SCARLET

A FRAMEWORK FOR CONTEXT AWARE COMPUTING

BY

PATRICK WAGSTROM

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Adviser

Chicago, Illinois
July 2003

ACKNOWLEDGMENT

I would like to thank my adviser Dr. Xian-He Sun for giving me the support and freedom I needed for my graduate studies. Portions of the initial implementation were done by Andrei Makhanov and the Java interface was created by Tyler Butler. This project would not be the success it is without the help and feedback of Brent Lagesse, Nehal Mehta, Naga Kunderu, and Vijay Gurbani. Furthermore, I would like to thank my parents and Kristina for their support and their continual reassurances that I would be able to finish this.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

Abbreviation	Term
API	Application Programming Interface
DNS	Domain Name Service
FPGA	Field Programmable Gate Array
HTTP	Hyper Text Transfer Protocol
IP	Internet Protocol
PDA	Personal Digital Assistant
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
WSDL	Web Services Description Language

ABSTRACT

Most computer programs are controlled strictly based on program parameters and user input. Context awareness is the ability for a computer program to obtain information and alter behavior based on sources other than user input. Information that may be used in a context aware system is the actual physical location of the device, the people in the room, physical data obtained from sensors, or information about the current computer. While there have been a variety of attempts to create context aware systems most leave the developer using proprietary tools or mandate a particular programming paradigm. Scarlet is a new system for developing context aware applications that harnesses the power of web services technologies to allow communication and flexibility among the devices in a context aware system. It has shown itself to be scalable, cross-platform, modular, and extensible through the development of several sample applications.

CHAPTER I

INTRODUCTION

The world is becoming more interconnected. Devices such as mobile phones that were designed to transfer only voice information are gaining the ability to transfer images, video, and data. At the same time personal digital assistants are gaining mobile phone capabilities and seeing an explosive growth in computational power. It is now possible to pay a few hundred dollars and get a handheld computer that is more powerful than a desktop computer from just a few years ago.

When Mark Weiser first described what we now call pervasive computing in 1991 he noted that "the most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it[36][1]". We've seen this happen with a variety of technologies over the past decade. Things that we take for granted now, such as cellular phones, personal digital assistants, and even the Internet were once restricted to the technological elite, far from the public eye. Now they're all part of our daily lives; visible in corporations, universities, and even the corner coffee shop.

These technologies have a broad impact on the way we do business and interact with others. Consider the action of organizing a group of friends for an outing to see a movie. Your friends may not be at home when you're trying to organize the outing, but may still want to see the movie with you. You can look up the times that the movie shows using the a web site that lists all movie show times for your city, call your friends on their mobile phones to let them know when and where you are seeing the movie, and finally download the map to your PDA so you don't get lost on the way there. This series of actions seems almost commonplace now.

However, there still are major issues with the integration of these technologies.

[1] numbers in brackets refer to entries in the bibliography

While they function well enough in their own regard, they don't operate well with other systems. In the previous scenario each action needed to be performed separately. The first action was using the Internet to look up movie show times. After looking up the show times, each of the friends had to be called and informed about the time and location of the movie. Finally, another service such as MapQuest[23] was used to get a map to the movie theater.

This is just one example, there are hundreds of situations we often face with technology where the lack of integration among the components makes the task difficult or even impossible. It would have been nice if in the web site could have automatically retrieved the schedules and locations of all the people interested in seeing the movie and then pick a theater and show time that work for all those involved. Without a scalable platform on which to develop communication solutions, the vision of a truly connected future may never arrive. Scarlet seeks to lay the foundation for such a communication system.

1.1 Introduction to Context Awareness

When any action is performed it has a context in which it is performed. This context provides additional information about the action or situation. For instance, if I were to run around while playing a friendly game soccer, such behavior would be considered normal, in fact I would be rewarded for that behavior if it contributed to the team effort. However, if I were to run around while taking an exam, I would be punished for such behavior. The difference is the context in which the action is performed, one is done in the context of an athletic event, while the other is done in the context of an academic assessment. In this case, context can be used to decide if an action is appropriate or not.

The previous two contexts of a soccer game and academic assessment are fairly easy to identify based on time and location. Throughout an academic semester class

may be held every Thursday evening at 6:25pm in room 113 while the intramural soccer team may have games at 2:00pm on Saturday on the intramural sports field. Although these two contexts are easy to identify based on easily observable parameters, many situations require more than just time and location to fully understand the context.

If we examine the concept of a smart classroom such as the example from the Reconfigurable Context Sensitive Middleware project we see an example of a situation that needs more than simple time and location information[37]. In this scenario a professor is using computer presentation software and an LCD projector to give a presentation to a group of students. The instructor has a handheld computer with a copy of the presentation material stored on it and would like like to distribute a copy of the slides to each student's handheld or notebook computer when he starts the presentation. The problem in this scenario is knowing when the professor actually begins the presentation. This requires analyzing and understanding the context of the professor's actions.

There are certain pieces of information that will prove helpful for the slide distribution program; these can be obtained by observing people giving presentations in a similar environment. In most cases while presenting, the presenter will stand near the projector, the lights will be dimmed to enhance the image from the projector, and the room will be quiet to allow everyone to hear the presenter. Observation allows us to see that once these three criteria are met it is a safe assumption that the presentation has started. Thus, there are three pieces of context information that inform us when to distribute the slides; the location of the professor, the level of light in the room, and the level of noise in the room.

Each of the aforementioned pieces of information helps us establish context for actions. Within Scarlet a single piece of information that helps establish context, such as "the light is on" or "it is 73 degrees outside," is called a context nugget. An

application or device that provides context nuggets is called a context provider while an application or device that requests context nuggets is called a context consumer. There may be some cases where an application utilizes context information to provide new pieces of context. In Scarlet these are called context aggregators. In the smart classroom scenario whether or not the professor is currently showing a presentation could be used as a context nugget for another program, so it may make sense to utilize a context aggregator.

Problems arise when we examine how a computer is able to process context. As a computer is simpler than a Turing machine, it is able to act only upon inputs given to the program. Thus, unless we want our slide distribution to be constantly distributing information, there must be a piece of information that acts as an input trigger to start sending the slides. Most current technology gets this information through invasive requests of the user, usually through a dialog box or other means. Such interaction causes a shift in the focus of the user, slowing them down and causing a degraded user experience.

What is required is a way to make applications aware of the surrounding context with minimal user interaction. Schilit and Theimer were the first to describe such a system and called it "context-aware"[31]. It was described it as the location, identities of nearby people, and objects and changes that occur in such information. Further work[4] refined this definition slightly to include other information such as the time of day, season, weather outside and other physical characteristics. For the purpose of this document, context is defined as information that describes a entities physical or programmatic state. Specifically, we are interested in information relevant to the execution of an application and the state of the associated user.

1.2 Review of Literature

There are a variety of systems available that provide context to applications at one level or another; an exhaustive listing and analysis of all such systems is not possible here. Instead, this section examines three representative projects that illustrate different ways of approaching the issue of context awareness. First we examine Reconfigurable Context-Sensitive Middleware (RCSM) from Arizona State University. RCSM uses an object request broker similar to those found in CORBA to handle context information. Next we examine One.world a Java based approach from the University of Washington that utilizes special features of the Java programming language to provide a complete framework for pervasive computing. The last system examined is Context Toolkit from Georgia Institute of Technology. Context Toolkit seeks to provide an interface for the exchange of context information.

1.2.1 Reconfigurable Context-Sensitive Middleware. Reconfigurable Context-Sensitive Middleware (RCSM) from Arizona State University is a system to provide context awareness and analysis in an environment of handheld and desktop computers with ad hoc organization[37]. In the context of RCSM, ad hoc refers to the links between various applications on different computers that need to be dynamically established and deleted based on changing contexts and network connectivity.

The bulk of the work in RCSM lies in an object request broker, called the R-ORB, that runs on each device in the system. Context is generated through a context expression and method signature that is expressed in a special context aware interface definition language (CA-IDL). This interface is then compiled to generate an adaptive object container (ADC). The ADC can then be used to generate stubs for a transport independent implementation of the context object. This implementation can be in any of the supported languages, such as C, C++, C#, or Java. This multi-tier abstraction helps ease the development of new sources of context information.

Communication with the context sources and the management of context is done internal to the running instance of R-ORB. Each piece of context is assigned a variable in the CA-IDL script for the ORB and the interactions among the context variables can be represented via boolean equations. The ORB then monitors the values of the variables and when a boolean expression is satisfied a method is invoked on a remote object, such as a method to distribute slides of a lecture to the group of students present.

There are many cases where a single device may not have all the requisite information necessary to active a context-sensitive method. In those cases, multiple R-ORBs can interact via the RCSM general inter-ORB protocol (R-GIOP). R-GIOP is an ad hoc networking protocol designed for situations where communication may not be guaranteed, such as in a wireless network. This allows an R-ORB to obtain context information from a remote R-ORB.

There are several points where RCSM excels and is greatly successful. Because it has been designed from the ground up as a system for context awareness in ad hoc environments, there is no infrastructure requirement for the system. Also, the base of the system is implemented in hardware through customized FPGAs and in software through non-interpreted programming languages, such as C. This means the code has a very small footprint, a requirement for handheld devices. Most importantly, RCSM provides a method to collect context information in which the programs that read the context information from the sensors can be agnostic about the how to transmit the information as that is taken care of by the bindings. This allows sensor developers to focus on reading the information from the sensor rather than how to send the information to other hosts.

However, there are some serious drawbacks to using RCSM. By utilizing custom protocols for communication, RCSM has rendered itself inoperable with the bulk of programming languages available. Also, by having all context processed in the

R-ORB and represented by boolean equations, there are conceivable situations where RCSM will not work. Finally, in order to add a new source of context for RCSM, one is required to extend the CA-IDL compiler. This is a huge drawback which greatly limits the efficiency of the system as developers of sensors must not only be aware of how to read the data from the sensor, but also understand the structure and code of the CA-IDL compiler[35].

