Subiquitous: Supporting Ubiquitous Computing

James A.G. Henry

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Manuel A. Pérez-Quiñones, Chair
Stephen H. Edwards
Steve R. Harrison

April 29, 2010
Blacksburg, Virginia

Keywords: Ubiquitous Computing, Software Architecture
Copyright 2010, James A.G. Henry
Ubiquitous computing describes a world in which technology invisibly assist us in our everyday activities. Unfortunately, development of ubiquitous software has fallen behind advances in available hardware and high-speed networking. Subiquitous is a software platform to support the development and deployment of applications in a ubiquitous computing environment. The goal of Subiquitous is to provide flexible support for a variety of ubiquitous application structures and distributions as well as to support the rapid development and zero configuration, user friendly set-up of those applications.

The Subiquitous system consists of two basic parts. First, it provides a client-server architecture to support the deployment and communication of Subiquitous applications. Second, it provides an application framework used to build Subiquitous applications. The framework, in collaboration with the Subiquitous server and client, provides service discovery, transparent and flexible communications, code distribution, and application organization.

To demonstrate Subiquitous contributions toward the improvement of ubiquitous software, a number of example ubiquitous applications were developed. The examples demonstrate: a) distribution of Subiquitous applications to different devices along the Model-View-Controller separation, b) running of the same application in multiple devices supporting easy communication between devices, c) a resource-server with multiple clients all sharing data from a single location, and d) an existing complex application with a Subiquitous wrapper that supports moving user interaction from one device to another. Each example application requires zero user configuration and includes no more than thirty lines of Subiquitous code to support user interaction across multiple devices in the home.
# Contents

1 Introduction  
1.1 Motivation ............................................................... 1  
1.2 Previous Work ......................................................... 2  
1.3 Ubiquitous Home ........................................................ 2  
1.4 Research Objective .................................................... 4  

2 Related Work  
2.1 Hardware-centric ...................................................... 6  
2.2 Application-centric .................................................... 8  
2.3 Summary .................................................................... 9  

3 Subiquitous Implementation  
3.1 SU Server ................................................................. 12  
3.1.1 Dynamic Service Discovery ........................................ 12  
3.1.2 Communication Service ............................................. 14  
3.1.3 Code Distribution Service .......................................... 15  
3.1.4 Contribution of SU Server to Solution .......................... 15  
3.2 SU Client ................................................................. 16  
3.2.1 SU Application Installation ........................................ 16  
3.2.2 Communication Service ............................................. 18  
3.2.3 User Interface .......................................................... 19  
3.2.4 Contribution of SU Client to Solution .......................... 20
3.3 SU Modules ......................................................... 21
  3.3.1 Structure ...................................................... 21
  3.3.2 Contribution of the SU Module to Solution .................. 23
3.4 SU Application ..................................................... 23
  3.4.1 Contribution of the SU Application to Solution ............... 25
3.5 Services ........................................................... 25
  3.5.1 Service Discovery ............................................. 25
  3.5.2 Communication ............................................... 26
  3.5.3 Code Distribution ............................................. 27
  3.5.4 Service Comparison .......................................... 29

4 Sample Applications .................................................. 31
  4.1 Instant Messenger ............................................... 31
  4.2 Picture Share .................................................... 33
  4.3 Lights Controller ............................................... 36
  4.4 YouTube Follower ............................................... 38

5 Conclusion ........................................................... 42
  5.1 Approach ......................................................... 42
  5.2 Evaluation ....................................................... 43

6 Future Work .......................................................... 45
  6.1 SU Application Installation ....................................... 45
  6.2 SU Client Interface .............................................. 46
  6.3 Code Distribution ............................................... 46
  6.4 Failure Handling ................................................ 47
  6.5 System Parameters .............................................. 47
  6.6 Implementation Issues ......................................... 47
List of Figures

3.1 Subiquitous Block Diagram. One SU Server per system, one SU Client per device, and many SU Apps per device. ............................................. 12
3.2 SU Server Class Diagram. ....................................................... 13
3.3 SU Client Class Diagram. .................................................... 17
3.4 SU Client User Interface. ................................................... 20
3.5 SU Module Class Diagram. .................................................. 22
3.6 Service Discovery Sequence Diagram. ................................... 26
3.7 Communication Sequence Diagram. ..................................... 27
3.8 Code Distribution Sequence Diagram. .................................. 28

4.1 Instant Messenger Application Interface. ............................... 31
4.2 Instant Messenger Application Structure. The IM Module is ClientShared and ClientRun ................................................... 32
4.3 Picture Viewer Application Interface. ................................. 34
4.4 Picture Sharer Application Structure. The Share Module is ClientRun and the View Module is ClientShared .............................. 35
4.5 Lights Controller Application Interface. ................................. 36
4.6 Light Controller Application Structure. The Light Module is ServerRun and the Control Module is ClientShared .......................... 37
4.7 YouTube Player Application Interface. ................................. 39
4.8 YouTube Player Application Structure. The Play Module is ClientRun and ClientShared .................................................. 40
List of Tables

2.1 Comparison of ubiquitous solutions ........................................ 10

3.1 Comparison of ubiquitous solutions including Subiquitous .............. 29
Chapter 1

Introduction

When Mark Weiser coined the term “Ubiquitous Computing” (ubicomp) he envisioned technologies that “weave themselves into the fabric of everyday life until they are indistinguishable from it” [22]. He noticed that computers were hindering actions by taking attention from the act and placing it on the tool. Instead, he believed computers should invisibly supplement our efforts to complete a goal. In this thesis I explore the current field of ubicomp research and its progress toward meeting Weiser’s vision. Using previous work as a guide, I show that zero configuration and user friendly deployment of ubiquitous applications, flexible support of application structure and distribution, and support for rapid development are all key features for a successful ubicomp platform. This thesis presents Subiquitous, a ubiquitous platform designed to meet these requirements. Through this research and the implementation of Subiquitous I hope to advance the field of ubiquitous software.

1.1 Motivation

Weiser’s first vision of ubicomp describes a world that contains hundreds of devices of varying size and complexity per person, all unobtrusively doing their part to make the tasks we face in our everyday lives easier to accomplish. In particular, he presents three devices, the tab, pad, and board as examples of the future of ubicomp. These devices are meant to mirror the common use of sticky notes, note pads, and whiteboards. He envisioned that these devices would be used the same way as their existing counter parts, not because they are required to complete a task but because they are convenient and helpful. Weiser ends this glimpse into the future of ubicomp by highlighting the issues that, at the time, must be resolved to make this vision a reality. The three challenges named were the advancement of cheap low power hardware, improvement of high speed wireless network protocols, and development of software that supports and takes advantage of the possibilities offered by ubicomp [22, 23].

In the years since this influential work great strides have been made in the advancement
and availability of hardware devices as well as high speed wireless networks. Progress in these fields has brought us closer than ever to Weiser’s ubicomp vision. Comparatively, development and adoption of ubiquitous platforms and applications has been less substantial. However, as the number of devices we integrate into our lives increases, so too does the need for these platforms. Users of ubiquitous technology are becoming burdened by the task of handling this confusing tangle of devices and would benefit greatly from a means of seamlessly connecting networked devices.

1.2 Previous Work

There has been a great deal of research devoted to achieving this vision of ubiquitous computing and addressing the increasing need for ubiquitous solutions in today’s computing environment. Weiser, while discussing the future of software for ubicomp, asked “is it better to have a single application with multiple windows, or many applications independently connected” [23]. Though the line blurs at times, ubicomp research can often be categorized using these two patterns. The group supporting multiple independently connected applications looks at ubicomp from a hardware-centric viewpoint. The applications in a ubiquitous environment exist to discover and utilize hardware components, such as sensors or a printer. In these environments, high level applications typically exist on one machine and make direct connections to resources in the system, leasing and sharing them with other applications [3, 8]. In other words, applications exist to use the hardware.

On the other hand, the group that promotes single applications with multiple windows looks at ubicomp in an application-centric way. In such an environment applications can exist that do not use any low-level hardware resources. The focus is on distributing applications across the network to devices that can use them. Applications can be separated and distributed according to software architectures, such as Model-View-Controller, or network architectures, such as Client-Server. In addition to hardware resources, applications can serve as software resources and systems can be augmented one additional application or device at a time. Connections between resources and applications are typically done through event or resource based communication. In this way an application can be updated by a component in the background with little knowledge or information about its existence [1, 9, 24]. In other words, hardware exists to support applications.

1.3 Ubiquitous Home

Imagine sports fans that want to be up to date on the latest sports information. Currently, in the morning, as they get ready for work, they may move from device to device tuning it to their favorite broadcast for sports information. They may start by moving into the bathroom, turning on the radio, and searching for the sports station. Then they might move
into the bedroom and switch the television to their favorite channel. Every time they switch locations they must turn off one device and configure another.

In a ubiquitous home, the same sports fan could be provided all of this information automatically. He may walk into the bathroom and the system could detect his presence, turn on a television embedded in the mirror, and tune to a sports channel leaving him free to enjoy the program as he brushed his teeth and shaved. Moving into the bedroom to get dressed the system could again detect his movement, shut off the television in the bathroom, and turn on the television in the bedroom starting the program exactly where it left off in the bathroom. A display in the kitchen could show today’s weather and news while the television program from the bedroom continued audio only form. While leaving the house he could use a display embedded in the wall near the door to turn off all the lights in the house and stop the system from monitoring his location.

In this vision of the Ubiquitous home, a number of integrated devices work together as a single system to provide a seamless service to the user. Such a system could work by having an application installed that is able to use any audio/visual output, such as the television, as a resource to present information to the user. The same application could listen for location events sent by a separate application monitoring the user’s location. The system would not be dependent on the number and type of displays or the complexity of the location monitoring and could be augmented one piece at a time. To start the system may use infrared motion detectors as a basic way of tracking the users location and have displays in only the bedroom and bathroom. However, the user could purchase a more complex camera based location monitoring system and, once plugged into the system, all interested applications would automatically begin to receive these more complete location updates. In addition, the user could buy another display to place in the living room and the application presenting sports broadcasts would automatically be able to utilize that display to expand the service.

