations. Due to these limitations, wearable sensor-based activity recognition (i.e., sensor-based activity recognition) [22] – which requires less data processing – is more commonly used. Accelerometer and gyroscope are the most widely used body-worn sensors for activity recognition. Although these sensors are accurate in monitoring activities, the continuous sensing can quickly drain the battery of mobile devices in a short amount of time [10].

In the health-monitoring field, activity monitoring can provide the ability to monitor and diagnose patients using continuously generated data. Physicians can detect deviations from a typical routine of a patient's current physical status. Patients undergoing physical therapy, gain the benefit of being able to have their condition monitored in more details and therapists can ensure their patients improved the quality of care with more accurate evaluations [23].

Activity recognition has also become a key component in several commercial products. For example, game consoles such as the Nintendo Wii[2] and the Microsoft Kinect[3] rely on the recognition of gestures or body movements to enhance the game experience. While these systems are originally developed for the entertainment sector, they have also been used for other types of applications such as personal fitness training [24]. Furthermore, some sports products such as the NikePlus[4] running shoes which integrate motion sensors and offer athletes feedback on user's performance.

2.2.2 Mobile Sensing Platforms

Energy consumption in mobile phones is a well-studied research topic in literature from various perspectives [10, 25–27], each offering a different approach. To my knowledge, there is no existing approach to economizing by enabling sharing of sensing data among different applications.

Early efforts to build frameworks for mobile sensing applications have tended to trade off accuracy for lower

[2]http://www.nintendo.com/wiiu
[3]http://www.microsoft.com/en-us/kinectforwindows/
[4]https://secure-nikeplus.nike.com/plus/

power usage by implementing algorithms which reduce the amount of sensor data. Other frameworks [28, 29] have tried to reduce power usage by opportunistically offloading sensor data processing to back-end servers in the cloud infrastructure. Other techniques rely on adopting a variety of duty cycling techniques that manage the sleep cycle of sensors on the device to trade off the amount of battery consumed against sensing accuracy and latency [30]. The drawback of these approaches is that they are not applicable to continuous sensing scenarios.

Adapting sampling rates for human activity recognition is a well-studied research topic. SpeakerSense [31] uses a low sampling rate to detect whether a speaker exists, and switches to a high sampling rate once a speaker has been detected. Similarly, SociableSense [32] lets the sensors operate at a high sampling rate only when interesting events happen. If there are no interesting events, the sensors are set to run at a low sampling rate. SpeakerSense and SociableSense use what I call multi-modal sensing to sample data only at the rate required. In my efforts to improve programmability of such sensing requirements, I developed ModeSens, which separates applications' multi-modal sensing concerns from their functional concerns.

Another approach is to achieve energy savings through optimizing sensor duty cycles (i.e., periodic sensing and sleeping instead of continuous sensing). EEMSS [30] is an energy-aware framework for human activity recognition. EEMSS tries to achieve energy savings by shutting down unnecessary sensors, and carefully selecting sensor duty cycles. A sensor management scheme is used to determine the minimum sampling lengths and intervals for a set of sensors to recognize user states and to detect state transitions. Mercury [33] uses a similar strategy to reduce energy consumption by disabling and enabling sensors dynamically. ShareSens addresses the orthogonal concern of sharing sensor feeds.

Senergy [34] offers an API that can be used by developers of context-aware applications to specify Latency, Accuracy, and Battery (LAB) requirements of the used sensors in their applications. Senergy focuses on the optimization of continuous context monitoring, energy, latency and accuracy while meeting developers' LAB requirements. The approach essentially allows an application's inherent flexibility to be used to find opportunities to conserve energy.

The growing popularity of continuous sensing has also attracted the attention of hardware designers to develop hardware processors to support it. For example, the Little Rock project [35] at Microsoft Research is developing energy efficient co-processors for a mobile device, dedicated to the task of continuous sensing. The tasks of duty cycle management, sensor sampling, and signal processing are offloaded to the new co-processor, allowing the primary CPU to sleep more frequently, saving the overall power consumption. The Little Rock architecture gives programmers the flexibility to choose between the primary processor and the co-processor for the various tasks in their applications, but it also makes application development more challenging.

2.3 Formal Approaches

First, in Section 2.3.1, I review the Actor model of concurrency [6], which is used in this research as the underlying model for distributed computations. In Section 2.3.2, I review the key concepts and principles of Service-Oriented Architecture (SOA). Then, in Sections 2.3.3 and 2.3.4, I review different types of approaches for modeling and composing mobile services, as well as multi-modal sensing.

2.3.1 Actors

Actors [36] provide a formal model for concurrent systems. Actors are autonomous concurrently executing primitive agents (i.e., active objects). Each actor encapsulates a thread of control, an object (state and methods) and a mailbox. Only an actor's encapsulate thread may access its object directly and change its state. Actors communicate using asynchronous messages. A message's recipient takes a message from its mailbox and executes its corresponding method, possibly sending messages to other actors, or creating new actors. Messages are not necessarily processed in the order they are sent, but fairness in the delivery of messages is guaranteed. Recipients of messages are identified by globally unique addresses (i.e., actor names). An actor can only communicate with actors whose addresses it possesses. Actor names cannot be guessed.

The key semantic properties of the Actors model are described as follows: (i) encapsulation of state; (ii) atomic execution of a method in response to a message; (iii) fairness in scheduling actors and the delivery of messages; and (iv) location transparency enabling distributed execution and mobility.
As a result of processing a message, an actor can concurrently:

- Create a finite number of new actors
- Send a finite number of messages to other actors
- Designate a new behavior to process subsequent messages

Actors are becoming increasingly important as the model of choice for building large-scale interactive systems. There is a growing number of implementations of Actors, including programming languages such as Scala [37] which supports actors through its Akka library [38], Erlang [39], Pony [40], SALSA [41], etc. In addition, there are also actor libraries to support Actor programming with existing languages, such as Actor Architecture (for Java) [42], ActorFoundry (for Java) [43], Pulsar (for Python) [44], CAF (for C++) [45], Orleans [46] (for Microsoft's .NET framework), etc. These languages and libraries have been used in many industrial projects such as Twitter's message queuing system, Facebook chat, Vendetta's game engine, etc. [47].

2.3.2 Service-Oriented Architecture

Service-Oriented Architecture (SOA) [48] is a software paradigm that provides a way to design flexible, scalable, and reusable services over a network. In SOA, a service is defined as a logical representation of a business activity that has a specified outcome. Services can be composed to create a new service which provides a new business value. An architecture for service-oriented systems has three parts: a provider, a consumer, and a registry. Providers publish their services on registries, which provide an interface for consumers to search for services and then invoke them. A service is a loosely-coupled and self-describing computation element. This is achieved through encapsulation and communication through message passings. In SOA, a set of interface languages are used to permit a service to export sufficient information so that consumers can discover and connect to it. The key principles of SOA are described as follows [48]:

- Service Abstraction. Services act as black boxes, that is their inner logic is hidden from the consumers.
- Service Autonomy. Services are autonomous and independent of control.
- Service Reusability. A service-oriented system's logic is divided into various services in order to support reuse of code.
- Service Longevity. Services should be designed to be long lived.
- Service Location Transparency. Services can be consumed from anywhere within the network which enables distributed execution and mobility.

In SOA, applications are primarily distributed systems of composed services. These services can belong to different organizational domains. Therefore, there is no single line of authority regulating their interactions, which enforces consumers to either trust the provider to deliver the expected service, or establish a contract with it. A contract describes an agreement between a number of services that determines rights and obligations on its signatories.

Orc [49] is a programming language designed for supporting service-oriented computing. Orc provides uniform access to computational services, including distributed communication and data manipulation. Orc enables a concurrency-first style of programming, in which programmers start with a concurrent program, instead of adding concurrency only when it is required. In Orc, concurrent computations (called Orc expressions) are built by composing primitive operations using a set of combinators and declarations. Executing an Orc expression results publishing values. These publications may occur at different times, which means that Orc naturally supports asynchronous events. The primitive operations in an Orc program are called sites, which are callable values whose execution is outside the Orc semantics and can be implemented in any language. Therefore, sites allow an Orc program to interact with the outside world.

2.3.3 Mobile Service Models

Mobile services are increasingly attracting significant interest. There are a number of related works in the literature which offer support for mobile services, ranging from efficient communication protocols (e.g., CoAP [50]), to publish/subscribe systems for mobile environments (e.g., [51] , [52]), to service composition models (e.g., [53], [54]).

CoAP [50] is specially designed protocol for use with resource-constrained devices (e.g., mobile devices) over low-bandwidth network links. CoAP fulfills the requirements of the mobile environment domain such as providing support for asynchronous message exchange, multicast capabilities, low overhead, and implementation simplicity.

Huang et al. [51] attempted to extend the publish/subscribe model to operate in a mobile environment, where events can be generated by moving mobile devices or users, and subscribers can request delivery of events at their mobile devices. The proposed model consists of three main parties: event sources, an event brokering system, and event displayers. An event source generates events based on changes its surrounding environment. Then the generated events are published to the event brokering system, which matches them against a set of subscriptions, submitted by event displayers in the system. However, the described system assumes a broker backbone that has a dedicated broker-based sub-network located on a static backbone and consisting of a predefined set of servers, which limits the application of this approach on a larger scale.

Existing solutions to service composition use two different approaches: a template-matching approach and a dynamic approach.

In the template-matching approach, the client requirements (i.e., service specs) are expressed as a request template, and through composition, a system would identify services to populate the target service's specs within the request template. However, with the dynamism involved in mobile environments, the client requirements have to be met by exploiting available resources, even when an exact match does not exist. Furthermore, the traditional models of template-matching service composition [55–57] do not consider the mobility of service providers (i.e., contributors) and requesters (i.e., clients) in mobile environments. So they cannot be used directly for mobile services because these models have an implicit assumption that services are running on stationary servers. This assumption does not hold true for modern mobile devices equipped with various sensors and a significant computation power.

In the dynamic approach, the service specs have to be met by exploiting available resources, even when an exact match does not exist. Our work is more closely related to that research focused on supporting dynamic service composition. However, existing solutions which implement that approach focus on state-space model [58], wherein a huge number of states need to be checked before composition. These solutions for dynamic composition are called state-based solutions.

One of the earliest attempts to provide dynamic service composition in mobile environments is proposed in [53] where a composition service protocol is implemented through a middleware called PICO. Mobile services can be initiated and consumed through that middleware. This service composition approach models mobile services as directed attributed graphs, maintains a repository of service graphs and dynamically combines multiple basic services into complex services. However, two issues limit the application of this approach on a larger scale: service selection and detection of equivalent services. During the step of state-space exploration, a state can be generated more than once for analysis purpose. To avoid analyzing the same states repeatedly, it is necessary to remember the states already explored by storing them in the memory. However, this can lead to storing a huge number of them in the memory and eventually to state-space explosion [58], making it difficult to support more complex compositions. Our approach for service composition avoids state explosion as it does not depend on the state of composable services.

Deng et al. [54] describe a mobile service provisioning architecture to solve the problem of service selection and composition of mobile services when both service requesters and providers are mobile. They propose a mobility model to describe mobile user moving behavior. The mobility model is used to transfer the mobility-aware service composition problem into an optimization problem and then the Krill-Herd (KH) algorithm [59] is used to solve it. However, such an approach can be impractical when applying the mobility model to each provider (i.e., contributor) in the system because of the complexity of solving the composition problem, which leads to significant decrease in performance. In comparison, our approach for service composition does not suffer from performance issues when composing services because each composition rule has a set of light-weight constraints which applied on the input services before the composition process takes place.

2.3.4 Multi-Modal Sensing

Multi-modal behavior has previously been examined in various mobile device contexts. I present below some existing frameworks for programming sensing modes in a mobile environment.

In [60], the history of contexts of a device together with the actions previously carried out by the device, define a mode which then determines which actions are to be carried out in that mode. The author presented a finite state machine model for the situation-aware mobile applications, where each state represents a mode that a mobile device behaves at a time, and each mode is characterized by the historical context record and the internal action record of that device. Furthermore, the finite state machine is used to build a spanning tree of state machine which could be used in verifying that a corresponding situation-aware application's requirement is valid, i.e., the requirement does not specify any situation that will never be true or never trigger any actions.

Matic et al. [61] attempted to identify the social mode in which a user might be. The goal of this work was to detect face-to-face social interactions between people on a small spatiotemporal scale using built-in sensors in mobile devices (e.g., accelerometer, microphone and GPS). The proposed approach detects spatial setting through parameters of interpersonal distances and relative body orientations sensed by mobile phones. Recognition of speech activity is done using the accelerometer sensor by detecting vibrations at the chest level that are generated by vocal chords during phonation.

Chiu et al. [62] present an advisory system for the transportation domain, which offers routing advice based on the perceived mode of traffic flow. In MOSDEN [16], mobile devices determined to be in particular modes (such as being in a moving vehicle) can be tapped for data in a *sensing as a service* context. MOSDEN also provides a mobile sensing as a service framework that operates on smartphones to capture and share sensed data between multiple distributed applications and users.

In contrast, I am specifically interested in *sensing modes*, which tell us the sensing requirements in an application's environmental, interest or resource context.

Chapter 3

CSSWare: An Actor-Based Middleware for Mobile Crowd-Sourced Services

This chapter is organized as follows: Section 3.1 describes a set of domain-specific mechanisms to supporting crowd-sourced services using multi-origin communication. Section 3.2 and 3.3 present the design and prototype implementation, respectively. Section 3.4 evaluates the approach in two ways. First, it illustrates the ease with which new services can be implemented on the platform. Second, it presents experimental results showing scalability, performance and energy efficiency of the approach.

3.1 Multi-origin Communication

It turns out that a large class of crowd-sourced services exhibit a similar pattern of interaction, where members of a *crowd* contribute bits of information from their respective contexts, which are then aggregated to create useful information for clients. This pattern of interaction is orginally defined in [7] as *multi-origin (multi-sender) communication*, which involves aggregation of the messages received from a group of senders (referred here to as the constituency) into a *group message* to be sent on behalf of the group to one or more intended recipients.

This thesis shows how to apply this approach in the domain of crowd-sourced services. This is illustrated in Figure 3.1, where senders a_1 through a_n autonomously send messages with the intention of them being sent to a destination in aggregate form.

To be precise in the presentation of multi-origin communication, it is specified in terms of the Actor model [6]. The different parties involved in a multi-origin communication are represented using actors, and the required communication is defined in terms of asynchronous actor messages.

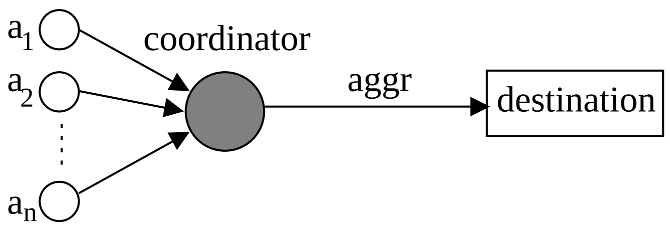


Figure 3.1: Multi-Origin Communication

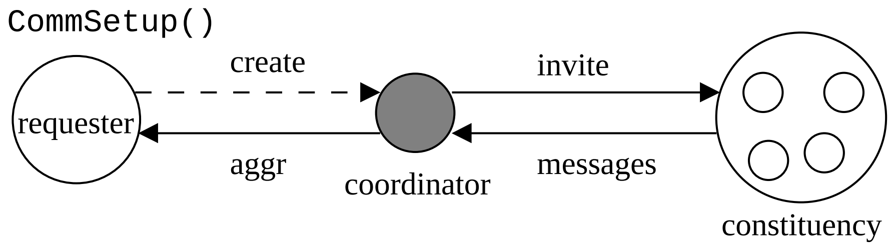


Figure 3.2: Multi-Origin Communication Setup

The requester of a multi-origin communication makes a function call in order to launch the communication. The call passes two parameters, the first specifying the potential contributors – the constituency – to be invited to participate in the communication, and the second specifying an aggregation method. As illustrated in Figure 3.2, an invocation of this function results in the creation of a new coordinator actor capable of coordinating the communication, which is next told to invite the constituency to participate. The coordinator then sends invitations to the members of the constituency (the contributors) to send their messages; when applicable, it also sends them parameters advising on how to construct their contributions (such as by tapping into a set of sensors, or soliciting input from the user), how often to send them (once or periodically, how frequently), etc.

I assume that each contributor is an actor with a method to receive these requests, and the capability to create the types of messages. Given that there are a relatively small number of sensors on mobile devices, the parameters could simply be specifying which sensors to be tapping into, with what frequency, and what periods to be averaging the feeds over, etc. However, coordinators for some services may be more interested in hearing about higher-level events – such as a restaurant client sitting down at the table, finishing eating, paying the bill – which would require more significant local processing to generate than simply receiving sensor feeds. This could be supported in various ways: by migrating an actor with the required behavior to the sender, by sending the code as a parameter to create an actor locally, or simply by frequently updating the sender-side application to include the functionality needed by every type of request.

As the contributors send their messages, the messages are aggregated by the coordinator according to its own behavior, to generate group messages on behalf of the contributors.

Two types of such setups are introduced. The first – one-off multi-origin communication – is to solicit a group message from a number of contributors with a termination condition and a timeout. This would be the type of communication used to serve one-time requests, such as to hold a census or an election, or to satisfy a one-off request to recommend a restaurant with a short waiting time. The second – continual multi-origin communication – is to solicit a continual feed of group messages from a number of contributors. This would be useful for a service provided over the web or through a mobile application where site visitors or application users seek up-to-date information (say) on restaurant waiting times in a neighborhood. For some services, such as the one for restaurant recommendations, the choice of one or the other setup would depend on the frequency of requests, the number of potential contributors of messages, etc. For instance, it would not be useful to be maintaining up-to-date information about all restaurants when there are very few requests for recommendations; however, it would be wasteful to solicit one-off communications for frequent requests.

3.1.1 One-Off Multi-Origin Communication

In a one-off multi-origin communication, the coordinator actor expects at most one message from any contributor. It collects messages until either a sufficient number of messages has been received (as can be tested using a termination function), or a timeout has been reached; it then proceeds to aggregate the messages, and sends the aggregate to the requester on behalf of the contributors. An example of a multi-origin communication with timeout would be an electronic voting service, where the coordinator expects no more than one vote from each voter and there is a deadline by which all votes must be in.

Figure 3.3 illustrates the execution of a one-off multi-origin communication using an actor event diagram [6]. In the figure, $contrib_1$ through $contrib_n$ are the prospective contributors. There is a clock actor to which the requester sends a request to notify the coordinator when the timeout has been reached. I assume that the clock is local to the coordinator and has a way of notifying in a timely manner. The requester initiates the communication by calling the function `oneOffCommSetup(coordClass, constit, termCond, timeout)`, where `coordClass` is the desired behavior of the coordinator, `constit` is a list of contributors, `termCond` is a function to test the termination condition indicating receipt of a sufficient number of messages, and `timeout` is a time when the coordinator would stop accepting messages from the contributors.

Once the coordinator is created, it sends announcements to all contributors, and begins collecting messages. The coordinator expects to receive the maximum of one message from each contributor. After the timeout is reached, the coordinator sends a message to the requester with an aggregate of all responses.

The coordinator actor's behavior can be defined by extending the *multicall* selective blocking broadcast operation defined in [7] with support for timeouts, or directly using the following three methods:

- `announce(constit)`, used by the requester to instruct the coordinator to solicit messages from members of the constituency.
- `sendMessage(msg)`, used by the contributors to send their messages to the coordinator.
- `timeout()`, used by the clock to tell the coordinator that the timeout has been reached.

A contributor actor's communication behavior is defined by one method: `receiveAnnouncement(serviceParams)`. This is the method invoked when the solicitation is received from the coordinator, and it carries out the computations specified in `serviceParams` in order to create its message.

Figure 3.4 shows pseudocode for the `oneOffCommSetup` function. The `createCoordActor` function creates a new coordinator actor with the termination condition and application-specific customization initialized in its behavior, and returns the coordinator name. Once the coordinator has been created, a message is sent

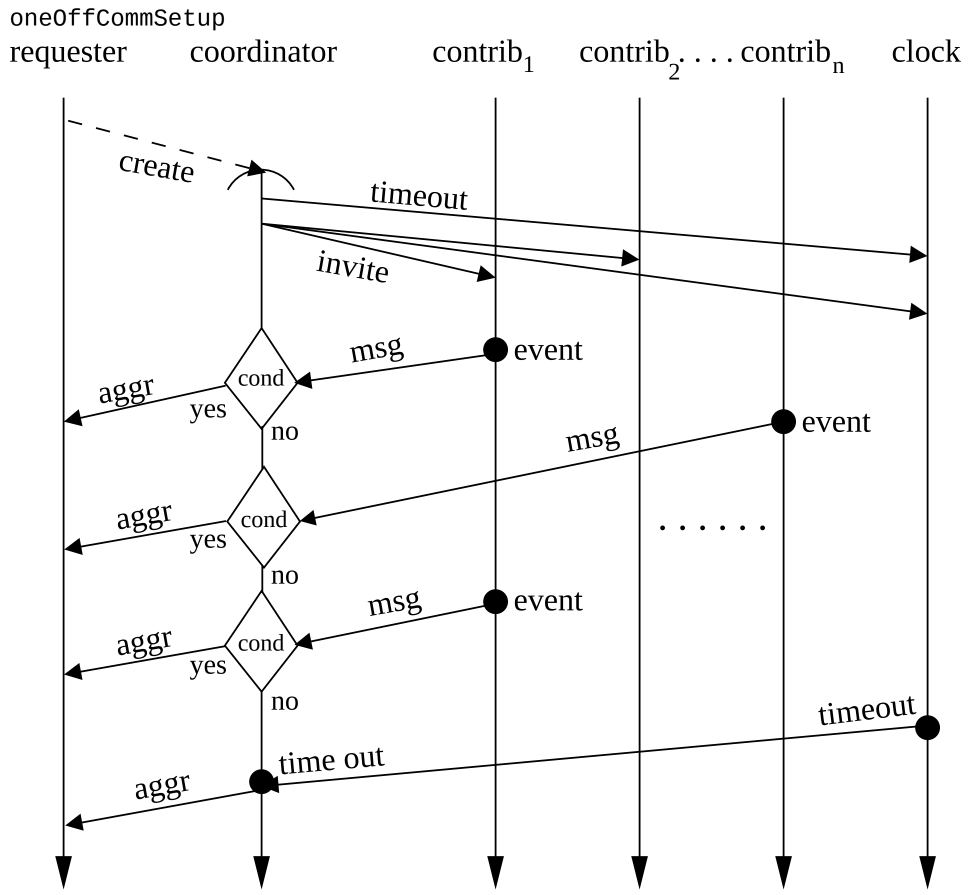


Figure 3.3: One-Off Multi-Origin Communication

to the coordinator to broadcast an announcement to all contributors. Another message is sent to the clock actor instructing it to notify the coordinator when the timeout is reached.

Here we assume that there is a static timeout set by the coordinator in the communication setup. One possible way to generalize this approach is to reset the clock after receiving a group of contribution messages to allow for *dynamic context-sensitive timeouts.* For example, a clock-reset message could be sent from the condition function to the clock after receiving a new contribution message. Another approach is to use multiple timeout intervals, wherein each of the timeout intervals corresponds to one type of contribution messages. When a timeout interval elapses for any of these types of messages, all messages with that type will be discarded.

3.1.2 Continual Multi-Origin Communication

In a continual multi-origin communication, the coordinator expects multiple messages from each contributor over time, and periodically aggregates them and sends updates to the communication's requester. When a new message arrives, the coordinator checks whether it warrants an update, or whether the interval for which it was to collect messages has passed. In either case, it forwards an aggregate of messages received since

```
void oneOffCommSetup(coordClass, constit, termCond, timeout)
{
    coordinator = createCoordActor(coordClass, termCond, custom);
    coordinator <- announce(constit);
    clock <- timeoutSetup(coordinator, timeout);
}
```

Figure 3.4: Pseudocode for oneOffCommSetup

the beginning of the interval to the requester. An example of continual communication would be that of a restaurant recommendation service available over the web, which attempts to offer up-to-day information to site visitors. The service could also be customized for individual visitors, based on their geographic locations, preferences, etc.

Figure 3.5 illustrates the execution of a continual multi-origin communication using an actor event diagram. $contrib_1$ through $contrib_n$ now send multiple messages over time, reporting local updates. Also, the clock actor periodically (i.e., after every `interval` period of time) notifies the coordinator of the passage of an interval, at which time the coordinator computes a new aggregate.

A continual communication is initiated by the requester by calling the function `continualCommSetup( coordClass, constit, updateCond, interval)`, where `coordClass` is the desired behavior of the coordinator, `constit` is the list of prospective contributors, `updateCond` specifies the condition in which the requester should be immediately updated,[1] and `interval` specifies the intervals at which the coordinator would be notified by the clock.

Once the coordinator has been created, it broadcasts an announcement to all contributors, and then waits to receive messages. Contributors either send updates periodically or when they observe an interesting event (such as a change in the level of activity in a restaurant, for example).

A coordinator actor's behavior is defined by the following methods:

- `announce(constit)`, used by the requester to instruct the coordinator to solicit messages from members of the constituency.
- `sendMessage(msg)`, used by the contributors to send messages to the coordinator.
- `interval()`, used by the clock to inform the coordinator of the passage of each interval.

A contributor actor's behavior is defined by one method: `receiveAnnouncement(serviceParams)`. This is the method invoked when the solicitation is received from the coordinator, and it carries out the computations specified in `serviceParams` required for creating its messages.

[1]This should also lead to resetting of the interval with the clock; this is not shown in the event diagram to avoid making it too crowded.

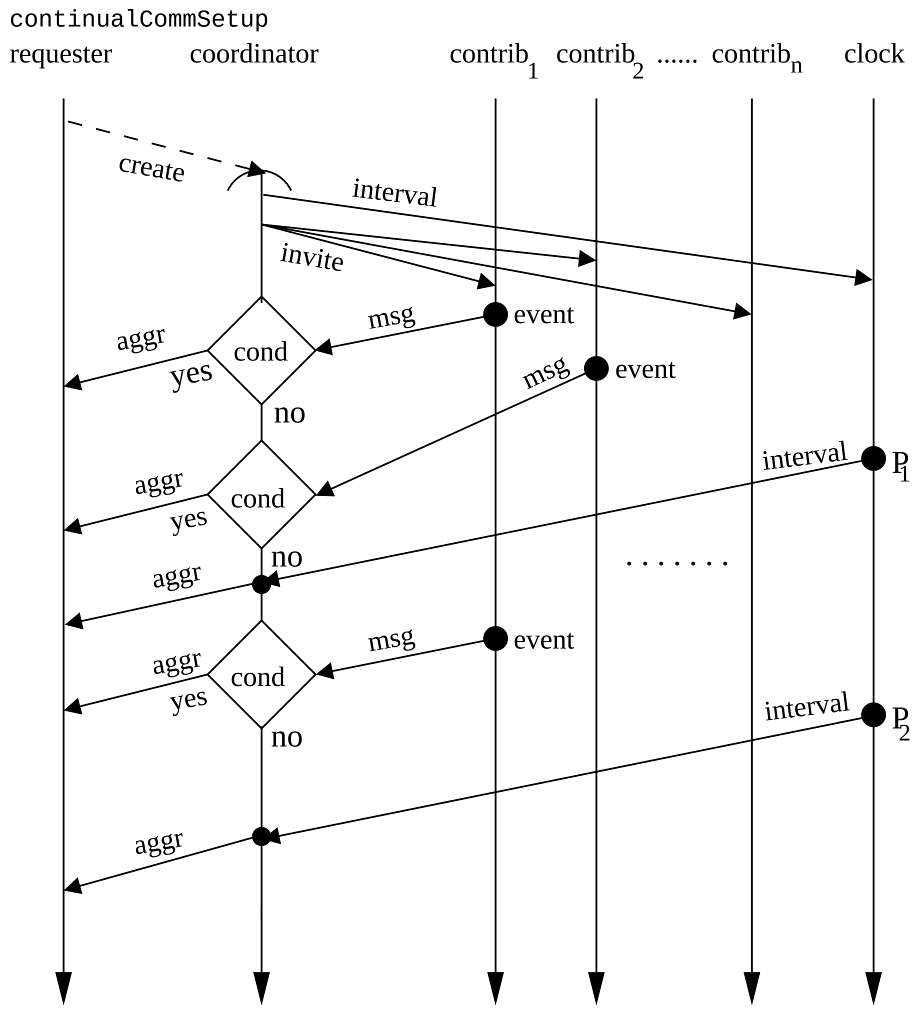


Figure 3.5: Continual Multi-Origin Communication

Figure 3.6 shows the pseudocode for function `continualCommSetup`. The `createCoordActor` function creates a new coordinator actor with an update condition and application-specific customization initialized in its behavior, and returns the coordinator name. Once the coordinator has been created, a message is sent to the coordinator[2] to broadcast an announcement to all contributors. Another message is sent to the clock actor instructing it to notify the coordinator every time the required interval has passed.