1.2.2 One.world. One.world[17], from the University of Washington is a Java based approach to provide a complete system for pervasive computing. Under a One.world environment, a user could have a contact list application that runs on her desktop at work. When she goes home she can access her contact list from her home video game console. On the road she may be able to access the system from her mobile phone, or she could use the computational facilities present in her location to access her contact list. No matter where she goes her information is accessible and usable as long as she has network access.

One of the most basic premises of One.world is that the current computing infrastructure is unable to address the requirements for a distributed pervasive architecture. Specifically, the current modes of interaction are not well suited for the wide variety of hardware devices in use. While we now see a huge variety of devices connected to the Internet and able to communicate over the Internet, most applications are still built for desktop computers and have the same windowed interface style that has been prevalent for the last twenty years. Secondly, network connectivity is not a guarantee. Even in highly modernized areas, such as Chicago, there are network outages that make it impossible for information to leave the network. The only way to protect against such outages is to have multiple physical connections, but this is not practical or even possible in most situations. Thirdly, as the Internet is a network of networks, there are a wide variety of administrative domains involved. This makes

it very difficult to see if a lost message was due to a policy decision on some a network or because of a genuine network failure [18].

The strength of one.world comes in the way that information is passed from one system to another. Realizing that architectures read bytes differently, one.world does not rely on byte streams for information, but rather stores information as tuples. An individual tuple can be read from, written to, or taken (a procedure where it is read and delete atomically). These tuples allow formulated data structures to be passed from one entity to another with ease.

Another interesting feature of One.world is that adaptability lies in the program level[19]. This is done because of perceived problems that are inherent in transparent adaptability. For instance, an application may need to adapt behavior when the computer it is running on switches from a 100Mbps wired network to an 11Mbps wireless network. If the adaption were done at the infrastructure level the application, which could be a movie player, would never know it had to change some parameters to best use the available bandwidth. When done at the application level, the application knows to switch to a lower quality video or audio stream.

One.world has been successfully used in the Labscape project at the University of Washington. Labscape provides a ubiquitous computing environment for cell biologists in which all devices are networked together and information is automatically transferred from one source to another, eliminating the need for lab notebooks The original implementation of Labscape was done using standard TCP sockets and another implementation was done using One.world. The TCP socket implementation was found to not be flexible enough and too slow for the needs of the users. A reimplementation in One.world made the system quite usable and functional[1].

However, the usefulness of One.world comes at a cost. One of the major drawbacks is that it imposes a specific programming model upon application developers. This is done to enhance security and ease manageability, but ends up limiting where

One.world will be useful as it may prove difficult to adapt to new programming strategies such as aspect oriented programming. In the Labscape study this was not found to be a severe issue, however more work will need to be done to verify that. Another major drawback of One.world is that it relies on Java and methods specific to Java for much of the functionality. This includes serialization of tuples and methods from one process to another. Thus, an implementation of One.world in another language would not be compatible with the current implementation of the system.

1.2.3 Context Toolkit. The Context Toolkit[8, 9] from Georgia Institute of Technology is a system that is focused solely on context awareness. It has six main services that it provides; encapsulation of sensors, access to context data through a network API, abstraction of context data through interpreters, sharing of context data through a distributed infrastructure, storing of context data and history, and basic access control for privacy protection.

The system consists of several components. Context widgets mediate access between a user and an application, as such they function very similar to widgets found in GUI applications. Context aggregators are specialized widgets that have all the capabilities of standard context widgets but also have the ability to take information from multiple widgets to form a single piece of context, this is the basis for their implementation in Scarlet. A context interpreter is an application that can utilize the low level information provided by context widgets and convert it into useful information. In the smart classroom example, a context interpreter would examine the current light and noise levels and check the position of the presenter and determine if a presentation is taking place or not. In a wireless network a context interpreter could be used to read the signal strengths from various wireless networks and provide an approximation of location from that information.

All of the communication in the context toolkit is done by sending XML[3]

messages over HTTP[13]. Each of the elements in the system; widgets, aggregators, and interpreters; are able to run independently of one another and are accessed through a simple HTTP server. This allows the components to be reused by multiple context-aware applications.

The Context Toolkit has already been used for a variety of applications, including an intelligent in-out board to manage the presence of staff and "Dummbo", an intelligent white board. These systems utilize iButtons[7] for context awareness. When a user is present at the white board they insert their iButton into the receptacle. When two iButtons are docked a context event it triggered to indicate a discussion is taking place and Dummbo begins to record the audio from the discussion and store the writings on the board. The in-out board works by having participants place their iButton in a receptacle while working at a location. This information is then accessible over a variety of interfaces.

The Context Toolkit has very few limitations, however there are few that are worth mentioning. The basic mode of communication is XML over HTTP. However, there isn't a standard encoding method specified for complex data types in the XML messages. Also, because it is straight XML and not some other protocol, implementing the system in a new programming language requires a significant amount of work because of the need to write a new parser for the XML schema. Because the system is tightly tied to HTTP, there are situations where the overhead of HTTP is significant and an alternative protocol may be a better choice. In addition the system was not designed to work in an ad hoc environment this limits the effectiveness in creating truly pervasive spaces.

1.3 Overview of Work

Scarlet is designed to be the first logical step in developing a ubiquitous and pervasive computing system. Rather than trying to address all of the aspects of

pervasive computing, it addresses only providing context information to applications. Many of the shortcomings of other systems are addressed within Scarlet such as communication via a platform and transport neutral protocol, execution in ad hoc environments, and providing a flexible and easy interface for sensor designers to add new sensors.

Chapter 2 describes the background of Scarlet, the goals of the system, and provides some brief analysis of the system. Chapter 3 describes the components of the Scarlet system and the required functions that each component must perform. Chapter 4 discusses the initial implementation of the system and provides an analysis of the performance and size of the system on several different architectures. Chapter 5 provides examples of sample applications that utilize the Scarlet framework. Chapter 6 concludes the document and provides information of how Scarlet may be worked into a future system for pervasive computing research.

CHAPTER II

AN OVERVIEW OF SCARLET

Research in distributed computing is one of the most fundamental areas of research in computer science. Simply put, it is research that examines how different computing sources can communicate and utilize resources amongst themselves. Within distributed computing there is also research on mobile computing. Mobile computing takes distributed computing a step further and allows the computing sources to move around both virtually in the network and physically.

Pervasive computing is another step beyond mobile computing. Whereas mobile computing may have focused on only a few nodes that may be moving and interacting with each other, pervasive computing seeks to weave computing into everything and have it all work together. Satyanarayanan identifies four additional research areas in pervasive computing beyond those of mobile and distributed computing[30]. Those areas are:

- Effective use of smart spaces
- Invisibility
- Localized scalability
- Masking of uneven conditioning

Smart spaces are physical locations that are augmented with computational resources to enhance the human experience. These resources may be sensors that automatically detect the presence of particular people and their location or perhaps measure a physical property of the room such as light or noise. Or the resources may be computers and displays that augment the user experience by providing more information to the users present in the room.

The Access Grid[16], developed by Argonne National Laboratory, is one example of a smart space. Cameras in the room film what is currently going on in the room and microphones record conversations in the room. This data is then sent to participants at other sites, called nodes, that are participating in the conference. Ceiling mounted projectors are able to display a large format display the video streams on a wall to create the impression of being in the same room with participants at other access grid nodes. There are some customized applications to allow for distributed presentations and data visualization. A set of computers manages the interfaces to the entire system and provides features such as highlighting the current speaker and echo cancellation.

Effective use of such spaces means that handheld, notebook, and other forms of computers automatically are able to interface with the environment. For instance, if a presentation was taking place, a user watching the presentation with a notebook computer would be able to automatically download the slides of the current presentation and view them without any interaction from the presenter.

Another of the goals is invisibility, or allowing the users to interact with the system without knowing it. Currently most human-computer interaction is invasive; when a person is using a computer they are very aware of it as we must interact with computers through methods other than those used to interact with people. In a pervasive and ubiquitous system the user should not have to change their method of interaction to interface with the computer systems in the room. This is especially important because if everything in the room has a computer chip in it, changing the method of interaction for each device can be quite time consuming and inconvenient.

The system should also be able to easily scale. For instance, the system may work well when there are only three people present in a room, but how will the system work when it is expanded for use in a shopping mall, airport, or stadium?

Finally, there is no way to guarantee the homogeneity of resources across all

spaces. At work a user may have a 100Mbps uplink to the Internet while at home he may be limited to 128Kbps. The applications in the environment must know how to handle both of these situations and adapt to them. Uneven conditioning is the result of having differing numbers and quality of resources depending on location. In most current applications, a drastic difference in the quality or availability of resources will cause an application to cease functioning. For these applications to adapt some form of user intervention is required. An example of such a situation is viewing video over the Internet; when the user requests the video file they must also select what speed of Internet connection they have. If for some reason the throughput from the video server degrades, because of congestion or other reasons, the video may pause and become choppy. To adapt the user must return and manually select the lower quality stream, which has the drawback that when more bandwidth is available, it won't improve the quality of the video.

Designing a system to properly address all four of these issues is a daunting task and is not the goal of Scarlet. Instead Scarlet takes a philosophy similar to that of many of the command line utilities for Unix, do only your job and try to do it very well. Then using these simple tools it is possible to build complex systems to handle complex interactions. This also greatly simplifies the design and debugging of the system.

Scarlet is designed to be a system for context awareness, similar those described in chapter 1, building on the strengths and overcoming the weaknesses of such systems. Eventually, it is hoped, that Scarlet will be just one component of a much larger, vibrant, pervasive and ubiquitous computation environment. Scarlet focuses on the following smaller issues which are highly related to the more general issues of pervasive computing research.

- Cross-platform compatibility

- Scalability
- Modularity
- Extensibility

Each of these features will be described in brief in this chapter, with additional information available with the discussion of the implementation of the system.

2.1 Cross-Platform Compatibility

For the consumer desktop market it is often assumed that the user is running some version of Windows on an Intel or AMD processor. Making such an assumption alienates some users, such as those who use a MacOS or Linux for their desktop operating system, but their total number is generally under 5% of the total computing population. In many vendors' eyes this is an acceptable loss.