This vision describes a consumer model for ubiquitous computing, in which users can purchase devices separately to build up their ubiquitous environment as the need or opportunity arises. These devices could be purchased in different locations and produced by different companies but all would share a “Ubiquitous Ready” label indicating they will automatically work with a ubiquitous environment. All of these devices would connect to each other wirelessly over the home local area network. Access to the system would be restricted to the local network and privacy would be provided by network security at the router.

Which software approach can best support future progress toward this vision of ubiquitous computing? A device, such as a temperature gauge, could be a small standalone product that has the means of unobtrusively displaying its own information. Another option is that the temperature gauge is a discoverable device that can be found, queried, and will respond to the requesting device. A different possibility is that the device broadcast data to be picked up by any device on the network able of using this information or it could provide capable devices on the network with a user interface to display and interact with its information and options. Each of these options improves upon the last and builds a more complete vision of
ubicomp. The most effective system should provide the last, most robust option, and ideally support the others. Only an application-centric approach provides this level of flexibility within a ubiquitous environment.

### 1.4 Research Objective

Ubiquitous software must be advanced to the level of today’s hardware and wireless networks in order to make this vision a reality. However, current ubiquitous software solutions are complex, lack versatility, and many remain theoretical. Of the solutions proposed in the literature, application-centric platforms provide the greatest flexibility in supporting ubiquitous environments. The research objective of this thesis is the exploratory development of an application-centric ubiquitous platform for the development and deployment of ubiquitous applications. We envision a home ubiquitous computing environment that mixes desktop computing with consumer ubiquitous devices. Users should be able to join the system, utilize available applications, and add resources with little to no effort or configuration, much like home desktop computing today. The platform should support any applications that would like to take advantage of the accessibility of ubiquitous environments and remain flexible towards how these applications are structured and distributed. Finally, the platform should promote rapid application development by providing assistance to developers creating robust programs using ubicomp features. Ubiquitous software has fallen behind advancements in hardware and networks but a platform with all of these features will bring ubiquitous applications a step closer to providing the accessibility, convenience, and ubiquity required by Weiser's original vision.

In the remainder of this thesis, I begin by reviewing related research in the field of ubicomp. The discussion for this research will be framed by the hardware-centric and application-centric distinction and I will compare their achievements to the vision of ubiquitous computing. Next, I present and describe the features of Subiquitous, a platform to support ubiquitous applications. The goal of Subiquitous is to meet all of the requirements named above, particularly user friendly zero configuration set-up and use, flexible support for a variety of ubiquitous application structures and distribution across devices, and rapid development of ubiquitous applications. To assess the effectiveness of the Subiquitous system I have implemented a number of proof-of-concept applications. These ubiquitous applications will be explained and I will discuss their features in relation to the goals of Subiquitous. Finally, I discuss the conclusions and ideas for future work drawn from developing the Subiquitous platform and the ubiquitous applications that use it.
Chapter 2

Related Work

In an early, genre defining paper from the man who coined the phrase “Ubiquitous Computing”, Weiser discusses ubiquitous computing, explaining that its goal is to integrate computers seamlessly into everyday life [22]. He goes on to explain that ubicomp is in opposition to a number of trends being advanced in computer science, including virtual reality. Virtual reality attempts to replace the real world with a simulation whereas ubicomp attempts to invisibly augment the real world. Weiser goes on to describe three possible ubiquitous devices of different scales, tabs, pads, and boards, which can mirror the use of sticky notes, notepads, and whiteboards respectively. He envisions hundreds of tabs, a dozen pads, and one or two boards will be scattered and placed all over a ubiquitous environment much like their modern equivalents are today. They will provide the same convenience as their counterparts but also allow for other functions facilitated by their connection to one another, such as using a pad to change the information on a board.

Weiser next touches on the technological advancements that will be required in the coming years to make ubicomp a reality. These include cheap, low-power devices and displays, software targeted at the ubiquitous environments, and high speed wireless networks. In a second paper published later, Weiser continues his discussion of the challenges facing computer science in advancing ubicomp [23]. He frames this more detailed discussion by again describing tabs, pads, and boards. Weiser goes into more detail here but the major categories remain the advancement of hardware, networks, and software for the ubiquitous environment. In the years following the publication of these papers, progress in cheap low-powered devices and high speed wireless networks has advanced quite a deal and these technologies are prevalent in today’s world. On the other hand, ubiquitous software has not shared the same rate of advancement or adoption.

While describing a shared drawing application, Weiser questions whether it is “better to have a single application with multiple windows, or many applications independently connected?” He also asks what is the best option for handling communication between ubiquitous software [23]. He indicates that ubiquitous applications should do more then make other devices,
such as a printer, appear to be connected to a user’s computer. Ubiquitous applications must “exploit the unique capabilities of physically dispersed computers” [22]. To do this, platforms will need to be created with ubiquitous applications in mind. These platforms must support dynamic systems with devices constantly connecting and disconnecting. A successful system must be flexible in supporting communication between applications and the distribution of one application across multiple devices. Finally, to be in line with the principles of ubiquitous computing a successful platform should be invisible to the user. This work leads future researchers to wonder what applications will be possible with ubiquitous computing and what the best methods to support this vision are.

Since these two influential papers, many researchers have attempted to further categorize and define the research required in the field of ubiquitous applications. In one such paper, Raatikainen, et al. discuss middleware support for ubiquitous applications [16]. This vision paper outlines the challenges faced in meeting the requirements of middleware support for mobile and pervasive computing. The authors promote the importance of reconfigurable applications in supporting the user of ubicomp. This includes application partitioning, co-operation, and movable service sessions. Application partitioning is the easy separation of an application into distributable parts. Co-operation is the simultaneous execution of applications and their parts on separate machines. Movable service sessions are application states that follow the user from device to device. The requirements these authors place on a platform for supporting ubiquitous computing is summed up best by the statement that these platforms “should support fast service development and deployment. It should make it easy to divide the application logic into co-operating parts, to distribute and configure these components” [16].

We will now take a look at research that has been done to meet Weiser’s original vision of ubiquitous computing. The ways in which this research meets or fails to meet the requirements of platforms for ubiquitous applications will be discussed. This discussion will be framed by two general categories of platform approaches, hardware-centric and application-centric ubiquitous software.

### 2.1 Hardware-centric

Many service discovery techniques have been created to deal with increasingly dynamic networks of connected devices. Protocols such as Jini and UPnP allow users to easily discover and use devices that provide a service to the network such as a printer or webcam. Harihar and Kurkovsky discuss the strengths and weaknesses of Jini in support of one type of ubiquitous environment, namely pervasive environments [8]. Using a multicast request protocol Jini provides dynamic service discovery. Through these devices the Jini federation can offer services to the system to be consumed by other member devices. Jini also supports runtime downloading of the code required to utilize a service by passing serialized, instantiated objects from the service to the service user. Communication between services is normally
Related Work

handled through remote method invocation or a tuple spaces implementation. Jini is service oriented and hardware-centric; its main goal is to connect an application running on a machine to the hardware resources needed to provide a service to the user.

Friday, et al. discuss the limitations of Jini and other service discovery techniques in meeting the requirements of a ubicomp system [6]. Service discovery techniques are difficult to scale. Each time a service is needed a query is broadcast and a response may be sent from multiple services. This can cause heavy network traffic and responses may be missed. Both of these can frustrate users and work against the technology fading into the background. Jini does allow services to provide code to clients, providing better access to the services features. However, Jini’s support of code distribution is limited along with its support for application flexibility. Support for application partitioning and communications are rudimentary and it is difficult to access a services current state, e.g. currently printing. These limitations lessen its effectiveness as a flexible solution to meet the requirements of the ubicomp vision.

Middleware solutions are also being explored as a means to lessen the complexity of large dynamic networks. These systems add a level of abstraction above the low-level software and hardware infrastructure that can be used to more easily and rapidly develop robust applications that take advantage of the network and its resources. These solutions have proved promising in tackling ubiquitous computing needs. Metaglue [3] is a “specialized language for building systems of interactive, distributed computations” which the authors believe provides “general computational properties and requirements for” intelligent environments [3]. Metaglue consists of a runtime platform allowing “agents” to utilizes Metaglue’s capabilities as well as a Java extension to support creating new Metaglue agents. Metaglue’s capabilities include: configuration management, establishing and maintaining the configuration each agent specifies, establishing communication channels between agents, maintaining agent state, introducing and modifying agents in a running system, manage shared resources, event broadcasting, and support for debugging.

Metaglue agents refer to each other by capabilities, decoupling the resource from the device it is running on. When resources are found a direct connection is made from the resource to the requesting device. Metaglue handles information such as agent configuration and state in a database. While this is a flexible system, it is still hardware-centric, facilitating applications running on single devices to utilize sensors and other hardware to complete complex tasks. Metaglue does not allow for easy application partitioning. Many functions not directly related to the ubiquity of applications are implemented by the Metaglue system, adding further complexity.

Many middleware systems are bound to particular pervasive environments with a limited functional scope, such as supporting office meetings [18]. In addition, these systems are focused on simplifying the use of an existing set-up while installation and initialization typically remain complicated. The specialization of these frameworks, as well as other hardware-centric middleware solutions, limits their usefulness as general ubiquitous platforms.
2.2 Application-centric

Unlike hardware-centric solutions, application-centric platforms focus on flexible support for a variety of ubiquitous application separation, distribution, and communication needs. Arregui, et al. discuss ubiquitous application communication [1]. The authors believe that in order to meet the new requirements of the growing ubiquitous world a shift must occur from a system-centric to a user-centric approach. The boundaries of systems must be blurred and application must be “seamlessly integrated with activities through ubiquitous user interfaces.” They present STITCH, a middleware for building user-centric, context aware applications from distributed heterogeneous components including hardware and software. STITCH combines resource based programming such as tuple spaces with event based programming to provide the best of both messaging paradigms. As opposed to hardware-centric, this approach provides more flexible, loose coupling between components. The authors state that events “are very well adapted for dealing with mobility, disconnection and reconfiguration that are most common in ubiquitous environments.”