Most examples of crowd-sourced services fit the *continual* type of multi-origin communication, where members of the constituency send messages on a continual basis rather than just once. From here on, I will refer to continual multi-origin communication as simply multi-origin communication.

Appendix A presents two case studies to illustrate the use of the two multi-origin communication primitives which have been discussed in this chapter.

[2] $a \leftarrow m(p)$ means message m with parameters p is sent asynchronously to actor a.

```
void continualCommSetup(coordClass, constit, updateCond, interval)
{
    coordinator = createCoordActor(coordClass, updateCond);
    coordinator <- announce (constit);
    clock <- intervalSetup(coordinator, interval);
}
```

Figure 3.6: Pseudocode for continualCommSetup

3.2 CSSWare Design

The design of the CSSWare middleware builds on the domain-specific mechanisms for multi-origin communication described in the previous section.

As illustrated in Figure 3.7, the sensing crowd becomes the constituency whose input is solicited. The service continually aggregates the feeds arriving from the crowd to create up-to-date custom views for various types of clients. For example, if the service were for recommending restaurants, one interface could be for prospective diners, another for the restaurant managers making real-time staffing plans, yet another could be for a vehicular routing system interested in improving downtown traffic flow at lunch time.

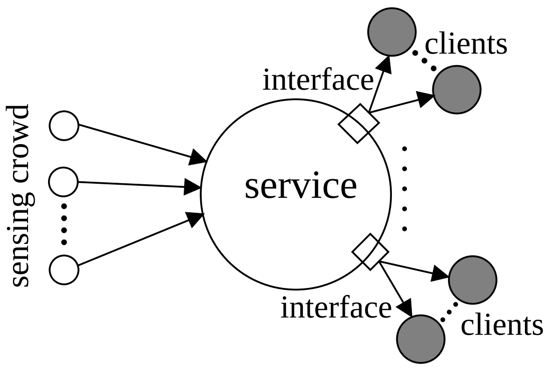


Figure 3.7: Crowd-Sourced Service

Figure 3.8 illustrates how the distributed run-time system for the middleware is organized with parts executing on the service platform, on devices of members of the constituency, as well as client devices. In the rest of this section, I discuss these three parts separately.

3.2.1 Service Platform Side

The service designer uses the service creation API to create and launch a new crowd-sourced service. A set of parameters stating service specifications is passed through the API. These specifications identify the contributors to be invited to participate in the service, the aggregation method to be used, as well as a

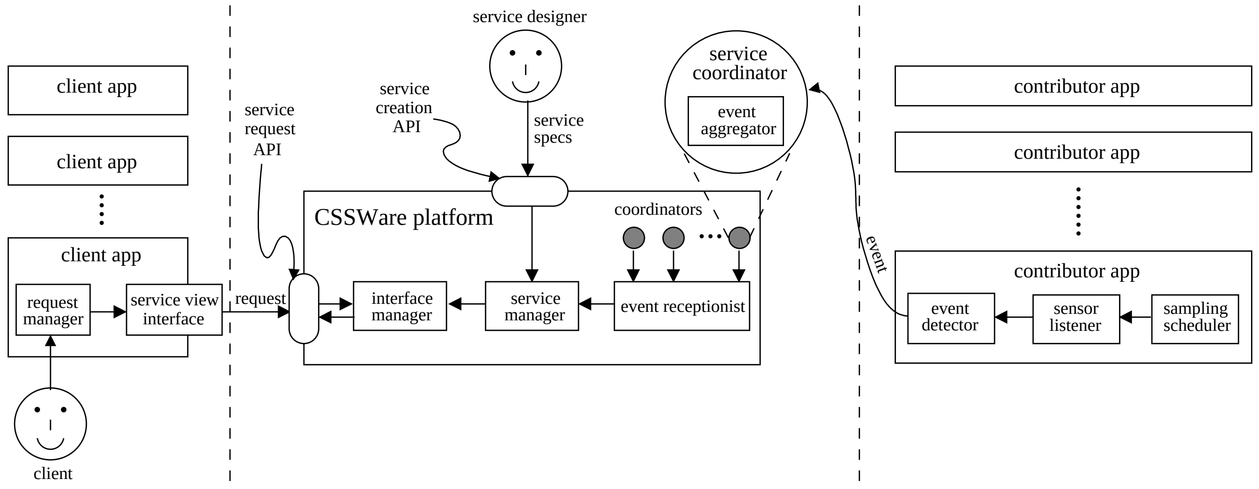


Figure 3.8: System Architecture

description of the feeds solicited from the contributors in terms of specific events of interest, such as arrival at a restaurant, being seated at the table, etc.

To launch a new service, the service manager (see the server in Figure 3.8) creates a new service coordinator to coordinate the communication between the contributors and the CSSWare platform, which is capable of coordinating the communication between the contributors and the CSSWare platform. Next, it sends invitations to the contributors to send their events – when one is detected – to the coordinator. It also sends them parameters advising on how to detect events, construct their messages, and how often to send them (once or periodically, how frequently, etc.).

Contributor events received by a service coordinator are handled by its event aggregator, which in turn reports the events in aggregate form to the CSSWare platform's event receptionist. The aggregated events are then passed on to the service manager, which processes them to update the service's state, which is forwarded to the service interface manager to deliver appropriate views requested by clients through custom interfaces.

3.2.2 Contributor Side

To launch a service, the platform's service manager sends invitations to contributors to participate in the service. It also sends them parameters advising on how to detect events and construct their messages (i.e., sensing parameters). Event detection is carried out by dedicated *event detection actors*, who generate event feeds using relevant sensor feeds, which are then sent to the service coordinator.

An optimizing *sampling scheduler* schedules the sampling of each sensor based on the sensing requirements received from the service coordinator for each service being served at the time. More details about the sampling scheduler will be discussed in Chapter 4.

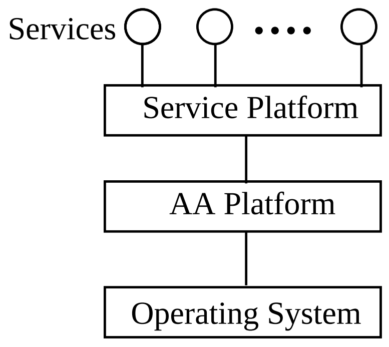


Figure 3.9: System Design

3.2.3 Client Side

A service can have various types of clients subscribed to different views of the service's state, each provided by a custom interface. When a client requests subscription to a particular type of view, the request manager inside the client app constructs a custom view subscription request. This request is passed on to the service view interface, which is transmitted through the service request API of the CSSWare platform (see Figure 3.8). The platform adds the client to a list of subscribers to that view of the service, and begins sending it all updates.

3.3 CSSWare Implementation

This section presents the prototype implementation of CSSWare middleware. I have prototyped it as an actor system. In particular, the prototype implementation is built using the CyberOrgs [63] extension of Actor Architecture (AA) [42], a Java library and runtime system for distributed actor systems.[3]

The mobile side – contributor and client sides – for the systems is prototyped as a self-contained mobile application implemented over AA ported to Android.

As shown in Figure 3.9, crowd-sourced services run over the CSSWare platform, which runs over the CyberOrgs runtime system. The implementation for both CSSWare and ModeSens has two parts: a server implementing a crowd-sourced service platform (about 7,800 lines of code), and a mobile app supporting both client and contributor functionalities (about 4,900 lines of code).

3.3.1 Service Platform Side

To launch a new service, first, the requested service's meta data (i.e., its title and description) is added to the list of published services, which lists active services visible to contributors. Next, the service manager creates a service actor which invites potential contributors to send their events to the service's coordinator.

[3]Actor Architecture is a platform for implementing systems of actors [6], which are autonomous concurrently executing primitive agents (i.e., active objects) which communicate using asynchronous messages, CyberOrgs extension adds mechanisms for resource coordination of actors.

It also sends them parameters advising on how to construct their contribution messages. After inviting the contributors, a new service view is created in the service request API to serve clients' requests.

As contributors to a service detect and send events, the events are aggregated by the coordinator and reported to the service manager through the event receptionist (see Figure 3.8). The service manager collects aggregated events until a sufficient number of them have been received (as determined by a sufficiency condition provided by the service designer in the form of a function) and then updates the service state, revising the custom service views available to the clients.

3.3.2 Contributor Side

For the contributor (and client) side, I have ported CyberOrgs to Android OS, and implemented a self-contained application over it which runs on the Android OS (ver. 5.1). The current implementation supports contributions based on feeds from the GPS, accelerometer, microphone, magnetometer, gyroscope, pressure, humidity, temperature and light sensors. A set of high-level sensor events has been pre-implemented in terms of these (low-level) sensor events – as executable specifications – which a service designer can draw from and customize by providing parameters. These high-level events form the basis for service events. For each high-level sensor event feed, the list of required low-level feeds is provided in the form of a list, where each entry identifies a sensor and specifies the rate at which it should be sampled. These specifications are typically only a few lines of code, varying between 7 and 18 lines of code for the triggers used in the example service prototypes. The code for using high-level sensor events to generate the service events is typically even shorter. The current prototype does not have a way for a service designer to add an entirely new high-level sensor or service event types; ongoing work is developing a way to allow that.

As shown in Figure 3.10, the runtime system executing on the Android device has two components: the sampling scheduler and the event detector.

Sampling Scheduler. As described in Section 3.2.2, the sampling scheduler sets a sampling rate for each sensor based on the received sensing parameters. The scheduler optimizes sensor sampling feeds by opportunistically sharing them between different service feeds. It then sends these requirements to individual sensor listeners, which then sample sensor data at the required sampling rate.

In Chapter 4, I describe further details of this process and introduce the ShareSens API through which the sampling scheduler can be accessed by any mobile application.

Event Detector. Because the data sampled from a sensor can be for multiple event feeds, the data is filtered to extract the sub-feed pertinent to each event feed being served, and only that sub-feed is forwarded to the relevant event detection actor. An event detection actor monitors the sensor feed it receives for event triggers; when it sees one, it fires the event off to its service coordinator.

An event detector does not maintain a local record of the triggered events itself; all events are sent to the service coordinator.

Because the contributor side of the system will likely execute on battery-operated mobile devices, it is

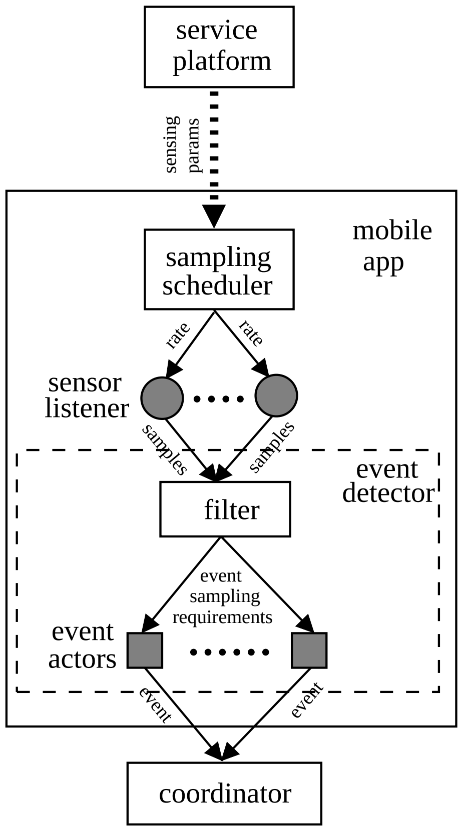


Figure 3.10: Contributor Side

important that contributors have the ability to either develop or adopt simple resource consumption policies to avoid undesired battery drain. I hope to utilize the fine-grained resource management features already present in the CyberOrgs [63] extension of Actor Architecture, which I have used in my prototype. For now, I have implemented a feature allowing a service designer to specify resource limits after reaching which the contributor device would stop contributing feeds.

3.3.3 Client Side

The client side of the platform is implemented as part of the Android application implementing the contributor side. When a new service is launched, each client receives a notification about the launch. Multiple views are supported through custom interfaces installed by the service designer. A client interested in subscribing to a service can examine available views using the *service view interface* (see Figure 3.8), and then use the service request API to subscribe to the desired view.

There is a collection of four general purpose view interfaces pre-implemented in the platform, which averages at about 85 lines of code (the largest at about 100 and the smallest at 75 lines).[4] Although

[4]These 350 lines of code are included in the previously mentioned roughly 4,900 lines of code for the Android application's implementation.

these interfaces are sufficient for the examples I have implemented, and for services with similar client-side requirements, additional interfaces would need to be implemented for different types of services. In my current prototype, there is no way for service designers to program these interfaces themselves; however, I plan to provide a way for new (general purpose or custom) interfaces developed by service designers or other parties to be installed or added to a repository from which they could be installed.

3.4 Evaluation

This section evaluates the CSSWare model in two ways. First, the programmability advantages of using CSSWare in terms of the orders of magnitude lower number of lines of code required for launching a new service. Next, the experimental results demonstrating scalability, performance and data-contributor side energy efficiency of the approach.

3.4.1 Programmability

The benefit of using CSSWare is relatively obvious: a fewer number lines of code to create new services. I illustrated the ease with which new services can be launched by presenting source code for prototype implementations for two qualitatively different types of services. My objective here is not to compare these implementations with equivalent services built on top of existing frameworks (such as [1–5]) because it is difficult to utilize these frameworks for implementing such services. Rather, I compared them with equivalent built-from-scratch services I implemented.

As shown in Table 3.1, the prototype restaurant recommendation service presented in this section required 19 lines of main service specification code; in comparison, an equivalent standalone service I implemented required 6,142 lines of code. A Twitter-like messaging service I implemented, similarly required 21 lines of main service specification code instead of 4,768 lines of an equivalent standalone service. Both services required less than 160 lines of additional relevant code from available libraries of aggregation functions, sensor-events specifications and service view interface. These 160 lines of code are included in the previously mentioned roughly 4,900 and 7,800 lines of code for the Android application's implementation and service platform's implementation, respectively.

Table 3.1: Lines of Code Comparison

Service	Standalone	CSSWare			
		service specification	aggregation function	high-level sensor events	view interface
Restaurant Recommendation	6,142	19	9	53	94
Twitter-like Messaging	4,768	21	16	31	75

To further simplify the specification of crowd-sourced services, I have developed a simple graphical user interface to allow clients to specify the service events, transitions between events, and any outputs to the service. Figure 3.11 shows this interface.

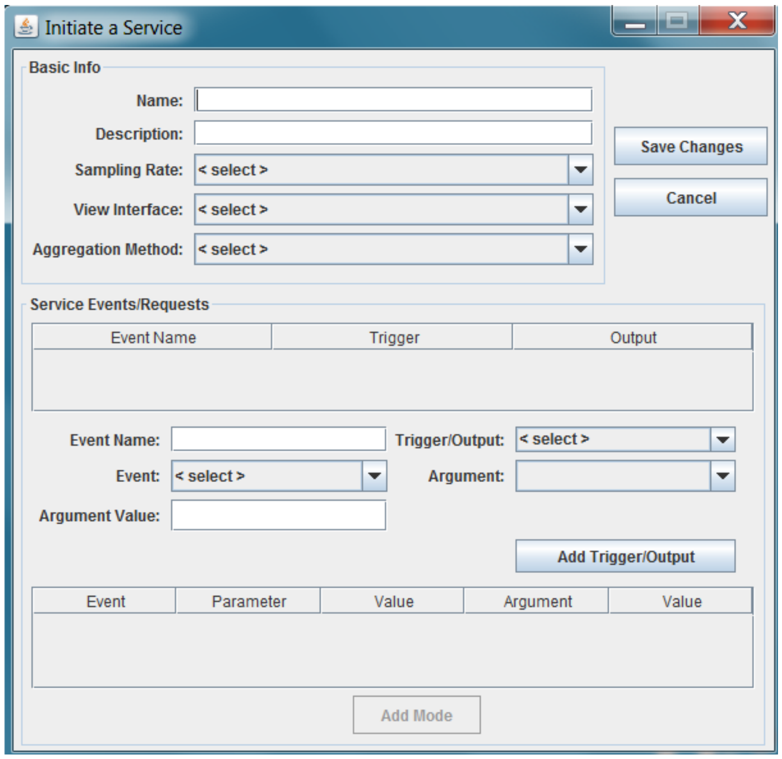


Figure 3.11: The GUI for Creating Crowd-sourced Services

Below, I present prototype implementations of these two qualitatively different services to illustrate the ease with which new services can be programmed.

Restaurant Recommendation Service

Consider the type of restaurant recommendation service previously described in Chapter 1, where mobile devices of people visiting restaurants in a neighborhood automatically send real-time updates about the service they are receiving to a service provider, which then aggregates this information for people searching for restaurants. I assume that information required for generating these feeds can be gathered automatically by the devices by tapping into various sensors to determine when someone arrives at a restaurant, when they are waiting to be seated, when they sit down, when they are served, when they finish eating, and when they leave. The information could be coarser or finer grained depending on the device, usage habits, quality of the behavior detecting software, etc. These updates from personal mobile devices could then be aggregated by a service provider to rank restaurants according to criteria such as the amount of wait time before being seated, the length of time taken dining (shorter or longer, as preferred), the total amount of time that the user could expect to travel to the restaurant, dine, and be back at work. The ranking could also consider the server's meta-knowledge about the number of people being sent to various restaurants by the service.

Figure 3.12 shows the service graphically, and Figure 3.13 presents my code implementing such a service as a `createSensorService()` method. First, a number of service variables are initialized: a method to be used by the coordinator to aggregate contributions, `aggrMethod`, and the default sampling rate to be used for sensor feeds when a rate is not explicitly specified, `samplingRate`. `aggrMethod` is initialized here to a general purpose method for computing the average; it is to be used by the coordinator to compute average waiting time. Other services could use other available aggregation methods; the prototype provides a selection of them. I also provide a template for a service designer to add a new (custom) aggregation method.

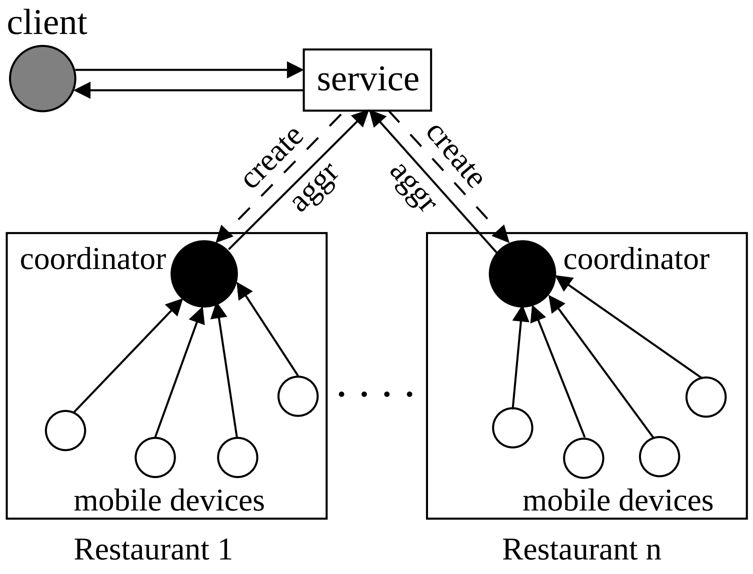


Figure 3.12: Restaurant Recommendation Service

Two types of service events are defined. The first, `locationEvent`, is defined in terms of a number of parameters. The "trigger" parameters identify high-level sensor events, which become the basis for service events. For example `enterPlace` recognizes entering a location (a restaurant in this service). The "output" parameters identify the service events to be sent to the coordinator; here, `visitTime` computes the difference between `enterPlace` and `departPlace`. Additional parameter types are parameters that are available to the various methods; for example, `updateInterval` is available to `visitTime` as a parameter to decide the frequency of feeds to send to the coordinator.

Similarly, `activityEvent` specified a different sensor feed related to observations of the restaurant client's activity. It uses various sensor feeds. The triggers detect activities of "sitting down" or "being still," the latter using the `stillTime` parameter, which are then used as the basis for a `waitTime` service event to be sent to the coordinator.

Finally, the service is created as an instance of the `CrowdService` class, and launched. The constructor for `CrowdService` takes as parameters a `title`, a `description`, the sampling rate to be used for sensor feeds `samplingRate`, the list of events (i.e., `locationEvent` and `activityEvent`) and the aggregation method

```
void createSensorService()
{
    /* initialize service variables */
    int aggrMethod = SvcEnum.average;
    int samplingRate = SensorManager.SENSOR_DELAY_NORMAL;

    /* define a service event */
    ServiceEvent locationEvent = new ServiceEvent (SvcEnum.sensorEvent,
     new List<EventParam>(){
       createParam(``trigger'', SvcEnum.enterPlace),
       createParam(``trigger'', SvcEnum.departPlace),
       createParam(``argument'', SvcEnum.geoRadius, 10),
       createParam(``argument'', SvcEnum.placeType, ``restaurant''),
       createParam(``argument'', SvcEnum.updateInterval, 30),
       createParam(``output'', SvcEnum.visitTime)});

    /* define a service event */
    ServiceEvent activityEvent = new ServiceEvent (SvcEnum.sensorEvent,
     new List<EventParam>(){
       createParam(``trigger'',SvcEnum.sitDown),
       createParam(``trigger'',SvcEnum.still),
       createParam(``argument'',SvcEnum.stillTime,1),
       createParam(``output'',SvcEnum.waitTime)});

    /* create and launch the service */
    CrowdService service = new CrowdService(title, description, samplingRate,
     new List<ServiceEvent>() {locationEvent, activityEvent}, aggrMethod);
    service.launch();

}
```

Figure 3.13: Restaurant Recommendation Service

`aggrMethod`. Once the service has been created, `launch` is called to launch the service, which creates the coordinator actor to coordinate the communication between the contributors and the service, which then invites the contributors to begin sending their event feeds.

Twitter-like Messaging Service

A service like Twitter serves a number of purposes, which include transmission of personal, organizational and news updates, social networking, coordination of collective action, and sharing or propagation of opinions. Increasingly, it has also served as a source of information for journalists, opinion makers, politicians, etc. to acquire a sense of public sentiment. There are a handful of specific message formatting devices (particularly hashtags) which are created and subsequently adopted by contributors to indicate a relationship with existing messages and conversations, and which enable some degree of analysis of sentiment. Here I show how the mechanisms presented in this paper can be used to implement a service which allows users to both contribute their opinions, and obtain aggregate information helpful in assessing contributor sentiment.

First, I assume that potentially relevant contributors are somehow invited to participate in discussions. Once invited to contribute, a contributor can *propose* a message by simply typing it. Because of the nature of this service, a contributor of a message is also simultaneously a client who gets to see other messages. On proposing a message, the service shows the contributor (now also a client) a list of existing messages in the discussion which contain keywords from the proposed message (or are similar in a more meaningful way). At this point, the contributor decides whether to proceed with contributing their message as a new message to the discussion or to add support to one of the existing messages. This decision is made by the contributor by voting for either their own message or one or more existing messages, or both, by distribution their 1.0 vote among them.

Figure 3.14 shows the service graphically and Figure 3.15 presents my code implementing it. To define events, I abstract the observation of a user's contribution as an event sensed by the keyboard sensor. I use three events – `msgSentEvent`, `createDiscEvent` and `addMsgEvent` – corresponding to the three types of activities a user can engage in.

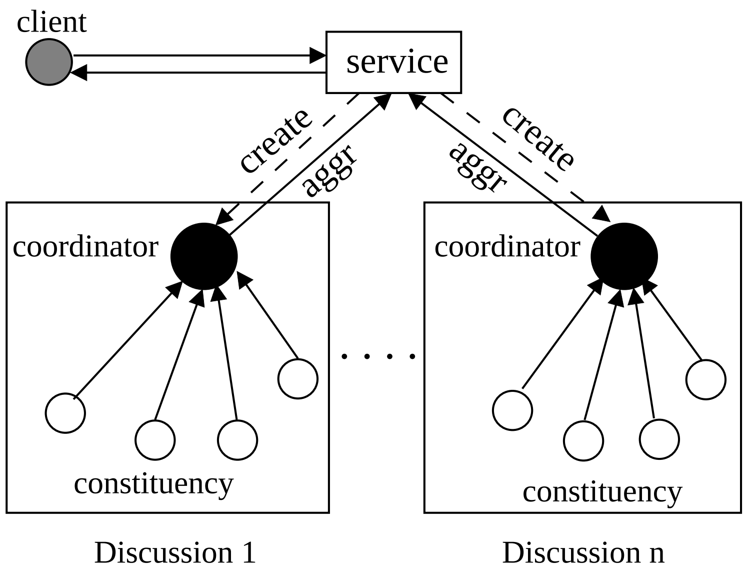


Figure 3.14: Twitter-like Messaging Service

Next, the set of events to be reported to the coordinator is defined. A `createDiscEvent` fires when a contributor sends a request to create a new discussion with the identified constituency `discConstit`. When the service receives this request, it assigns a new discussion ID to identify the discussion topic by, and creates a dedicated discussion coordinator for that discussion, and sets up an `updateInterval` specifying the lengths of the intervals after which the service would receive updates from the coordinator. The coordinator in turn announces the discussion to contributors (`SvcEnum.inviteConstit`). Once invited, the contributors are free to send messages to the discussion coordinator in the form of asynchronous messages. An `addMsgEvent` fires when the service receives a (`msg`) from a contributor. This `msg` is either a new message drafted by the contributor, or an existing message previously sent to the service. On receiving a `msg`, the discussion

```
void createMessageService()
{
    /* initialize service variables */
    int defaultSamplingRate = SensorManager.SENSOR_DELAY_NORMAL;
    int aggrMethod = SvcEnum.msgRanking;

    /* define a service event */
    ServiceEvent createDiscEvent = new ServiceEvent(SvcEnum.msgEvent,
     new List<EventParam>(){
        createParam(``trigger'', SvcEnum.rcvReq),
        createParam(``argument",``constituency'', discConstit),
        createParam(``argument",``updateInterval'', 1),
        createParam(``output'', SvcEnum.inviteConstit)});

    /* define a service event */
    ServiceEvent addMsgEvent = new ServiceEvent(SvcEnum.msgEvent,
     new List<EventParam>(){
        createParam(``trigger'', SvcEnum.rcvMsg),
        createParam(``argument",``message'', msg),
        createParam(``output'', SvcEnum.updateList)});

    /* define a service event */
    ServiceEvent voteEvent = new ServiceEvent(SvcEnum.msgEvent,
     new List<EventParam>(){
        createParam(``trigger'', SvcEnum.sendVote),
        createParam(``output'', SvcEnum.msgText),
        createParam(``output'', SvcEnum.msgWeight)});

    /* create and launch the service */
    CrowdService service = new CrowdService(title, description, samplingRate,
    new List<ServiceEvent>(){voteEvent,createDiscEvent,addMsgEvent},aggrMethod);
    service.launch();

}
```

Figure 3.15: Twitter-like Messaging Service

coordinator first updates the ranked message list (`SvcEnum.updateList`) to reflect the new message received, and then checks to see if it is time to aggregate received messages and report back to the service. When it is time to aggregate,[5] it aggregates the updates and reports them to the server using an update message, which invokes the corresponding method in the server. The server's update method updates the state of the discussion, and then for every entry in the list of service subscribers, sends them the view that they are subscribed to. A `voteEvent` is reported to the coordinator when a new message is drafted by the contributor. A message contains some text (`msgText`) as well as the proportion of the contributor's vote for the message, `msgWeight`. Each contributor has a total of 1.0 vote for any discussion, which they are free to distribute between various messages under that discussion.