However, once we begin to interact with embedded and portable devices, the story changes drastically. Common handheld computers run operating systems such as PocketPC 2002, PalmOS, and even Linux. The processor on the systems vary, coming from manufacturers such as Intel, Motorola, Texas Instruments, and Hitachi.

As a user gets more devices, the variety and number of possible combinations increases at a frightening rate. Relying on a platform specific method for communication leaves the device in its own private world, unable to ever have the possibility of communicating with other devices. This gives rise to the first major requirement of Scarlet, it must be cross-platform.

To this point, most work on delivering a cross-platform solution has focused on using the Java programming language, systems such as One.world[17] have chosen this approach. However, such a solution is not truly cross-platform as users are trapped in a world where all components must be written in Java because of the

reliance on Java specific methods such as object serialization to pass information from one host to another. In addition, standardizing on a single programming language may enforce certain programming paradigms, such object oriented in Java, upon application developers. Thus, execution speed and choice of programming paradigms are sacrificed for the perception of cross-platform compatibility.

This is why the core of Scarlet is not a single application, but rather a set of methods and interactions. By specifying the communication methods for the system, rather than the actual object model and internal structure of the system, we obtain a much more flexible and cross-platform solution.

This concept is similar to the ideas used in the Open Grid Specification Architecture (OGSA)[15], which forms the basis for Globus Toolkit 3.0[14]. Previous versions of the Globus Toolkit were a collection of mostly unrelated executables glued together by a large set of Unix shell scripts. There was no standard protocol for service discover, invocation, or utilization. OGSA and Scarlet both utilize web services technologies for communication. This includes the use of the Web Services Description Language (WSDL)[5] to describe the methods of a service and the Simple Object Access Protocol (SOAP)[2] to actually utilize the system.

SOAP is an XML encoding that allows the preservation of complex data structures while passing them form application to application. Because SOAP has a standard encoding model, it doesn't matter if the applications reside on drastically different systems. As long as the data is properly sealed in a SOAP envelope each system will be able to read and parse the information. SOAP also allows the naming of various data elements in the envelope to help preserve the structure. In this sense the information is stored in method similar to the tuples used by One.world. The exact format of a tuple is specified by the WSDL document for that particular service and operation.

WSDL is another XML language that allows for robust definitions of the meth-

ods, messages, and bindings provided by a web service. Within a WSDL document are one or more service bindings. If a service is available over multiple protocols then there will be a binding for each protocol. Each binding contains a port type that defines the methods that are available through the binding. Each method defined within the binding has a definition for its input and output messages. Each message then has a formal definition for the components that make up the message. In this hierarchy we can precisely define the input and output required by a service.

This use of SOAP for method invocation does cause Scarlet to incur additional overhead when passing binary information, especially in streaming data such as video or audio, due to the need to encode the data using base 64 encoding. However, the flexibility provided by Scarlet allows location of the resources within the framework and data transfer to take place through other out-of-bounds methods.

Furthermore, neither OGSA or Scarlet specify a particular programming language that must be used to develop for the system. Although the reference implementations for OGSA and Scarlet are written in Java and Python, respectively, one could, if so driven, write an implementation of the system in a different programming language. In a similar vein, neither of the two systems specify a programming paradigm, such as a procedural or object oriented, leaving the developer free to pursue solutions appropriate to the task.

2.2 Scalability

There are two different aspects of scalability that Scarlet focuses on; one is scalability in terms of computational power needed to run the system, the other is scalability in terms of number of systems participating in the environment.

The driving force for cross-platform compatibility is also the driving force for developing a scalable system. In a truly pervasive environment there is the need to integrate information from a huge variety of providers. Some providers may be

running on a super computer, while others are on desktop computers, or handheld devices. Still other context providers may be running on embedded systems, such as an RCX brick[26].

Designing a system for the least common denominator would result in a profound lack of features. While designing only for high end systems would result in many devices being unable to take advantage of the infrastructure. Thus, Scarlet takes a compromise between the two; it provides a moderate amount of functionality inherent in the system, but allows lower end devices to not implement all of the features of the architecture.

The other aspect of scalability, the number of devices that are using the infrastructure is heavily impacted by communication bottlenecks and single points of failure. It is for this reason that Scarlet seeks to maximize peer-to-peer communication wherever possible. Context providers that provide notification operations can future reduce bandwidth usage as Scarlet has the flexibility to allow multicast dissemination of context. However, this functionality is net yet fully implemented.

The only place that isn't true peer-to-peer communication is in service discovery. Discovery of new services within Scarlet is done in a method similar to that of the Domain Name Service system[24] with a caching name server. In Scarlet resource discovery is done via a two layer hierarchy. A node running Scarlet will have a local registry running. This local registry is able to communicate with a domain registry that serves a physical area, such as a room in a building. Under the Scarlet architecture, when a request for a resource is made, it is first sent to the local instance of Scarlet. If the local instance has enough information to satisfy the request, it will provide the information. If it does not have enough information, the query will proceed to a domain registry. The domain registry, which receives periodic updates from all of the local registries in the area, will then search the list of resources and provide a response. The local registry then caches the response from the domain registry for

future reference. From this point on communication is done directly with the context provider that was discovered. This is very similar to how the World Wide Web works. After the initial query is made, no further communication is necessary with the domain name server until the record for that host expires.

2.3 Modularity

Another issue the comes up as a corollary to the system being cross-platform and scalable is that of modularity. Modularity is the ability of the system to run with only some of the parts of it enabled, or with different versions of common parts. Scarlet's architecture is designed from the ground up to be modular and flexible.

The Scarlet runtime is broken up into a variety of components which are described in detail in chapter 3. The only component of the system that is required for every instance of Scarlet is the base, also called the Scarlet runtime module. The base contains the code needed to start up a HTTP SOAP listener and code to provide notifications to consumers if no local registry is present. Additional components are loaded at run time depending on the configuration of the system.

An example of where modularity is beneficial to the system is on a handheld device. If the device does not provide context information to remote sources, then it will not have any context providers and thus the runtime will not load the module for providing context. This saves memory and CPU cycles for other tasks on the handheld device.

Another example of a benefit of modularity is the ease with which components may be swapped in and out for different implementations of the same component. It may be the case that a system provides a context nugget to other context consumers, but it may not have the need or ability to provide the full set of services. Thus, it is possible to switch the standard context provider module for a more simplified one.

2.4 Extensibility

It would be naïve to think that Scarlet could, in its current form, satisfy all of the possible requirements for a context aware system in a pervasive computing environment. It is for this reason that Scarlet has extensibility at the very core of the system.

The module architecture was designed to allow easy replacement of components with more advanced or refined components. The communication protocols chosen are bound to a specific platform, programming language, or even transport protocol.

Much of this extensibility comes from the use of SOAP[2] and WSDL[5] for object access and object description. While currently most of these systems use HTTP[13] for their transfer protocol, it is possible to use to use other transport and invocation methods such as FTP[25], SMTP[20], BEEP[29] or another yet undeveloped protocol to transfer the requests from one host host to another. However, the restriction still exists that both ends of the communication must understand the protocol. Both specifications also support the ability to add more data data encodings at runtime through the use of XML schema imports.

Although the use of SOAP causes additional overhead over straight XML or binary serialization techniques, it allows multiple programming languages to be used. Whereas if Java binary serialization were it used, only Java programs could read the objects, similar restrictions apply for C#'s XML serialization and Python's object pickling interfaces.

CHAPTER III

SCARLET COMPONENTS

Scarlet can be thought of as having four different interacting layers that work together to provide context awareness. Like the OSI model for network communications, each layer in Scarlet communicates only with the layers above and below it and has virtual communication with the same layer on remote hosts. A visualization of this layered system can be seen in figure 3.1 with a summary description provided by table 3.1.

At the highest level is the user who interacts with the system, this is not considered to be a level exclusive to Scarlet, but rather is a necessary component for any context aware system. The user interacts with underlying application through any of a variety of operating system dependent methods, usually this is the windowed style user interface that is prevalent on Windows, Unix and MacOS. At the next level is the context aware application; this includes context providers, context consumers, and context aggregators. These communicate with each other by first sending messages through the API to the modules of Scarlet. Examples of applications at this level are temperature sensors, location sensors, and the presentation program from the smart classroom example.

Proceeding down the model, the next level is the modules. These modules provide the core functionality of the system for services such as provider and consumer

Table 3.1: Scarlet Level Descriptions

Level	Role
Users	Interact through operating system interfaces
Context Aware Apps	Utilize or provide context information
Scarlet Modules	Provide registration, subscription, and service location
Scarlet Base	Communicate with other instances of Scarlet

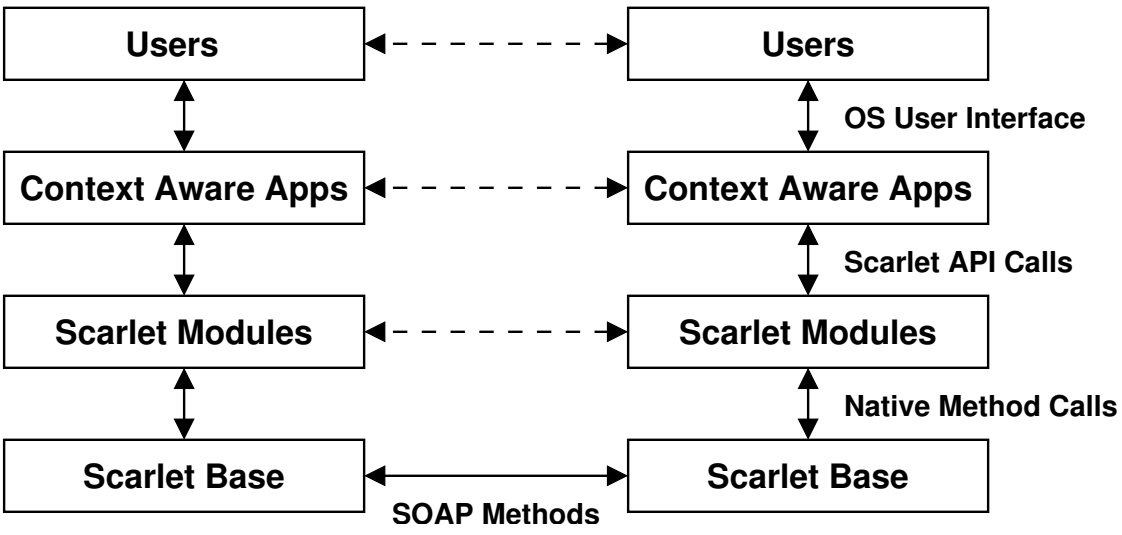

Figure 3.1: Levels in Scarlet

registration, service discovery, and security. The individual requirements and specifications for each module are detailed later in this chapter. Modules of Scarlet may interact directly with modules residing in the same instance, or may pass messages to the Scarlet base to communication with other instances of Scarlet.