Hoareau and Maho discuss ubiquitous application distribution [9]. Their work is founded on the belief that “a ubiquitous application is supposed to render its services everywhere”. The authors suggest that ubiquitous applications should be hierarchical structures, high level applications created by grouping lower-level components. In particular, this paper describes a scheme for distributing hierarchical components in a dynamic pervasive environment. The main challenge is dealing with the heterogeneity and volatility of the hosts. Using a deployment descriptor, components with the hierarchy are distributed to the proper hosts. These components provide interfaces to the application that can be activated or deactivated according to availability of the main application. The authors go into great detail about the deployment specifications, actions of deployment, and result of disconnections, however, their research does not address communication between distributed components on separate clients. Although some hardware-centric approaches do support shared code and research has been done to improve code sharing capabilities of protocols such as Jini [2], these do not provide the flexibility of code distribution required by ubiquitous applications.

Zaidenberg, et al. present a tool which combines ubiquitous application communication and code distribution into a single tool [24]. The authors propose an architecture for supporting the building of ubiquitous applications constructed of numerous interconnected modules. In order to facilitate this, their architecture provides “communication and interconnection of modules, service discovery, a platform for easy deployment, (and) dynamic re-composition without restart.” These services are provided by two recent technologies: OMiSCID[5], Object Oriented Opensource Middleware for Service Communication Inspection and Discovery, and OSGi[10], Open Services Gateway initiative, as well as a central computer overseeing operations and containing all available modules. OMiSCID provides service discovery and communication while OSGi facilitates the dynamic deployment of modules. The messages between modules are in XML and the connection between modules are direct, like sockets, as opposed to an event/tuple spaces produce consume model. They detail an example personal
ubiquitous assistant using this framework.

Many people may be familiar with the X Window System [17], also known as X11, present in UNIX based operating systems. The X Window System is a software framework and network protocol to support the development of application graphical user interfaces that can be used on distributed computers. In other words, an application that implements the X Window protocol can provide a graphical interface to a computer across a network from which it can be controlled. In many ways, this is an application-centric approach for desktop computing. Its focus is on the distribution and access of applications across distributed devices. However, there are a number of limitations which make it a poor solution for ubiquitous computing. The information which it communicates between machines and the way in which it does so is rigid. The X Window System is set up using a client-server architecture where low-level graphical events, such as mouse click coordinates, are sent directly between machines. As a result, the X Window System does not support anything but the distribution and use of thin graphical clients to access application resources. Ubiquitous platforms should support the use of distributed resources that are not graphical in nature; the X Window System fails to do this. Ubiquitous platforms typically focus on accessing resources from multiple devices at the same time from each connected computer. The X Window System, on the other hand, focuses on accessing resources in a single machine from multiple computers. Finally, the X Window System is not zero-configuration and as a result does not invisibly integrate into everyday tasks as a ubiquitous solution should.

### 2.3 Summary

Many of the research ideas presented in these papers come close to meeting the requirements set by Weiser’s vision of ubicomp. In summary, Weiser envisioned a world where information provided by computer software blends naturally into our everyday lives and activities. From previous work, I have composed the following set of key features required for an effective ubiquitous application platform:

1. zero-configuration and user friendly set-up and use
2. flexible support of application structure and distribution
3. support for rapid development

I argue that hardware-centric approaches constrain the true possibility of ubicomp and fail to meet these requirements. Their feature sets remain too limited, complex, or are relevant to only a particular use. On the other hand, the more recent application-centric approach to ubicomp provides the flexibility required to take full advantage of ubiquitous environments. However, these solutions remain theoretical, partial, or require third-party code. Table 2.1
Table 2.1: Comparison of ubiquitous solutions

<table>
<thead>
<tr>
<th>Feature</th>
<th>Application-centric</th>
<th>Hardware-centric</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X Windows</td>
<td>OMiSCID/OSGi</td>
</tr>
<tr>
<td>Zero-configuration</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Dynamic configuration</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Low Overhead</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Application partitioning</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Co-operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Movable Sessions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Native Resource Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Repeated Resource Discovery</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Direct Communication</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Event-based</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Resource-based</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

compares the characteristics of previous work in the field of ubicomp software. The solutions are divided by their preference for either hardware-centric or application-centric features.

The next chapter presents an explanation of the Subiquitous platform, an application-centric ubiquitous framework designed to address these key features and support ubiquitous applications.
Chapter 3

Subiquitous Implementation

To meet the requirements described above I developed the Subiquitous platform. Subiquitous was built from the ground up using the Java programming language. No extensions or third party software was used. Subiquitous consists of three parts: the SU Client, which runs on each participating machine; the SU Server, which runs on one machine and monitors and supports the whole system; and the SU Module framework, which provides support code for the development of SU Applications.

In Figure 3.1 we can see a block diagram of Subiquitous. Built on the Subiquitous framework are the standalone SU Client and SU Server applications which handle the deployment and communications of SU Applications. Each machine connected to the system runs an instance of the SU Client and only one instance of the SU Server exists per system. Each SU Client can only connect to one SU Server, so the SU Server defines a Subiquitous system. The SU Module is part of the Subiquitous development framework and is extended by developers. Any number of developer implemented SU Modules can be combined to create as SU Application which run on the SU Clients.

In combination, the mechanisms supplied by Subiquitous facilitate the development and deployment of applications written to take advantage of a ubiquitous computing environment. In particular, the functionality provided to assist developers include dynamic service discovery, tracking of service availability, and transparent communication. To assist the home user in deployment and use of the system provides low-configuration dynamic installation, code distribution, and module organization. The following section contains a description of the inner working of each part of the Subiquitous platform.
3.1 SU Server

The server performs a central and vital role in the Subiquitous system. While in development I considered the idea of a decentralized system much like Jini, in which clients would be in charge of connecting with each other and maintaining the system. However, I decided that the benefits of a centralized server outweighed the cost of relying on one machine and program for the stability of the entire platform. For the ubiquitous home scenario that I presented earlier, it seemed reasonable that the home would have a computer that could function as a server. There is only one instance of the server program running for each Subiquitous network to which all participating SU Clients connect. The server can run on a dedicated machine or it can run on a machine shared with an SU Client. This centralized server approach cuts down on network traffic and complexity. Instead of requiring a multitude of requests for available services and a web of direct connections between SU Clients and SU Applications each client is only required to make a single connection to the server [6]. The SU Server’s three main roles within Subiquitous include dynamic service discovery, a communications service, and a code distribution service. Figure 3.2 shows the six primary classes related to the operation of the SU Server.

3.1.1 Dynamic Service Discovery

Service discovery protocols add a layer of indirection on top of network communication that allow zero-configuration connections to be made between services running on dynamic and heterogeneous networks [8, 13]. This ability makes service discovery an attractive option for handling connections in ubiquitous computing environments. Typically, service discovery works by sending a multicast datagram packet requesting a particular service from the network. All systems providing services listen for this packet and when they receive one
requesting their service they respond with the information required to make a direct connection, including at least the IP address and port number. Once the requesting system receives this information a direct connection is made to the service.

Service discovery systems are often used to query the network for available services and create multiple direct connections to each desired service, JINI is one such system [6, 8]. There are a number of protocols available from research or software vendors that could have been used for Subiquitous. However, the discovery requirements of Subiquitous are straightforward and the features included with third party protocols would have added unnecessary complexity. In addition, recall there are many issues that come with relying on dynamic service discovery as the only means of discovering and establishing communication to services [6]. The limitations and inconsistencies associated with service discovery can cause user frustration and contribute to a perception of system instability [8]. As a result, service discovery was
used in a very limited and deliberate way in Subiquitous to maximize its benefit and reduce
the likelihood of it causing issues. For Subiquitous, a simple service discovery protocol was
implemented for the sole purpose of locating the SU Server. It allows the SU Server to run on
any IP address and available port within the network while still allowing the SU Clients to
connect with no user configuration. The process and operation of the SU Server’s dynamic
service discovery will be discussed in detail in Section 3.5.1.

3.1.2 Communication Service

The SU Server is the hub for all communications within the Subiquitous system. All commu-
ications between SU Applications running on connected devices go through the SU Server.
Once a message is generated and sent from an SU Client, the SU Server takes control of
forwarding the message to its intended destinations. Subiquitous shares many characteris-
tics with a number of traditional communication protocols including: events, resource-based,
and remote method invocation. Event-based communication typically allows an interested
application to subscribe to the events produced by a service. When the service produces an
event a direct connection is made to all listeners to communicate the event. Some SU Ap-
plications could be seen as producing events and other SU Applications do listen for these
events, but Subiquitous does not require the more rigid producer and subscriber model, as
the producer does not need to have any knowledge of those receiving the event.

Resource-based communications, such as tuple spaces [19], allow members to request avail-
able information from the services on the system. Sometimes this information is produced
prior to the request; alternatively, it can be produced in response to a request. In Subiqui-
tous, an SU Application could send a request to the entire system for available information
of a specific type. If an SU Application providing this information is available it will re-
spond and the requesting application will pick it up as a normal message. If a response is not
available after the request is sent the SU Application will not lock waiting for a response.

Remote method invocation (RMI) allows services to communicate using method calls. A
direct connection is made to an available service. Methods of the service can then be called
as if the code were available locally. Allowing remote method calls supports communication
between services using any object [11]. While Subiquitous allows any object to be the medium
of a communication, Subiquitous communication is not dependent on a direct connection.
Using the SU Server to relay messages as opposed to requiring a direct connection between
two clients to send messages allows the same message instance to be sent to many clients.

The communication service also has the responsibility of maintaining SU Client status.
SU Client status is contained within a LocationInfo object which includes the availability of
the client for communication and a list of the available modules. When the SU Server receives
a message containing LocationInfo as a result of module availability changing, the server im-
mediately updates its record of the SU Client that sent the update. The SU Server will then
broadcast this information to all of the connected SU Clients to update any SU Applications
that may be currently utilizing information about other SU Clients. SU Applications can also request all the information available from the SU Server as a vector containing a LocationInfo object for each connected SU Client. The other event that can trigger a broadcast of status information is the loss of connection between the SU Server and an SU Client.