[5]If messages are infrequent, a clock is used by the service to interrupt the coordinator at the end of each interval.

3.4.2 Experimental Evaluation

I experimentally evaluated CSSWare regarding performance, scalability and energy efficiency. My experiments were conducted on a prototype Actor-based implementation of CSSWare. On the contributor side, I used a Samsung Galaxy Note II phone with a 1.6GHz quad-core processor and 2GB of RAM running Android OS ver 5.1. The server ran on a Windows 7 laptop equipped with a 2.6GHz quad-core Intel i7 processor and 8GB of RAM.

I installed instrumentation in the server and mobile application (i.e., contributor and client) parts of my prototype restaurant recommendation service to measure the processor time was taken to perform various tasks. Instrumentation was also added to the contributor side to measure the energy consumption of sensing.

Each experiment presented in this chapter is carried out for ten trials, then I took the average of these trials' results.

Performance and Scalability

Service Platform Processing Demand. To evaluate the scalability of the server, I measured the resources required to host a service.

I created and launched a set of instances of the previously described restaurant recommendation service with their required frequencies of event feeds distributed over a normal distribution function. Specifically, I picked 150 random values with an average of 6.7 (SD 3.45), with the intension to be added up to 1,000.[6] I created 150 services with the randomly chosen feed frequency requirements, adding up to a cumulative feed frequency of 1,000 feeds per second. Each service received feeds from 10 restaurants. Note that the event feeds here are feeds of higher level events detected at the contributor end; these are not the raw data received at a high frequency from the sensors. In other words, the average frequency of 6.7 events per second per service would mean that something interesting is observed at some contributor device related to the service at the rate of 6.7 per second. Furthermore, I used a window size of 20 for recently received feeds for any window; this is the number of recent feeds which were used to compute a score for the restaurant. For this local aggregation, I simply maintained the average wait time for the restaurant, which required O(1) amount of time to maintain. These local aggregates for restaurants fed into the creation of a global aggregate in the form of a ranked list of the restaurants based on their scores, which amounted to a single step of *insertion sort* to maintain a sorted list, with an O(n) cost.[7]

Table 3.2 separately shows the one-time processing costs involved in the creation of a new service as well as on-going processing costs as each event feed is received and processed. Creating service and coordinator actors – the former also including parsing the service's metadata (i.e., title and description) and adding the new service to the published service list – took 13.04ms and 11.67ms on average, respectively. Initializing the

[6]I manually changed two average frequencies in order to make the 150 values precisely add up to 1000.

[7]Although this performs well for the small number of restaurants, it would be more efficient to use a binary search tree to keep a large number of restaurants sorted.

global view for the service required 7.84ms. In terms of on-going costs, receiving and parsing an incoming event feed required 7.35ms on average. The cost of local aggregation to keep track of the average of the last 20 waiting times for a restaurant was 0.024ms on average. This aggregation has $O(1)$ complexity. As shown in the table, I also measured costs for $O(\log n)$, $O(n)$ and $O(n^2)$ complexity local aggregation functions. These aggregation functions implemented different algorithms which have these time complexities. Particularly, I used a binary search tree to keep the last 20 waiting times sorted, which has $O(\log n)$ complexity; I implemented a quick select algorithm to find the median value of the last 20 waiting times, which has $O(n)$ complexity; and I used insertion sort to maintain a sorted list of these 20 values, which has $O(n^2)$ complexity. The global aggregation for ranking the ten restaurants incurred an average processing cost of 0.95ms.

Table 3.2: Average Processing Time at the Server Side in ms

One-Time Per-Service Costs	**Mean**	**SD**
Create a service actor	13.04	2.63
Create a coordinator actor	11.67	1.74
Create a service view	7.84	0.98
Total processing time	32.55	5.35
Per-Event-Feed Costs	**Mean**	**SD**
Process an event feed	7.35	1.11
Local aggregation ($O(1)$ cost)	0.024	0.0021
Local aggregation ($O(\log n)$ cost)	0.078	0.0083
Local aggregation ($O(n)$ cost)	0.280	0.0349
Local aggregation ($O(n^2)$ cost)	0.680	0.0987
Global aggregation (10 Restaurants)	0.95	0.17
Total processing time ($O(1)$ local aggregation)	8.325	1.28

To put these numbers in some context, given the 8.325ms required per feed on an on-going basis, about 120 event feeds could be processed by a server of my configuration per second. This could support a single service where 120 events are being collectively detected by the contributors every second, or 10 services which are each receiving about 12 feeds per second on average, and so on. In a broader context still, assuming 40% of the population dines out at a meal time,[8] assuming the diners are distributed somewhat evenly over a period of two hours, and each diner's device is sending 3 events over the course of their meal (indicating arrival, seating, departure) a server of my modest configuration could process 288,288 diners' data, equivalently data for a city of about 720,720 people. In practice, data from a small fraction of the diners could be used, allowing service for an order of magnitude higher population.

That said, the global aggregation function assumed only ten restaurants. Although this may be reasonable because individuals requiring restaurant recommendations are not likely to be close to hundreds of restaurants, narrowing down the selection before aggregation would mean custom global aggregations, each costing the

[8]Zagat 2014 restaurant survey reported that an average American ate out or bought 47% of their lunches or dinners

0.95ms. However, this custom aggregation could happen on the client's own device, without impacting the server's scalability. Alternatively, for a truly global aggregate for a city with (say) 10,000 restaurants, an $O(\log n)$ binary search tree could be used to keep the restaurants sorted; only the top few would ever need to be fetched, limiting the fetching cost.

Contributor Processing Demand. On the contributor side, again, I separately measured the initial cost of handling a new service's request for a contribution, as well as the on-going cost of serving the service.

The average total of measured one-time cost was 53.35ms (SD 3.41). The on-going costs measured were per sensor feed: every time a piece of raw data was received from a server, its average total processing cost amounted to 8.68ms (SD 1.02). A finer breakdown of this total is presented in Table 3.3.

Table 3.3: Average Processing Time at the Contributor Side in ms

One-Time Per-Service Costs	**Mean**	**SD**
Process a service invitation	32.51	2.32
Initiate service	20.84	1.09
Total processing time	53.35	3.41

Per-Sensor-Feed Costs	**Mean**	**SD**
Schedule samples	2.81	0.45
Filter a sensor feed	4.16	0.49
Detect a service's event	1.71	0.08
Total processing time	8.68	1.02

To put this on-going cost in perspective, about 115 sensor feeds per second could be handled on a device of my configuration (assuming no other computations are executing). If an average service requires as many as 10 data samples per second (from a variety of sensors), 11.5 of such services could be supported; if an average of 1 data sample per second is required per service, a more likely scenario, 115 services could be simultaneously contributed to.

Client Processing Demand. For the client side as well, I measured the one-time processing costs of accessing a new service, as well as the on-going costs of receiving updates.

As shown in Table 3.4, the average total of measured one-time costs was 35.53ms. The total of measured per-refresh on-going costs amounted to 60.9ms on average, with 28.7ms (SD 3.9) for processing the update, and 32.2ms (SD 6.4) for display. In other words, a client could be simultaneously subscribed to and receive updates from 16 services every second. This is not very meaningful considering that more than half of the processing cost is for graphically displaying the update, which is not likely to happen simultaneously for more than only a few services. If I assume that only one service's updates are actually displayed at a time, more than 30 services could be supported in the background where interesting updates could lead to notifications, invitations to display, etc.

Table 3.4: Average Processing Time at the Client Side in ms

One-Time Per-Service Costs	**Mean**	**SD**
Parse a service notification	21.32	2.21
Subscribe to a custom view	14.21	1.84
Total processing time	35.53	4.05

Per-View-Update Costs	**Mean**	**SD**
Parse a service update	28.7	3.9
Display a service view	32.2	6.4
Total processing time	60.9	10.3

Energy Consumption of CSSWare vs. Standalone Services

Finally, a set of experiments was carried out to measure the overall improvement achieved in energy consumption by using CSSWare's sampling scheduler on the contributor device. I used the PowerTutor software [64] for my energy measurements.

To measure the overall improvement in energy consumption, I made measurements of energy used by CSSWare and identical standalone services implemented without using CSSWare. Table 3.5 shows the total amount of energy used by each sensor for the entire experiment duration. As shown in Table 3.5, the sampling scheduler improved energy consumption of accelerometer and gyroscope sensors by up to 24.60% and 26.63%, respectively. However, the percentage savings depend entirely on the number of requests being served, because although the energy used is roughly linear in the cumulative sampling rate of all requests for the standalone services, for CSSWare, it depends almost entirely on the highest frequency being requested at the time, from which other requests are also served.

Table 3.5: Energy Consumed using CSSWare vs. Standalone Services in mJ

Sensor	**Standalone**		**CSSWare**	
	Mean	**SD**	**Mean**	**SD**
Accelerometer	2,646	132.3	1,995	102.6
Gyroscope	14,653	761.4	10,751	645

Overhead Analysis. To determine the non-sensing overhead of CSSWare, I measured the energy consumed by the contributor device side of the framework, albeit without the actual sensing. The average energy consumed was measured to be 70.4 mJ for the accelerometer, and a similar 79.6 mJ for the gyroscope sensor. In percentage terms, this was roughly 4% of the total energy consumed in the accelerometer experiments, and 0.8% for the gyroscope sensor, the difference explained by the order-of-magnitude larger overall energy demand of the gyroscope sensor itself.

Chapter 4
ShareSens: An Approach for Energy-Efficient Mobile Sensing

This chapter presents ShareSens, an approach for efficiently supporting the sensing needs of mobile distributed services. ShareSens is implemented as an Android API for requesting sensor feeds, which enables sharing of overlapping feeds between services. This chapter is organized as follows: Section 4.1 and 4.2 present the design and implementation of ShareSens respectively. Section 4.3 presents my evaluation of ShareSens.

4.1 ShareSens Design

ShareSens also uses the Actor model [6], which is an increasingly influential model for concurrent systems. My design of the ShareSens is shown in Figure 4.1. An optimizing *sampling scheduler* is used to schedule the sampling of each sensor based on the sensing requirements received from apps being served at the time. The scheduler attempts to optimize the sampling rate of each sensor exploiting opportunities for different apps to share sensor samples when possible.

ShareSens handles two types of sensing requests, fixed – specified using a single required sampling rate – and flexible – specified using a range of acceptable rates, from the lowest acceptable rate to the highest rate which, if available, can be made use of. Furthermore, because the approach is most beneficial when multiple requests can be merged to serve the requests simultaneously, my prototype implementation makes further assumptions about the nature of requests, which will be discussed below.

Even though fixed sampling rate requests are essentially subsumed by flexible rate requests, there are advantages to handling sets of fixed-only rate requests separately. This is primary because flexible rate requests present an additional opportunity to serve applications at rates above their minimum requirements, which adds a degree of complexity. When all requests are for fixed rates, that complexity can be avoided.

When handling fixed-only rate requests, on receiving a new request, ShareSens's scheduler checks if the current sampling rate – sufficient for serving all currently served requests – can also satisfy the new sampling rate being requested. To ensure that the data samples are equidistant in time, the scheduler needs to compute the *least common multiple* (LCM) of the sensor's current sampling rate and the rate being requested, to determine a rate that would serve both existing requests and the new request. This, however, presents a challenge.

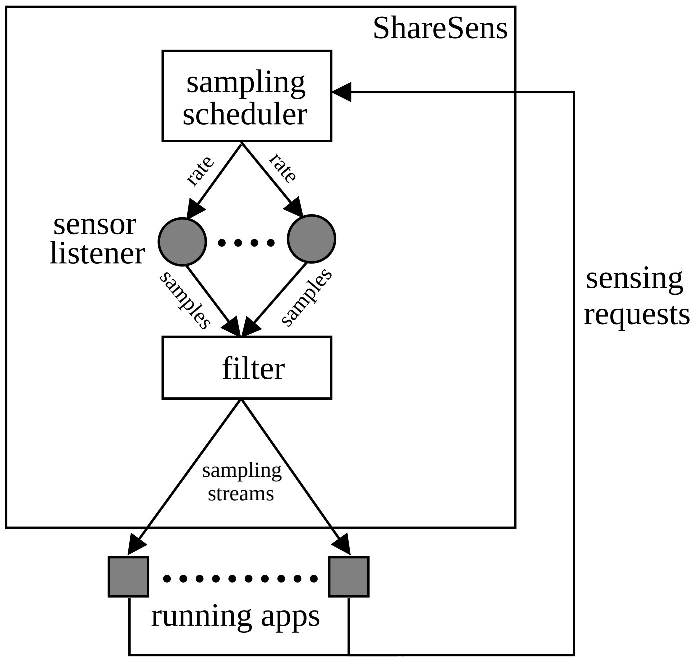


Figure 4.1: ShareSens Architecture

Because of the way LCM is defined,[1] it is only in special circumstances that $LCM(x, y)$ is lower than the sum of x and y, meaning that using the LCM of two requested sampling rates would rarely be better than serving them independently. Particularly, the LCM of two numbers is lower than their sum when the numbers are equal or one is the multiple of the other. For this reason, finding a covering sampling rate is most beneficial when the sampling rates requested by applications are in such a relationship among themselves. It turns out that this is possible to achieve by setting up preferred sampling rates for applications to pick from – as the Android Sensor API already does by offering $5Hz$, $16Hz$, $50Hz$, $100Hz$ with designated purposes of UI, normal, game, highest, respectively[2] – so that they are of the form $x * 2^n$, making the LCM of any two requested sampling rates simply the larger of the two. In the prototype, I use $10Hz$, $20Hz$, $40Hz$, $80Hz$; however, we can easily add a lower rate of $5Hz$, or change the scheme to begin with $12.5Hz$ so that $100Hz$ can be served.

Once the sampling rate to be used has been decided by the scheduler, it is passed on to the sensor listener for sampling sensor data at that rate. The sensor feed is then received by a dedicated *filter* for the sensor which extracts from the feed the samples required for each application.

4.1.1 Fixed-Only Sampling Rate

Algorithm 1 shows the steps taken by ShareSens in handling fixed-only sampling rate requests. Each sensing request specifies the sensor s to be sampled and the rate r at which it should be sampled. When a new

[1] $LCM(x, y) = (x * y)/GCD(x, y)$; and $LCM(x, y, z) = LCM(LCM(x, y), z)$
[2] These use-case focused sampling rates are in addition to the ability to request specific sampling rates.

request is received, the scheduler checks if the sensor is already scheduled to serve other requests. If not, it sets it up to be schedule from then on. If the sensor is already being scheduled to serve existing requests, the scheduler determines the rate of sampling which will be sufficient for also serving the new request. Assuming that requests are limited to the preferred sampling rates, this simply requires comparing the new request with the existing sampling rate, and picking the larger of the two. In general, it would require a more complex process computing LCMs of different numbers of requests to be served, and check if serving them independently would be more efficient than serving them from a covering rate. Because of the computational complexity of this method, as well as the low likelihood that the LCM of requests would be lower than their sum – let alone the LCM of all requests – my implementation and experiments assume that requests pick from the preferred sampling rates of $10Hz$, $20Hz$, $40Hz$ and $80Hz$.

The computation involved in this algorithm only has fixed costs with respect to the number of requests being served.

Algorithm 1 Fixed-Only Sampling Rate Request Handling Algorithm

```
1:  procedure SENSOR SCHEDULING($s, r$)                          ▷ sensor name (s) and sampling rate (r)
2:      if $\neg SamplingScheduler.isSensorFound(s)$ then
3:          $SamplingScheduler.add(s, r)$;                             ▷ add $s$ to the scheduler
4:          $ShareSensManager.createSensorListener(s)$;   ▷ create a new sensor listener actor for $s$
5:          $ShareSensManager.createSensorFilter(s)$;            ▷ setup a sensor filter actor for $s$
6:      else                                                        ▷ $s$ is already scheduled
7:          if ($r > SamplingScheduler.currentRate$) then
8:              $SamplingSchedule.adoptSamplingRate(s, r)$;         ▷ adopt the sampling rate to $r$
9:          end if
10:     end if
11: end procedure
```

4.1.2 Flexible Sampling Rate

Algorithm 2 shows the steps taken by ShareSens in handling a flexible sampling rate request. Flexible requests come in the form of a range of possible sampling rates rather than just one. Particularly, there is a low rate, which is the lowest rate required by the application, and there is a high rate, which is the maximum rate which the application can take advantage of when available. The implied intent is that although the application minimally requires the low rate, if a higher rate is available because of other requests being served, the application would like to receive it.[3]

This is achieved as follows. As in the fixed case, if the sensor is currently not serving requests, it is asked to begin collecting data at the low rate. However, if it is already sensing, the current rate is compared with

[3]In a related project, I am exploring a pricing model for sensing requests in a *Sensing as a Service* context, where requesting applications/services could share the expense of receiving a higher sensing rate.

the requested low rate to see if it readily serves the request (i.e., whether the requested low rate is less than or equal to the current sampling rate). If it does not, the low rate is adopted as the new sampling rate for the sensor. Recall that this works because I assume that the sensing rates pick from the preferred rates, and the higher of two sampling rates is always also the LCM of the two. Next, the scheduler checks whether this raising of the rate has created opportunities for existing requests to be served at higher rates, and sets up the filter to do exactly that. Alternatively, if the low rate of the new request is already served by the current sampling rate, the scheduler looks for the opportunity to serve the new request at the highest rate already available.

Finally, note that the advantage of applications picking only from the preferred sampling rates is even more pronounced for the flexible case because the complexity of finding the lowest common multiple (LCM) for the range of rates acceptable to each application would be very high – significantly increasing the computational cost of accommodating every change in sampling rate requests.

In terms of time complexity, when the `for` loop is executed to look for opportunities for serving existing requests a higher rate, *O(n)* amount of time is taken, where n is the number of requests currently being served; the rest of the computation takes constant time with respect to the number of requests.

Algorithm 2 Flexibile Sampling Rate Request Handling Algorithm

```
1:  procedure SENSOR SCHEDULING($s, l, h$)          ▷ sensor name (s) and sampling rate range (l,h)
2:      if $\neg SamplingScheduler.isSensorFound(s)$ then
3:          $SamplingScheduler.add(s, l)$;
4:          $ShareSensManager.createSensorListener(s)$;
5:          $ShareSensManager.createSensorFilter(s)$;
6:      else                                        ▷ $s$ is already scheduled for sampling
7:          if ($l > SamplingScheduler.currentRate$) then
8:              $SamplingSchedule.adoptSamplingRate(s, l)$
9:              for each $request$ in $SamplingScheduler.getRequestList(s)$ do
10:                 if ($request.h > SamplingScheduler.currentRate$) then
11:                     $ShareSensFilter.setRate(request, SamplingScheduler.currentRate)$;
12:                 else
13:                     $ShareSensFilter.setRate(request, h)$;   ▷ set the filter to provide $h$ to application
14:                 end if
15:             end for
16:         else if ($h > SamplingScheduler.currentRate$) then
17:             $ShareSensFilter.setRate(request, SamplingScheduler.currentRate)$;
18:         else
19:             $ShareSensFilter.setRate(request, h)$;
20:         end if
21:     end if
22: end procedure
```

Discussion

I assume that requests need to begin being served immediately, and that applications have no tolerance for stale data.

Under these assumptions, for ShareSens to be beneficial, there must exist a covering sampling rate which is better than the sum of the sampling rates being requested. As I have explained above, this happens most readily if the requests are for sampling rates of the form $x * 2^n$ (or ranges defined in terms of them). When the covering sampling rate is not better than the sum of the requested sampling rates, it would be better to serve the requests separately.

That said, if there is flexibility in when the sensing takes place, or if the applications can tolerate some delay between when data is collected and when it is made available to the application, that would create opportunities for scheduling sensing tasks to increase sharing of data. In both of these scenarios, caching of sampled data would also be beneficial. The extent of these opportunities will depend on the types of sensors involved and the needs of the applications.

4.2 ShareSens Implementation

A prototype of the system described in the previous section has been implemented as an Android app. As Figure 4.2 shows, our implementation is built over Actor Architecture (AA) [42], a Java library and runtime system for programming and executing distributed actor systems.[4] I have ported AA to the Android OS for supporting the mobile app. Android OS was not modified in any way.

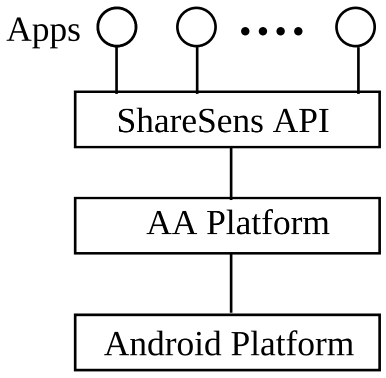


Figure 4.2: ShareSens Platform and APIs

4.2.1 Sensor APIs in Android

Android offers a sensor API to read sensor data, which provides access to event handlers to capture sensor events. Android provides two event handlers through an interface called `SensorEventListener`: an

[4]More precisely, it is an extension of AA for supporting resource-bounded actor computations based on the CyberOrgs model [63]. CyberOrgs also adds mechanisms for resource coordination of actors, which we plan to use in the future.

`onSensorChanged` event is triggered when a sensor reports a new sensor value, and an `onAccuracyChanged` event is triggered when a sensor's accuracy changes.

The `SensorManager` class implements an Android service which provides various methods for accessing and listing sensors, and registering and unregistering sensor event listeners. To start up the sensor service, an app has to retrieve a handle to `SensorManager`, and uses the handle to make registration or unregistration calls for its event handlers.

4.2.2 ShareSens API

Figure 4.3 shows the implementation of the ShareSens API. The `ShareSensManager` class – which extends the Android platform's `SensorManager` abstract class – implements a background service to host actors executing in the system, and defines methods for accessing and managing sensors. Each sensor is encapsulated by a `SensorServiceActor`, which collects data samples collected by the sensor. At runtime, the `ShareSensManager` is responsible for computing the sampling frequency which would satisfy all requests for each sensor's data, and keeps the relevant `SensorServiceActor`s informed about it. Once an application registers a `SensorListener` actor for collecting sensor data, it begins receiving a custom sub-stream of sensor data collected by the `SensorServiceActor` just for itself. The customization is done by a special filter actor – one for each sensor – which knows the requirements of all applications' sampling requests, and accordingly extracts and forwards the required streams from the feed received from a `SensorServiceActor` to applications' `SensorListener`s. A `SensorListener` then passes the feed it receives from the filter to its application. Both control and data communication happen using asynchronous message passing.

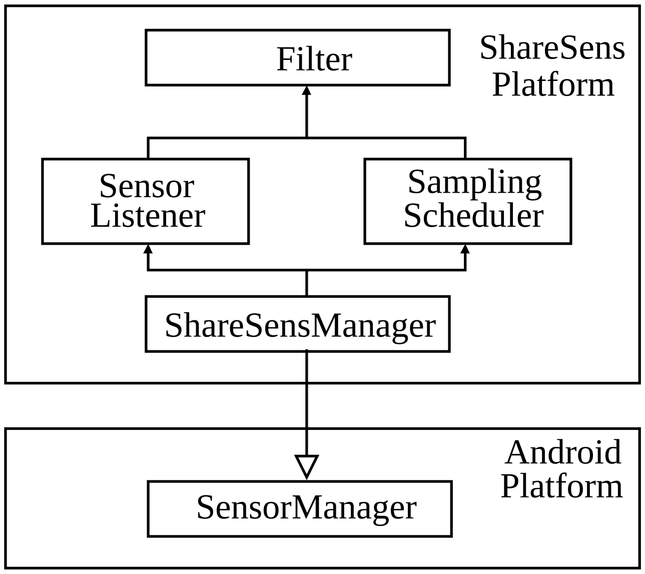


Figure 4.3: ShareSens Class Diagram

As discussed previously in Section 4.1, ShareSens assumes that applications pick from a small number of sampling rates available, selected both to increase opportunity for sharing sampled data as well as to enhance performance of the scheduling mechanism. Particularly, `ShareSensManager` expects the sampling rates requested to be in terms of four values (roughly matching similar choices offered by the Android API): (i) SENSOR_DELAY_NORMAL ($10Hz$) which is suitable for screen orientation changes; (ii) SENSOR_DELAY_UI ($20Hz$) which is suitable for the user interface; (iii) SENSOR_DELAY_GAME ($40Hz$) which is suitable for games; and (iv) SENSOR_DELAY_FASTEST ($80Hz$) which is the fastest rate at which sensor data is provided. I have picked these values for convenience; they can be changed as long as they have the desired property described in Section 4.1 to facilitate sharing of sampled data. Note also that my approach aims to deliver samples to applications collected at equal intervals.

ShareSens handles both fixed and flexible sampling requests, the latter specified in terms of a range of sampling rates. Figure 4.4 shows a code snippet illustrating how a fixed sampling rate would be requested by an application. The code implements `ShareSensListener` and uses `ShareSensManager` to register an accelerometer sensor with the normal sampling rate (`SENSOR_DELAY_NORMAL`). A reference to the sensor server is obtained to identify the sensors available on the device. This requires creation of an instance of the `ShareSensManager` class by calling the `getSystemService()` method and passing it the `SENSOR_SERVICE` argument. A reference to the accelerometer sensor can be obtained using the getDefaultSensor() method by passing in the type constant for a specific sensor (`Sensor.TYPE_ACCELEROMETER`).

```
public class ShareSensFixedRateExample implements ShareSensListener
{
  private ShareSensManager mShareSensManager;
  private Sensor mAccelerometer;

  @Override
  public ShareSensFixedRateExample() {
   mShareSensManager=(ShareSensManager)getSystemService(Context.SENSOR_SERVICE);
   mAccelerometer = mShareSensManager.getDefaultSensor(Sensor.TYPE_ACCEL);
   mShareSensManager.registerListener(this, mAccelerometer, ShareSensManager.
     SENSOR_DELAY_NORMAL);
 }

  @Override
  public final void onSensorChanged(SensorEvent event) {
   // Do something with the sensor event.
  }

  @Override
  public final void onAccuracyChanged(Sensor sensor, int accuracy) {
   // Do something here if sensor
   // accuracy changes.
  }
}
```

Figure 4.4: ShareSens API Usage Example for Fixed-Only Rate

Similar to the sensor APIs in Android, raw sensor data is monitored by implementing the `onAccuracy-`

`Changed()` and `onSensorChanged()` callback methods provided through the `ShareSensListener` interface.

Programming of flexible sampling requests is illustrated in Figure 4.5. Here, `ShareSensManager` allows the programmer to specify a range of sampling rates by stating the low and high rates. The code shown in Figure 4.5 is very similar to the one for the fixed sampling rate (Figure 4.4), except that the `registerListener` method is now overloaded (see line 12) to take the low and high sampling rates, `minSamplingRate` and `maxSamplingRate`. Again, the rates which can be requested are assumed to be from the preferred ones. In other words, all preferred rates between the low and high values are meant to be included in the range.

```
1  public class ShareSensFlexRateExample implements ShareSensListener
2  {
3    private ShareSensManager mShareSensManager;
4    private Sensor mAccelerometer;
5    private int minSamplingRate = 20;
6    private int maxSamplingRate = 40;
7
8    @Override
9    public ShareSensExample() {
10     mShareSensManager = (ShareSensManager)getSystemService(Context.
         SENSOR_SERVICE);
11     mAccelerometer = mShareSensManager.getDefaultSensor(Sensor.TYPE_ACCEL);
12     mShareSensManager.registerListener(this, mAccelerometer, minSamplingRate,
         maxSamplingRate);
13   }
14
15   @Override
16   public final void onSensorChanged(SensorEvent event) {
17     // Do something with the sensor event.
18   }
19
20   @Override
21   public final void onAccuracyChanged(Sensor sensor, int accuracy) {
22     // Do something here if sensor
23     // accuracy changes.
24   }
25 }
```

Figure 4.5: ShareSens API Usage Example for Flexible Rate

4.3 ShareSens Evaluation

The benefit of using ShareSens is relatively obvious: a lower cumulative sampling rate would lead to power savings related to the sampling. In this context, I evaluated ShareSens with two goals in mind. First, to confirm the opportunity for offering higher sampling rates “for free” when sampling request rates are flexible. Second, and more significantly, to assess the cost of using ShareSens, and to establish its limits: when does ShareSens stop being beneficial?