At the bottom level is the Scarlet base module. This module provides the glue between all the modules and also has the responsibility for loading modules into the system. The base module also includes the SOAP server that is used to communicate with other instances of Scarlet.

3.1 Module Overview

As described in Chapter 2, Scarlet is based on a componentized architecture. The base, also called the Scarlet runtime is the core component of the system. It loads other components at runtime to configure the system appropriately. An overview of all of the components can be seen in figure 3.2.

In addition to the components that are loaded at runtime by the Scarlet base,

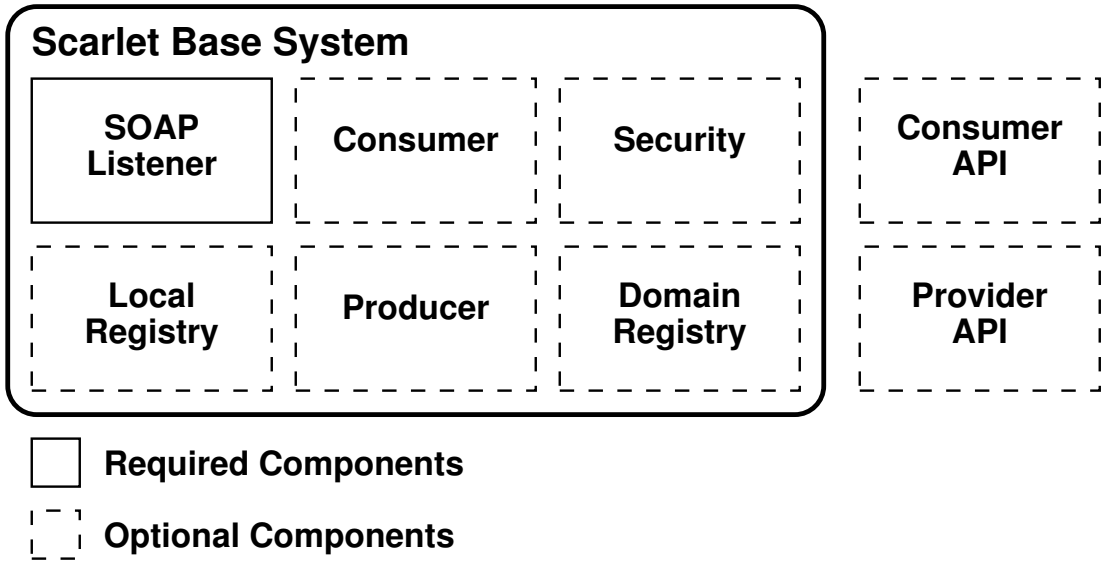

Figure 3.2: Scarlet Components

there are two other components that exist outside the bounds of the system, the consumer and provider APIs. These exist outside the core of the system to provide increased modularity and also because of the difficulties inherent in combining program modules written in different programming languages. Having the APIs outside allows the creation of context providers and consumers that are written different languages than the base of the system.

3.2 Scarlet Base

The Scarlet base is the core component that is required in all implementations of Scarlet. It has the responsibility of performing initial configuration tasks, loading the requested modules, and starting up a SOAP server for remote clients to connect to. The SOAP server may be secured via TLS, but doesn't need to be. For the default SOAP over HTTP the server listens on TCP/IP port 2707, while running the SOAP over HTTPS listens on port 2747.

The base is also responsible for forwarding all communication destined for a

module. Thus, if a module has a service that it wishes to provide to the entire Scarlet environment, it must register that method through the base module.

In addition, the base module must provide several methods to external clients. These methods allow clients to obtain information about the instance of Scarlet running at that address. Those methods are:

- `base.version` - returns a string containing an identifier for what version of Scarlet the base is running. An example of such a string is "`pyScarlet $Revision 1.14$ $Date: 2003/06/14 15:43:43 $`". This is intended to be an informational method, similar to HTTP server headers, and thus there is no specific formatting requirement.

- `base.modules` - returns an array of the modules that are running on the current system. On a system that has both context providers and consumers attached the return value could be "`['base', 'provider', 'registry', 'consumer']`"

Internally, the SOAP listener portion of the base must have a way to dynamically register and unregister methods with the server. This is needed for context providers as they may start up and shut down as they please. There is no restriction on the connection handling execution model within the SOAP listener, however it is recommended that it use either a multi-threaded or an asynchronous core to minimize delay.

The individual implementation of the Scarlet base is free to implement more functionality than this, however, all new methods that are accessible must have the prefix `base` to indicate that they reside in the base module of the system. It also is important to note that the base system should load some other modules as it can't do anything useful without them.

3.3 Local Registry

The local registry has the responsibility for responding to search requests from local context consumers, storing information about local providers, and communication with the domain registries.

As discussed briefly in section 2.2, from a context consumer's point of view, the system is designed to act similarly to a caching name server. A context consumer can make requests to the registry. If the registry does not have the information to fulfill the request it will forward the request on to the domain registry. When a response is received from the domain registry, it will be cached for a finite period of time. In this method, if a particular context provider is frequently requested, it will not result in frequent requests to the domain registry.

Figure 3.3 helps to illustrate how such requests are made. The messages are numbered 1 to 4 in the order that they are made by the various components. Solid lines are requests while dashed lines are responses. In addition, the diagram has been simplified some by leaving out additional components such as the SOAP listener, APIs, and consumer module. Initially the context consumer makes the request for a service to the local registry. The local registry, which in this example does not have knowledge of such a service, proceeds to query the domain registry via the message labeled as 2. The domain registry is able to locate the service and passes back the information about the service to the local registry. Finally the local registry passes back the information to the client via the message marked as 4.

However, in another method similar to that of DNS, the local registry only caches entries for remote services for a finite amount of time, usually five minutes. This is because a pervasive network is usually far more dynamic than the networks that DNS was designed to first serve. In our previous example, after the local registry received information about the service, all future requests, such as a request for the WSDL of the service or a query about the methods the service provides, will be

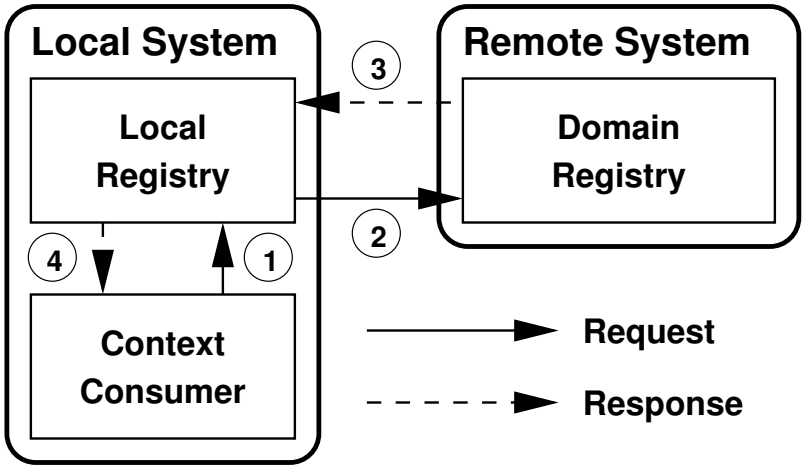

Figure 3.3: Multi-level Registry Query

answered by the local registry via the cached information.

In addition, the local registry must support updating the domain registry with information about the providers that are registered with the same base instance of Scarlet. When a new provider registers with the instance of Scarlet, it is also entered into the local registry. This allows Scarlet to run without a domain registry as requests may be satisfied locally. Periodically, the local registry will upload the information about the local providers to the list of domain registries. Through this method other hosts are able to locate providers without the need for a global network view.

The local registry also plays a large role in giving Scarlet the ability to dynamically configure itself with only minimal user interaction. This is done in conjunction with the domain registry. An instance of the local registry also opens up a UDP socket that listens for announcements from a domain registry. The local registry then automatically adds domain registries it has heard from to the list of domain registries and will update them on the next cycle. More about how this works to dynamically configure Scarlet can be seen in section 3.4.

The local registry also has methods that it must implement. These methods

are designed to provide a stable base for all implementations of the Scarlet infrastructure. In addition to the standard arguments passed, each of these commands has an additional option on whether or not to forward all requests to the domain registry. The default is to forward to the domain registry only when no responses are available from the local registry. Other options available are never forward and always forward to the domain registry.

- `reg.listServices` - returns a multi-dimensional array that lists all services present in the registry. Each entry in the array is a tuple containing the hostname and port where the service can be reached at, name of the service, and a globally unique key that is used to uniquely identify that instance of the service.

- `reg.findServiceByName` - this method works similar to listServices, with the exception that it takes a single argument, the name of the service. If there is a service with the given name in the registry, the information will be returned in the same tuple format as that of `reg.listServices`.

- `reg.findServiceByKey` - each service is accessible by both its name and a key. The key can be used to ensure that communication is always with the same instance of a service. This method takes a key and returns information about the service that has that key in the standard tuple format. More information about the keys and how they work can be found in section 3.5.

- `reg.findServiceByMethod` - a single service may have multiple methods. This method allows the consumer to specify the name of a method that they wish to call, such as `getTemperature`, and will return all services that support that method. Optionally the consumer can specify an additional method signature information such as the input and output parameters to ensure that the proper method is accessed.