### 3.1.3 Code Distribution Service

As seen in Chapter 2, previous research explored code distribution within ubiquitous environments [2, 9, 24]. Some of these systems are targeted at distributing a particular portion of application, for example the user interface. In other cases, they use third party technologies that add further restrictions, complexity, and overhead to the development process or the final application. Jini requires code, such as a user interface, to be sent from the service to the client as a serialized version of the instantiated object on the service side. While the client must also separately download any support code [15]. This technique is intended for the transfer of simple service interfaces and places a practical limitation on the complexity of the code that can be distributed.

On the other hand, Subiquitous allows any code to be distributed and run as long as it has an SU Module entry point. Code distribution requires zero-configuration on the part of the end user. The SU Server automatically handles distributing the appropriate code to the SU Clients in the system, reducing the burden placed on the user when adding new service or connecting to the system with a new machine. The user can install a new application on one client and it will automatically be available on every other client with no configuration or interaction required from the user. Also, the SU Server stores all of the shared code available on the system. As a result, a user can connect to a Subiquitous enabled network with a device running an SU Client, such as a laptop, and seamlessly receive all available services from the system and share all available services on the machine.

### 3.1.4 Contribution of SU Server to Solution

The SU Server contributes a great deal to both the goal of zero-configuration set-up and the goal of flexible support for application structure and distribution. Dynamic service discovery allows the SU Server to be located at any IP address on the network and running on any port while remaining discoverable to the SU Clients. This results in zero-configuration for the user when initializing the system by starting the server or when connecting clients to the running system. In addition, the SU Server handles the majority of the code distribution logic. It not only facilitates the immediate distribution of newly installed applications it also stores shared code in order to synchronize newly connected SU Clients. This eliminate any need for the end user to perform configuration in order to fully utilize the system when installing new SU Applications or connecting an unsynchronized SU Client.
The SU Server also contains the majority of the communication logic. This communication architecture goes a long way in supporting Subiquitous’ goal of providing flexible support for a variety of application-structures and distributions. The ability of the communication to act in both an event-based and resource-based manner while allowing any object to be the subject of the communication places few restrictions on how SU Applications can be connected between multiple SU Clients. The SU Server is also the backbone of Subiquitous’ code distribution mechanism and therefore plays an integral part in meeting the goal of supporting flexible application structure and distribution. Although the SU Server’s involvement in this process is invisible to the developer and therefore does not directly promote the idea of structural variety, without the server this flexibility would not be available.

3.2 SU Client

All of the SU Applications on a connected device are run through an SU Client. Along with the SU Module class, the client provides these applications access to the features of Subiquitous, such as communication and locating available services. The SU Client also organizes, configures, and runs the SU Applications for the user with no required user configuration. Previous work typically builds these features in the agents themselves as opposed to having a separate application which shares them between all agents [3, 24]. The SU Client code could be used to mirror this design; no issues would arise from running one SU Client per SU Application on a machine or an SU Application could easily be developed that contained an SU Client. However, the SU Client is meant to be shared by all the SU Applications running on a participating device.

In addition to the features above, having a shared SU Client simplifies application development and eases the operation of the ubiquitous application for the end user. In particular it decreases the amount of code that the developer is required to implement to create new modules and, therefore, promotes rapid application development. As well, it provides users with one centralized point for interacting with all the SU Applications available on a particular device. Next, I will describe the facilities of the SU Client to get a better understanding of its role in the platform and in meeting the goals of Subiquitous. Figure 3.3 shows a class diagram containing the six most important classes to the operation of an SU Client.

3.2.1 SU Application Installation

SU Applications can be added to the client in two ways, either through direct user installation or by receiving modules from the SU Server. When installing an application directly the user points the SU Client to a JAR file package containing SU Modules making up an SU Application. Each JAR file should contain a manifest describing the contents of the package. The SU Client first attempts to read this manifest. If the manifest is found the
Figure 3.3: SU Client Class Diagram.
SU Client next checks the manifest to determine whether the packages contain *client run* modules. The manifest and the types of SU Modules, i.e. *client run*, are discussed in detail in Section 3.4. If it does, the client attempts to save the JAR file to the client storage folder. The SU Client compares the package name of the new package to each of the currently saved JAR files. If there is no conflict the file is saved with a unique file name. Otherwise, if a conflict does occur, the installation process is halted and the user is notified. Currently, support for automatically updating packages using versioning has not been implemented.

If the save is successful, the client then begins to iterate over all of the names of the SU Modules that are labeled as *client run*. The name of each module is added to the client configuration file so that it will be run automatically the next time the client is re-started. The next step is to instantiate and run the module code. The key to this process is the Java ClassLoader. Class loaders are used to specify the path on which to look for the code defining java classes when they are loaded. Java allows class loaders to be created dynamically at run-time which allows the instantiation of classes whose names and locations were not specified at compile-time [14]. Subiquitous uses a special class loader called the JarFileLoader which allow JAR files to be included in the specified search path. Using the JarFileLoader the JAR files saved by the client are added to the class path and the modules contained within can be loaded and instantiated. For each *client run* SU Module, the SU Client uses this process to create an instance of the specified module. The SU Module is then connected to the client’s communication object and is executed in its own thread.

After each *client run* module is started the SU Client again reviews the manifest. If the package contains either *server run* or *client shared* SU Modules the file needs to be sent to the SU Server for proper distribution. This constitutes the other way in which SU Applications can be added to the SU Client, by receiving a shared SU Application from the SU Server. SU Clients are automatically sent new SU Applications by the server as they become available. SU Applications are sent from the SU Server as JarPackages. When receiving packages from the SU Server, the SU Client is only concerned with the *client shared* modules and will only receive packages if they contain such modules. The client first attempts to save the JAR file from the byte array using the same checking and saving procedure as described above. Once this is complete the SU Client reads the manifest iterating over all of the *client shared* module names and loading them using the same JarFileLoader process as above. After the JAR file has been saved and the correct modules are loaded the JarPackage object received from the SU Server is disposed of.

### 3.2.2 Communication Service

Once SU Modules are added to the SU Client and begin running they can immediately begin using the SU Client to send and receive messages to and from other SU Modules. The core technology behind Subiquitous communication is Java’s built-in support for object serialization. Using object serialization, a live object and its state stored only in memory can
be flattened into a sequence of bytes [7]. These bytes can then be used in a number of ways including saving them to file or broadcasting them over a network connection. An SU Module can send any object as a message as long as it implements the Serializable interface. Within the SU Client, the object in charge of all communications is the ModuleCommunicator. There is only one instance of this object per SU Client and SU Modules are given a reference to this object when they are added to the client.

To send a message, an SU Module only needs to pass the ModuleCommunicator the object to send and a destination. When messages are received from the SU Server each SU Client is responsible for directing the message to the appropriate SU Module. To do this the ModuleCommunicator first reads the module identification information of the intended recipient stored in the Entry object. It then iterates over the client modules and sends the subject of the Entry object to any SU Modules matching/listening to that module id. The object is then used by the SU Module to perform its programmed action.

The ModuleCommunicator is also in charge of sending and receiving updates on SU Client status. Any time an SU Application is added by either of these means described above, an updated list of SU Modules available on the SU Client is sent to the SU Server. This information is sent in a LocationInfo object which contains a vector of one ModuleInfo object for each module available for communication. In addition these objects contain the unique identifiers of the location or module and a user identifiable display name. When this information is sent to the SU Server it is then relayed to all other connected clients. When the SU Client receives this type of information from the SU Server it immediately notifies any running SU Applications to the availability of services with which they may communicate. For example, if an SU Application for controlling lights within a house is currently showing a list of the rooms with lights and one of these rooms becomes unavailable, the SU Application and interface will be immediately updated with this change.

### 3.2.3 User Interface

Once SU Applications have been added to the SU Client, it is important to provide some simple way for users to interact with them. For this task the SU Client provides a user interface, the ClientView. This provides a common interface for interacting with the SU Client, its installed SU Modules, and viewing client information.

The view included by default with the client is meant to follow a common desktop metaphor for visually organizing the available modules. Each SU Module should contain a number of methods for controlling the visual aspects of the SU Module if they exist. The ClientView uses these methods to determine if the SU Module contains a displayable interface, to set visibility and maximum bounds, and to monitor the state of the interface. The default ClientView uses these methods to provide a module toolbar, see Figure 3.4. To the far left of the toolbar is a start menu button. This button brings up a menu where users will find a list of all the displayable SU Modules currently available on this SU Client. Clicking one
of the SU Modules’ names will show that SU Module’s user interface and place a label on the toolbar indicating that it is available for interaction. The toolbar can also be docked to the top or bottom of the screen and when an SU Module’s interface is maximized it will be bounded to the screen space left by the toolbar.

On the far right of the toolbar is a settings button that presents a menu containing available SU Client actions and options. One menu options available is to install a new SU Application. When the user selects this option they are presented with a file chooser dialog to find and select JAR files to install on the SU Client. A user interface like the one shown in Figure 3.4 is meant to facilitate the users interaction with the Subiquitous system, providing a simple common interface to add new SU Applications, discover available applications, and use those applications.

3.2.4 Contribution of SU Client to Solution

The SU Client’s major contribution in meeting the goals of the Subiquitous platform comes from facilitating zero-configuration and user friendly set-up and use of ubiquitous applications. First the SU Client serves as a central hub for accessing all of the modules available to the user. By default this hub is presented using the ClientView in the familiar desktop metaphor of a start menu and taskbar. The hope is by providing a familiar interface technique to locate and utilize available SU Applications, the end user will find interacting with the Subiquitous system user friendly. In addition the interface provides utilities for graphically maintaining the system, including installing new SU Applications. All the end user is required to do is point the SU Client to the desired package and the SU Application will automatically be installed locally and distributed across the network accordingly. The client contributes to the goal of flexible application distribution by being able to receive SU Modules from the server and send appropriate SU Modules to the server.