Precisely because of the opportunistic nature of the advantage of using ShareSens – there are savings only when there is an opportunity to share sensing data – I decided against using complete applications for evaluating it. The rationale for this decision was that it would be difficult to sufficiently control the sensing requirements of real(istic) applications for controlled experiments. Also, in order to obtain meaningful results, we would have to install instrumentation to isolate the situations where the opportunities for shared sampling did exist. More importantly, these efforts would be aimed at first creating and then isolating situations which are of interest, defeating the point of having the real(istic) applications in the first place. An evaluation approach explicitly built around the interesting situations – opportunities for sharing sensed data, and transitions between these opportunities – would achieve the evaluation goals more effectively. I chose the latter approach.

I carried out a number of experiments with both fixed and flexible sampling requests, with multiple goals: to confirm the expected reduction in sensors' sampling rates as a result of using ShareSens, with concomitant reduction in energy consumption; to confirm the opportunity to serve flexible requests of applications at sampling rates higher than the lowest rates; to determine the energy footprint of using ShareSens, and finally to establish whether using ShareSens stops being beneficial beyond a certain frequency of changes in requests by applications.

4.3.1 Experimental Setup

My experimental work was done on a Samsung Galaxy Note II phone running Android OS version 5.1,[5] and executing the prototype implementation of ShareSens.

Because of the opportunistic nature of the advantage of using ShareSens, the experiments are designed primarily to establish that (a) when there is an opportunity presented by overlapping sensing requests, energy saving actually materialize, and (b) when the opportunity does not exist, my approach does not do worse than Android's API.

I artificially generated a set of sensing loads to simulate application requests for continuous sensing for two different sensors, the accelerometer and the orientation sensor. Specifically, I used three loads which I refer to as light, normal, and heavy. For each experiment, measurements were carried out over the period of 60 seconds, where fresh requests could arrive at 5-second points. With the exception of one low rate sensing request active over the entire 60 seconds period, other requests were inserted at randomly selected points, and each was active for 5 seconds at a time. The maximum number of requests for a certain rate were restricted as well, with fewer requests for higher sampling rates than those for lower rates, implicitly assuming that higher sampling demands – such as for fine-grained health monitoring requiring 80 samples per second – would be less frequent.

Each experiment presented in this chapter is carried out for ten trials, then I took the average of these

[5]This was the current version of Android OS at the time of our experiments, which were carried out in 2015; newer versions of Android OS have not significantly changed the relevant sensing mechanisms.

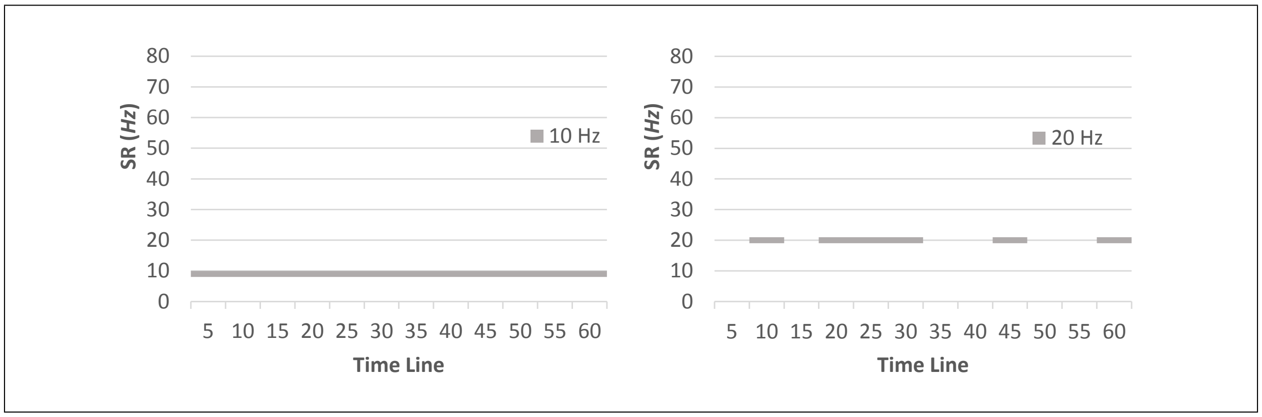


Figure 4.6: Fixed-Only Rate Requests (Light Load)

trials' results.

Fixed Sampling Rate

Figures 4.6, 4.7 and 4.8 show the light, normal and heavy sensing loads used in the experiments for the fixed sampling rate (SR) requests. In each case, there was a 10 Hz request which lasted the entire 60 seconds. Additionally, 20 Hz, 40 Hz, and 80 Hz requests were inserted at randomly selected points, each for 5 seconds at a time. The maximum number of requests of a certain sampling rate were limited depending on the frequency: 6 of 20 Hz, 4 of 40 Hz, and 2 of 80 Hz. The same three sets of measurements were carried out for the accelerometer and the orientation sensors.

Flexible Sampling Rate

Figures 4.9, 4.10 and 4.11 show the light, normal and heavy sensing loads used in the experiments for the flexible sampling rate (SR) requests. The loads used here are similar to those in the fixed case in terms of when the requests arrive. The main difference is in that the individual requests are for ranges of sampling rates, rather than fixed rate, implying that the requesters would be satisfied with the low rates, but would like higher sampling rates when available for "free." In all loads, there is one request for a sampling rate between 10 Hz and 20 Hz for the entire 60 seconds interval. Additionally, there are requests for ranges of 20 − 40 Hz, 40 − 80 Hz, and 80 − 80 Hz.[6] The maximum number of requests different types are 6, 4 and 2 respectively. The same three sets of measurements were carried out for the accelerometer and the orientation sensors.

[6]Note how a fixed request can be represented as in 80 – 80 Hz. Although flexible requests conceptually subsume fixed requests, collections of fixed-only requests can be implemented more efficiently because of the absence of opportunity to serve requests at higher rates than the low rates in the ranges.

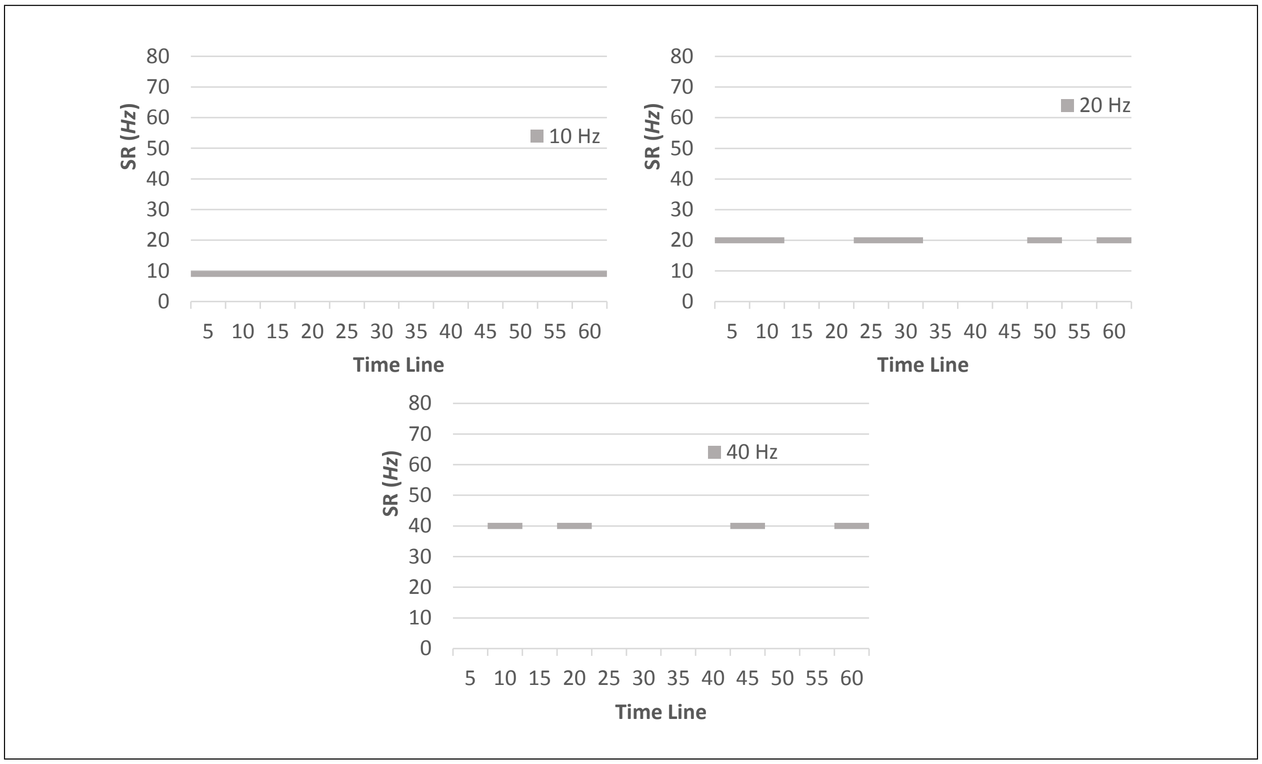


Figure 4.7: Fixed-Only Rate Requests (Normal Load)

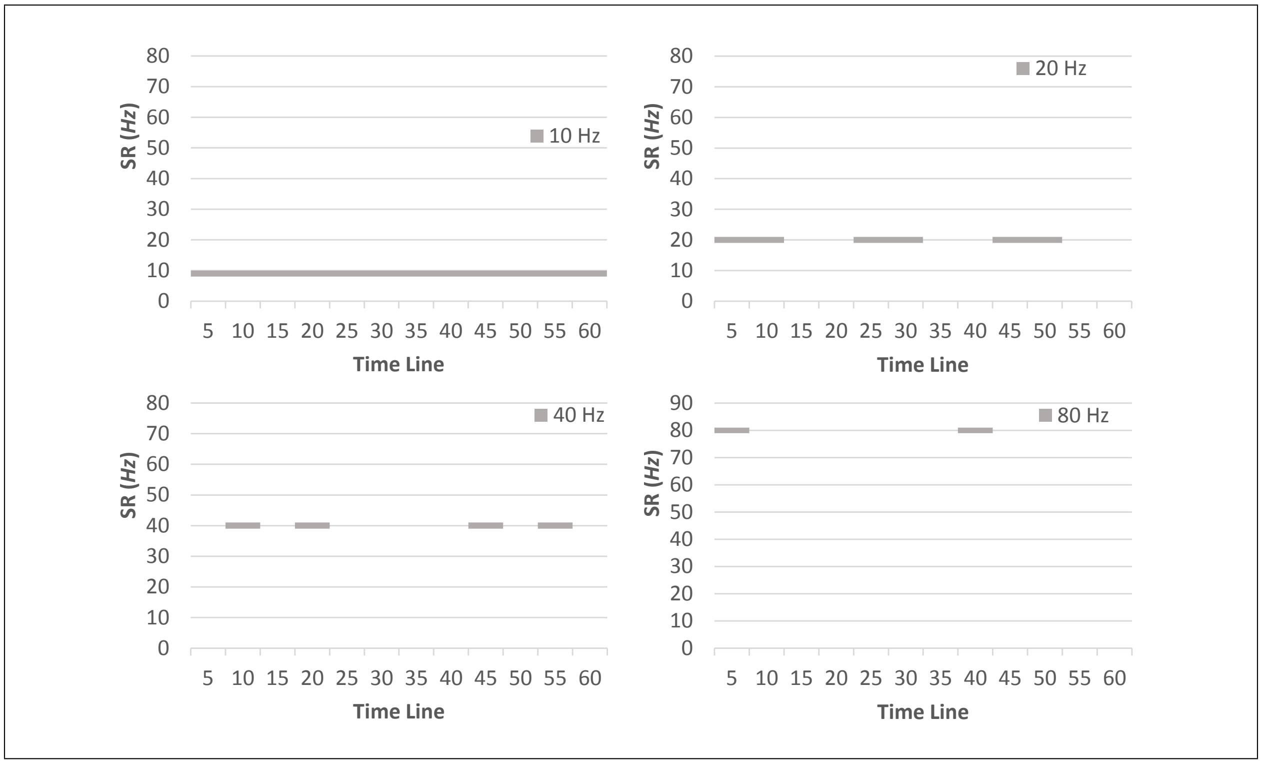


Figure 4.8: Fixed-Only Rate Requests (Heavy Load)

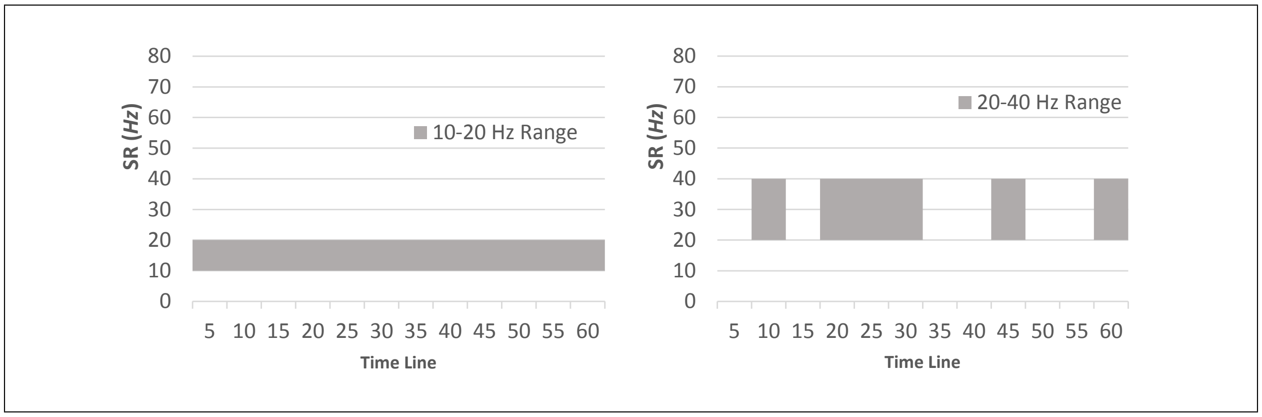


Figure 4.9: Flexible Rate Requests (Light Load)

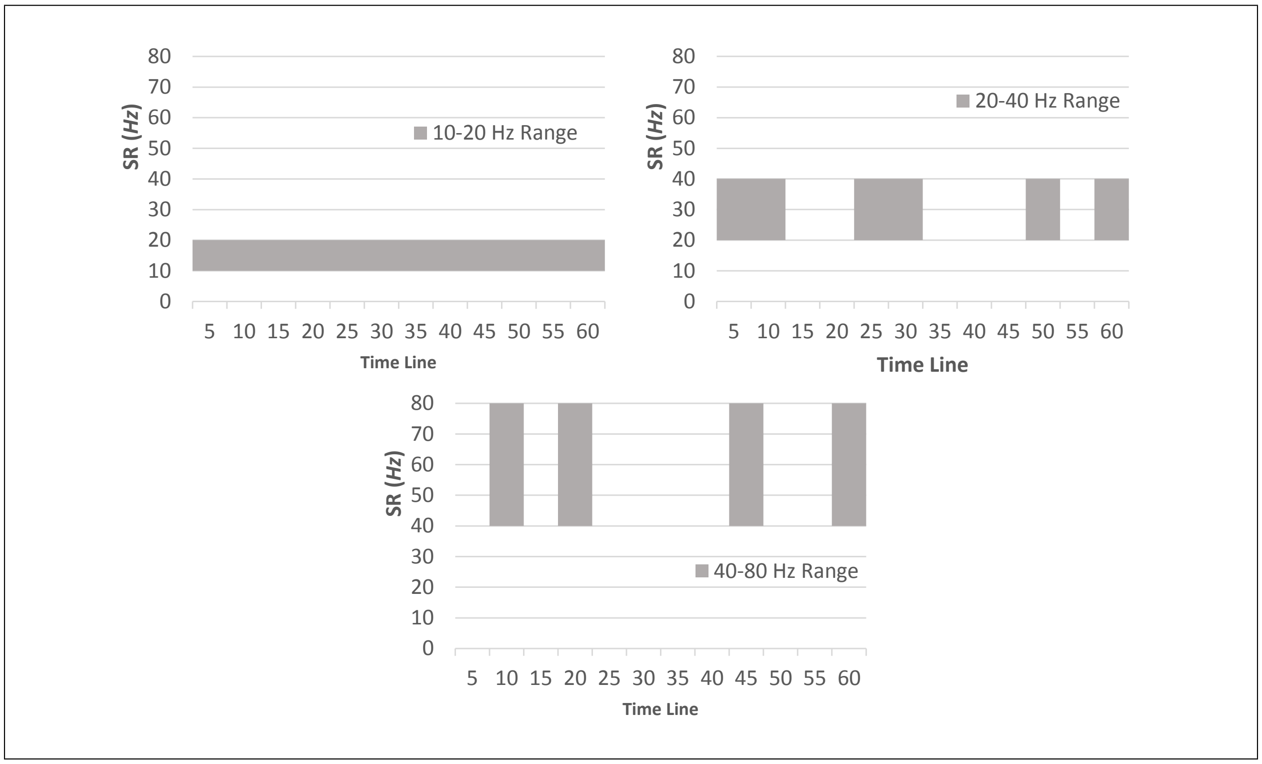


Figure 4.10: Flexible Rate Requests (Normal Load)

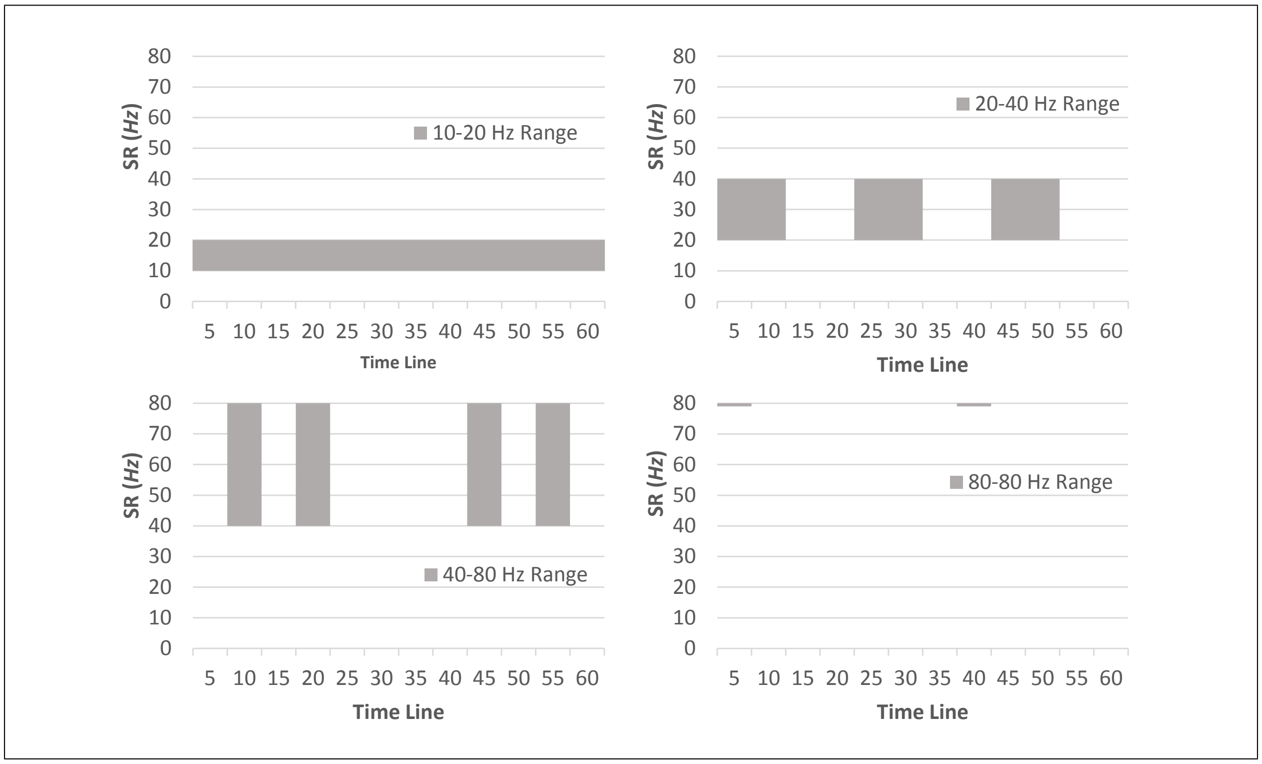


Figure 4.11: Flexible Rate Requests (Heavy Load)

4.3.2 Opportunistic Raising of (Flexible) Sampling Rates

For the case of flexible sampling requests, I collected data about when requests were served at a higher sampling rate than the low rate in the request. Recall that flexible requests specify a range of acceptable sampling rates, with the intent of specifying the minimum required rate, and the highest rate which the requesting application may be able to take advantage of. At times when different requests are being served with different low (i.e., required) sampling rates, the rate used for sampling data would be the highest of these low rates. Consequently, there is an automatic opportunity to serve those with lower low rates with a higher sampling rate. Figures 4.12, 4.13, and 4.14 show the actual sampling rates which were served in my experiments for the flexible rate requests.

4.3.3 Energy Consumption of ShareSens vs. Android's Sensor API

For this set of experiments, I used the PowerTutor software [64] for power measurements.

As intended, for the fixed-only sampling rate experiments, the actual rate at which ShareSens sampled a sensor was always the highest requested rate (in accordance with Algorithm 1 presented in Section 4.1). For the flexible rate experiments, as well, I observed the actual sampling rate to be correct – i.e., the highest low rate requested – in accordance with Algorithm 2. Figure 4.15 compares the actual sampling rates used by

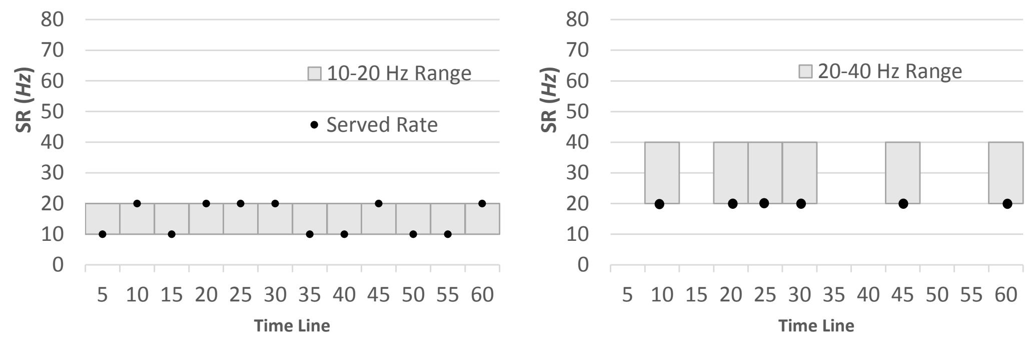


Figure 4.12: Opportunistic High Sampling Rate for Flexible Request (Light Load)

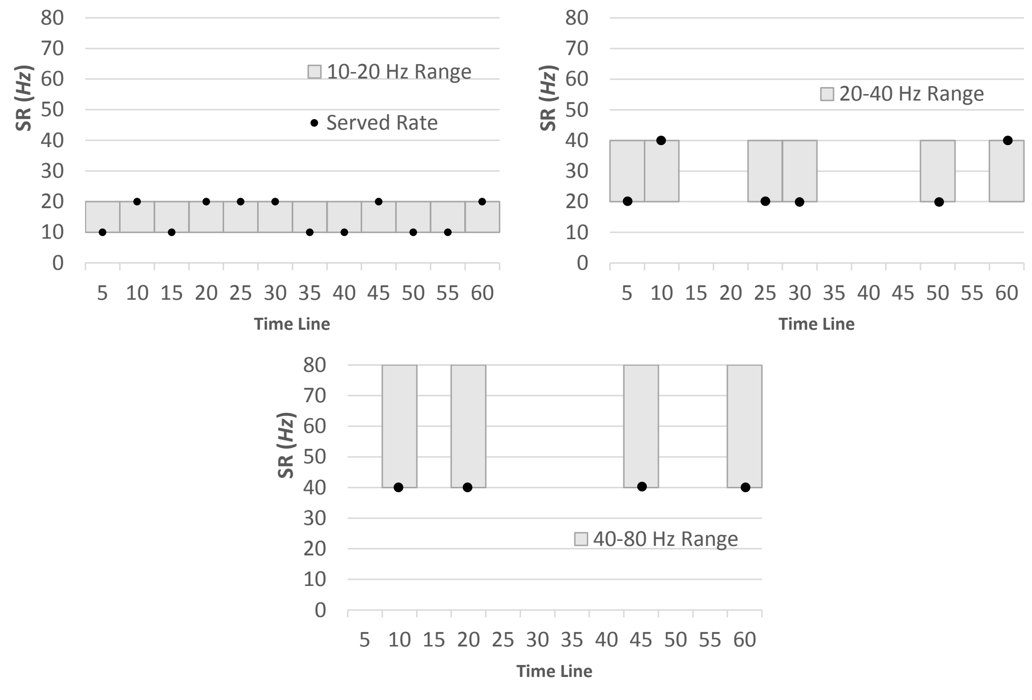


Figure 4.13: Opportunistic High Sampling Rate for Flexible Request (Normal Load)

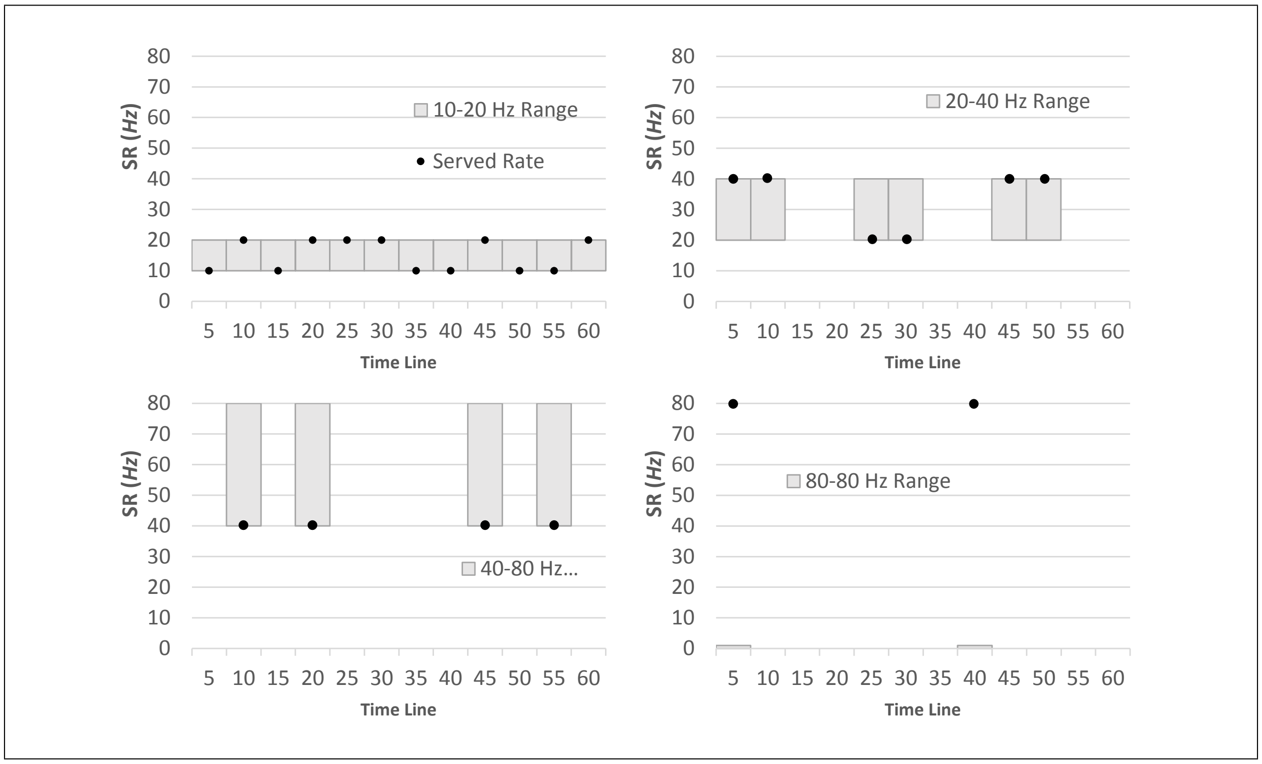


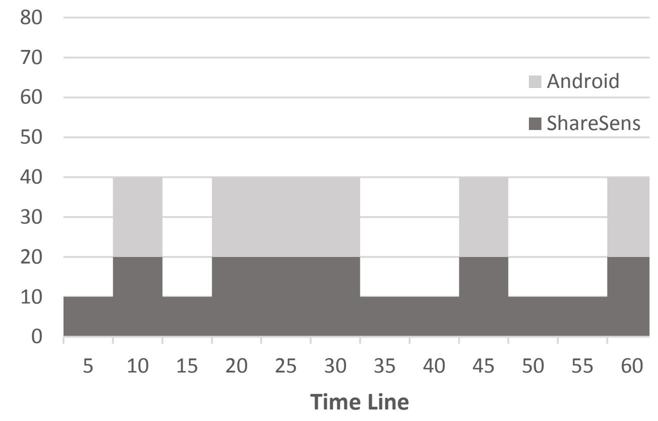
Figure 4.14: Opportunistic High Sampling Rate for Flexible Request (Heavy Load)

ShareSens with those used by the Android API for identical flexible rate loads. Recall that both algorithms assume that sampling requests by applications pick from the preferred sampling rates of the form $10 * 2^n\ Hz$ in the spirit of similar preferred sampling rates offered by the Android API.