- `reg.listServiceMethods` - in some cases it may be helpful to list all the methods that a particular service provides. Given a service name or key this method returns an array with all of the methods of the provider. In the case of multiple providers with same name running on different hosts, only the first set of results is returned.

- `reg.getWSDL` - after locating a particular service, it may be necessary to do some additional checking to ensure that it is the service that is desired. This method returns the WSDL for a given service, thus allowing the client to ensure that it is the desired service. The WSDL can also be used by advanced consumers to automatically generate for a set of context providers at runtime. An example of an advanced consumer can be seen in section 5.1.

- `reg.listDomainRegistries` - provides a listing of all domain registries that the local registry knows about. This is intended primarily as a debugging function, but may be useful for some consumers, such as service browsers. Unlike the other methods listed, this method never gets propagated to the domain registry.

3.4 Domain Registry

The domain registry has the responsibility for collecting information about the context providers that are registered at each local registry. This is a multi-step process that is similar to the methodology that a hosts use to obtain an IP address through DHCP[11, 12].

The domain registry must periodically make announcements to remote systems on a specified port, in this case 2708. Each announcement consists of a single UDP/IP packet that is sent to the broadcast address. Because of this, all instances of listeners to port 2708 on each network the domain registry is located on will receive notification of the domain registry's existence. An announcement is a small string that contains

three parts separated by colons: the normal TCP port that the registry is listening on, the TLS secured port the registry is listening on and the name of the registry. If the domain registry is not running on a TLS port then the port will be listed as -1.

The domain registry does not need to be present on all systems in the environment. In most cases a particular environment can function just fine with a single domain registry. Multiple domain registries are helpful in cases where fail over support is required. In the case that a local registry has multiple domain registries in its cache it should forward requests it is unable to fulfill to the domain registry that has most recently sent an announcement. If no results are returned, it can try forwarding the request on to another domain registry.

Methods provided by the domain registry have the prefix "`domain`" and are generally the same as those provided by the local registry. The exception being that the method `listDomainRegistries` has been replaces by `listLocalRegistries` which returns all the local registries.

3.5 Provider Module

The provider module is only needed on systems that have context providers and, as the name suggests, it has the support functions that are needed by context providers. Currently there are two main functions that the provider module must support; registration and unregistration of context providers and provision for notification methods.

When a context provider wishes to register itself, it sends a SOAP request to the Scarlet base that has information about the service. The request contains information that can be used to uniquely identify a provider: the WSDL file that describes that service, the address where the SOAP listener for the service is running, and a private key that is chosen by the provider.

The WSDL file is used by Scarlet to get information about the context provider's

name and the methods that are provided. In the future it may be used to provide information about alternative transports or the actual SOAP listener location, however right now this is not the case. This decision was done because it allows for greater dynamic configuration of the system. It makes it easier to create a simple provider and install it another system with zero configuration.

The address should be passed in as a tuple that gives the hostname and the port that the SOAP listener for the provider is active on. In most cases this should be running on the localhost or over a Unix domain socket if the system supports it, this provides some degree of integrity to the services as is prevents direct access from remote hosts.

The private key is a string that is chosen by the application that is shared with the provider module. It is important to note that this is not used for encryption. Rather, the string is used to validate the request that is made to unregister the provider with the Scarlet base.

When the module receives a request to register a provider, it must first ensure that there is currently no other provider that is registered with the same name. If there is a provider with the same name, the module then tries appending numbers to the name until an unclaimed name is found. So if there are three providers registered that are all called "`TemperatureService`", they will be registered as "`TemperatureService`", "`TemperatureService0`", and "`TemperatureService1`".

After determining a name for the new provider, the module then generates a reference key that is used to identify the service (this should not be confused with the private key used for registration above). The reference key should be generated using some method that will not easily generate the same value twice, using an MD5 hash of the current time or a series of bytes from the system entropy pool both work quite well. In this way, a consumer using a reference key can be sure of communication with same provider instance.

Finally, the provider module must register the new provider with the local registry and the local SOAP listener. The methods are registered with the SOAP listener in the format `name.method`. So if `TemperatureService` had a method called `getTemp`, it would be made accessible through the base as `TemperatureService.getTemp`. In addition each method is also registered under a special reference key method. This is obtained by concatenating the string "key_" and the actual reference key. Thus if the reference key for the service was "11aba5b58ad19dab9b64bcc20fd0eba7", method name `key_11aba5b58ad19dab9b64bcc20fd0eba7.getTemp` would also access the same method.

After the context provider has registered itself with the provider module, it must then register the functions with the local SOAP listener that was passed to the provider module. In the case where an operation called `getTemp` was present in the WSDL, an operation called `getTemp` must be registered with the local SOAP listener.

When a provider wishes to unregister itself a similar process is done in reverse. First the provider module must check to see if the proper private key was passed, if it wasn't, it will not unregister the module. After the private key has been verified, the provider will be removed from the local registry. After removing it from the registry all the functions associated with the provider are removed from the SOAP listener. At this point it is no longer possible to call the function in question. This also helps by not allowing requests for methods that no longer exist.

In addition to the standard request-response method of obtaining context information from a remote service operation, the provider module also has support for notification operations, which are mirrored off the WSDL methods for notification.

When a provider registers with the provider module, the module parses the WSDL file and picks out which operations are request-response and which operations are notification. Those operations which are request response have two operations bound to the SOAP listener instead of just one, a subscribe and unsubscribe operation.

For example, if a service called `TemperatureService` had a notification operation called `notifyTemp` that provides a notification stream when the temperature outside changes and a request-response method called `getTemp` that returned the temperature at that moment, the following operations would be registered with the SOAP listener:

- `TemperatureService.notifyTemp.subscribe`
- `TemperatureService.notifyTemp.unsubscribe`
- `TemperatureService.getTemp`

Unlike the request-response operations, no local methods need to be bound to the provider's individual SOAP listener as all data is sent to the remote clients via the `prov.publish` operation on the provider module. Thus, it is conceivable to create a context provider that does not utilize an instance of a local SOAP listener.

The addition of notifications to the Scarlet architecture has several advantages. First of all, it allows us to create light weight context providers as a context provider that has only notification methods requires no local SOAP listener. Secondly, it allows use another method of distributing context information and ensuring that the information is synchronized. When a context provider calls `prov.publish` the information is sent to all the context consumers who are subscribed to the operation. Thirdly, it allows for future expansion as it might one day be possible to use alternative transport protocols such as multicast.

To support all of this functionality, the provider module must implement the following methods:

- `prov.register` - registers a provider with the instance of the Scarlet SOAP listener and adds an entry for the function into the local registry.
- `prov.unregister` - removes a provider from the Scarlet SOAP listener and the local registry. Usually called right before a provider shuts down.

- `prov.publish` - allows a provider to send an update to a group of consumers who are subscribed to a particular notification operation.

3.6 Consumer Module

The consumer module handles a variety of functions to assist context consumers in their operations. The main methods that it handles are methods to register and unregister services along with helper functions for the management of subscriptions to context providers.

Consumers, like providers, are required to register before they start using the system. This allows the system to allocate resources for computation before the computation begins. This information is also needed for tracking what remote services are currently in use and how they are being used.

After registration the consumer module serves as a gateway for finding services. When a service is requested, the consumer module locates the service and returns a handle to the service. While initial designs of the system had registry requests coming directly from the context consumers, this method allows more adaptability in the system, it also helps to eliminate superfluous service lookups.

When the client requests a particular operation from a context provider, the handle is passed to the consumer module along with the name of the operation to execute and any additional arguments that may be necessary for the context to be obtained. The consumer module is responsible for issuing the SOAP request to the appropriate module.

The consumer module also provides the support methods for subscription and unsubscription of notification methods. This is primarily done to provide support for multiple consumers on the same instance of Scarlet without having to replicate the data multiple times. The number of consumers subscribed is kept via a reference counting system, when there are no more consumers subscribed to a notification

operation, the local Scarlet base will send an unsubscription notice to the notification source to stop sending updates.

This also allows for easy caching of values from a notification source. This means that when there are multiple context consumers that are subscribed to the same notification operation, no communication is necessary to the service that has the operation to add additional context consumers.

- `cons.register` - registers a consumer with the local instance of the Scarlet.
- `cons.unregister` - detaches a consumer from the local instance of Scarlet.
- `cons.subscribe` - subscribes this consumer to a notification method from a given provider. The specified provider will then send all future notification context nuggets for that method to the consumer until an unsubscribe message is sent.
- `cons.unsubscribe` - unsubscribes a consumer from a notification method of a provider.
- `cons.getValue` - returns the value of a notification stream that the consumer is currently subscribed to. This is done as a temporary measure until a more appropriate method can be discovered.

Future expansion of this module will assist in resource adaptability and provide helper functions for resource discovery. This will be done by creating specialized registry query functions for consumers. These functions instead of returning a handle to the remote Scarlet instance, will return an identifier, much like a file identifier, that is used for future context requests. This will allow the modules to transparently change over to a different context provider if the first one fails.

3.7 Security Module

Scarlet has a mechanism for security built into the system as it was easier to build the system from the ground up with security concerns than to add them later. Most of the security in Scarlet comes from one of two different mechanisms, access control lists or transport layer security. The access control lists can be defined in the system configuration. Using these lists it is possible to limit access to particular methods or groups of methods within the system on an allow/deny basis. This prevents unauthorized clients from accessing a context provider. Currently this is done on an IP address basis.

The second form of security comes from the option to use transport layer security[10] (also known as TLS or secure sockets layer). This is the same technology that is used to secure e-commerce transactions. TLS is not always a viable option as it requires the use of large numbers, typically handled by floating point units on CPUs, however most handheld devices do not have floating point units and must resort to much slower floating point unit emulation.

The security module is not accessible from outside of the Scarlet base, so it registers no methods with the Scarlet base provider, instead all configuration must be done through other methods, such as a configuration file. Because of this, the actual functions of the security module are dependent on the implementation that is being used.