The client also contributes to the goal of rapid development of ubiquitous applications. The
client works together with the SU Module class to provide the functions required of SU Applications, further lessening the load placed on the developer. Although the SU Module class provides the access point, the majority of these are merely hooks to more complex code provided by the SU client. The best example of this is module communication. The SU Module class may provide methods to receive communications but the ModuleCommunicator is provided by the SU Client and performs the actual communication protocol. The developer may not have to deal directly with the SU Client while implementing their application but without the SU Client, SU Modules and SU Applications would be all but useless.

3.3 SU Modules

The most basic component and the foundation of all SU Applications designed to run on Subiquitous is the SU Module abstract class. SU Modules follow many of the paradigms set by previous work in agent based middleware [3, 18]. Like previous work, implementing applications to utilize the features of the platform requires extending a provided class. Each SU Module in Subiquitous extends the SU Module class provided in the API. Extending this class ensures the module being developed will have all the functions required for use on this platform. Figure 3.5 shows the main classes used by an SU Module. Notice there is some overlap with the SU Client class diagram, Figure 3.3, signifying the close connection between the operation of the SU Modules and the SU Client in which they run.

3.3.1 Structure

The SU Module class contains a number of default implementations of action and helper functions. These include setting the communication object, providing SU Module information, adding listeners, and a function to determine all the SU Clients that have a specified SU Module. The SU Module class constructor takes two parameters that must be specified by the developer, the moduleID and the displayName. The moduleID is used for communication and will be discussed in detail below. The displayName is an identification or description of the SU Module that users will see when interacting with the system. In reality defining these two parameters is the only thing required to extend the SU Module class. This allows developers huge flexibility in how they want to implement their SU Modules and SU Applications. In theory, a developer could create a single run SU Module by placing all the code in the constructor and leaving every other method empty.

The SU Module class also contains a number of abstract functions left for the developer to implement with code specific to their implementation. These abstract classes are what allow SU Modules to take full advantage of the Subiquitous system. These included functions that are called when the SU Module is sent a message, when communication with the system is first established, and when the availability of SU Modules on the system changes. This could
Figure 3.5: SU Module Class Diagram.

be used to prompt an action or update the SU Modules knowledge of services available on the network. There are also abstract functions intended to be used by the SU Client to control how the SU Module is displayed. These include reporting if the SU Module has a displayable interface and controlling interface visibility as well as adding a ComponentListener and setting maximum bounds of the available interface. The SU Module class also implements the Runnable interface and the run function is left abstract to be implemented by the developer, see Figure 3.5. This ensures that all SU Modules will natively have the option of running on a separate Java thread. This is very important for ubiquitous applications as it allows multiple modules to be run in parallel and respond in real-time to user interaction and requests.

Extending the SU Module class is the only requirement to run using Subiquitous. In addition, SU Modules can import a number of additional classes provided by the Subiquitous framework and API for added functionality. Classes typically used in SU Modules include CommunicationServerProtocol to define communication types and destinations. Also, SU Modules communicate using a ModuleCommunicator object shared by all SU Modules on the same SU Client. The ModuleCommunicator provides the SU Modules with information about available system services and a mechanism for sending any serializable object to specified
locations for a specified SU Module.

### 3.3.2 Contribution of the SU Module to Solution

The SU Module architecture contributes to all of the goals of the Subiquitous system. Its most major contribution is toward facilitating rapid development of ubiquitous applications. By extending the SU Module class the developer is not only making his application compatible with Subiquitous but also gaining many default implementations of important methods. In addition, the abstract methods guide the developer to implement remaining methods that are implementation specific or provide less critical functionality. These features of the SU Module class promote rapid development of SU Applications, allowing developers to focus on the logic specific to their program or interface by reducing or eliminating their need to implement key ubiquitous functions.

### 3.4 SU Application

In Subiquitous, one or more SU Modules and any supporting code make up an SU Application. The first advantage of this composite structure is that one application can receive messages from multiple SU Modules and also provide multiple SU Module services. The other benefit is that this promotes application partitioning, allowing applications to be distributed across multiple clients following any number of different software or network architectures. For example, an SU Application could be configured with a model and controller backend running on one device and a lightweight interface view on each of the other connect clients. This contributes to the flexibility of Subiquitous for supporting a wide variety of application structures and development styles.

This level of flexibility could result in confusion and a difficult configuration process to direct modules to their proper destination. Without a solution such a system would be unreasonably complex for an end user to install and maintain. Subiquitous addresses this issue by packaging all of the supporting code for an SU Application into one or more Java Archive (JAR) files. These packages contain all of the dependencies of the developed code, making the package self-sufficient. The SU Modules making up an SU Application can be packaged together or separately. A single JAR package could contain SU Modules providing any number of different services and SU Modules needed to interact with these services. On the other hand, an SU Application could be split between JAR packages, with the SU Modules providing services in one JAR and the SU Modules providing user interfaces in another. The ability to package an SU Applications in separate JARs allows for less network traffic when distributing the application as the client shared or server run parts can be bundled and subsequently sent with only the code required for that portion of the SU Application.
Each SU Application JAR package must contain a module manifest to define how its SU Modules should be distributed. The manifest file is at the heart of the flexibility of SU Application structures and distribution. This manifest file is created by the application developer and defines the types of SU Modules contained within the JAR and consequently, how they should be distributed.

Each SU Module making up an SU Application can be defined as any combination of the following:

- **ClientRun**, run only on installation SU Client
- **ClientShared**, distribute to all SU Clients
- **ServerRun**, distribute to and run on the SU Server

Defining an SU Module as ClientRun means that it will only be run on the SU Client on which it was originally installed and it does not need to be sent to the SU Server for distribution. Defining an SU Module as ClientShared means that it will be sent to the SU Server to be distributed to each connected SU Client and it does not need to be run on the SU Client on which it was installed. Finally, defining an SU Module as ServerRun means that it will be sent to the SU Server and the SU Server will run it as if it were an SU Client. ServerRun SU Modules do not need to be distributed beyond the SU Server and do not need to be run on the installation SU Client. The SU Module qualifications can be used in combination. For example, an SU Module defined as ClientRun and ClientShared would be run on the SU Client on which it was installed and would be distributed to every other connected SU Client.

This simple set of module manifest qualifications provides a great deal of flexibility in how SU Applications will behave on the system. They can be used to distribute SU Applications across devices according to any number of software architectures, such as Model-View-Controller, or network architectures, such as client-server or peer-to-peer. In addition making simple changes to the definitions in the module manifest can completely change the behavior of SU Applications without touching the code. For example, a file sharing SU Application could have a backend file resource server and a frontend file browser interface. If the server is defined as ClientRun the file server would only run on devices on which it was manually installed and those file could be made available to any connect devices with a ClientShared frontend. If the manifest was changed to make the file server ClientShared and the frontend interface ClientRun, then the file server would be distributed to every connected SU Client and make their files available only to the SU Client on which the SU Application was manually installed.

The manifest also contains the unique application identifier much like a Uniform Resource Identifier (URI) to identify the SU Application package for distribution. Like the SU Module definitions, the identifier is defined in the module manifest by the application developer. If
an SU Application is being installed either manually or from the SU Server, its identifier is compared to the identifiers of the SU Applications currently installed on that SU Client. If the same identifier is found the SU Application is not installed and in the case of a manual installation the user is notified.

### 3.4.1 Contribution of the SU Application to Solution

The SU Application architecture facilitates two goals of the Subiquitous system, namely flexible support for application structure and distribution as well as zero-configuration set-up. The simple packaging of SU Applications in JAR files with a module manifest places a small burden on the developer in order to provide zero-configuration set-up for the end user. The low requirements to extend the SU Module class as well as the freedom to implement only desired abstract methods and weave application code anywhere into the Module structure facilitate SU Application flexibility. In addition the ability to place multiple SU Modules in one SU Application where they are needed, for example to bridge two classes or send and receive information from another SU Application, provides flexibility in how SU Application as a whole will work and can be structured and distributed between SU Clients while still running as a single application.

### 3.5 Services

Each of the services provided by the Subiquitous framework span a number of the structures described above. Most require the SU Server, SU Clients, and SU Applications to work together and each perform their part to function properly. We have seen some details of each of their separate jobs in these processes, in this section we will look at an overview of how these parts combine to provide a single service.

#### 3.5.1 Service Discovery

Dynamic service discovery is used to allow an SU Client to locate the SU Server with no user configuration. This discovery process is essential to the functionality of the system as once an SU Client discovers the SU Server a permanent direct connection is made which is used for all subsequent communications.

Figure 3.6 displays a sequence diagram of this process. When the SU Server starts, it opens a ServerSocket on any available port number. At the same time it begins listening for any service discovery requests in the form of multicast datagram packets. When an SU Client starts, it begins searching for the SU Server by sending out one of these datagram packets requesting a response from the SU Server (1 in Figure 3.6). The SU Client then begins
listening for a response to its request. When SU Server receives a request that matches its own service descriptor it responds with a multicast packet containing the descriptor with the IP address and port number of the SU Server specified (2). When the SU Client requester receives this response, it uses the IP address and port number information contained in the descriptor to make a direct connection to the SU Server (3). Once a direct connection is established using a Java Socket, the SU Client is identified using its unique identifier or it is given an identifier if it does not have one and all subsequent communications use the TCP/IP protocol (4). When a new identifier is given the SU Client stores this information in its configuration file.

### 3.5.2 Communication

Communications between SU Modules is a process which include the SU Module, SU Client, and SU Server. As mentioned earlier Subiquitous communications are flexible and allow any object to be sent as the subject of a message. Figure 3.7 displays a sequence diagram of the communications process between two SU Modules.

SU Module communications is initiated when an SU Module sends its SU Client ModuleCommunicator a message object and a message destination (1 in Figure 3.7). The destination is specified by passing any number of location identifiers and a single SU Module identifier. SU Modules can also specify that the message should be sent to any combination of any or all locations and modules allowing for anything from a direct message for a specific client service to a broadcast message. For example, a device that reads temperature could ask the ModuleCommunicator to send an update message to all locations containing a temperature listener.