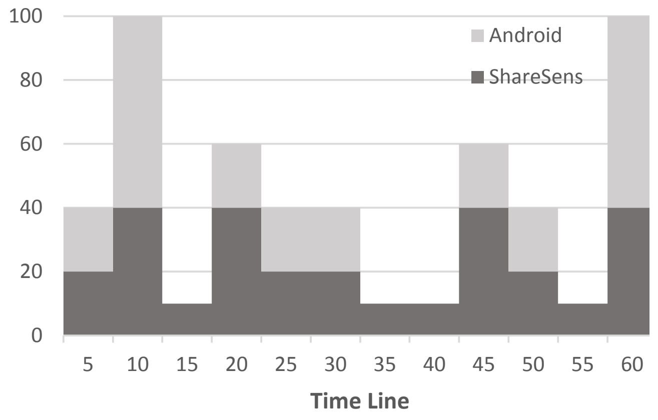
Figures 4.16, 4.17, 4.18 and 4.19 show the power use measurements for the accelerometer and orientation sensor experiments for fixed-only and flexible sampling requests, respectively. ShareSens delivered energy savings as expected, effectively reducing the power use to the requirement for the actual rate at which the sensor was sampled.

Surprisingly, using the Android API led to power consumption that was superlinear with respect to the sum of all sampling rates requested by the concurrent sensing tasks; most notably, in my tests, I found that multiple requests adding up to a cumulative sampling rate of (say) $80Hz$ used less power than a single request for $80Hz$, leading to a negative correlation between the number of independent sensing streams and the power required to serve them, for the small number of streams I tested for.

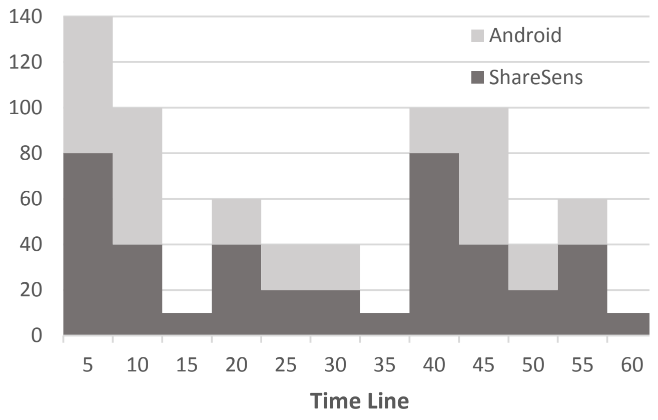
For completeness, in Tables 4.1 and 4.2, I present the total amount of energy used for the entire 60 seconds sensing loads used in my experiments. The percentage saving in the energy consumed when using ShareSens (in comparison with the Android API) depends entirely on the number of requests being served, and the difference between the sampling rates they request. This is because although the energy used is roughly linear in the *cumulative* sampling rate of all requests for the Android API, for ShareSens, it depends



(a) Light Load

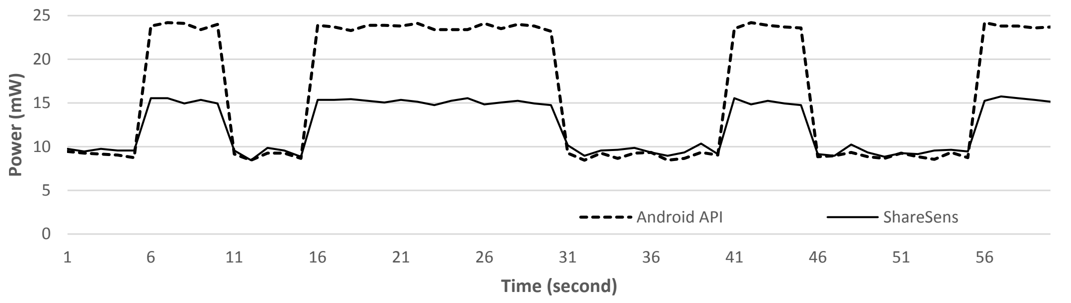


(b) Normal Load

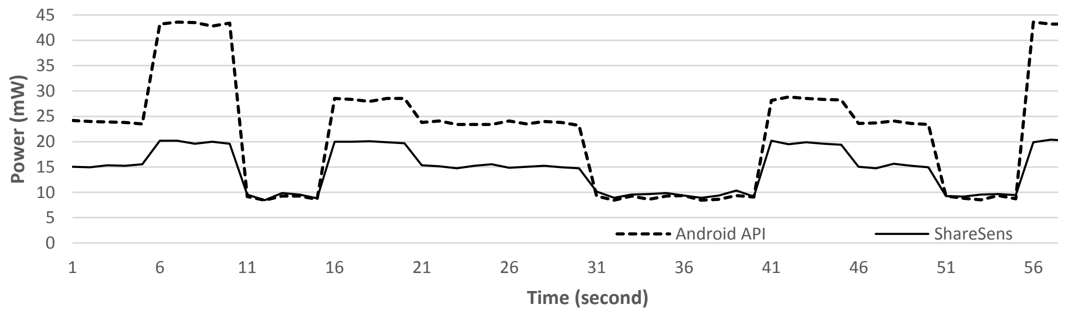


(c) Heavy Load

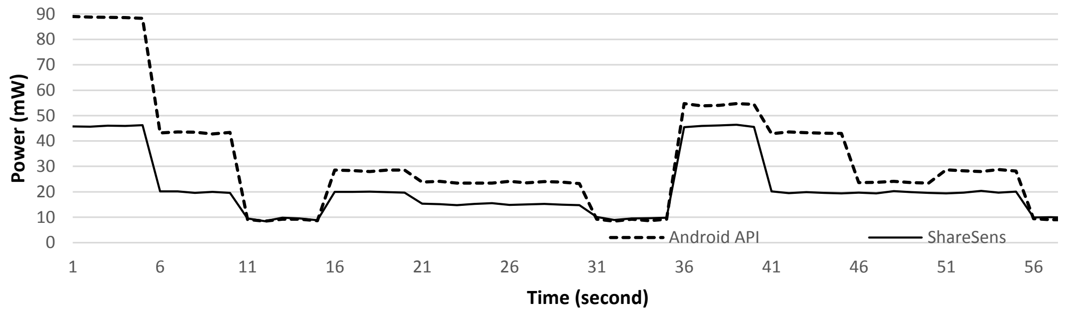
Figure 4.15: Actual Sampled Rate for Flexible Rate Experiments



(a) Light Load Experiments

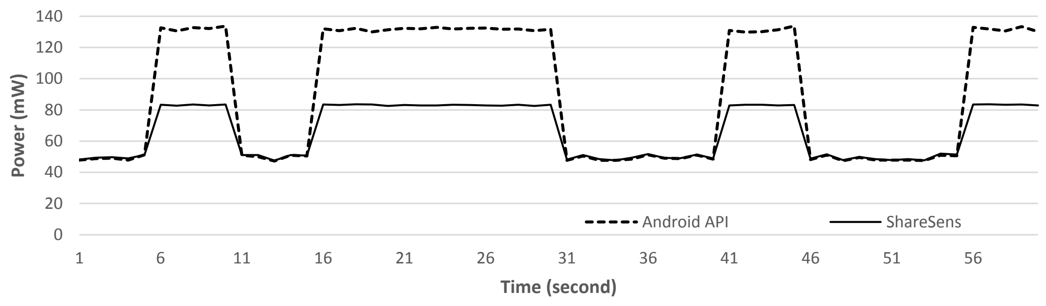


(b) Normal Load Experiments

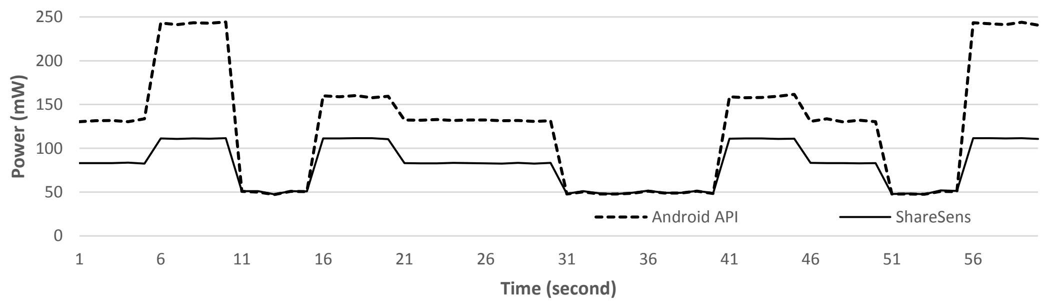


(c) Heavy Load Experiments

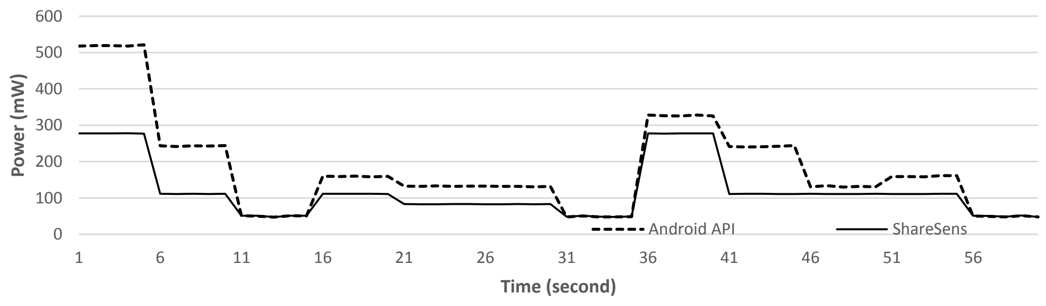
Figure 4.16: Fixed-Only Rate Requests: Power Consumption for Accelerometer



(a) Light Load Experiments

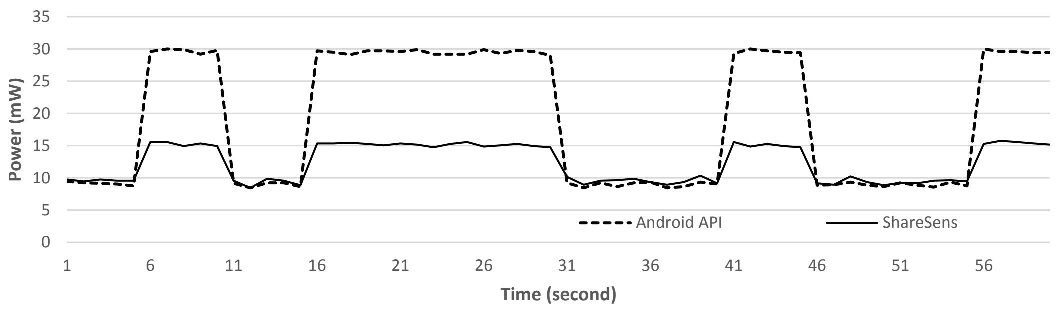


(b) Normal Load Experiments

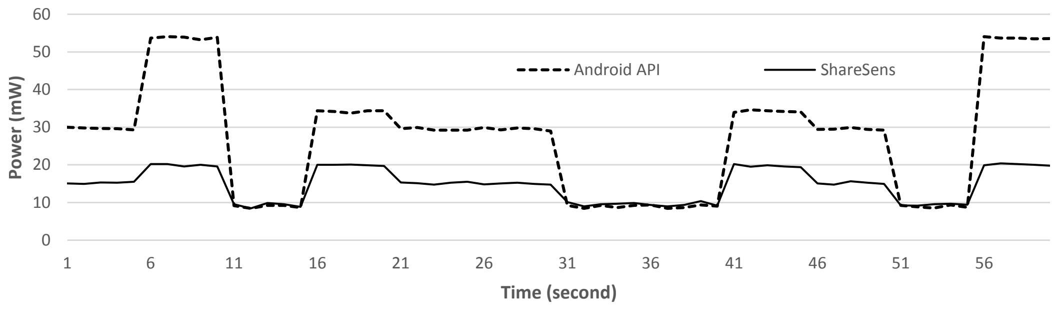


(c) Heavy Load Experiments

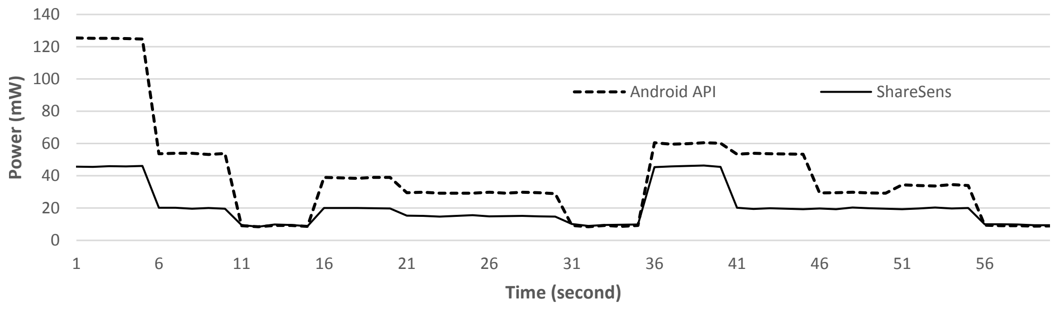
Figure 4.17: Fixed-Only Rate Requests: Power Consumption for Orientation Sensor



(a) Light Load Experiments

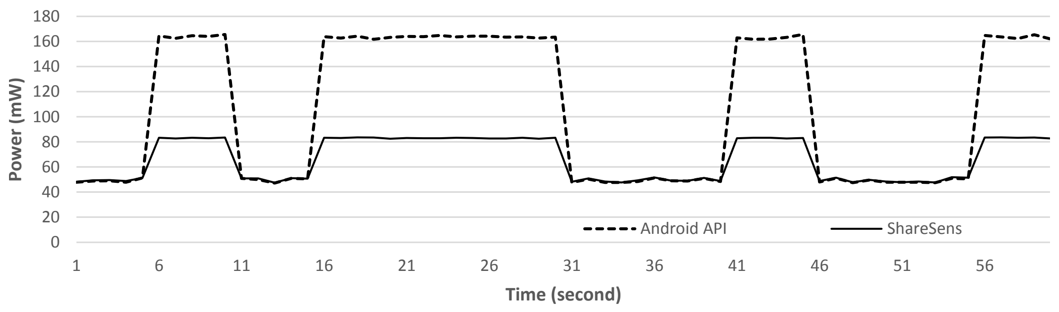


(b) Normal Load Experiments

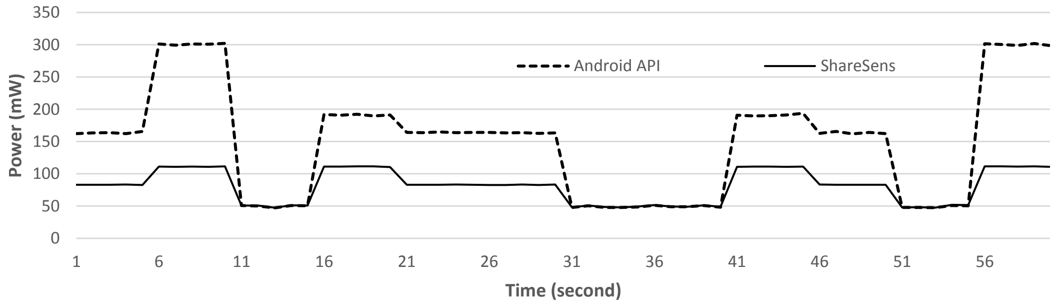


(c) Heavy Load Experiments

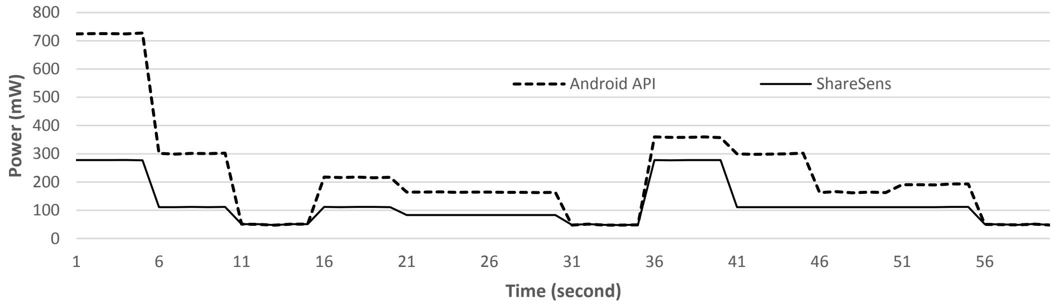
Figure 4.18: Flexible Rate Requests: Power Consumption for Accelerometer



(a) Light Load Experiments



(b) Normal Load Experiments



(c) Heavy Load Experiments

Figure 4.19: Flexible Rate Requests: Power Consumption for Orientation Sensor

almost entirely on the actual sampling rate used by ShareSens, which is equal to the highest rate requested for fixed-only requests, or the highest of the low rates requested for flexible requests.

Table 4.1: Fixed-Only Rate: Energy consumed for different workloads in mJ

Sensor	**Workload**	**Android**		**ShareSens**	
		Avg.	**SD**	**Avg.**	**SD**
Accelerometer	Light	982	49	737	38
Accelerometer	Normal	1,370	82	886	41
Accelerometer	Heavy	1,921	104	1,245	70
Orientation	Light	5,427	282	3,980	239
Orientation	Normal	7,638	367	4,876	234
Orientation	Heavy	10,960	504	7,123	356

Table 4.2: Flexible Rate: Energy consumed for different workloads in mJ

Sensor	**Workload**	**Android**		**ShareSens**	
		Avg.	**SD**	**Avg.**	**SD**
Accelerometer	Light	1,156	58	742	39
Accelerometer	Normal	1,649	99	894	41
Accelerometer	Heavy	2,405	130	1,255	70
Orientation	Light	6,384	332	3,986	239
Orientation	Normal	9,175	440	4,884	234
Orientation	Heavy	13,655	628	7,134	357

4.3.4 Transition Frequency Overhead

Finally, I examined the impact of the delay in transitioning from one rate of sampling a sensor to another, which is observable in the power consumption graphs.

I measured the two sources of delay in transitioning from one sampling rate to another: the processing time taken by ShareSens to determine the new sampling rate, and the time is taken by a sensor to change its sampling rate. These measurements are shown in table 4.3. As previously discussed in Section 4.1, the complexity of the computation involved is constant for the fixed-only rate case, and linear (in the number of requests being served by a sensor) for the flexible rate case. As a result, the measured time of 0.636 ms is for the 3 to 4 requests used in my experiments, and will change if there are more requests. Regardless, the dominant delay in making these transitions is the sensors' delay in switching to a new sampling rate. For the accelerometer and orientation sensors, the total per-transition cost (including both CPU time and sensor delay) for the fixed-only sampling rate experiments was 6.29ms and 10.24ms, respectively, and for the flexible sampling rate experiments was a similar 6.806ms and 10.756ms, respectively.

Table 4.3: Transition Delay in ms

	Mean	**SD**
Scheduler CPU Time (Fixed-Only Rate)	0.12	0.0063
Scheduler CPU Time (Flexible Rate)	0.636	0.046
Sensor Transition Delay (Accel.)	6.17	1.13
Sensor Transition Delay (Orient.)	10.12	1.39

It is difficult to definitively translate these delays into a frequency of sampling rate transitions beyond which using ShareSens will no longer be useful, without a much more detailed analysis. However, considering that as long as ShareSens's scheduling does get a chance to select sampling rates, it can deliver energy savings over Andriod's API, there is some value in determining the total number of transitions which can happen per second. That number turns out to be roughly 160 or 100 depending on whether it involves the accelerometer or the orientation sensor.

In any reasonable scenario, I would expect there to be only a few transitions every second, which is about two orders of magnitude fewer, suggesting that the transition time would not be a significant concern.

Chapter 5
ModeSens: An Approach for Multi-modal Mobile Sensing

This chapter presents the ModeSens's approach to modeling and programming multi-modal sensing requirements of applications. This chapter is organized as follows: Section 5.1 presents the model and implementation of the approach. Section 5.2 evaluates ModeSens in two ways. First, I state the programmability benefits of separating sensing mode transition concerns from functional concerns of applications. Second, I present experimental results on performance and energy costs of using ModeSens.

5.1 Model and Implementation

A natural way to represent multi-modal sensing is by using a finite state machine. Particularly, I specify a machine $M = \langle S,\ \Sigma,\ \delta,\ s_0 \rangle$ (Figure 5.1), where S and Σ are non-empty finite sets of states and inputs respectively, with $s_0 \in S$ being the initial state. δ: $S \times \Sigma \rightarrow S$ is the state transition function. Σ contains triggers for mode change, defined in terms of recently sensed data. Sensing of specific data can fire a trigger $t \in \Sigma$ which leads to state transition from s_i to s_j.

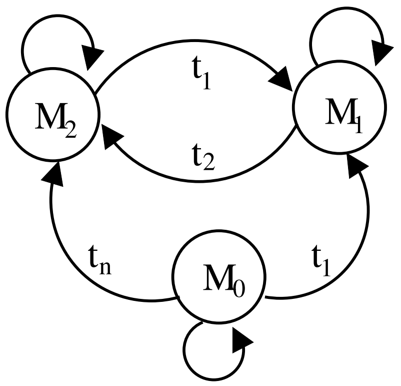


Figure 5.1: Multi-Modal Sensing

Similar to CSSWare, I have prototyped this mechanism for multi-modal sensing as a self-contained mobile application implemented over the CyberOrgs [63] extension of Actor Architecture (AA) [42] ported to Android.

5.2 ModeSens Evaluation

In this section, I present the evaluation of ModeSens for both the programmability benefits, as well as the experimental evaluation of performance and energy costs of using ModeSens. Note that the latter is intended more to document these cost than to compare with the alternative of mixing mode transition concerns with functional concerns, which would obviously still have the same monitoring, triggering and mode transition costs.

5.2.1 Programmability

My approach to multi-modal sensing offers programmability advantages by separating concerns of the sensing mode transition from the application's functional code, which would otherwise be (and are) mixed. To simplify the specification of the multi-modal system, I have developed a simple graphical user interface to allow programmers to specify the finite state machine for the mode transitions, by defining new modes, transitions between modes, and any outputs to the application. The lower half of Figure 5.2 shows this interface; the complete figure shows an interface for specifying a crowd-sourced service, which is used by CSSWare.

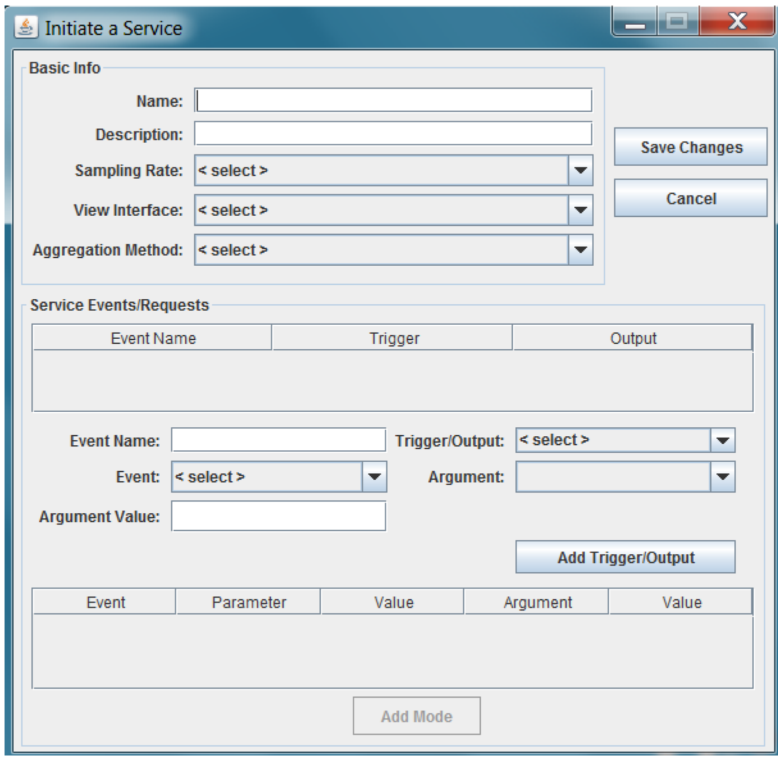


Figure 5.2: The GUI for Creating Crowd-sourced Services

5.2.2 Experimental Evaluation

My experimental evaluation used a specific case study involving human activity recognition as a sensing application, which ran on a Samsung Note II (1.6 GHz quad-core, 2 GB RAM) running Android 5.1. I defined four activity modes, namely stilling (i.e., being still), walking, bicycling and driving. Each mode had its sensing requirements for one or more of the accelerometer, gyroscope and GPS sensors, which represented the functional requirements for the application. The stilling mode required accelerometer data, walking mode required accelerometer and gyroscope data, and the bicycling and driving modes required accelerometer and GPS data.

In addition, the mode transition logic also relied on sensed data. Although it is not required in general, here I assumed that all sensors were to be sampled to detect mode transition triggers at a sampling rate of $1Hz$. In other words, for the sensors being already sampled for the current mode's functionality, only one of a presumably larger number of samples was accessed every second for trigger detection purposes; for the remaining sensors, new sensing was required. The latter could also be pulled from sensing done for other applications using ShareSens mechanism.

Each experiment presented in this chapter is carried out for ten trials, then I took the average of these trials' results.

Performance. I separately measured the ongoing cost of monitoring for detecting mode transition triggers, and the sensing and processing delays in carrying out the triggered mode transitions.

The trigger detection mechanism checked for a trigger on the arrival of every new feed set, and examined a window of recently sensed data to detect a trigger. I found the trigger detection cost to depend primarily on the size of the window of recently sensed data considered. For my case study, I used a window of size 12, which seemed sufficient for detecting mode transition triggers.

Table 5.1 shows the processing time measured for the ongoing cost of monitoring for mode transition triggers for each of the modes. The time (in milliseconds) is for both acquiring and processing the set of feeds (one feed from each sensor) to check for a transition trigger. *Old* applies when data already being collected for the mode's function can be utilized; *New* applies when fresh sensing is required. Because the sensing for detecting triggers was at the rate of 1Hz in my case study, this cost is also in milliseconds per second. The delay in transitioning individual sensors from a current sampling rate to a different one were measured to be $6.21ms$ (*SD:* 1.21), $10.71ms$ (*SD:* 1.76) and $17.39ms$ (*SD:* 2.36) for the accelerometer, gyroscope and GPS sensors, respectively. Table 5.2 shows the cost of making a transition from one of the four modes to another. This involved the sampling rate changes for the sensor involved plus a small amount of processing cost required to call for the changes. The sampling rate change cost was incurred only if a previously unsampled sensor was to be sampled in the new mode, or vice versa; the delay between a pair of modes is symmetric.