3.8 Consumer API

Both the consumer and provider APIs are not listed as part of the Scarlet base because they actually reside outside of the base module. These APIs reside on systems where there are context providers or consumers and are intended to provide an easy interface for developers to interface with the system by masking the intricacies

of calling the appropriate functions and processing SOAP queries.

In general the consumer API must support functionality to allow an application to locate services, invoke services, and subscribe to updates from various services. However, how this is done is best left up to the individual API, as over-mandating it would make it difficult to implement for a variety of programming paradigms. More information about the prototype API can be found in section 4.2.

3.9 Provider API

The provider API serves a similar purpose to that of the consumer API in that it resides outside of the Scarlet base and shields the developer from a lot of the intricacies of the system. However, the provider API needs to be slightly more complex because a context provider need to have a simple SOAP server running if if supports request-response methods. If provides only notification methods, then no SOAP server is necessary.

Once again, the actual structure of the API will vary greatly depending on the language that the API is designed for. More information about the prototype provider API implementation can be found in section 4.3.

CHAPTER IV
IMPLEMENTATION DETAILS

Along with the specification of Scarlet, I have created an initial implementation of Scarlet. This implementation has been shown to run on a wide variety of systems, from the Sharp Zaurus PDA up to enterprise class servers from Sun Microsystems. It is built on the following tool sets:

- Python 2.2
- PyXML 0.8.2
- SOAPpy 0.10.1

Python[34] is a cross-platform object oriented programming language that was first developed by Guido von Rossum in 1991. While it does not yet enjoy the following that languages such as Perl and Java do, it has several advantages that make it a good choose for the development of Scarlet. Some of the advantages over Java were described in a 2001 memo by Julian Taylor, an employee of Sun Microsystems and a member of the team that develops the Java programming language[32]. In the analysis Taylor points out that for a simple program, such as one that prints "Hello World", an implementation in Python requires less than 1/5th the memory of it's Java counterpart.

Another great advantage of Python is that it is truly an open source language. One can go and download the source code for the language and the interpreter and recompile it for new platforms. While this is also true for languages like Ruby, Perl, and TCL, it is not true for the standard implementations Java and C#. This openness has made Python available on most platforms where a suitable C development chain exists. Ranging from highly interconnected supercomputers, such as the Earth Simulator, down to handheld devices such as the Apple Newton and the Sharp Zaurus.

PyXML[27] is an implementation of various methods for reading and processing XML[3] formatted files developed by the Python XML special interest group. Unlike the standard XML parsers for Java and Perl, PyXML is written in C and loaded into the Python interpreter at run time. This greatly increases the speed of reading and manipulating XML documents while simultaneously decreasing memory requirements.

SOAPpy[33] is an implementation of both SOAP server and SOAP client routines for Python. It is a pure Python package that is built on to of PyXML. As of version 0.10.1 it is close to being fully compliant with the SOAP specification and has shown sufficient ability to interact with other SOAP implementations.

4.1 Runtime Execution Model

The system begins by starting up an instance of the base module in the Python interpreter. This module reads a configuration file that resides in the users home directory. The configuration file tells the base what modules are supposed to be loaded, the TCP/IP ports that services are running on, and how to log information messages. Whenever a value is requested from the configuration file, a default value is also provided. This allows the system to begin running under a default configuration if there is no configuration file present.

After reading the configuration file, the base instantiates up the SOAP listener thread. This is, itself, a threaded SOAP server, allowing multiple clients to connect and make requests at the same time. The reference to the SOAP listener thread is preserved in the base module to allow other modules to register the appropriate services. The SOAP listener is the primary interface to Scarlet. By default it listens for connections on all available network interfaces, but can be configured to only listen on specified interfaces.

Next the system iterates over the list of modules that were specified in the

system configuration. For each module that is requested it attempts to load the module. Because modules may use other modules, there is a hierarchy for the order in which modules should be loaded. Modules are loaded in the following order: local registry, provider, consumer, security, domain registry.

A diagram to help illustrate the system start up can be seen in figure 4.1. The column on the left is the main thread, tasks that are in the column on the right can be done in parallel. The key concept is that all methods are registered with the external SOAP listener before the SOAP listener actually begins listening for requests. This helps to ensure that methods are not available for external scarlet instance before the appropriate objects are instantiated.

Each of the modules runs in a separate thread inside of the Python interpreter. The system utilizes the global interpreter lock in Python and set of thread locks to ensure that there are no race conditions with memory elements in the system. When each thread is initialized it is registered back with the Scarlet base instance and then registers the appropriate methods with the base SOAP listener instance. After all threads have been initialized they are all started at the same time.

For the most part the threads sit idle, and thus consume very little resources. Periodically each thread awakens to execute various functions, such as registry updates, and check the system execution state. If the system has entered a shutdown state, then each thread unregisters all of the methods from the base SOAP listener and terminates all other activities.

The SOAP listener thread is an exception to the above statement. This thread is constantly listening for new connections. When a connection is received a new thread is spawned to handle the connection, this is made easy through the use of the `ThreadedSOAPServer` module of SOAPpy. After the thread has been spawned, the SOAP envelope will be opened and the appropriate method will be found in the dictionary of registered methods. If there is no method registered, then a fault will

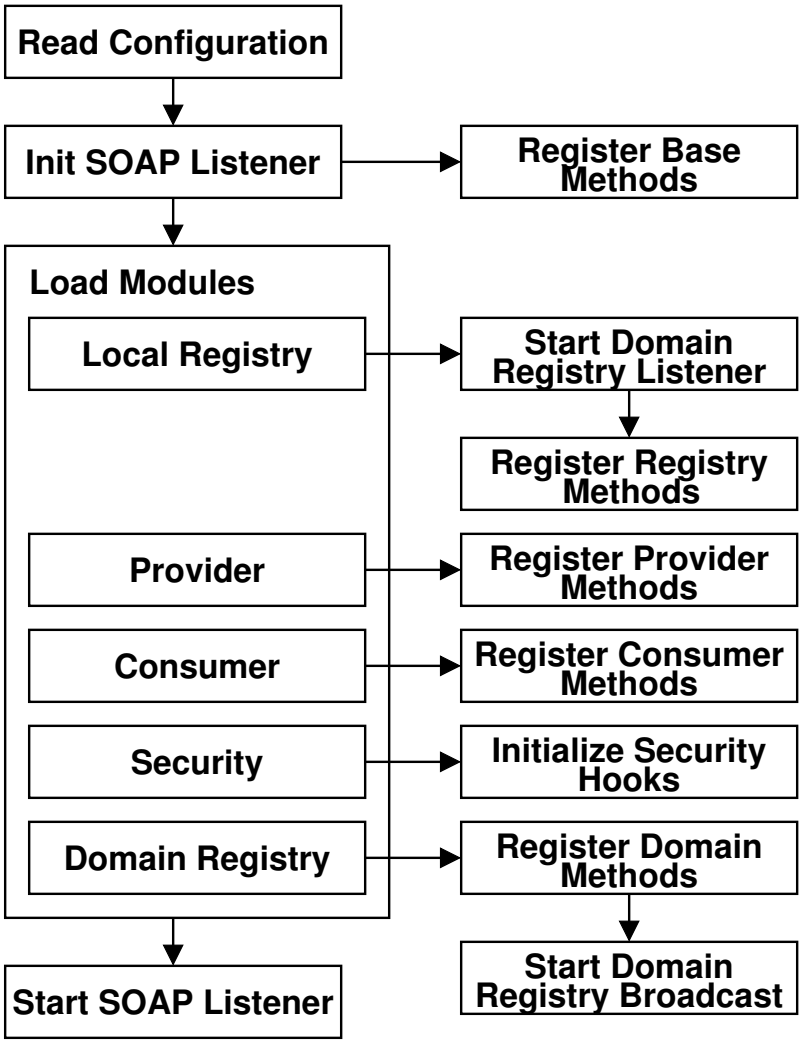

Figure 4.1: Scarlet Start Up Procedure

be returned to the requesting client. If the method is present, the `apply` function will be used to invoke the method and pass the appropriate keyword arguments.

Because of the way that Python is structured and the fact that everything is an object within Python, methods of context providers can be activated as a regular method even though they reside in a different memory space and a SOAP request must be made to accomplish the task. If we examine how a process actually registers in the Python implementation we see how this is possible.

4.2 Provider API Implementation

The provider API is the more complex of the two APIs as it needs to hide some of the issues with creating a SOAP server and locating an appropriate port for the server to run on. It is an object oriented API for Python and starts by having the application create an instance of the `ProviderAPI` object.

The creation of the `ProviderAPI` object also starts up a simple SOAP server. The server listens only on the loopback interface and selects a port number from the ports that are free, starting with port 40000.

The next step is that the context provider must register itself with the Scarlet base instance. This is done by passing in the filename where the WSDL file can be found to the `register` function of the API. The provider API does no checking to make sure the WSDL file is correct at this time, instead it reads the file and passes it as a string to the provider module of the Scarlet base for registration of the functions.

After this point, the context provider registers the appropriate functions that it supports by calling `registerFunction` and passing in a reference to the function and an optional function name. If the function name is not in the list of supported functions from the WSDL, then the registration cannot take place. This is done to help maintain naming consistency. It's important to note that only the operations that are of the request-response type need to be registered via this method as notification

methods are handled via the provider module within the Scarlet base.

The final thing that needs to be done is to start up the SOAP listener instance. This will allow the Scarlet base to forward requests through to the context provider and obtain context information. There are two different options for execution in this case. The first option is to use a listener that runs a separate thread by calling the `serve_thread` command. This allows the provider to continue and do any other functions that are needed without worrying about how to handle the listener. The second is to call the `serve_forever` command. This can be done when the provider has no background execution that is necessary for proper execution.

From this point on the provider is free to execute as it normally would as all the interfaces have been initialized. When a remote client requests an operation that the provider has, the function that was registered via the `registerFunction` system call will be executed.

If the provider has any notification methods that are supported by it, publishing to the methods is a simple matter of calling the `publish` method of the `ProviderAPI` object and passing in the name of the operation to publish to and the new value. The provider module within the Scarlet base will automatically take care of the rest including managing subscription and forwarding the information on to the subscribed consumers.