Once the ModuleCommunicator is sent the object and a destination, this information is
placed inside of an Entry object. The Entry object has variables to store the destination locations and module identifier as well as a byte array to store the serializable object. Once the Entry object is created it is sent on to the SU Server in the default communications object, a Message, to be relayed to the correct locations (2).

When the SU Server receives a Message object from one of the SU Clients it first determines if it contains an Entry to be relayed or some other type of communication. If it is an Entry, the server first reads the targeted SU Clients. If the Entry is directed to any location the server picks an SU Client at random that contains the designated SU Module and sends the entire Message object to that SU Client (3). Otherwise, the server begins iterating over all the connected SU Clients either looking for a match to the specified destination locations or sending the message to each location with the specified SU Module if directed to all SU Clients. Matches are made based on the unique client identifier. If an Entry is directed to an SU Client that is not currently available it is stored temporarily on the SU Server in a queue. As SU Clients join, these stored messages are checked and if a match is found the messages are sent to the SU Client in the order they arrived. Currently the persistence of undelivered messages depends on the frequency of these messages, as the queue is of a fixed size. Storing messages is meant to alleviate issues caused by unreliable network connections.

### 3.5.3 Code Distribution

The code distribution service involves SU Applications, SU Clients, and the SU Server. Code Distribution ensures that each of the SU Clients connected to the Subiquitous system has access to all of the resources available on the network. Figure 3.8 displays a sequence diagram of code distribution caused by the installation of a new SU Application and caused by the
connection of a new SU Client.

When an SU Application is installed to an SU Client, either manually (1 in Figure 3.8) or on receipt from the SU Server, the SU Client reviews the module manifest available as part of the JAR Package. If the package contains either ServerRun or ClientShared SU Modules, as defined by the manifest, the file needs to be sent to the server for proper distribution. A JarPackage object is created to transfer the JAR file to the server. The JAR file is contained within a JarPackage object. This object also contains the name of the origin JAR file and a String representation of the manifest file for easier access. This JarPackage is sent to the server and the server then handles the process of distributing the SU Modules to each connected SU Client (2).

The SU Servers part in distributing code is simple yet integral to this function within Subiquitous. The process of code distribution for the server begins when it receives a Message containing a serialized JarPackage object. This is normally the result of a new application being installed on a connected client or a new client that provides some service connecting to the system. The JarPackage contains all the information required for storing and dis-
tributing the pieces of the SU Application required to utilize it from each of the SU Clients. This includes the name of the original JAR file, the module manifest defining the type of SU Modules it contains and its unique identifier, and a byte array representation of the contents of the JAR file containing the SU Application.

The SU Server reads the JarPackage manifest and takes the required actions. If the package contains an SU Module to be run on the SU Server the byte array is saved as a JAR file on the hard drive of the server device and, using the same process used by the SU Client described above, the appropriate SU Modules are started. Similar to the SU Client, any saved JARs containing ServerRun SU Modules will now be run anytime the SU Server starts. If the manifest states that the JarPackage contains ClientShared modules then the entire JarPackage is serialized and stored on the hard drive for later distribution if the server does not already have this package. The JarPackage is then wrapped in a Message and sent to the SU Clients that may not have this package (3). The JarPackage remains stored on the server and when a new SU Client connects (4) the SU Server sends it all the SU Application packages available (7). This synchronizes the new SU Client with the rest of the system, ensuring it has all the available SU Applications and access to system resources.

### 3.5.4 Service Comparison

Now Subiquitous’ features can be compared to those of work previously found in the field of ubiquitous software. As can be seen in Table 3.1 Subiquitous provides many of the most important services provide by a number of different ubiquitous software solutions. These services are vital to providing zero-configuration and user friendly deployment, flexible

<table>
<thead>
<tr>
<th></th>
<th>Application-centric</th>
<th>Hardware-centric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subiquitous</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X11</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OMiSCID/OSGi</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Metaglue</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Jini</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Zero-configuration</strong></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Dynamic configuration</strong></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Low Overhead</strong></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Application partitioning</strong></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Co-operation</strong></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Movable Sessions</strong></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Native Resource Management</strong></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Repeated Resource Discovery</strong></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Direct Communication</strong></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Event-based</strong></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Resource-based</strong></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
support of application structure and distribution, and support for rapid development of ubiquitous applications.

In the next chapter I will show examples of how Subiquitous’ flexibility allows developers to quickly create a wide variety of ubiquitous applications.
Chapter 4

Sample Applications

4.1 Instant Messenger

The Instant Messenger application mirrors the use of popular messaging applications, such as AOL Instant Messenger and MSN Messenger, allowing users on separate machines to send text communications to each other in real-time. The application is meant to run on every available client on the system that is capable of displaying it. The user interface looks the same on each machine on which it is run. When a user starts this application they are presented with a communication window, see Figure 4.1. On the far left of this window is

![Figure 4.1: Instant Messenger Application Interface.](image-url)
Sample Applications

Figure 4.2: Instant Messenger Application Structure. The IM Module is ClientShared and ClientRun

a list of the clients that have the Instant Messenger application and are available to receive messages. To the right of this list are two text areas and a send button stacked vertically. The top text area displays a history of the communications between users in the system. The next text box is where the user types a message to send, and clicking the button sends the typed message. Selecting one or more client names from the list on the left controls which clients will receive the message and which messages will be displayed in the history. For example, selecting “Kitchen” and “Living Room” from the list of available clients will send messages to both of those clients and display messages received from and sent to either of those client in the history text box. When a message is received it will immediately appear in the history textbox if the sending client is selected. Otherwise, if the application is closed it will open and the user is notified of a new message by the name of the sending client becoming highlighted in yellow in the selection list.

This application contains only one SU Module and a few supporting classes packaged in a JAR file. When this application is installed the SU Module, InstantMessenger, is started on the installing SU Client and Subiquitous distributes the JAR file to each of the connected SU Clients. Each SU Client runs an instance of the same SU Module. Figure 4.2 shows the structure of the Instant Messenger application. The pink boxes represent the same SU Module running on each connected SU Client.

This SU Application implements a peer-to-peer network architecture and is distributed accordingly. When the user opens the InstantMessenger window the SU Module first scans the list of available SU Clients from the ModuleCommunicator to locate those SU Clients that also have the InstantMessenger application and places them in a selection list. When a user attempts to send a typed message, the InstantMessenger reviews the selection list and adds each SU Client selected to the list of message recipients. The message is sent inside an InstantMessage object that contains an identification of the SU Client sending the message, the time the message is sent, and the text of the message itself. The SU Server then sends the message to each SU Client on the receiving list. When a message is received by an instant messenger client it is stored in an array of message histories organized by the timestamp indicating the time when sent. The SU Clients selected from the list are scanned and any messages stored in the array from the selected SU Clients are displayed in the InstantMessengerWindow.
This is a simple example of both Subiquitous code distribution and communication support. Subiquitous is able to distribute code in a way that it is mirrored on each of the installed clients. This occurs with no input from the user installing the application. Users are provided with the same application on every client on the system. All of these things reduce the burden placed on the user. Subiquitous also allows this application to send one message to the server to be distributed to any number of other clients.

This application consists of only four classes, two of which import Subiquitous code. In total this application is around two hundred lines of code with around thirty lines referring to Subiquitous code. The vast majority of the code implements the user interface. The developer does not have to implement any communications protocol, which is the main focus of this application, allowing them to focus on rapidly developing an application for the best user experience.

As an example of the flexibility provided by Subiquitous, with a negligible number of additions this Instant Messenger application could be transformed from a peer-to-peer architecture to a centralized server architecture. To do this a new SU Module would need to be created as part of this SU Application. This new SU Module could be ServerRun so that only one instance would exist per Subiquitous system. The new ServerRun SU Module would track and report available instant messenger clients, receive and redirect instant messages, and could provide client authentication. The ClientShared SU Modules would remain very similar to their current implementations other than they would send all communications and requests through the new ServerRun SU Module.

### 4.2 Picture Share

The Picture Share application is meant to facilitate the viewing of images stored on one device, such as a computer or camera, from another machine on the network. The backend of this application is meant to run continuously, providing access to the images on that device. The front end can run on any client connected to the system and provides a graphical user interface for viewing available images in both thumbnail and full size format.

When this application is started the user is presented with a window that contains a list of available clients sharing images and an area to display images of the selected client, see Figure 4.3. When a user selects a particular client by double clicking its display name in the selection list, thumbnails for each image shared by that client are shown in the display area. The user can then browse the thumbnails and select one or more pictures. At the bottom of the display area is a button that allows them to view the selected images. When pressed, the first of the selected images will appear full size in the display area. The image will resize to match the viewable size of the display area. After a period of time the next image in the user selection will be shown in the display area. This will continue in a loop, like a slideshow, until the user chooses to view the thumbnails of a client again.
This application contains two modules and a few supporting classes all packaged in a single JAR file. When this application is installed the backend immediately begins running on the installing client and Subiquitous handles distributing the JAR file containing the frontend to each of the connected clients. Figure 4.4 shows the structure of the Picture Sharer application. The green box represents the SU Module that constitutes the backend running on the SU Client on which the application is installed manually. The pink boxes represent the SU Module providing the picture viewer interface and distributed to each of the other connected SU Clients.

The backend SU Module, PictureSharer, runs continuously on its own thread and recursively searches the specified directory for valid images files. When an image file is found a thumbnail of the original image is create by scaling a BufferedImage. These thumbnails are saved in a vector and the module waits 2 minutes to look for new images. The frontend SU Module, PictureViewer, runs on each of the connected SU Clients and sends requests to the backend for data. When a user selects a backend from which to view images, first a message is sent via Subiquitous communications requesting all of the thumbnails from the PictureSharer. Information on the requesting SU Client is stored in a queue waiting till a response is available. If the backend is currently performing an image search, the search is sped up.