Table 5.1: Ongoing Cost Per Feed Set of Monitoring for Mode Transition Triggers in ms

Mode	**Accel.**	**Gyroscope**	**GPS**	**Total**
Stilling	Old: 0.39 (*SD:* 0.05)	New: 4.50 (*SD:* 0.54)	New: 7.31 (*SD:* 0.81)	12.2
Walking	Old: 0.39 (*SD:* 0.05)	Old: 0.39 (*SD:* 0.05)	New: 7.31 (*SD:* 0.81)	8.09
Bicycling	Old: 0.39 (*SD:* 0.05)	New: 4.50 (*SD:* 0.54)	Old: 0.39 (*SD:* 0.05)	5.28
Driving	Old: 0.39 (*SD:* 0.05)	New: 4.50 (*SD:* 0.54)	Old: 0.39 (*SD:* 0.05)	5.28

Table 5.2: Mode Transitions Delay in ms

	Stilling	**Walking**	**Bicycling**	**Driving**
Stilling	X	12.32 (*SD:* 0.98)	19 (*SD:* 1.33)	19(*SD:* 1.33)
Walking	12.32 (*SD:* 0.98)	X	31.32 (*SD:* 3.45)	31.32 (*SD:* 3.45)
Bicycling	19 (*SD:* 1.33)	31.32 (*SD:* 3.45)	X	1.61 (*SD:* 0.15)
Driving	19 (*SD:* 1.33)	31.32 (*SD:* 3.45)	1.61 (*SD:* 0.15)	X

Energy Consumption. A set of experiments was carried out to measure the ongoing energy cost incurred by the sensors for additional sensing for detecting mode transition triggers, as well as the cost of change the sensors' sampling rates to carry out the mode change. I used the PowerTutor software [64] for energy measurements. As shown in Table 5.3, the ongoing costs were measured to be $0.63mJ$ (*SD:* 0.03), $1.24mJ$ (*SD:* 0.08) and $1.93mJ$ (*SD:* 0.13) for the accelerometer, gyroscope and GPS sensors respectively. These per feed set costs amounted to mJ per second for my case study, because the sensing for detecting triggers was at the rate of 1Hz. The cost of changing a sensor's sampling rate was measured to be $2.42mJ$ (*SD:* 0.12), $3.18mJ$ (*SD:* 0.15) and $4.05mJ$ (*SD:* 0.21) for the accelerometer, gyroscope and GPS sensors, respectively.

Table 5.3: The Energy Consumption of Sensors for Multi-Modal Sensing in mJ

	Accelero.	**Gyro.**	**GPS**
Cost Per Transition	2.42 (*SD:* 0.12)	3.18 (*SD:* 0.15)	4.05 (*SD:* 0.21)
On-Going Cost Per Feed Set	0.63 (*SD:* 0.03)	1.24 (*SD:* 0.08)	1.93 (*SD:* 0.13)

Chapter 6
Towards a Formal Model for Representing Mobile Distributed Services

This chapter presents an early model, MobDisS (Mobile Distributed Services), for representing mobile distributed services. MobDisS provides a mechanism for specifying the creation and manipulation of services. This chapter is organized as follows: Section 6.1 presents the definition of a mobile service. Section 6.2 describes a set of composition rules through which more complex services are constructed from simpler ones, and gives examples to demonstrate the concept of service composition. Section 6.3 presents operational semantics for MobDisS. Section 6.4 discusses the connections between multi-origin communication and CSSWare in the context of the model. Section 6.5 summarizes this chapter.

6.1 Services

A service receives input contributions from some contributing source, processes them, and creates output contributions for some client. We call these contributions as *service feeds*. In a system of services, we treat every client and contributor as a service: they are called the client service and the contributing service of the particular service. In other words, the client service could simply be an end user receiving a feed from some service, but not necessarily producing a derivative service for another service; a contributing service, similarly, could be just a sensor without any service contributing to it.

We model a service by a set of *ports* and a set of *agents*. Ports can be one of two types: input or output. A service receives input feeds from contributing services through its input ports and sends output feeds to client services through its output ports. Agents implement the service's logic and convert the feeds arriving from contributing services into the feeds required by client services. Our definition for a service is as follows:

Definition 1 (Service):
A service has one or more input ports, which receive feeds from contributing services, and one or more output ports, which send feeds to client services, and a set of agents, which are responsible for implementing the service's logic.

From this definition, a service has two types of components: ports and agents. Both ports and agents are active objects. Consequently, a service can be defined as a set of active objects. Input ports are receptionists of data. Therefore, contributing services must know the names of input ports of client services in order to send data messages – through their output ports – to them.

6.1.1 Communication between Services

In a system of services, the required communication between services is carried out by sending and receiving asynchronous messages. Messages can be of one of two types: control messages or data messages. Control messages (also called inter-service messages) are communicated between services for administrative purposes; while the data messages are used to send service feeds from contributing services to client services. Each service has a dedicated agent called the *service coordinator*, which is responsible for handling control messages between its service and other services in the system. When a control message is sent to a service, the message is received by its coordinator agent. The rest of agents implement the service logic and process the received data messages.

Figure 6.1 illustrates the interaction between services. A rectangle represents a service's boundary. Each service has a set of ports using which it could communication with other services in the system. The figure also illustrates the components of a service. Ovals are agents serving the service, white circles are input ports, black circles are output ports, and the lines with arrows represent message flows; the service encapsulates a set of agents implementing its logic. These agents are invisible to other services. To interact with a service, both contributing and client services have to be connected to the service's ports: to send messages to the service, contributing services are connected to its input ports; to receive messages from the service, client services are connected to its output ports.

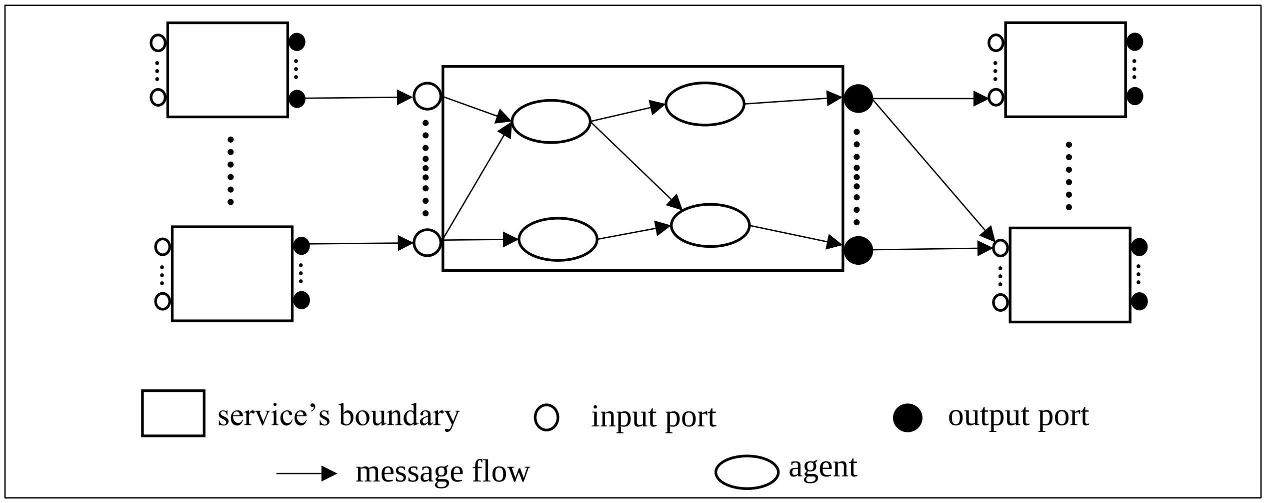


Figure 6.1: Communication between Services

6.2 Composition

We define services compositionally. In other words, simpler services can be composed to create more complex services by applying a set of composition rules. A service connects to other services using input and output ports. Messages can be received by a service at its input ports and can be sent out to other services from its output ports. An external observer can only see the input ports and the output ports of the final composed service.

We make a clear distinction between the composition of the definitions of services and the composition of running services. Here, we present how to compose the definitions of services using a set of composition rules. Later in Section 6.3, we will discuss the composition of running services.

Making services composable offers two benefits: First, from the programming perspective, composing services facilitate the reuse of designs and implementations of existing services. Particularly, a platform or a middleware could use a set of service composition rules to allow service designers to compose simpler contributing services, which could be as simple as sensor services, to create new services utilizing them. However, composing services is not the only way for creating new services. Services can also be defined directly by first creating its sets of ports and agents, and then connecting it to the required set of contributing and client services through its input and output ports, respectively. Second, from the definitional perspective, compositional definition allows for properties of services to be proven inductively.

A service s is written as:[1]

$$[\![I : \alpha : O]\!]_s$$

where s is the service's unique name, I is a set of input ports, α is a set of agents of the service, and O is a set of output ports. To be precise in our presentation of services, we assume that the computations involved in a service are carried out by actors [36]. Actors are active objects, which use asynchronous messages to communicate. In particular, we use actors to model ports and agents. An actor with name $a \in \alpha$ and behavior b is represented by $(b)_a$.

6.2.1 The primitive service

The primitive service is a *one-time service*, which has one input port, one agent, and one output port. It simply forwards a one-time service feed received from a contributing service, which is connected to the service's input port, to a client service, which is connected to the service's output port.

Consider a simple alarm service which serves one-time requests. The service's client expects at most one feed message from the alarm at a specific time in the future; when that point of time is reached, an alarm message is sent to the client.

The primitive service shown in Figure 6.2, which has one contributing service sending it a one-time service

[1]A table of notations (Table 6.1) is included at the end of this chapter.

feed, ζ, through its input port, i (white circle), which it sends to a client service through its output port, o (black circle), can be written as:

$$[\![\{i\} : \{a\} : \{o\}]\!]_s$$

where i is an input port, o is an output port, and a is an agent which has a forwarder behavior. For improving readability, from here on, we will refer to $\{a\}$, $\{i\}$ and $\{o\}$ as simply a, i and o, respectively.

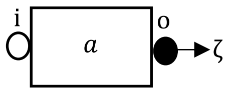


Figure 6.2: Primitive Service: One-time Service

A service feed ζ is represented in our model as a *timed data feed*, with its temporal and contextual constraints which define when and how this feed can be collected and delivered to a client service, respectively. Formally,

$$\zeta = \langle \gamma \ , \ s \ , \ td \ , \ d_\zeta \rangle$$

where γ denotes an expression which is evaluated by a service to obtain the sensing task that is carried out by the service,[2] s represents the client service which will receive the feed, td is the delivery time of the feed, and d_ζ is the maximum time delay between the sensing time and the delivery time. If the sensed data is a result of aggregating multiple feeds, then d_ζ becomes the maximum time delay between the oldest feed and the delivery time of the aggregate.

6.2.2 Composition Rules

There are three service composition rules: union, output merge and serial composition.

Union Composition

The union composition rule is used to compose a number of services by combining their ports and agents, as well as their service feeds. The purpose of this composition is to create a new service which utilizes the definitions of a number of existing services, to widen the scope of coverage provided by them over *time*, *space*, or both.

First, services can be composed over time by combining their output feeds temporally, but not necessarily combining their ports and agents. However, the delivery times of these output feeds must be different. We refer to the output service of this composition as a *temporally distributed* service. A temporally distributed service receives continual input feeds from a number of contributors, and periodically aggregates these feeds and sends continual output feeds to the service's clients. An example of a temporally distributed service would

[2]After evaluating the γ expression, the service replaces it with the sensed data.

be that of a restaurant recommendation service available over the web or through a mobile application, which attempts to offer up-to-day information to site visitors.

Second, services can also be composed over space by combining their ports and agents. As a result, the output feeds of the composed service can be considered as the union of all output feeds of the services being composed, without any restrictions on their delivery times. Therefore, the time union composition can be considered a special case of the space union composition. For instance, this composition would be useful to create a new service which takes advantage of some existing services covering different geographical locations in a city, observing various events of interest in the same location, collecting signatures in an organization or community for an electronic petition using personal smartphones, etc.

The union composition first creates a new service with empty sets of input ports, agents and output ports. If the services are composed over space, then both ports and agents of the services being composed, which retain the old behaviors, are cloned and added to their corresponding sets ports and agents of the new (composed) service. Figure 6.3 shows an example of composing $n > 1$ services over space using the union composition rule. Each contributing service s_j, which has a set of input I_j, a set of agents α_j and a set of output ports O_j, produces a set of feeds Z_j during the execution of the service, for $1 \leq j \leq n$, where n is number the contributing service to be composed. Each output port of the services being composed can produce one or more service feeds during the execution of the service. As shown in the figure, all ports and agents of the services being composed, as well as their output feeds, are combined into the composed service.

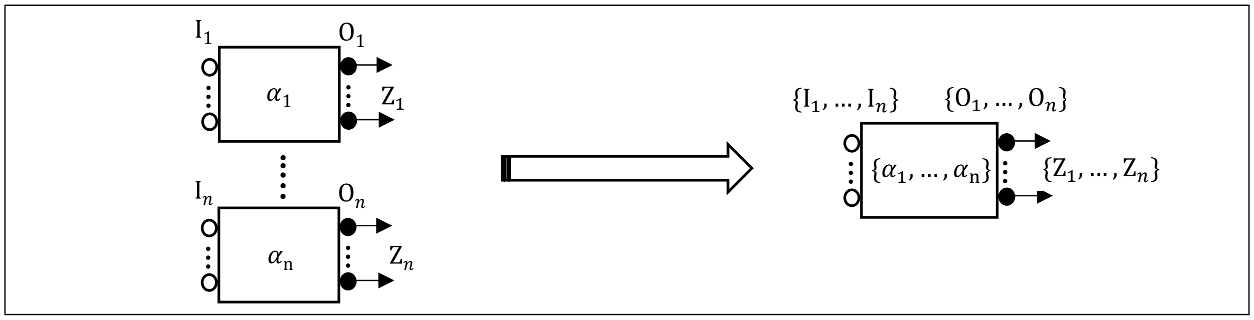


Figure 6.3: Composition: union

We use the composition operator symbol | to represent one of the composition rules defined in this section. The union composition of $n > 1$ services can be described as follow:

$$\overset{\cup}{|}{}_{j=1}^{n} [\![I_j : \ \alpha_j \ : \ O_j]\!]_{s_j} \Rightarrow [\![\bigcup_{j=1}^{n} I_j : \ \bigcup_{j=1}^{n} \alpha_j \ : \ \bigcup_{j=1}^{n} O_j]\!]_{s_c}$$

where $n > 1$, s_c is the name of the composed service, and the domains of all α_j ($Dom(\alpha_j)$) are disjoined for $1 \leq j \leq n$.

Next, I present two examples to illustrate how services can be composed using union composition over space and time.

Example: Online Petition Service. Consider an online petition where a client wants to collect a number of signatures in an organization or community using personal smartphones. Figure 6.4 illustrates how an electronic petition service can be created by composing a number of one-time services over space using the union composition rule. Each contributing service s_j, which has a set of input I_j, a set of agents α_j and one output port o_j, produces only one signature feed ζ_j during the execution of the service, for $1 \leq j \leq n$, where n is number the contributing service to be composed.

The online petition service can be created by composing $n > 1$ one-time services over space using the union composition rule, as follows:

$$\overset{\cup}{|}_{j=1}^{n} [\![I_j : \alpha_j \ : o_j]\!]_{s_j} \Rightarrow [\![\bigcup_{j=1}^{n} I_j : \bigcup_{j=1}^{n} \alpha_j \ : \bigcup_{j=1}^{n} o_j]\!]_{s_c}$$

where $n > 1$, s_c is the composed (online petition) service, and $Dom(\alpha_j)$'s are disjoined for $1 \leq j \leq n$.

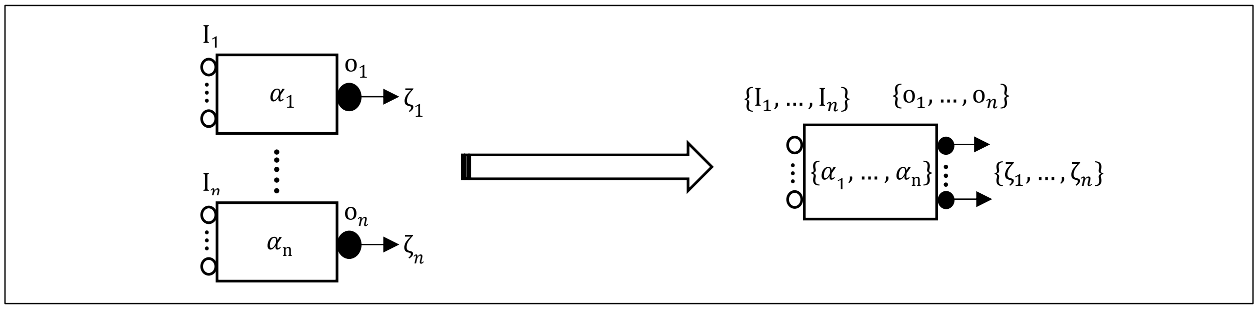


Figure 6.4: Example: Online Petition Service

Example: Periodic Alarm Service. A periodic service can be considered a particular case of the temporally distributed service. It can be created by composing a number of one-time services over time if the time interval between every two adjacent service feeds is the same, given that the service feeds are sorted by their delivery times. An example of a periodic service would be that of a periodic alarm service which serves continual requests. The service's client expects periodic notifications from the alarm at specific points of time in the future; when one of these points of time is reached, an alarm notification message is sent to the client.

Figure 6.5 illustrates how the periodic alarm service can be created by composing $n > 1$ one-time alarm services over time using the union composition rule. Each contributing service s_j has one input port i_j, one agent a_j and one output port o_j; each output port produces at most one service feed ζ_j during the execution of the service, for $1 \leq j \leq n$, where n is number of the contributing service to be composed. s_c is the name of the composed service with i_c and o_c as its input and output ports, respectively. This composition can be described as follow:

$$\overset{\cup}{|}_{j=1}^{n} [\![i_j : a_j \ : o_j]\!]_{s_j} \Rightarrow [\![i_c : \bigcup_{j=1}^{n} a_j \ : o_c]\!]_{s_c}$$

where $n > 1$; s_c is the name of the composed (periodic alarm) service with i_c and o_c as its input and output ports, respectively; and $Dom(\alpha_j)$'s are disjoined for $1 \leq j \leq n$. This composition happens only if this condition $dt(\zeta_j) - dt(\zeta_{j+1}) = const > zero$ holds, for all temporally adjacent $\zeta_j, \zeta_{j+1} \in Z_c$, where $1 \leq j < n$, Z_c is a set of feeds produced by the composed service, and dt is a function which returns the delivery time of a service feed ζ.

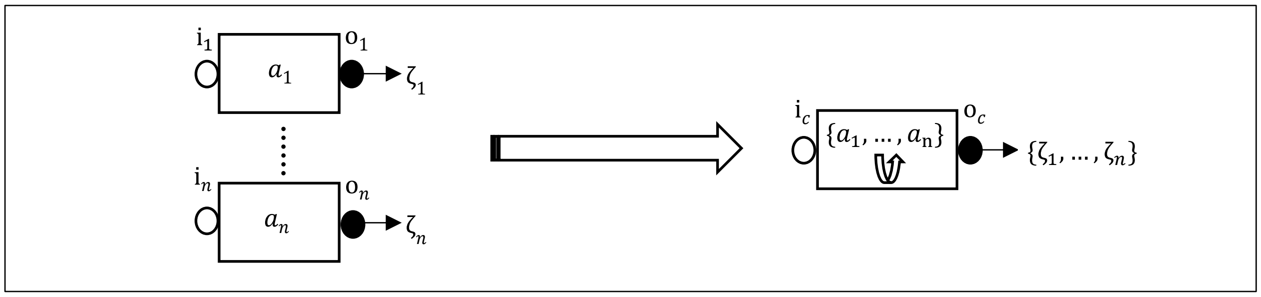


Figure 6.5: Example: Periodic Alarm Service

Output Merge Composition

The output merge composition rule is used to compose a number of services by merging their output ports. The purpose of this composition is to create a new service which enables a single client to receive service updates collected by a number of contributing services aggregated in some order. That said, the input feeds can be received by the composed service and processed as required by the service's logic. Alternatively, they can be somehow aggregated. In its simplest form, the service simply forwards out each feed received in its original form. In more interesting forms, it can process received feeds in permitted ways, both to create aggregate feeds to be forwarded, and to make decisions about whether and when to forward aggregates [9].

It turns out that unlike the previous composition rule, the output merge composition rule requires a set of parameters to be provided at the composition time. To compose a number of services by merging their output ports, the composition receives as parameters the set of output ports to be merged and one behavior. The composition first creates a new service with empty sets of input ports, agents and output ports. Then, the input ports of the services being composed, which retain the old behaviors, are cloned and added to the input ports of the composed service. The composition then creates a new output port, which becomes the output port of the composed service, and creates a new agent actor with the provided behavior targeting the newly created output port. Finally, the composition transforms the output ports to be merged into agents which have the newly created agent as their target.

Figure 6.6 shows an example of composing $n > 1$ services using the output merge composition rule. Each contributing service s_j, which has a set of input I_j, set of agents α_j and set of output ports O_j, produces a set of feeds Z_j during the execution of the service, for $1 \leq j \leq n$, where n is number the contributing service to be composed. α_c is the set of agents of the composed service. Z_c is a set of feeds produced by the composed service. o_c is the output port of the composed service (i.e., the newly created output port). This composition can be described as follow:

$$\underset{S^o,b}{\overset{\cap}{|}}{}_{j=1}^{n} \; [\![I_j : \alpha_j : O_j]\!]_{s_j} \Rightarrow [\![\bigcup_{j=1}^{n} I_j : \bigcup_{j=1}^{n} \alpha_j \cup \{(b)_{a_c}\} \cup S^o : (\bigcup_{j=1}^{n} O_j - S^o) \cup \{o_c\}]\!]_{s_c}$$

where $n > 1$, S^o is a set of output ports to be composed, b is the provided behavior, s_c is the name of the composed service, o_c is the newly created output port, a_c is the name of the newly created agent actor with a behavior b targeting o_c, and $Dom(\alpha_j)$'s are disjoined for $1 \leq j \leq n$. Each output port in S^o, which retains the same behavior, is added to the set of agents of s_c, and they all have their target to the newly created agent a_c.

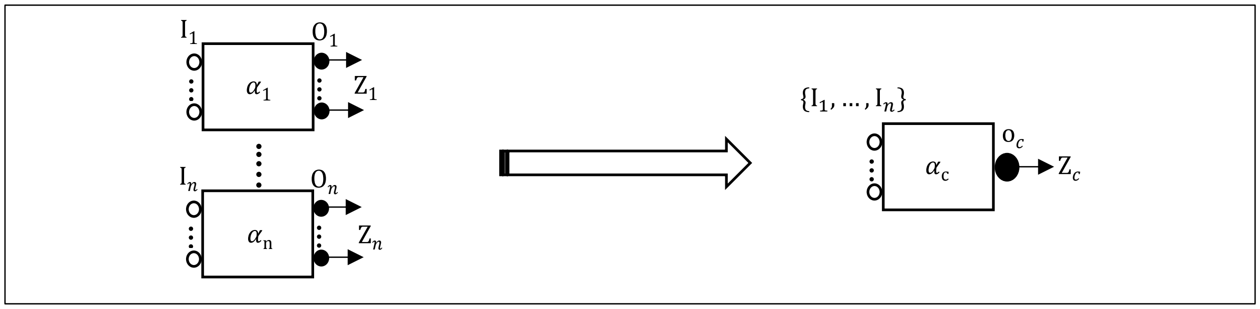


Figure 6.6: Composition: Output Merge

Example: Electronic Voting Service. Imagine a scenario where smartphones are used in an electronic election organized by a government. The type of communication in the electronic voting service is defined as a one-off event-based communication with a timeout [9], in which the electronic vote service expects at most one message from any contributing service. It collects messages until either a sufficient number of messages has been received (as can be tested using a termination function), or a timeout has been reached; it then proceeds to aggregate the messages (i.e., count and classify votes), and sends the election result to the client.

Figure 6.7 illustrates how an electronic voting service can be constructed by composing a number of one-time services using the output merge composition rule. Each contributing service s_j, which has a set of input I_j, a set of agents α_j and one output port o_j, produces only one vote feed ζ_j during the execution of the service, for $1 \leq j \leq n$, where n is number the contributing service to be composed. α_c is the set of agents of the composed service. o_c is the output port of the composed service. ζ_c is the output vote result produced by the composed service. This composition can be described as follow:

$$\underset{S^o,b}{\overset{\cap}{|}}{}_{j=1}^{n} \; [\![I_j : \alpha_j : o_j]\!]_{s_j} \Rightarrow [\![\bigcup_{j=1}^{n} I_j : \bigcup_{j=1}^{n} \alpha_j \cup \{(b)_{a_c}\} \cup S^o : (\bigcup_{j=1}^{n} \{o_j\} - S^o) \cup \{o_c\}]\!]_{s_c}$$

where $n > 1$, S^o is a set of output ports to be composed, b is the provided behavior which could be implemented as a function that counts and classifies votes, s_c is the composed (electronic voting) service, o_c is a newly created output port, a_c is the name of the newly created agent actor with a behavior b targeting o_c, and $Dom(\alpha_j)$'s are disjoined for $1 \leq j \leq n$.

Figure 6.8 shows an implementation code of the merge behavior b_{a_c}. In this example, we assume that the client wants to know the winning candidate of the election if the total number of votes exceeds a threshold,

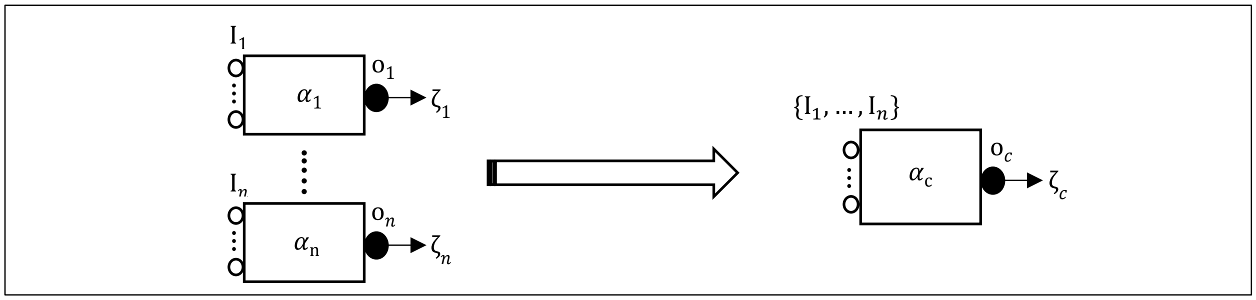


Figure 6.7: Example: Electronic Voting Service

or a deadline is reached. When a new vote feed is received at the a_c actor, the function *cond* is used to check whether that condition is reached. In *cond*, t is the expiration date; we also assume a function $time()$ which can obtain current time of the voting system. The votes received by the a_c actor are all placed into its internal queue q, until enough votes have been received to make the function call $cond(a_c, q, t, total, threshold, vote)$ true. At this point, the contents of q are aggregated and sent back to the *client* along with total number of collected votes for the winner candidate (using *pr* operator, which simply creates a pair), and a_c actor changes its own behavior to become a sink and consume all future messages without doing anything.

```
$b_{a_c} = rec(\lambda b.\lambda a_c.\lambda client.\lambda cond.\lambda q.\lambda vote.$
        $let\{total := 0, threshold := 50\}$
         $seq(putq!(q, vote),$
           $if(cond(a_c, q, t, total, threshold, vote),$
             $seq(send(client, pr(vote, total), ready(sink))),$
             $ready(b(self)))))$

$cond = rec(\lambda a_c.\lambda q.\lambda total.\lambda threshold.\lambda vote$
          $if(emptyq?(q),$
            $false,$
            $if(or((total > threshold), (time() > t)),$
              $true,$
              $if(eq(getq(q), vote),$
                $a_c(delq!(q), t, total + 1, threshold, vote)))))$
```

Figure 6.8: Electronic Voting Service: Merge Behavior

We assume four queue operations: $getq(q)$ retrieves the element at the front of queue q; $putq!(q)$ adds an element at the end of queue q and returns the modified queue; $delq!(q)$ deletes the element at the front of queue q and returns the modified queue; $emptyq?(q)$ determines whether queue q is empty. Also, we use the following syntactic sugar: *seq* is used for expressing sequential composition; *rec* represents a definable call-by-value fixed-point combinator.

Serial Composition

The serial composition rule is used to create a new service by composing a number of services serially. The composition connects some of the output ports of one service to some of the input ports of another service. The purpose of this composition is to create a new service which enables its client to consume feeds from a *pipeline* of services, where the feed flow is from the first service to the second service, and so on. In its simplest form, each service in this pipeline simply forwards out each feed received in its original form. Alternatively, it can process the received feeds before forwarding it to the next service.