Before a program terminates, the final thing that it must do is call `unregister`. This tells the Scarlet base instance that the provider is no longer executing and that it should unregister all of the methods. The code for a very simple provider that serves as a hit counter is shown in figure 4.2. The WSDL is not shown, but it would include a single operation called `count`.

```
01: from scarlet.api.provider import *
02: ctr = 0
03: def count():
04:     global ctr
05:     ctr = ctr + 1
06:     return ctr
07:
08: if __name__ == "__main__":
09:     provObj = ProviderAPI()
10:
11:     provObj.register('simpleCounter.wsdl')
12:     provObj.registerFunction(count)
13:     try:
14:         provObj.serve_forever()
15:     except KeyboardInterrupt:
16:         pass
17:     provObj.unregister()
```

Figure 4.2: Simple Counter Provider

4.3 Consumer API Implementation

The consumer API is simpler than the provider API because of the fact that consumers lack the need to run a SOAP provider. Like the provider API, the core of the consumer API is the `ConsumerAPI` object which represents the connection to the Scarlet base instance. After initializing the object the consumer calls the `register` function to register itself with the base. After this the program may execute however it wishes. It can subscribe to a notification operation by using the `subscribe` method, or it can locate services by using the `findServiceByName` and `findServiceByMethod` operations. Updates to notification methods are obtained by calling `getValue` and passing in the service name and operation name that the consumer is subscribed to. Upon finishing the consumer should call `unregister` to ensure that all resources are cleaned up.

An example of a simple context consumer is shown in figure 4.3. This context consumer connects to the local Scarlet instance and prints out the current value from the simple counter service that was shown in the last section.

```
01: from scarlet.api.consumer import *
02: consObj = ConsumerAPI()
03: consObj.register()
04: service = consObj.findServiceByMethod('count')
05. print "Counter Value is: %d" % ( service.count())
06. consObj.unregister()
```

Figure 4.3: Simple Counter Consumer

Table 4.1: Scarlet Memory Consumption on Sharp Zaurus SL-5500

Components	Memory
Base	2816KB
Base, Registry, Consumer	2912KB
Base, Registry, Consumer, Provider	2964KB
Base, Registry, Consumer, Provider, Domain	3204KB

4.4 Performance Details

Performance testing was conducted on several systems to ensure that the initial implementation met the original goals for the system. In each case the system was running a variant of Linux and the memory consumption for the processes were measured by examining `/proc/X/status` where X was process ID of Scarlet.

The Sharp Zaurus SL-5500 was used to test performance on handheld systems. The operating system on the devices was changed from the factory original to OpenZaurus 3.2[21] to provide more flexibility for development and more options for memory control. The devices feature a 206MHz Intel StrongARM processor, 16 megs of flash ram and 64 megs of general purpose ram. The general purpose ram was split, providing 24 megabytes for additional storage, and 40 megabytes for program execution. The memory consumption of various Scarlet configurations can be seen in figure 4.1.

For desktop class computers, an AMD Athlon running RedHat Linux 7.3 and Linux kernel 2.4.20 was used. The machine has a 700MHz processor and 768MB of

Table 4.2: Scarlet Memory Consumption on AMD Athlon Linux

Components	Memory
Base	5112KB
Base, Registry, Consumer	5176KB
Base, Registry, Consumer, Provider	5240KB
Base, Registry, Consumer, Provider, Domain	5600KB

ram for program execution. The memory consumption for the different configurations can be seen in figure 4.2.

There are several interesting things to note from these figures. The most obvious thing is that memory consumption is significantly smaller on the handheld than on the desktop system. Although the Python executable is larger on the handheld, the memory footprint ends up being smaller because it has fewer shared libraries that must be loaded at run time.

Another thing of note is that adding more components takes much less memory than originally thought. This is largely because of the fact that new modules run in threads rather than in processes. The use of threads with a common parent allows each module to share the code for the Python interpreter, this is highlighted by the relative small increase in code size for each additional module.

Even though I was rather pleased with the overall memory consumption of the base system, on the handheld it does show that memory consumption may soon become an issue. The main reason is because each context provider runs in its own address space, and thus has its own instance of the Python interpreter. While this would be a necessity on a system where the base is written in a different language, it shows that there is merit to embarking on work that allows the base to load the context providers directly into the same Python interpreter.

However, this does not solve the problem of the memory consumption with context consumers, as each consumer must also have an instance of the Python in-

terpreter. Because these are separate applications that also have other functionality, it makes sense to leave these outside of the main Scarlet system. Thus, if there was a trade-off decision to be made about whether to develop an enhanced client or provider API, the preference should probably go to the client.

4.5 Other Implementations

After the initial Python implementation, work has begun to allow other languages to interact with Scarlet through the consumer and provider APIs. At this time there are Scarlet API libraries for Perl, Java, and C++. Preliminary work has started on a C# API using the Mono runtime of the Common Language Infrastructure. In addition to the API libraries, there has been preliminary work on a C language implementation of the Scarlet base. This implementation is targeted at smaller systems such as handheld computers.

CHAPTER V

SCARLET APPLICATIONS

The easiest way to demonstrate some of the properties of the system is to show a set of sample applications that utilize the Scarlet framework. A wide variety of applications have been developed for the system, this chapter will describe three such applications, the service browser, wireless signal strength monitor, and a television assistant. Scarce hardware resources have limited the ability to create a wider variety of context aware services. Section 5.4 details how some of the examples from other context aware systems could be implemented using Scarlet.

5.1 Graphical Service Browser

This tool is not meant to be the primary method of interacting with Scarlet, but rather is designed to be a development and administration tool. There are two versions of this program, one was written in Python and utilizes the QT widget toolkit for support on Unix, MacOS X and handheld devices such as the Sharp Zaurus. The other is a more advanced tool written in Python and uses the wxWindows widget toolkit for support under Windows, Unix, and MacOS X.

A screenshot of the QT based browser running on the Sharp Zaurus can be seen in figure 5.1. The user interface is intentionally minimal so as better function on the 240x320 display on most current handheld devices. When the application starts it attempts to contact a local instance of Scarlet to locate domain registries that have sent announcements to the device. The user then selects a registry and is presented with a list of all context providers the registry currently has knowledge of. Selecting a provider results in obtaining a list of operations for the service, and selecting an operation will allow the user to obtain context information by querying the particular operation from the provider.

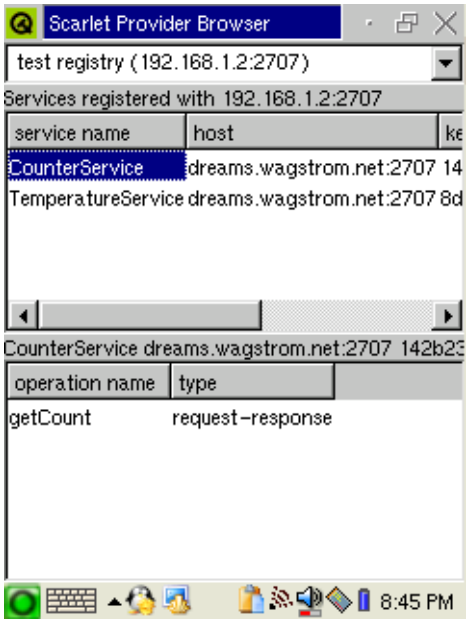

Figure 5.1: Handheld Graphical Service Browser

The wxWindows based service browser, a joint project with Andrew Makhanov, functions in a similar manner, but it is tuned for desktop computers that provide a large display area. A screenshot of the browser can be seen in figure 5.2.

The basic functionality of the program is similar to that of the browser for handheld devices, with a few noticeable extensions such as fetching and processing of the WSDL file for the service. This allows the program to dynamically build more complex user interfaces specially suited for each context provider. This is especially helpful for providers that require additional information before they can provide context. For instance the temperature context provider requires the zip code of the location in order to return the context information. The WSDL file for the service indicates that the input message requires a string to be passed, so the browser automatically builds a user interface with a field to enter a zip code. For the return value, the different elements are automatically parsed and labeled according to the WSDL description.

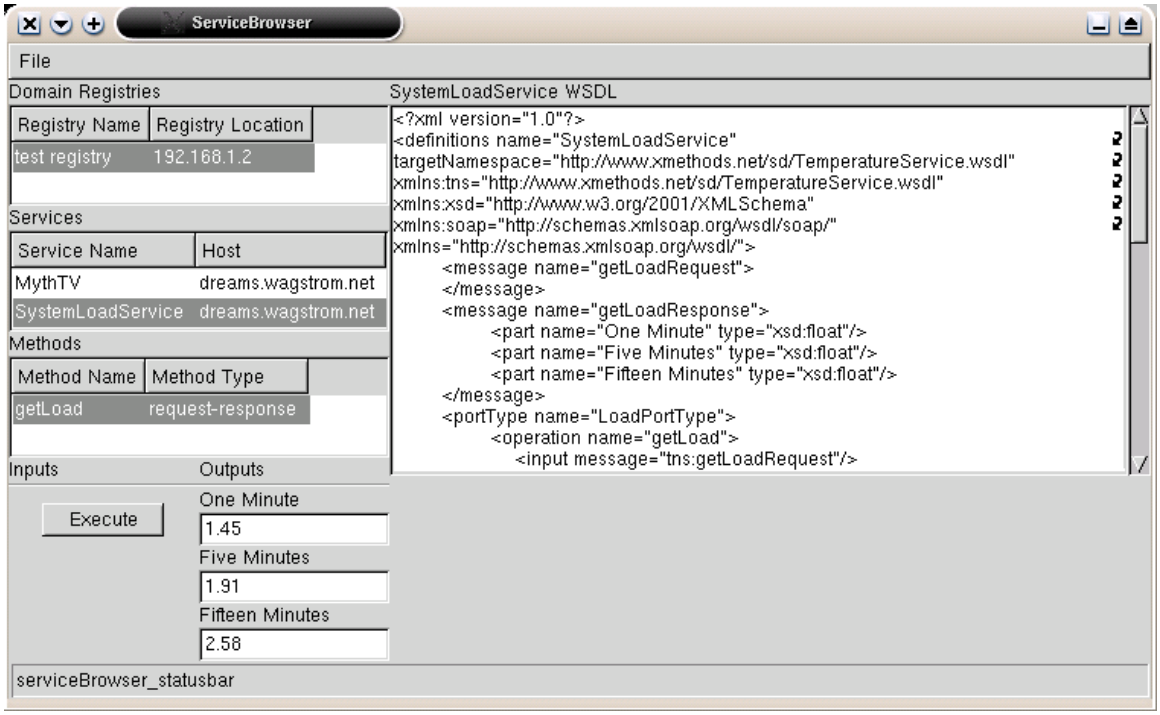

Figure 5.2: Desktop Graphical Service Browser

5.2 Wireless Strength Monitor

On traditional wired networks, such as ethernet, one can reasonably assume that slow network performance is the result of network congestion. Solutions to help avert congestion related issues usually involve an upgrade to the physical network medium, either by utilizing a higher speed network or by installing network switches instead of standard hubs.