Once the search is complete, the vector of thumbnails is placed in a PictureShareMessage
Figure 4.4: Picture Sharer Application Structure. The Share Module is ClientRun and the View Module is ClientShared

which also contains an indicator of the message contents, i.e. a set of thumbnails, and the backend location identifier. This message is sent to any locations that have made requests and are currently waiting for the search to end. When the PictureViewer receives the thumbnails it displays them all for the user in a special JList made to display images. When the user selects a few of the images to view full size, a new request is made to the corresponding PictureSharer for the first image in the selected group. The requested image is read as a string of bytes and sent back to the requesting SU Client. The viewer immediately displays the received image and requests the next one in the selected group. If the next image is received before the specified wait period is complete then the PictureViewer waits. Otherwise the next image is displayed as soon as it arrives.

For this application Subiquitous is required to run a different SU Module on the connected SU Clients than is running on the installing SU Client. Subiquitous, places the burden of directing the operation on the developer. As far as the user is concerned they merely need to install the application on a device that should share images and immediately those images are available through the same interface on every connected client. Also, when the user moves the device to a new Subiquitous system the interface is automatically sent to each of the SU Clients on the new system. This allows users to focus on the action of viewing images while the question of how to viewed shared images is handled invisibly.

The Picture Share application is an excellent example of the flexibility afforded to an SU Application through change only to the module manifest. In its current state images are only shared from SU Clients who have had the Picture Share application manually installed. By changing one line in the module manifest we can change the image server backend from ClientRun to ClientShared. If this is done, both the backend and the frontend SU Modules would be distributed to all connected SU Clients when the Picture Share application was installed on any SU Client. At that point every SU Client would be sharing any images it had and could view images available from any SU Client on the system.

Here Subiquitous communications are used to send much larger and more complex messages between clients. Also this application uses a request and response system much like resource-based communication, where the PictureViewer asks for information and the PictureSharer responds. This displays the flexibility of the Subiquitous communication system to support
different protocols as determined by the developer. This application also takes advantage of the ability of Subiquitous to run different code on different clients as well and running module on their own thread.

Using Subiquitous, the developer of this application is able to focus on the image retrieval and thumbnail creation logic. The Picture Share application contains six classes, with two that import from the Subiquitous framework. In total there are around 450 lines of code, around twenty of which refer to Subiquitous. Again, over half the non-Subiquitous lines were dedicated to creating the graphical user interface.

### 4.3 Lights Controller

The Lights Controller application is targeted towards homes that have a light system that can be controller through a computer interface, such as X10 [12] or Z-Wave [21]. The backend of this application is meant to run on a machine that is always on and connected to the hardware required for sending signals to the lights. The frontend runs on any client machine that is connected to the system and provides a graphical user interface for changing the status of the lights. When the frontend interface is first started it displays a list indicating all of

![Figure 4.5: Lights Controller Application Interface.](image)

Figure 4.5: Lights Controller Application Interface.
the locations with and available light control backend. When the user selects a location a new window appears presenting them with some image representation of the area containing the lights and icons representing the lights available of control, see Figure 4.5. The image representing the location is set on the backend and could be a picture of a room or a blueprint of a house. The icons for available lights are in the shape of a light bulb and have a white or yellow fill indicating an ON/OFF status as well as a blue or black border indicating whether they are the selected icon. Although the icons location on the image is set by the backend, the user is still able to move the icons by dragging them around the window. Moving the icons on the frontend does not change the backend information. The user can also click on the icon to request the backend to change the light’s ON/OFF status. After the change succeeds the frontend icon will be updated to reflect the current status of the light.

This application contains two SU Modules and a number of supporting classes that provide the graphical user interface all packaged in one JAR file. When a user installs classes that provide the graphical user interface all packaged in one JAR file. When a user installs this SU Application using one of the SU Clients, the first SU Module, LightsModule, begins running on the installing SU Client or on the SU Server. Where it is run depending on the SU Application’s manifest and whether it lists this SU Module as ClientRun or ServerRun. This module provides the backend and should be installed on the SU Server or an SU Client that has hardware required for controlling lights. For the sake of the demo this light control is simulated and the LightsModule is given a graphical representation to observe the effect of changes made from the other SU Modules on the participating SU Client machines.

The Lights Controller application also contains a ClientShared SU Module. As a result, the SU Server distributes the entire JAR file to each of the connected SU Clients. Upon receipt, the clients begin running the second SU Module, LightsControlModule. This SU Module runs on each of the connected SU Clients and sends control events to the backend. Figure 4.2 shows the structure of the Instant Messenger application in its ServerRun state. The green box represents the model backend running on the SU Server. The pink boxes represent the SU Module providing the controller and view to each connected SU Client.

When the LightsControlModule is started by the user, it first reviews the system information stored in the ModuleCommunicator to locate SU Clients that have the LightsModule and places those found in the list for selection. Once a user makes a selection the LightsCon-
trolModule requests the initial state including light positions, on/off status, and the background image of the LightsModule model. When the state is returned it is displayed using a LightsControlPanel with a LightIcon representing each controllable light. The LightsControlModule listens to the LightsControlPanel for graphical events such as a LightIcon being clicked. When this event occurs it is immediately sent by the LightsControlModule as a LightsControlMessage to the proper backend using Subiquitous communications. The message contains an indicator of the event type, in this case LIGHT_POWER_CHANGE, the LightIcon on which the event occurred, and the sender of the event. When the LightsModule receives this message it attempts to alter the status of the light. If this occurs correctly the LightsModule broadcast a message to all SU Clients with the LightsControlModule on the network indicating that the light has been changed. This message is the same as the one above other than it contains the updated light model. When this broadcast message is received by the LightsControlModule on each SU Client, its display, the LightsControlPanel, is updated to reflect the change.

This application is a good demonstration of Subiquitous’ flexible support for a variety of different ubiquitous application designs, separating and distributing application as distinct parts. In particular this application is divided according to the Model-View-Controller software architecture. The code for the model is run on one SU Client while the view is distributed to all the other connected SU Clients. The SU Modules act as the controller connecting the multiple views to the model backend. This is all done without any user configuration. From the users point of view actions on one client are reflected seamlessly to all available clients. When a device with the LightsControlModule is moved to a new Subiquitous system it is automatically able to access any available controllable lights connected to the new system.

This application also demonstrates Subiquitous’ support for flexible communication protocols. In this application messages act much like events. When an action is taken on a view the controller requests a change to the model via a message. When the model is updated an event indicating this change is sent as a message to each of the model views.

Using Subiquitous the developer can focus on how to best divide the application into it MVC parts. The Lights Controller application contains eight classes, only two of which import from the Subiquitous framework. In total this SU Application contains around 500 lines of code with 30 or less referring to Subiquitous features. The majority of the remaining code is to create the function of the graphical user interface and its MVC separation.

4.4 YouTube Follower

The YouTube Follower application allows users to move from client to client while the video they are watching seamlessly follows them through the environment. This application provides the same graphical user interface to each client machine connected to the system.
When the user starts this application they are presented with a window that allows them to search for videos, create playlists, and watch videos, see Figure 4.7. To the left of the window is a textbox for entering search terms, below this is a list displaying the results of the search, and below that is a button to add select search results to a playlist. When the user double clicks on one of the videos in the playlist it will begin playing on the YouTube player to the right. The player also has simple controls including a play/pause button, a “follow me” button, a progress bar that shows and allows selection of the progress of the video being played, and a timer that shows the time of the current point in the video out of the total video time. When the player reaches the end of the current video it will begin playing the next video on the playlist.

When the user clicks the “follow me” button they are presented with a simple window containing a button that says “Move”. When this button is pressed the video will stop playing and the application will close. At this time all other clients with this application will present a small window containing a button that says “Here”. When this button is pressed its text will change to “Move” and the player window will open. The window will show the entire playlist from the original client and the video that was being watched on the original client will begin playing from the point at which the user left that client. The user can then press the “Move” button and continue to move from client to client. This application could be supplemented by a separate application that monitors user location and sends out...
Sample Applications

Figure 4.8: YouTube Player Application Structure. The Play Module is ClientRun and ClientShared

updates which could be used to start and stop videos without requiring the users to press a button to indicate their location.

This SU Application contains a number of third party libraries including SWT [20] and DJNativeSwing [4]. These libraries provide an expanded set of graphical user interfaces and a java web browser for displaying the YouTube player and utilizing the YouTube API. The web browser library also allows the java code to call and receive information from JavaScript within an HTML page as well as display HTML pages. This application can be run on its own but is wrapped in a Subiquitous module to allow for the ubiquitous functionality.

This application contains only one module, YouTubeFrame, which along with all of the supporting classes and libraries is packaged in a single JAR file. When this application is installed on an SU Client Subiquitous distributes the JAR file to each of the other clients on the system. Figure 4.8 shows the structure of the YouTube Follower application. The pink boxes represent the same SU Module running on each of the connected SU Clients.

When a user searches for videos and creates a playlist it is stored as a vector of VideoInfo. VideoInfo contains the display name, video id, and the time at which to start the video. When the user indicates that they are moving to a different SU Client the YouTubeFrame request information on the currently playing video from the YouTubePanel. The video is stopped and the information is returned in a VideoInfo object marking the progress time of the video. Information of the current video and the playlist vector are placed in a VideoMessage object and the SU Module uses Subiquitous to broadcast this message to all of the SU Clients that have the YouTube Follower application. When the other SU Clients receive this message they store its contents locally. Once the user indicates that they have arrived at another SU Client the playlist and video are loaded from the stored message and begin playing automatically from the time where the video stopped.

Subiquitous is flexible enough in its code distribution to distribute the module and all of its dependencies and run them on every client available. From the users perspective all they have to do is start watching a movie using a familiar looking application, move to another computer, and continue watching the move from where it left off on the previous machine. This application’s use of Subiquitous is not complex but the result is an exciting demonstration of how ubiquitous support can supplement a familiar user action and of the
type of ubiquitous application that can be created when the developer can focus on the application logic rather than low-level details.

This application can run as a standalone program and as such the majority of the code has nothing to do with ubiquitous functionality. A large amount of this code uses third party libraries. The YouTube Follower application contains eight classes, of which one imports from the Subiquitous framework. In total it contains around 600 lines of code, less than 10 of which are references to Subiquitous framework. The developer is able to focus on the application logic and then supplement the completed application with ubiquitous functionality.