The composed service can be created by composing a number of services in serial if a port-map, $\mathcal{M}$, is provided. Each entry in $\mathcal{M}$ has the form $(o_{s_k},\ i_{s_{k+1}},\ b_k)$, where o_{s_k} is the name of an output port of the k^{th} service being composed, $i_{s_{k+1}}$ is the name of an input port of the $(k+1)^{th}$ service being composed, and b_k is a binding behavior for a newly created agent actor which connects between o_{s_k} and $i_{s_{k+1}}$, for $1 \leq k < m$, where m is the number of entries in $\mathcal{M}$.

Figure 6.9 shows an example of composing $n > 1$ services using the serial composition rule. Each contributing service s_j, which has a set of input I_j, a set of agents α_j and a set of output ports O_j, produces a set of feeds Z_j during the execution of the service, for $1 \leq j \leq n$, where n is number the contributing service to be composed. I_c, O_c and α_c are the sets of input and output ports, and agents of the composed service, respectively. Z_c is a set of feeds produced by the composed service. This composition can be described as follow:

$$\underset{\mathcal{M}}{\overset{\twoheadrightarrow}{|}}{}_{j=1}^{n}\ [\![I_j\ :\ \alpha_j\ :\ O_j]\!]_{s_j} \Rightarrow [\![(\bigcup_{j=1}^{n} I_j) - (\bigcup_{k=1}^{m}\{i_k\}) :\ \bigcup_{j=1}^{n}\alpha_j \cup \bigcup_{k=1}^{m}\{(b)_{a_k}\}\ :\ (\bigcup_{j=1}^{n} O_j) - (\bigcup_{k=1}^{m}\{o_k\})]\!]_{s_c}$$

where $n > 1$, and $Dom(\alpha_j)$'s are disjoined for $1 \leq j \leq n$; $\mathcal{M}$ is a provided port-map, with its k^{th} entry has the form of $(o_{s_k},\ i_{s_{k+1}},\ b_k)$, which is a triple identifying an output port o_{s_k} of the k^{th} service, which connects to an input port $i_{s_{k+1}}$ of the next service, and a behavior b_k of a newly created agent actor a_k which implement that binding behavior, for $1 \leq k < m$, where m is the number of entries in $\mathcal{M}$; and s_c is the name of the composed service.

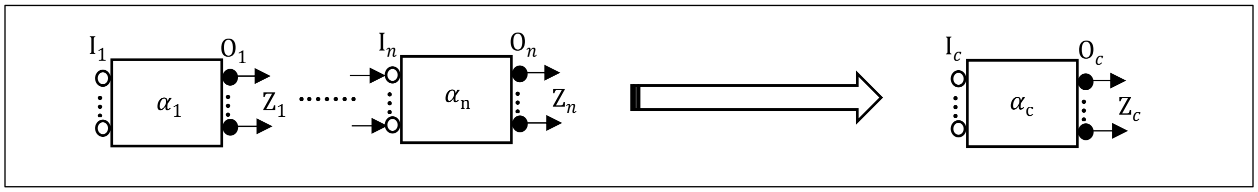


Figure 6.9: Serial Composition

Example: Document Approval Service. Imagine a scenario where smartphones are used to hierarchically obtain formal approvals from a group of people in a government or an organization for an important document such as a project plan, a contract proposal, etc. A document approval service can be created to help in sending the document to the target people, keeping track of which people have approved the document, and guiding them on how they can formally record their approval.

Figure 6.10 illustrates how a document approval service can be created by composing a number of one-time services using the serial composition rule. Each contributing service s_j, which has one input i_j port, one agent a_j and one output port o_j, produces only one document feed ζ_j during the execution of the service, for $1 \leq j \leq n$, where n is number the contributing service to be composed. α_c is the set of agents of the composed service. ζ_c is the output signed document produced by the composed service. This composition can be described as follow:

$$\underset{\mathcal{M}}{\overset{\twoheadrightarrow}{|}}{}_{j=1}^{n} \llbracket i_j : a_j : o_j \rrbracket_{s_j} \Rightarrow \llbracket i_1 : \bigcup_{j=1}^{n} a_j \cup \bigcup_{k=1}^{m} \{(b)_{a_k}\} : o_n \rrbracket_{s_c}$$

where $n > 1$, and $Dom(\alpha_j)$'s are disjoined for $1 \leq j \leq n$; $\mathcal{M}$ is a provided port-map, with its k^{th} entry has the form of $(o_{s_k},\ i_{s_{k+1}},\ b_k)$, which is a triple identifying an output port o_{s_k} of the first service, which connects to an input port $i_{s_{k+1}}$ of the next service, and a behavior b_k of a newly created agent actor a_k which implement that binding behavior, for $1 \leq k < m$, where m is the number of entries in $\mathcal{M}$; and s_c is the name of the document approval service.

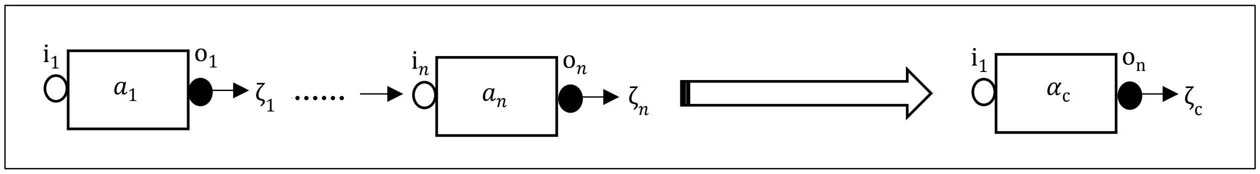


Figure 6.10: Example: Document Approval Service

Figure 6.11 shows the implementation code of the binding behavior b_{a_k}. In this example, we assume that each person (service) involved in the approval process signs the received document *doc* if it is valid as can be checked using the function *isvalid?*(*doc*). The a_k actor then send the signed document forward to the next person represented by the *client* argument, and a_k actor changes its own behavior to become *ready* to receive future messages.

```
$b_{a_k} = rec(\lambda b.\lambda a_k.\lambda client.\lambda sign.\lambda doc.$
        $seq(if(isvalid?(doc), send(client, sign(doc))),$
            $ready(b(self))))$
```

Figure 6.11: Document Approval Service: Binding Behavior

6.3 Operational Semantics

This section presents the operational semantics for MobDisS. As mentioned earlier, MobDisS relies on the Actor model [36] for concurrency to model ports and agents of a service, which use asynchronous messages to communicate. We define the state of a service as follows:

Definition 2 (Service State):
A service is denoted as s and is written as:

$$[\![I : \ \alpha : \ O]\!]_s$$

where s is the service's unique name, I is a set of input ports of the service, α is a set of agents of the service, and O is a set of output ports of the service. Here, we are only interested in modeling the interactions between services, so we do no show data messages with local recipients within a service. Because agents are modeled as actors, α can be considered as an actor map which maps a finite set of actor addresses to their behaviors. A coordinator actor of a service s, written as $a_s \in \alpha$, receives all messages sent to s. Notice that there is a one-to-one mapping between s and a_s; i.e., for each service s, there is exactly one coordinator agent.

Definition 3 (MobDisS Configuration)
The instantaneous snapshot of a system of services is called a MobDisS configuration. A MobDisS configuration represents the state of a finite set of long-lived services, a finite set of contracts between services which define how services are connected to each other, and a finite set of control messages between services. A contract is negotiated between two services when one of them wants to consume service feeds produced by the other. A MobDisS configuration can be represented by a 3-tuple:

$$\langle S \mid C \mid M \rangle$$

S is a set of services. C is the set of contracts between the services, where each contract $c \in C$ has the form $(s_1, \ s_2, \ map)$, where s_1 is the name of the first service, s_2 is the name of the second service and map is a name table which says which output ports of the first service are connected to which input ports of the second service. The connections are represented using $(o \ ; \ i)$ pairs where $o \in O$ is an output port of s_1 and $i \in I$ is an input port of s_2. The contracts involving a service $s \in S$ can be written as $co(s) \subset C$. M is a finite set of control (inter-service) messages in the system which are communicated between services, and are handled by coordinator actors of the communicating services.

MobDisS assumes that there is a special service, called the *directory service*, which has up-to-date information about the capabilities of all contributing services in the system.[3] The directory service fits the service definition (Definition 1) because it has non-empty finite sets of input ports, output ports and agents, so it can be considered as a service. The directory service can be implemented as a federated hierarchy of directory services which can be distinguished by different metrics such as geographical locations, type of services, etc. Each contributing service in the system must be registered with the directory service in order to participate in serving clients.

[3]For example, the service capability of a sensor service is determined by the maximum sampling rate of that sensor.

6.3.1 Service Request

One way to represent the requirements of a service is to represent them as sets of timed data feeds. For each of these feeds, the service needs to secure appropriate sensor data feeds and carrying out the required aggregations and customizations for different clients. This representation of a service request gives the ability to a client to represent their service requirements without being too rigid. If the client is too rigid in defining their request, the service request is likely to be rejected if the directory service is too busy at these points of time. A service request is represented by ρ and is defined as follows:

Definition 4 (Service Request):
A service request ρ is a set of sets of timed service data feeds. Each set $\rho_i \in \rho$ has a sufficient number of service feeds for serving that request.

A service request is represented formally as follows:

$$\rho = \{\rho_1, \ldots, \rho_n\},\ \rho_i = \langle \zeta_1, \ldots, \zeta_m \rangle$$

where $n \geq 0$ is the number of sets in ρ, $m \geq 0$ is the number of service feeds in ρ_i, and ζ represents a single timed service feed.

Only one set $\rho_{selected}$ out of ρ needs to be served. The selection of this $\rho_{selected}$ can be said to be done by a function f as follows:

$$\rho_{selected} = f(\rho)$$

The necessary and sufficient condition for accepting a service request can be stated as follows:

Axiom 1: Accepting a Service Request. A service request ρ can be accepted in the system if and only if at least one member of ρ, $\rho_{selected} \in \rho$, can be served by a set of contributing services, $S_{\rho_{selected}} \subset S$.

We consider a continuous-time model for representing services. This continuous-time modeling approach provides more flexibility in describing such services because of their real-time requirements, which are represented as sets of temporally defined feeds. These feeds are constructed by aggregating sensor events – such as user activities, change in a geographical location, etc. – which occur at particular points in time and lead to a change in state of interest to a service.

We assume there is a global clock in the system. One method to achieve global clock synchronization in a sensor network is to synchronize the clocks in the whole network such that all the clocks have approximately the same reading at a global time point, as described in [65]. The main idea of this method is to start from a master node, adjust the clocks of its neighbors, and diffuse this clock adjustment to other nodes. However, this approach is not scalable for large networks because it requires the nodes in the entire network to participate in the synchronization process at the same time. Another method for global clock synchronization that is fully distributed and localized is presented in [66], called diffusion-based time synchronization. In this method, time synchronization is done locally, without a global synchronization initiator. The diffusion method achieves global synchronization by spreading the local synchronization information to the entire system.

6.3.2 Transition Rules

We use transition rules to describe the progress of a system of services. Here, I review the transition rules in actor semantics.

The operational semantics for actor systems are defined using a transition relation on actor configurations, which is an instantaneous snapshot of the system state. An actor configuration is written as:

$$\langle \alpha \mid \mu \rangle$$

where α is an actor map which maps a finite set of actor addresses to their behaviors; and μ is a multi-set of messages, a message $m \in \mu$ which has two parts a target a and message content v, is written as $m = \langle a \Leftarrow v \rangle$.

An actor a can be in one of two states: *idle* and *busy.* The state of an idle actor is written as $(b)_a$ (round brackets is the symbol of an idle actor), which means that the actor a is ready to accept a message. When a message m arrives, the actor's behavior b is applied to the incoming message value. The state of an busy actor a is written as $[b(v)]_a$ (square brackets is the symbol of a busy actor), which means that the actor is processing a message $\langle a \Leftarrow v \rangle$ by applying its behavior b to the message value v.

An actor expression e is either a value or it can be decomposed uniquely into a reduction context, R, with a hole filled with a redex r, written as $e = R[r]$. Intuitively, r is the sub-expression to be reduced next when the actor performs a computation step. R is the remainder of the actor's behavior.

The following rule shows that if an actor expression e reduces to expression e', then an actor a with behavior e will change its behavior to e':

$$e \xrightarrow{\lambda}_X e' \Rightarrow \langle [R[e]]_a \ , \ \alpha \mid \mu \rangle \rightarrow \langle [R[e']]_a \ , \ \alpha \mid \mu \rangle$$

where $X = Dom(\alpha) \cup \{a\}$ is the context in which e is reduced to e', and $Dom(\alpha)$ is the domain of α.

The following rule describes how an actor a sends a message $\langle a' \Leftarrow v \rangle$ to an actor a' using the *send* primitive:

$$\langle [R[send(a', v)]]_a \ , \ \alpha \mid \mu \rangle \rightarrow \langle [R[nil]]_a \ , \ \alpha \mid \langle a' \Leftarrow v \rangle \ , \ \mu \rangle$$

where $send(a', v)$ is a primitive used to send a message with communication value v asynchronously to an actor with the address a'. This leads to the creation of message $\langle a' \Leftarrow v \rangle$ on the right hand side.

The following rule shows how an actor a with a behavior b can receive a message when it is in the idle state:

$$\langle [R[ready(b)]]_a \ , \ \alpha \mid \langle a \Leftarrow v \rangle \ , \ \mu \rangle \rightarrow \langle [b(v)]_a \ , \ \alpha \mid \mu \rangle$$

where $\langle a \Leftarrow v \rangle$ is a message sent to an actor a, $ready(b)$ means that a is ready to receive a message. The actor a's new state becomes $b(v)$, that is, its behavior b is applied to the incoming message value v. Notice that the reduction context R is discarded because *ready* discards the rest of the computation.

The following rule shows how an actor a can create a new actor a' with a provided behavior b using the *create* primitive, and the name of the newly created a' is returned to a:

$$\langle [R[new(b)]]_a \ , \ \alpha \mid \mu \rangle \rightarrow \langle [R[a']]_a \ , \ [ready[b]]_{a'} \ , \ \alpha \mid \mu \rangle$$
$$a' \ is \ fresh$$

where a' is the fresh name of the newly created actor.

Next, I present transition rules of MobDisS.

Sending a service request

In MobDisS, a service is initiated by a client service which sends a service request ρ to the directory service with the intent of creating a new service. The client's requirements are expressed in that request ρ. The same service request can also be used to subscribe to an existing service.

The following transition shows how a client service sends a service request to the directory service:

$$\langle [\![I : \ [R[send(s_d, (\rho, s))]]_{a_s} \ , \ \alpha \ : \ O]\!]_s, \ S \mid C \mid M \rangle \rightarrow \langle [\![I : \ [R[nil]]_{a_s} \ , \ \alpha \ : \ O]\!]_s, \ S \mid C \mid \langle s_d \Leftarrow (\rho, s) \rangle, \ M \rangle$$

where s is a client service, s_d is the directory service, ρ is a service request, a_s is s's coordinator actor, and $send(s_d, (\rho, s))$ sends message (ρ, s) containing the received ρ and the client's name to the directory service s_d. This leads to the creation of message $\langle s_d \Leftarrow (\rho, s) \rangle$ on the right hand side, and actor a_s continues execution.

Search function

For convenience, we define a function *search* which is used by the directory service to determine the opportunity to serve a new service request ρ by selecting one existing service matching the requirements of ρ, or a set of contributing services which could collectively contribute to serving ρ. The *search* function takes as parameters a service request ρ and the name of the client service s requesting ρ, and returns one of three pairs: (1) $(\{s_m\}, c)$, a pair of the name of an existing service s_m which matches ρ's requirements and a new contract c created between s_m and s; (2) $(S_\rho, \emptyset)$, a pair of a set of potential contributing services S_ρ which could collectively contribute to serving ρ based on their capabilities and an empty set $\emptyset$ indicating that no contract is created at this point; or (3) $(\emptyset, \emptyset)$ to indicate that there is no way of serving ρ. The *search* function is defined as follows:

$$search(\rho, s) = (\{s_m\}, c) \mid (S_\rho, \emptyset) \mid (\emptyset, \emptyset)$$

Receiving a service request

On receiving a service request ρ, the directory service uses the search function to determine the opportunity to serve ρ. This transition is written as:

$$\langle [\![I : [R[ready(search)]]_{a_{s_d}}, \alpha : O]\!]_{s_d}, \ S \mid C \mid \langle s_d \Leftarrow (\rho, s) \rangle, \ M \rangle \rightarrow \langle [\![I : [search(\rho, s)]_{a_{s_d}}, \alpha : O]\!]_{s_d}, \ S \mid C \mid M \rangle$$

where s_d is the directory service, a_{s_d} is s_d's coordinator actor, s is the name of the client service which sent ρ, $\langle s_d \Leftarrow (\rho, s) \rangle$ is a service request message sent to s_d, and ρ is the service request. As a result of delivery of this message to actor a_{s_d}, a_{s_d} uses the *search* function to first searches for an existing service matching the requirements of ρ, or a set of contributing services which could collectively contribute to serving ρ.

Subscribing to an existing service

If a matching service is found, then the directory service tells the client service about this service, and a new contract is signed between the client service and the found service.

If *search*(ρ, s) evaluates to $(\{s_m\}, c)$, then the transition rule for subscribing to an existing service is as follows:

$$\mathit{search}(\rho, s) \xrightarrow{\lambda}_X (\{s_m\}, c) \Rightarrow$$
$$\langle [\![I : [R[\mathit{search}(\rho, s)]]_{a_{s_d}}, \alpha : O]\!]_{s_d},\ S \mid C \mid M \rangle \rightarrow \langle [\![I : [R[(\{s_m\}, c)]]_{a_{s_d}}, \alpha : O]\!]_{s_d},\ S \mid C' \mid \langle s \Leftarrow s_m \rangle,\ M \rangle$$

where $X = \{a_{s_d}\}$ is the context in which *search*(ρ, s) is reduced to $(\{s_m\}, c)$; s_d is the directory service; a_{s_d} is s_d's coordinator actor; ρ is the service request; s is the name of the client service which sent ρ; s_m is the name of an existing service s_m which matches ρ's requirements; c is a new contract created between s_m and s, which has the form $(s_m,\ s,\ (o_{s_m}\ ;\ i_s))$ where $o_{s_m} \in O_{s_m}$ is an output port of s_m and $i_s \in I_s$ is an input port of s; and $C' = C\ \cup\ \{c\}$. This transition happens only if and only if *search*(ρ, s) is evaluated to $(\{s_m\}, c)$. This leads to the creation of message $\langle s \Leftarrow s_m \rangle$ containing s_m which is sent to s, and the creation of a new contract c created between s_m and s on the right hand side.

Creating a new service

Here, we present the composition of running contributing services to create a new service based on the definition of ρ.

Consider a function *create*, which when provided as parameters a service request ρ, a set of contributing services, S_ρ, which could collectively contribute to serving ρ, and the name of the client service s requesting ρ, creates a new service which satisfies ρ's requirements; the *create* function returns the name of the newly created service s', and a set of contracts C_ρ between s' and S_ρ. The *create* function is defined as follows:

$$\mathit{create}(\rho, S_\rho, s) = (\{s'\}, C_\rho, c)$$

If there exist contributing services whose feeds are sufficient for generating ρ, then the service request is accepted. This operation is shown in the following:

$$\mathit{search}(\rho, s) \xrightarrow{\lambda}_X (S_\rho, \emptyset) \Rightarrow$$
$$\langle [\![I :\ [R[\mathit{search}(\rho, s)]]_{a_{s_d}}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid M \rangle \rightarrow \langle [\![I :\ [R[(S_\rho, \emptyset)]]_{a_{s_d}}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid M \rangle$$

where $X = \{a_{s_d}\}$ is the context in which *search*(ρ, s) is reduced to $(S_\rho, \emptyset)$, s_d is the directory service, a_{s_d} is s_d's coordinator actor, ρ is the service request, and s is the name of the client service which sent ρ. This transition happens only if and only if *search*(ρ, s) is evaluated to $(S_\rho, \emptyset)$.

The directory service then uses the *create* function to create a new service utilizing the contributing services. The transition rule for creating a new service is as follows:

$$\mathit{create}(\rho, S_\rho, s) \xrightarrow{\lambda}_X (\{s'\}, C_\rho, c) \Rightarrow$$
$$\langle [\![I : [R[\mathit{create}(\rho, S_\rho, s)]]_{a_{s_d}}, \alpha : O]\!]_{s_d}, S \mid C \mid M \rangle \rightarrow \langle [\![I : [R[(\{s'\}, C_\rho, c)]]_{a_{s_d}}, \alpha : O]\!]_{s_d}, S' \mid C' \mid \langle s \Leftarrow s' \rangle, M \rangle$$
$$\{s'\}\ ,\ C_\rho \text{ and } c \text{ are fresh}$$

where $X = \{a_{s_d}\}$ is the context in which $create(\rho, S_\rho, s)$ is reduced to $(\{s'\}, C_\rho, c)$; s_d is the directory service; a_{s_d} is s_d's coordinator actor; s is the client service which sent the service request ρ; s' is the newly created service producing ρ from C_ρ's contributions; C_ρ is a new set of contracts created between s' and S_ρ; c is a new contract created between s' and s, which has the form $(s',\ s,\ (o_{s'}\ ;\ i_s))$ where $o_{s'} \in O_{s'}$ is an output port of s' and $i_s \in I_s$ is an input port of s; $C' = C\ \cup\ C_\rho\ \cup\ \{c\}$; and $S' = S \cup \{s'\}$. This transition happens only if and only if $search(\rho, s)$ is evaluated to $(S_\rho, \emptyset)$. This leads to the creation of message $\langle s \Leftarrow s' \rangle$ containing the name of the new service s', the creation of new sets of contracts C_ρ and $\{c\}$, and the creation of a new service s' on the right hand side.

Rejecting a service request

A service request is rejected if none of the existing services match the target service's requirements expressed in ρ, and there do not exist sufficient contributing services for serving ρ (i.e., there are no sufficient contributing services for serving any member of ρ). If these two conditions hold, a request rejection message to sent to the client service which sent ρ.

If $search(\rho, s)$ evaluates $(\emptyset, \emptyset)$, then the transition rule for rejecting a service request is as follows:

$$\frac{search(\rho, s) \xrightarrow{\lambda}_X (\emptyset, \emptyset)}{\langle [\![I :\ [R[search(\rho, s)]]_{a_{s_d}}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid M \rangle \rightarrow \langle [\![I :\ [R[(\emptyset, \emptyset)]]_{a_{s_d}}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid \langle s \Leftarrow \emptyset \rangle,\ M \rangle}$$

where $X = \{a_{s_d}\}$ is the context in which $search(\rho, s)$ is reduced to $(\emptyset, \emptyset)$, s_d is the directory service, a_{s_d} is s_d's coordinator actor, and s is the client service which sent service request ρ. This transition happens only if and only if $search(\rho, s)$ is evaluated to $(\emptyset, \emptyset)$. This leads to the creation of message $\langle s \Leftarrow \emptyset \rangle$ which is a rejection message sent to s indicating that there is no way of serving ρ.

6.4 Discussion

Mobile distributed services often require complex communication, which is not adequately supported by existing communication mechanisms. In this thesis, I tried to represent these complex communications in terms of primitive asynchronous messages in Actor systems for the domain of crowd-sourced services. For instance, I have identified multi-origin communications as a key mechanism underlying such crowd-sourced services, and then implemented these mechanisms in the form of a distributed middleware, CSSWare. CSSWare allows a rich variety of crowd-sourced services to be implemented with relative ease, from crowd-sourced recommendation services for restaurants to social media services similar to Twitter. Consequently, service designers can focus their attention on service-specific details rather than having to implement underlying communication mechanisms from scratch.

MobDisS can be considered as a generalization of multi-origin coordination mechanisms. The model can be used to understand and represent the communication requirements of a wider class of mobile distributed services in different domains. This can be achieved this by first identifying common types of communications in the domain, and then implementing high-level support for them.

Furthermore, the composition rules of MobDisS define the boundary of the class of services which are supported by our model, and consequently, the class of services which can be built over CSSWare. In other words, services that cannot be created using these composition rules are not supported by our model and cannot be implemented using our middleware. For example, our composition rules cannot be used to create a service that has a self-loop, which connects an output port of a service to an input port of the same service. However, this type of loops can be implemented within the service's logic.

6.5 Summary

This chapter gave the definition of a service from two different angles: compositionally and directly. From the composition angle, we presented a set of composition rules for constructing complex services by composing simpler ones. From the direct creation angle, we defined a service as a set of active objects, which are ports and agents. A service receives input feeds from contributing services via its input ports and sends output feeds to client services via its output ports. The received feed messages are handled by the service's agents, which sends the processed feeds to output ports.

A service can be as simple as a primitive service, which could forward a single feed received from a contributing service connecting to its input port to a client service connecting to its output port. Because of the composition rules, we can build more complex services starting from these primitive services. Therefore, having a set of composition rules through which more sophisticated services are constructed from simpler ones is crucial to our model.

Finally, this chapter presented the operational semantics for MobDisS, which are defined by a transition relation on MobDisS configuration.

Symbol	**Description**	**Section**
s	Service	6.2
i_s	An input port of a service s	6.2
o_s	An output port of a service s	6.2
I_s	The set of input ports of a service s	6.2
O_s	The set of output ports of a service s	6.2
α_s	The set of agents of a service s	6.2
$(b)_a$	An actor a with behavior b ready to accept a message	6.2
ζ	Service feed	6.2.1
γ_ζ	The sensing expression of a service feed ζ	6.2.1
td_ζ	The delivery time of a service feed ζ	6.2.1
d_ζ	The time delay of delivering a service feed ζ	6.2.1
$\mid$	Composition operator symbol	6.2.2
s_c	The name of a composed service	6.2.2
$Dom(f)$	The domain of a function f	6.2.2
Z	A set of service feeds	6.2.2
e	A lambda-calculus expression	6.2.2
$\lambda x.e$	Abstractions	6.2.2
$seq(e_1, \ldots, e_n)$	Sequencing construct	6.2.2
$if(e_0, e_1, e_2)$	Conditional branching	6.2.2
$let\{x := e_0\}e_1$	Lexical variable binding	6.2.2
rec	A definable call-by-value fixed-point combinator	6.2.2
pr	A pairing operator	6.2.2
$\mathcal{M}$	A port-map used for serial composition	6.2.2
a_s	A coordinator actor of a service s	6.3
s_d	The directory service	6.3
S	A set of services	6.3
C	A set of contracts between the services	6.3
M	A set of control (inter-service) messages	6.3
ρ	Service request	6.3
$\langle \alpha \mid \mu \rangle$	An actor configuration with:	6.3
	α – an actor map	
	μ – a multi-set of messages	
$[b]_a$	The state of a busy actor a with behavior b	6.3

R	Reduction context	6.3
r	Redex	6.3
$\emptyset$	Empty set	6.3
$\xrightarrow{\lambda}$	The reduction relation for functional redexes, $e \xrightarrow{\lambda}_X e'$	6.3
$\rightarrow$	The reduction relation for configurations	6.3

Table 6.1: Notation

Chapter 7
Conclusions and Future Directions

In this chapter, I present the conclusion of the thesis in Section 7.1, and future directions of the research in Section 7.2.

7.1 Conclusion

With the growing ubiquity of sensors and mobile devices, it is more possible than ever to offer innovative services based on both what the millions of sensors on people's devices are sensing, as well as what individuals are willing to actively contribute. However, the lack of precise understanding, specification, and analysis of such services limit the widespread of them.

In this thesis, I focused on building mechanisms to support programming of mobile distributed services, and formal models to understand them. My research objective was to understand the properties of these services and to design and implement novel mechanisms to improve the programmability and performance of them.

I have argued in this thesis that many crowd-sourced services, including prominent social media services (if we consider their role of helping evolve collective messages), require similar communication mechanisms. I focused on one such mechanism – multi-origin communication – which allows a number of autonomous participants to contribute messages which can then be aggregated to create group messages on behalf of all. I introduced an approach to supporting crowd-sourced services using multi-origin communication, and presented the design and implementation of CSSWare, an Actor-based middleware for crowd-sourced services as a platform for launching such services. I illustrated the ease with which new services can be launched by presenting source code for prototype implementations for two qualitatively different types of services, each requiring less than 25 lines of main service specification code, with less than 160 lines of additional relevant code from available libraries of aggregation functions, sensor-events specifications and service view interface. Finally, I experimentally evaluated the scalability of the approach. Most notably, even our modestly configured server could potentially provide a restaurant recommender service to a population of millions; contributor devices could contribute to tens if not hundreds of services simultaneously; client devices could monitor tens of services.

I also addressed the challenge of satisfying the energy needs of a potentially large number of services requiring sensor data continuously. I developed an approach for optimizing the frequency of data collection by sensors, which takes advantage of the overlap in sensing requirements of various applications. I presented the design and prototype implementation of the ShareSens API for Android, which can be used by applications to request access to sensor data. The experimental evaluation showed that significant power savings can be achieved using ShareSens when there are overlapping sensing requests. With applications requesting flexible sampling rates, it is also possible to opportunistically deliver higher rates to applications than what they minimally require, essentially for free.

Furthermore, I discussed how to support multi-model sensing of mobile distributed services. I proposed ModeSens, an approach to programming mode transition concerns of multi-modal sensing applications separately from their functional concerns. The mode transition logic can be easily specified using an appropriate finite state machine, for which I have implemented a simple GUI. My evaluation was two-fold. First, I stated the obvious programmability benefits of using ModeSens. Second, I presented experimental results documenting the processing overhead, sensing delays and energy costs involved in using ModeSens for achieving mode transitions. These experimental results are not intended to compare ModeSens with the alternative of mixing mode transition code with functional code, which would carry out virtually the same monitoring, triggering and mode transitions, except with worse programmability.

Finally, I built a formal model, MobDisS (Mobile Distributed Services), for representing mobile distributed services, allowing them to be carefully studied. Services can be built by composing simpler services. MobDisS relies on the Actor model to define services. I precisely described the syntax and operational semantics of MobDisS.

7.2 Future Directions

I envision two major avenues for my future research: first, I would like to build on my experience in concurrency, formal modeling, mobile distributed services and mobile sensing. Second, I would like to expand the scope of my research to related areas of big data processing and Internet of Things (IoT).

7.2.1 Mobile Distributed Services

In CSSware project, I would like to develop mechanisms for service designers and third parties to add custom service view interfaces and aggregation functions. This will allow a larger variety of services to be implemented. Also, I would like to apply an approach based on CyberOrgs model [63] for fine-grained resource coordination to refining the sensor sampling scheduler, and more generally to manage the resource demands that a larger number of services may place on resource-constrained mobile devices.

In ShareSens project, I would like to look into opportunities for updating existing applications to use ShareSens. Although a tool could easily be developed to automatically change source code to use the ShareSens API instead of the default Android API, it is not obvious that this would lead to the best outcomes.

Particularly, programmers coding while assuming ShareSens's properties may very well choose different sampling rates for the various sensors from what they would choose when working with the default Android API.

A related question concerns fairness among different applications in terms of which applications can request how much sampling, and consequently use how much energy, and how to account for this when multiple applications piggyback on each other's sampling rates as is possible by requesting flexible sampling rates. One possibility would be to treat sampling privileges as precious resources, and use an approach based on CyberOrgs model to coordinate competition for them.

In a related project, I have developed programming support for implementing multi-modal sensing, where evolving sensing requirements of applications can be programmed separately from an application's functions. Although the primary goal of this project, ModeSens, is to achieve software engineering objectives such as modularity and reusability, multi-modal sensing – by definition – has dynamically evolving sensing requirements, which creates opportunities for ShareSens to exploit for power conservation. Particularly, I plan to compose ModeSens with ShareSens to further generalize the class of sensing tasks to which it can be beneficial.

Furthermore, ShareSens and ModeSens can work together to optimize more complex sensing requirements of applications. The problem of cumulating demand on sensors is a significant barrier to simultaneously serving multiple sensing applications on battery powered devices. My research work addressing this challenge has the potential of making it possible to simultaneously serve multiple sensing-heavy applications or services, or help realize *Sensing as a Service*.

In ModeSens project, I would like to develop a mechanism to allow mobile users to train new modes into their multi-modal sensing applications – ModeLearn. In particular, I would like to develop an online Activity Recognition (AR) system to be implemented on mobile devices using its built-in sensors.

I would like to generalize the MobDisS model to support a broader class of services in different domains such as Wireless Sensor and Actuator Network (WSAN), IoT, etc. I also would like to express and prove interesting properties of these services based on the model.

7.2.2 Related Research Areas

Potential future work in related research areas is presented in the following sections.

Big-Data Services

With sensors becoming increasingly ubiquitous, the amount of data generated by sensors and individuals is increasing exponentially. The increasing volume, variety and velocity of sensor data places a demand not only for the storage and compute resources, but also on the development of big data services which can process those continuous sensing workloads. I am interested in building data aggregation services which can take advantage of the data collected by these sensors in aggregate form, to offer useful services such as detecting medical emergencies and imminent natural disasters, or simply recommending the restaurant with the shortest wait time.

Resource Coordination Support for Big-Data Frameworks

The Scala programming language [37] with its Akka [38] library for supporting actors, has become highly influential in the development of large-scale systems. Apache Spark [67] – the leading big data analytics system – has been implemented in Scala which enables scalable, high-throughput, fault-tolerant processing of data workloads. I would like to examine the opportunities for resource control and coordination available in the Scala programming language, and its Akka library and installed mechanisms to create resource encapsulations for long-lived computations.

Internet of Things (IoT)

Although there are a growing number of technologies and applications focused on IoT [68], there is relatively limited foundational work on such applications. These applications sense interesting environments, they do computation, and they act in a way to serve some purpose. I would like to extend MobDisS to support services focused on IoT which will allow fundamental properties of such services to be rigorously studied. I also would like to study the equivalence and composition of such services.

Supporting New Computing Paradigms in Software Engineering

New computing paradigms such as IoT, mobile crowd-sourcing, could computing and big data processing, present new challenges that current programming languages do not always support satisfactorily. I am interested in revisiting the foundations of object-oriented and functional languages to better support these new computing paradigms.

Publications related to Dissertation

- **Ahmed Abdel Moamen** and Nadeem Jamali. Opportunistic Sharing of Continuous Mobile Sensing Data for Energy and Power Conservation. *IEEE Transactions on Services Computing (TSC)*, pages 1-14, 2017, to appear. [Related to Chapter 4]

- **Ahmed Abdel Moamen** and Nadeem Jamali. An Actor-Based Middleware for Crowd-Sourced Services. *EAI Endorsed Transactions*, pages 1-15, 2017, to appear. [Related to Chapter 3]

- **Ahmed Abdel Moamen** and Nadeem Jamali. CSSWare: A middleware for scalable mobile crowd-sourced services. *In Proceedings of the EAI International Conference on Mobile Computing, Applications and Services (MobiCASE '15)*, pages 181-199, Berlin, Germany, 2015. [Related to Chapter 3]

- **Ahmed Abdel Moamen** and Nadeem Jamali. ModeSens: An Approach for Multi-Modal Mobile Sensing. *In Proceedings of the ACM SIGPLAN Conference Companion on Systems, Programming, Languages and Applications (SPLASH '15)*, pages 40-41, Pittsburgh, USA, 2015. [Related to Chapter 5]

- **Ahmed Abdel Moamen** and Nadeem Jamali. CSSWare: An Actor-Based Middleware for Mobile Crowd-Sourced Services. *In Proceedings of the EAI International Conference on Mobile and Ubiquitous Systems (MobiQuitous '15)*, pages 287-288, Coimbra, Portugal, 2015. [Related to Chapter 3]

- **Ahmed Abdel Moamen** and Nadeem Jamali. ShareSens: An Approach to Optimizing Continuous Mobile Sensing Workloads. *In Proceedings of the IEEE International Conference on Mobile Services (MS '15)*, pages 89-96, New York, USA, 2015. [Related to Chapter 4]

- **Ahmed Abdel Moamen** and Nadeem Jamali. An Actor-Based Approach to Coordinating Crowd-Sourced Services. *International Journal of Services Computing (IJSC)*, vol. 2, pages 43-55, 2014. [Related to Chapter 3 and Appendix A]

- **Ahmed Abdel Moamen** and Nadeem Jamali. Coordinating Crowd-Sourced Services. *In Proceedings of the IEEE International Conference on Mobile Services (MS '14)*, pages 92-99, Alaska, USA, 2014. [Related to Chapter 3 and Appendix A]

References

[1] M. Ra, B. Liu, T. La-Porta, and R. Govindan, “Medusa: A programming framework for crowd-sensing applications,” in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’12, Lake District, UK, 2012, pp. 337–350.

[2] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor, “AutoMan: A platform for integrating human-based and digital computation,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA ’12, Arizona, USA, 2012, pp. 639–654.

[3] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos, “AnonySense: Privacy-aware people-centric sensing,” in *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’08, Breckenridge, USA, 2008, pp. 211–224.

[4] X. Liu, M. Lu, C. Ooi, Y. Shen, S. Wu, and M. Zhang, “CDAS: A crowdsourcing data analytics system,” *The Journal of the VLDB Endowment (PVLDB)*, vol. 5, no. 10, pp. 1040–1051, 2012.

[5] F. Ye, R. Ganti, R. Dimaghani, K. Grueneberg, and S. Calo, “MECA: Mobile edge capture and analysis middleware for social sensing applications,” in *Proceedings of the International Conference Companion on World Wide Web*, ser. WWW ’12 Companion, Lyon, France, 2012, pp. 699–702.

[6] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA, USA: MIT Press, 1986.

[7] H. Geng and N. Jamali, “Supporting many-to-many communication,” in *Proceedings of the ACM SIGPLAN Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ser. AGERE!@SPLASH ’13, Indianapolis, USA, 2013, pp. 81–86.

[8] N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. Campbell, “A survey of mobile phone sensing,” *Communications Magazine, IEEE*, vol. 48, no. 9, pp. 140–150, 2010.

[9] A. Moamen and N. Jamali, “Coordinating crowd-sourced services,” in *Proceedings of the Third IEEE International Conference on Mobile Services*, Alaska, USA, 2014, pp. 92–99.

[10] N. Vallina-Rodriguez and J. Crowcroft, “Energy management techniques in modern mobile handsets,” *Communications Surveys Tutorials, IEEE*, vol. 15, no. 1, pp. 179–198, 2013.

[11] U. Levine, A. Shinar, E. Shabtai, and Y. Shmuelevitz, “Condition-based activation, shut-down and management of applications of mobile devices,” *United States Patents, US 8,271,057*, 2009.

[12] R.-Y. Li, S. Liang, D.-W. Lee, and Y.-J. Byon, “TrafficPulse: A mobile gisystem for transportation,” in *Proceedings of the ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems*, ser. MobiGIS ’12, Redondo Beach, USA, 2012, pp. 9–16.

[13] M. Zook, M. Graham, T. Shelton, and S. Gorman, “Volunteered geographic information and crowdsourcing disaster relief: A case study of the haitian earthquake,” *The Journal of World Medical and Health Policy*, vol. 2, no. 2, pp. 7–33, 2010.

[14] L. Besaleva and A. Weaver, “CrowdHelp: A crowdsourcing application for improving disaster management,” in *Proceedings of the 2012 IEEE Conference on Global Humanitarian Technology*, ser. GHTC ’13, San Jose, USA, 2013, pp. 185–190.

[15] T. Uga, T. Nagaosa, and D. Kawashima, “An emergency earthquake warning system using mobile terminals with a built-in accelerometer,” in *Proceedings of the IEEE Conference on ITS Telecommunications*, Taipei, Taiwan, 2012, pp. 837–842.

[16] P. Jayaraman, C. Perera, D. Georgakopoulos, and A. Zaslavsky, “MOSDEN: Efficient opportunistic sensing using mobile collaborative platform,” in *Proceedings of The International Conference on Collaborative Computing: Networking, Applications and Worksharing*, ser. Collaboratecom ’13, Austin, USA, 2013, pp. 77–86.

[17] S. Nawaz, C. Efstratiou, and C. Mascolo, “ParkSense: A smartphone based sensing system for on-street parking,” in *Proceedings of the ACM International Conference on Mobile Computing Networking.* Miami, USA: ACM, 2013, pp. 75–86.

[18] H. Martín, A. M. Bernardos, J. Iglesias, and J. R. Casar, “Activity logging using lightweight classification techniques in mobile devices,” *The Journal of Personal Ubiquitous Computing*, vol. 17, no. 4, pp. 675–695, 2013.

[19] G. Bieber, T. Kirste, and M. Gaede, “Low sampling rate for physical activity recognition,” in *Proceedings of the 7th International Conference on PErvasive Technologies Related to Assistive Environments*, ser. PETRA ’14, Rhodes, Greece, 2014, pp. 15:1–15:8.

[20] L. Chen, J. Hoey, C. Nugent, D. Cook, and Z. Yu, “Sensor-based activity recognition,” *IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews*, vol. 42, no. 6, pp. 790–808, 2012.

[21] J. Sung, C. Ponce, B. Selman, and A. Saxena, “Unstructured human activity detection from rgbd images,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, St. Paul, USA, 2012, pp. 842–849.

[22] K. Yatani and K. N. Truong, “BodyScope: A wearable acoustic sensor for activity recognition,” in *Proceedings of the ACM Conference on Ubiquitous Computing*, ser. UbiComp ’12, Pittsburgh, Pennsylvania, 2012, pp. 341–350.

[23] S. L. Lau, I. Konig, K. David, B. Parandian, C. Carius-Dussel, and M. Schultz, “Supporting patient monitoring using activity recognition with a smartphone,” in *Proceedings of the IEEE International Symposium on Wireless Communication Systems*, York, UK, 2010, pp. 810–814.

[24] Y.-S. Lee and S.-B. Cho, “Activity recognition using hierarchical hidden markov models on a smartphone with 3d accelerometer,” in *Proceedings of the 6th International Conference on Hybrid Artificial Intelligent Systems - Volume Part I*, ser. HAIS’11, Wroclaw, USA, 2011, pp. 460–467.

[25] C. Hung-Ching, A. Agrawal, and K. Cameron, “Energy-aware computing for android platforms,” in *Proceedings of the International Conference on Energy Aware Computing*, Istanbul, Turkey, 2011, pp. 1–4.

[26] B. Dietrich and S. Chakraborty, “Managing power for closed-source android os games by lightweight graphics instrumentation,” in *Proceedings of the 12th Annual Workshop on Network and Systems Support for Games*, ser. NetGames ’12, Venice, Italy, 2012, pp. 1–3.

[27] U. Y. Ogras, R. Z. Ayoub, M. Kishinevsky, and D. Kadjo, “Managing mobile platform power,” in *Proceedings of the 2013 International Conference on Computer-Aided Design*, ser. ICCAD ’13, San Jose, USA, 2013, pp. 161–162.

[28] K. Kumar and Y.-H. Lu, “Cloud computing for mobile users: Can offloading computation save energy?” *The Journal of Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[29] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, ser. MobiSys ’10, San Francisco, USA, 2010, pp. 49–62.

[30] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh, "A framework of energy efficient mobile sensing for automatic user state recognition," in *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '09, New York, USA, 2009, pp. 179–192.

[31] H. Lu, A. J. B. Brush, B. Priyantha, A. K. Karlson, and J. Liu, "SpeakerSense: Energy efficient unobtrusive speaker identification on mobile phones," in *Proceedings of the 9th International Conference on Pervasive Computing*, ser. Pervasive'11, San Francisco, USA, 2011, pp. 188–205.

[32] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow, "SociableSense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing," in *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '11, Las Vegas, USA, 2011, pp. 73–84.

[33] K. Lorincz, B.-r. Chen, G. W. Challen, A. R. Chowdhury, S. Patel, P. Bonato, and M. Welsh, "Mercury: A wearable sensor network platform for high-fidelity motion analysis," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09, Berkeley, USA, 2009, pp. 183–196.

[34] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola, "The latency, accuracy, and battery (LAB) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications*, ser. OOPSLA '13, Indianapolis, USA, 2013, pp. 661–676.

[35] B. Priyantha, D. Lymberopoulos, and J. Liu, "LittleRock: Enabling energy-efficient continuous sensing on mobile phones," *The Journal of the IEEE Pervasive Computing*, vol. 10, no. 2, pp. 12–15, 2011.

[36] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *The Journal of Functional Programming*, vol. 7, no. 01, pp. 1–72, 1997.

[37] "The scala programming language," http://www.scala-lang.org, (accessed June 1, 2017).

[38] "Akka programming language," http://www.akka.io, (accessed June 1, 2017).

[39] "Erlang programming language," http://www.erlang.org, (accessed June 1, 2017).

[40] "Pony: An open-source, object-oriented, actor-model, capabilities-secure, high performance programming language," https://www.ponylang.org/, (accessed June 1, 2017).

[41] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with salsa," *The journal of SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, 2001.

[42] M. Jang, A. Momen, and G. Agha, *Efficient Agent Communication in Multi-agent Systems*, 2005, pp. 236–253.

[43] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: A comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09, Calgary, Canada, 2009, pp. 11–20.

[44] "Pulsar: Concurrent framework for python," https://pythonhosted.org/pulsar, (accessed June 1, 2017).

[45] "CAF: C++ actor framework," https://actor-framework.org, (accessed June 1, 2017).

[46] "Microsoft's Orleans: A framework to building distributed, high-scale applications in .NET," https://dotnet.github.io/orleans/, (accessed June 1, 2017).

[47] G. Agha, "Actors programming for the mobile cloud," in *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC)*, Marseilles, France, 2014, pp. 3–9.

[48] M. N. Huhns and M. P. Singh, "Service-oriented computing: Key concepts and principles," *IEEE Internet Computing*, vol. 9, no. 1, pp. 75–81, 2005.

[49] D. Kitchin, A. Quark, W. Cook, and J. Misra, "The orc programming language," in *Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, ser. FMOODS '09, 2009, pp. 1–25.

[50] P. P. Pereira, J. Eliasson, R. Kyusakov, J. Delsing, A. Raayatinezhad, and M. Johansson, "Enabling cloud connectivity for mobile internet of things applications," in *Proceedings of the 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, ser. SOSE '13, 2013, pp. 518–526.

[51] Y. Huang and H. Garcia-Molina, "Publish/subscribe in a mobile environment," *The Journal of Wireless Networks*, vol. 10, no. 6, pp. 643–652, 2004.

[52] A. Morales, R. Alcarria, E. Cedeo, and T. Robles, "An extended topic-based pub/sub broker for cooperative mobile services," in *Proceedings of the International Conference on Advanced Information Networking and Applications Workshops*, Barcelona, Spain, 2013, pp. 1313–1318.

[53] S. Kalasapur, M. Kumar, and B. A. Shirazi, "Dynamic service composition in pervasive computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 7, pp. 907–918, 2007.

[54] S. Deng, L. Huang, J. Taheri, J. Yin, M. Zhou, and A. Y. Zomaya, "Mobility-aware service composition in mobile communities," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 3, pp. 555–568, 2017.

[55] W. Tan, Y. Fan, M. Zhou, and Z. Tian, "Data-driven service composition in enterprise soa solutions: A petri net approach," *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 3, pp. 686 – 694, 2010.

[56] P. Wang, Z. Ding, C. Jiang, and M. Zhou, "Constraint-aware approach to web service composition," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 6, pp. 770–784, 2014.

[57] S. Deng, H. Wu, D. Hu, and J. L. Zhao, "Service selection for composition with qos correlations," *IEEE Transactions on Services Computing*, vol. 9, no. 2, pp. 291–303, 2016.

[58] H. E. Mansour and T. Dillon, "Dependability and rollback recovery for composite web services," *IEEE Transactions on Services Computing*, vol. 4, no. 4, pp. 328–339, 2011.

[59] A. H. Gandomi and A. H. Alavi, "Krill Herd: A new bio-inspired optimization algorithm," *The Journal of Communications in Nonlinear Science and Numerical Simulation*, vol. 17, no. 12, pp. 4831–4845, 2012.

[60] Y. Wang, "An FSM model for situation-aware mobile application software systems," in *Proceedings of the ACM Annual Southeast Regional Conference*, Huntsville, USA, 2004, pp. 52–57.

[61] A. Matic, V. Osmani, A. Maxhuni, and O. Mayora, "Multi-modal mobile sensing of social interactions," in *Proceedings of the IEEE Pervasive Computing Technologies for Healthcare*, San Diego, USA, 2012, pp. 105–114.

[62] D. Chiu, O. Lee, H. fung Leung, E. Au, and M. Wong, "A multi-modal agent based mobile route advisory system for public transport network," in *Proceedings of the International Conference on System Sciences*, Hawaii, 2005, pp. 92–99.

[63] N. Jamali and X. Zhao, "Hierarchical resource usage coordination for large-scale multi-agent systems," in *Massively Multi-Agent Systems I.* Springer Berlin Heidelberg, 2005, pp. 40–54.

[64] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES/ISSS '10, Scottsdale, USA, 2010, pp. 105–114.

[65] W. Su and I. Akyildiz, “Time-diffusion synchronization protocol for wireless sensor networks,” *IEEE Transactions on Networking*, vol. 13, no. 2, pp. 384–397, 2005.

[66] Q. Li and D. Rus, “Global clock synchronization in sensor networks,” *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 214–226, 2006.

[67] “Spark: A large-scale data processing framework,” http://spark.apache.org/, (accessed June 1, 2017).

[68] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Boston, MA: Springer US, 2011, ch. Internet of Things, pp. 307–323.

Appendix A

Case Studies For Multi-Origin Communication

This chapter presents two case studies to illustrate the use of the two multi-origin communication primitives discussed in Chapter 3. Both examples use the continual multi-origin communication primitive; one-off versions of the examples can be easily adapted from these solutions using the one-off communication primitive.

A.0.3 A Restaurant Recommendation Service

Consider the type of restaurant recommendation service introduced earlier in Section 3.4.1. This service can be launched by creating and launching of a service actor, which in turn makes a number of calls to set up continual multi-origin messages, one for each restaurant, each geographical area, etc., depending on the degree of distribution required or desired. The `start` method in Figure A.1 shows how this could be done if a separate coordination were needed for each restaurant. The restaurants of interest are chosen, assigned unique IDs, and placed in a `restIDList`. Then for each `restID`, mobile devices in and near the restaurant are identified, say by tracking automatic *check-ins*. Finally, a call is made to set up a multi-origin communication primitive for each restaurant, with the nearby devices identified as the constituency.

```
void start() {

    * choose restaurants to track; assign them IDs; place them in restIDList with coordinates *
     for each restID in restIDList {
         * collect names of devices in or near restaurant ID *
         continualCommSetup(restCoordClass, deviceNameList, sigChange, null, restID);
     }
}

rankedRestList getView(location, rankParams) {
     return rank(filter(restIDList, location), rankParams);
}

void update(stateUpdate, restID) {
    * update global state with restID's new state *
}
```

Figure A.1: Methods Defining Behavior of Restaurant Service Actor

Additional parameters specify the condition indicating significant change in the restaurant state warranting an update to the server, and null to indicate that there is no set interval at which updates must be made. Each of these calls creates a local restaurant coordinator which invites event updates from current diners' devices. The devices in turn have applications installed to tap into sensor feeds to recognize significant events, such as arriving at the restaurant, being seated at a table. If there are a number of similar services that the device's owner is interested in, then each would interpret the sensor feeds for the purposes of that service. As an event gets recognized by a device, it sends a message to its restaurant coordinator, invoking the coordinator's `sendMessage` method (Figure A.2). `sendMessage` records the event in `eventList` and checks to see whether the event represents a significant change in the restaurant's state, and if so, sends an `update` message to the restaurant service – known to the coordinator by its actor name `serviceName` – to report the change. Invocation of `update` in the service updates the global state with the new information. In a real system, it would also make sense for both the restaurant coordinators as well as the global service to use aging functions to lower the relevance of obsolete information.

```
void sendMessage(deviceName, event, restID) {
      * record received event in eventList *
      if (sigChange(eventList))
           serviceName <- update(aggr(eventList), restID);
}
```

Figure A.2: Methods Defining Behavior of a Regional Coordinator Actor

A user searching for restaurants would call the `getView` method on the server with `location` and `rankParams` as parameters, where `location` specifies the user's geographical coordinates, and `rankParams` specify the metrics by which to rank the restaurant (such as by the wait time). The server filters the restaurant list for relevance according to the user's location, and then creates a ranking using `rankParams` to be returned to the client.

A.0.4 Twitter-like Messaging Service

Consider the Twitter service introduced earlier in Section 3.4.1. This service can be launched by the creation and launching of the messaging service actor, whose behavior is to receive requests for creation of new discussions with identified constituencies. These requests are received in the form of `createDiscussion` message sends as shown in Figure A.3.

When the service receives this message, it assigns a new ID – `discussionID` – to identify the discussion topic by, and calls the continual multi-origin communication setup primitive `continualCommSetup` with parameters specifying the discussion coordinator's behavior (`discCoordClass`), the `constit`, `null` for the update condition, `updateInterval` specifying the length of the intervals after which the service should receive updates from the coordinator, and finally `discussionID` to tell the coordinator its discussion topic ID. This call creates a dedicated discussion coordinator for that discussion, which in turn announces the discussion to the constituency. Once invited, members of the constituency are free to send messages to the discussion coordinator in the form of an asynchronous message invoking its `sendMessage` method (shown in Figure A.4).

`sendMessage` takes as parameter a list `voteList` of *(message, weight)* pairs, where *message* is either a new message drafted by the sender, or an existing message previously sent to the service (a ranked list of which can be obtained by calling the `findMessages` method of the messaging server), and *weight* indicates the proportional weight that the sender intends that message to have of their vote. Each sender has exactly one vote for any discussion, which they are free to distribute between various messages under their discussion.

The service can have various types of users, subscribed to different views of the discussions' states provided by custom interfaces (see Figure 3.14). When a user requests subscription to a particular type of view – `viewType` – after authentication, it is sent the view (by having a `view` message sent to it), and is also added to a subscriber list to be sent future updates. The types of view may include a view for an analyst interested in tracking trends, or even a view for a message sender interested in staying uptodate about a discussion to possibly revise their votes.

On receiving a `sendMessage` message, the discussion coordinator first updates `recentUpdates` to reflect the new messages received, and then checks to see if it is time to aggregate received messages and report back to the service. If it is time,[1] it aggregates the updates and reports them to the server using an `update`

[1]If messages are infrequent, a clock could be asked by the service to interrupt the coordinator at the end of each interval.

```
void createDiscussion(discussionTitle, constit) {
     * assign unique ID to discussionTitle *
     continualCommSetup(discCoordClass, constit, null, updateInterval, discussionID);
}

void getView(userName, userID, discussionID, viewType, viewParams) {
     authenticate(userName,userID);
     userName <- view(filter(state, discussionID, userID, viewType, viewParams));
     * add userName's record to the subscriber list *
}

rankedMessageList findMessages(userName, discussionID, keywords) {
     * create ranked list of existing messages relevant to keywords *
     return * ranked message list *;
}

void update(votesUpdate, discussionID) {
     * update state with votesUpdate *
     for each entry e in subscriber list {
          e.userName <- view(filter(state, e.discussionID, e.userID,
                                    e.viewType, e.viewParams));
     }
}
```

Figure A.3: Methods Defining Behavior of Messaging Service Actor

message, which invokes the corresponding method in the server. The server's `update` method updates the state of the discussion, and then for every entry in the list of service subscribers, sends them the view that they are subscribed to.

The service maintains the current state for all discussions. In practice, the service itself could be distributed into a number of actors, each handing any number of discussions.

There are some noteworthy features of this approach. First, message contributors are authenticated, and the voting is fair in that each contributor has the same one vote in any discussion, which they may divide among the multiple messages they support. Second, the constituency for each discussion is explicitly specified. This would allow this approach to be used for holding credible votes. Third, the approach naturally aggregates by allowing contributors to vote for existing messages rather than having them send a fresh message each time.

```
void sendMessage(userName, userID, voteList) {
     authenticate(userName, userID);

     * record received votes in recentUpdates *

     if (currentTime >= lastAggregate + interval) {
         serviceName <- update(aggr(recentUpdates), discussionID);
         lastAggregate += interval;
     }
}
```

Figure A.4: Method Defining Behavior of a Discussion Coordinator Actor