This application would be perfect to integrate with an SU Application that monitors users’ locations within the ubiquitous environment. Currently, this application only listens for its own messages contains video information. Movement is indicated manually through the use of the “Move” and “Here” buttons. Instead of using these buttons to trigger the movement actions, an SU Module could be added to this SU Application that listens for location and movement information created by another SU Application. When the YouTube Follower application receives a movement event it could trigger the same actions that are currently being activated using the buttons.
Chapter 5

Conclusion

Ubiquitous computing should invisibly integrate itself into a user’s routine and take full advantage of the possibilities presented by physically dispersed computers. Since Mark Weiser coined the term “Ubiquitous Computing” exciting progress has been made in the development of affordable, low-power hardware and high speed wireless networks. Of the three challenges to meeting Weiser’s vision, the development of ubiquitous software has fallen behind. In order to advance ubiquitous software towards the current level of hardware and network technology, I have proposed three general requirements of an effective ubiquitous software platform. Such a platform must provide user friendly zero configuration set-up and use, flexible support of application structure and distribution, and support for rapid development of ubiquitous applications. Using these requirements as the goal, I have developed Subiquitous, a platform to support ubiquitous applications.

5.1 Approach

Despite the fact that cheap, low-power hardware and wireless, high speed networks have been broadly adopted, the current ubiquitous software utilizing these technologies is widely ignored. We believe this is because they view ubiquitous environments in a hardware-centric way. Many hardware-centric solutions focus on discovering hardware components and making them available on each connected device while the complexity of the component’s use remains. Other hardware-centric approaches have constrained themselves to being very good at supporting a single type of ubiquitous computing, for example pervasive environments. In addition to not being general solutions, they typical do not include zero-configuration set-up and focus on the use of complete systems that have been pre-configured. Hardware-centric approaches fail to meet the requirements for successful ubiquitous software platforms. They remain too complex and limited to fully support Weiser’s vision of ubicomp.

On the other hand application-centric approaches have started to see ubiquitous applications
as means to complete a user task, connecting to hardware resource if they are required. Subiquitous takes this application-centric approach. We believe that this addresses a new and unique method to reach Weiser’s vision of ubicomp, one that is more in line with users’ perspectives and actions. The entire Subiquitous system can be seen in a desktop metaphor familiar to users. In this way Subiquitous provides applications, which happen to be ubiquitous, that the user can utilize to complete a desired task just like they currently use a desktop computer. In addition, this also aligns with a consumer-centric approach to systems familiar to users. In this way a Subiquitous system can be built up and improved by adding individual devices or applications one at a time.

5.2 Evaluation

We demonstrated the success of Subiquitous in meeting its goals by developing a number of example applications. Subiquitous was successful in meeting its goal of providing zero-configuration and user friendly set-up and use. Dynamic service discovery eliminates the need for user configuration when initializing the Subiquitous system by starting the server or when connecting new clients. The SU Client provides users with a familiar interface for locating and utilizing available applications. Application packaging and automatic code distribution also contribute to the goal of zero-configuration and user friendly set-up and use. With these mechanisms the burden on the end user is reduced significantly when installing new modules and connecting unsynchronized clients to the system.

Subiquitous was successful in meeting its goal of flexible support for a variety of application structures and distributions. Application partitioning and co-operation across multiple SU Clients is supported by multiple SU Modules per SU Application, code distribution, and native Java threads. Connections between these SU Modules is provided by a robust and versatile communications mechanism. Each of the example SU Applications could undergo slight modification to alter their current operation. The manifest file alone enables a great deal of flexibility within SU Applications. It was shown that modification to the module manifest could result in a significantly different application operation, e.g. client-server or peer-to-peer, without changing any of the underlying code.

Subiquitous was successful in its goal of facilitating rapid development of ubiquitous applications. Extending the Module class in combination with the mechanisms provided by the Subiquitous client handles the majority of the logic required to develop ubiquitous features. This allows developers to focus on the logic specific to their application. It was shown that even complex, stand-alone applications could be wrapped by the Subiquitous framework to take advantage of the ubiquitous environment. In this way, other service discovery protocols, such as Jini and Bonjour, and even other ubiquitous software solutions, such as Metaglue, could be integrated with Subiquitous. In order to do this an SU Module would be created to serve as a bridge between the SU Applications running on the system and the third-party code. Through this SU Module, SU Application could request information could request and
send information to the third-party resources.

Other well known distributed application platforms, such as the X Window System, could be created within the Subiquitous framework. The X Window System uses a client-server network architecture which is fully supported by the Subiquitous framework. An SU Application mirroring the capabilities of the X Window System would contain a backend SU Module that could be ClientRun and a frontend SU Module that is ClientShared. It would then use event-based communications to relay low-level graphical events between the two distributed SU Modules.

I believe that Subiquitous is an important step in the right direction toward advancing the field of ubiquitous software It goes a long way to bring ubiquitous software to the level of current hardware and network technology. With a focus on zero-configuration and user friendly deployment, flexible support of application structure and distribution, and support for rapid development it provides the flexibility required of ubiquitous applications without losing sight of the feature important to living up to the vision of ubiquitous computing.
Chapter 6

Future Work

Throughout the development of Subiquitous and the example applications a number of issues and potential improvements were highlighted. This chapter details those items and presents possible solutions in order to motivate future work.

6.1 SU Application Installation

Although there is some flexibility, the ClientView and the corresponding SU Module methods promote a particular type of interface for both the SU Client and the SU Applications themselves. The current implementation of these methods is catered to a common interactive graphical desktop environment. As a result of this mindset ClientRun SU Modules can only be installed manually to SU Clients. In the future a greater variety of SU Client types should be supported. It is easy to imagine a situation in which users will not be able to directly interact with connected devices running SU Clients. In this case, an SU Client will be needed that receives commands and code required to trigger the installation of a ClientRun SU Module. This could be done by creating an SU Client that supports command line input and could be interacted with through SSH or telnet from another device. A possible solution to ease the development of specific SU Clients would be to create a robust SU Client API and provide it to developers. This API would provide an SU Client abstract class that implements the basic operations of the SU Client in relation to the SU Server and SU Applications. Developers could then extend this class to implement their own unique SU Clients designed for specific needs.
6.2 SU Client Interface

In addition, there is currently no support for automatically adjusting the SU Client or SU Application interfaces according to the device on which they are being displayed. Solutions to this situation have been presented in previous work [2]. In order to add this to Subiquitous information about the SU Client device capabilities would have to be made available. This information could be entered manually into the SU Client’s configuration or could be retrieved by scanning the device for its resources.

On a related note there is currently no support for distributing modules according to the resources available at a particular SU Client. For example, there is no reason to send an SU Module with a user interface to an SU Client with no display mechanism. Systems such as Metaglue have included such capabilities [3]. This issue could be solved using the same method as above or by allowing users to have some manual control over which SU Modules will be installed on particular SU Clients.

6.3 Code Distribution

Despite the success of Application packaging and automatic code distribution, we can think of a few improvements that could be made to the implementations of these systems. Currently, applications must be packaged separately according to how the developer would like them to be distributed. An improvement would be a one JAR package per application approach where the Subiquitous system did more to intelligently separate the package into its distributed parts and their dependencies. This could be done by either manually listing groups of classes that need to be sent together or by scanning the class code for its dependencies. Either way, once these dependencies are known the system could unpackage the JAR file and repackage the classes that contain the SU Modules that need to be distributed with their dependencies.

Another possible improvement to SU Application distribution is the ability to use version information to automatically update SU Applications without stopping the system. Examples of similar functionality exist in previous research [24]. Currently, matching SU Applications are based only on the identifier in the module manifest. If a match is found then the SU Application is not installed on the SU Client. By adding a version number to the module manifest the SU Client would be able to determine if this was an update to a previously installed SU Application. If the version number of the SU Application being installed is higher than that of the SU Application currently installed then the SU Modules related to that SU Application would be stopped, the updated copy installed, and the new version would be started.
6.4 Failure Handling

Another potential improvement is native support for SU Module failure handling. Currently if a module fails the rest of the system continues to work but this module will be in an indeterminate state and may or may not be available. Previous research has suggested methods for dealing with module failure [3]. To add this to Subiquitous the SU Client would need to catch any SU Module errors. When a fatal error occurs the SU Client could then record the SU Module which failed and use the normal JarFileLoader procedure to restart that particular SU Module.

SU Server and SU Client failure handling should also be improved. While testing the example SU Applications, it became apparent that issues occurred when an SU Client failed that was running a ClientRun SU Module of an application that also contained a ClientShared SU Module. In this case the SU Server would distribute the ClientShared version of that SU Application to the SU Client upon restart before the SU Client could install the ClientRun Module. Upon attempting to manually reinstall the ClientRun module the SU Client would refuse because it already had that SU Application installed from the SU Server. A simple solution to this issue is to make manual installations always override previously installed SU Applications.

6.5 System Parameters

During the development of the example SU Applications the power of the module manifest truly became apparent. One interesting idea that should be explored in the future is using the manifest file to contain SU Application parameters. These parameters could be sent to the SU Server and stored there for later modification and reference. Using these parameters one could make system wide changes to an SU Application. These changes could even be applied to SU Clients that are not currently connected and missed the parameter change event. Also, these parameters could be used to create a common look and feel between similar SU Applications. One could imagine an SU Application requesting a default system color from the SU Server parameters. Implementing this would be as simple as developing a protocol and SU Server storage scheme and then using the pre-existing communication mechanisms.

6.6 Implementation Issues

One of the most noticeable limitations remaining to be addressed in the Subiquitous system is the requirement that the content of messages between SU Modules be in the form of a byte array. The object must be sent as a byte array because the SU Server does not have
the required code in its class path to interpret the message object. When the byte array is received by an SU Application it must use an ObjectInputStream to read the byte array and then cast it as the appropriate object type. Although this is a minor inconvenience it does add complexity to the implementation of SU Applications. Further research needs to be done into possible solutions to this issue and whether the cost of those solutions is worth the minimal benefit of eliminating this issue.
Bibliography


Future Work