In wireless networks the same assumption cannot be made. Most wireless computer networks operate in the unlicensed 2.4GHz or 5GHz ISM bands. As a result of the unlicensed nature of the spectrum, there are a large variety of devices that may be utilization the spectrum while not participating in the network, thus causing interference. Thus, we see how in a wireless network, slow network performance may not be caused by network congestion, but may instead be the result of interference

from a high end cordless telephone or a microwave oven. Distance and location also play more of a role than on wired networks as the physical terrain may act to block the wireless signals and degrade performance.

One method that can be used to help diagnose poor network performance is to monitor the status provided by the wireless network interface card. Within Linux this can be done by examining the contents of `/proc/net/wireless`. This file is not a normal file as it resides in the Linux `proc` file system. It's contents are continually updated and it doesn't take up room on the disk. Instead, it acts as standard kernel interface to some of the internal functions of the wireless network cards.

This was one of the first context providers developed for Scarlet and the code for the provider ends up being remarkably compact and can be seen in figure 5.3. Within this program, a large amount of the code is dedicated to parsing the data read in from the file while all the code needed to interface with Scarlet is contained in lines 22 to 31.

Lines 1 and 2 are commands that inform Python what modules this program is using. The system library, `sys`, is needed to have the provider exit in the case of an error, and the Scarlet provider API, `scarlet.api.provider` is needed to initialize communication. Lines 4 through 13 are the code that actually gets the wireless signal strength. In Linux the state of the current wireless connection can be found on the third line of `/proc/net/wireless`. Line 6 tells the program to always read from the beginning of the file, while lines 7-12 parse the input to return the interface name, signal quality (a function of strength and noise), signal strength, and signal noise. Line 13 then returns all the information back to the SOAP provider.

The main portion of the program begins on line 15. Initially an attempt is made to open `/proc/net/wireless` on lines 16-20. If the file does not exist, the provider prints an error message and exits. On line 22-25 is the initialization for the context provider. The WSDL file `WirelessStrengthService.wsdl` is used to provide

```
01: import sys
02: from scarlet.api.provider import *
03:
04: def getSignalStrength():
05:     global wirelessFile
06:     wirelessFile.seek(0)
07:     lines = wirelessFile.read().splitlines()
08:     useful = lines[2].split()
09:     interface = useful[0][:-1]
10:     quality = useful[2][:-1]
11:     strength = useful[3][:-1]
12:     noise = useful[4][:-1]
13:     return ( interface, quality, strength, noise )
14:
15: if __name__ == "__main__":
16:     try:
17:         wirelessFile = open("/proc/net/wireless", "r")
18:     except:
19:         print "Unable to open /proc/net/wireless"
20:         sys.exit()
21:
22:     provObj = ProviderAPI()
23:
24:     print provObj.register('WirelessStrengthService.wsdl')
25:     print provObj.registerFunction(getSignalStrength)
26:     try:
27:         print provObj.serve_forever()
28:     except KeyboardInterrupt:
29:         pass
30:     wirelessFile.close()
31:     print provObj.unregister()
```

Figure 5.3: Wireless Strength Provider Code

information about the service to the Scarlet base. The method `getSignalStrength`, defined on lines 4-13, is then registered as a remotely accessible method on line 25. Finally, lines 26-29 tell the program to run forever unless it stopped with keyboard interrupt and lines 30-31 close all remaining files and remove the system from the Scarlet base.

The WSDL file for the service can be seen in figure 5.4. It is a fully standards compliant WSDL file. Although it may seem quite complex, a quick analysis reveals that it is much simpler than first glance. The first seven lines of the program establish the appropriate name spaces for the service description, such lines are required for any WSDL file. Line 8 declares a message called `getSignalRequest` with constituent parts, similar to declaring a function as taking the `void` parameter in C. Lines 9-14 create another message called `getSignalResponse` that returns four parameters, one string and three floating point numbers. Lines 15-19 are the declaration for the operations, in this case only one operation is supported, `getSignalStrength`. The input and output messages are specified to be those that were just declared. Lines 21-36 provide the binding for the operation to SOAP, specifically SOAP over HTTP. These lines will be very similar for every service that runs SOAP over HTTP. Finally, lines 37-44 provide the definitions of the service and tie everything together.

5.3 Television Assistant

Almost anyone who owns or has used a personal video recorder, such as a TiVo, will be quick to explain how such devices have forever changed the way that television is viewed. These devices usually have features like an interactive program guide, the ability to pause live television, commercial skipping, remote scheduling of programs, and even use previous viewing habits to make recommendations of other programs that the viewer may enjoy.

With the advent of inexpensive television tuner add-on cards, the desktop com-

```
01: <?xml version="1.0"?>
02: <definitions name="WirelessStrengthService"
03:              targetNamespace="http://patrick.wagstrom.net/research/scarlet/WirelessStrengthService.wsdl"
04:              xmlns:tns="http://patrick.wagstrom.net/research/scarlet/WirelessStrengthService.wsdl"
05:              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
06:              xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
07:              xmlns="http://schemas.xmlsoap.org/wsdl/">
08:     <message name="getSignalRequest" />
09:     <message name="getSignalResponse">
10:         <part name="interface" type="xsd:string"/>
11:         <part name="quality" type="xsd:float"/>
12:         <part name="strength" type="xsd:float"/>
13:         <part name="noise" type="xsd:float"/>
14:     </message>
15:     <portType name="SignalStrengthPortType">
16:         <operation name="getSignalStrength">
17:             <input message="tns:getSignalRequest"/>
18:             <output message="tns:getSignalResponse"/>
19:         </operation>
20:     </portType>
21:     <binding name="SignalStrengthBinding" type="tns:SignalStrengthPortType">
22:         <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
23:         <operation name="getSignalStrength">
24:             <soap:operation soapAction=""/>
25:             <input>
26:                 <soap:body use="encoded"
27:                 namespace="urn:scarlet-WirelessStrengthService"
28:                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
29:             </input>
30:             <output>
31:                 <soap:body use="encoded"
32:                 namespace="urn:scarlet-WirelessStrengthService"
33:                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
34:             </output>
35:         </operation>
36:     </binding>
37:     <service name="WirelessStrengthService">
38:         <documentation>
39:             Returns the strength of the wireless signal for the host
40:         </documentation>
41:         <port name="SignalStrengthPort" binding="tns:SignalStrengthBinding">
42:             <soap:address location="scarlet://localhost:2707/"/>
43:         </port>
44:     </service>
45: </definitions>
```

Figure 5.4: Wireless Strength Provider WSDL

puter is now a useful and viable platform to replicate the functionality of such devices. MythTV[28] is an open source project that provides standard PVR functionality and more to computers running Linux.

The television assistant program is an application that runs on a handheld device, such as a Sharp Zaurus, that uses context information from MythTV to enhance the overall viewing experience. The most basic functionality is that of an enhanced remote control that works over radio frequencies rather than traditional infrared communications. This removes the requirement for line of sight to control the television. Also, this allows customized controls to be developed and utilized via the touch screen on the handheld.

The next set of functionality is that of a program guide. The application is able to request the listings of all available currently showing programs from MythTV and display program guide information without the alter the viewing oft the current program by bringing up an on screen display. This can be helpful if one person in a group wants to see what else is on television without disturbing the entire group.

The final functionality, and perhaps the most useful, is the ability to view segments of previous recorded programs on the handheld. This is done by instantiating an instance of a VideoLan[22] server on the computer running MythTV and a VideoLan client on the handheld. This mode is still fairly limited due to a variety of factors, the primary two being the processing power of the handheld and the available bandwidth in the wireless network.

Standard NTSC television is captured at a resolution 720x480 pixels at 30 frames a second. At this resolution, a full quality MPEG-2 encoded video stream will produce about 15 megabits of data a second, which is higher the the 11 Mbps theoretical max of 802.11b wireless networks. This results in the need to either save the programs at a lower quality or transcode the programs to lower quality while sending to the handheld device. Unfortunately, in most cases it is undesirable to save

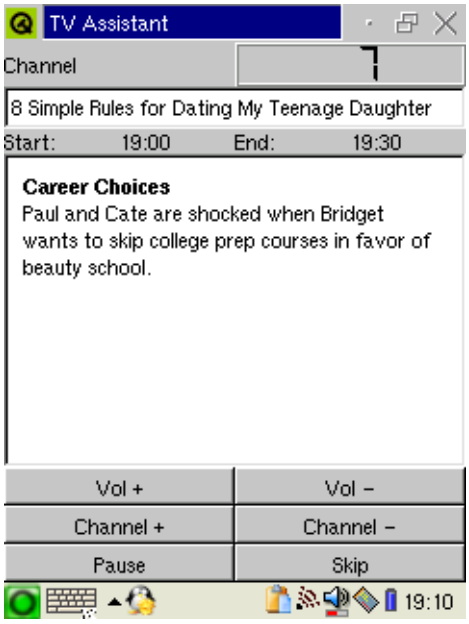

Figure 5.5: Television Assistant Application

at a lower quality and neither MythTV or VideoLan support transcoding on the fly.

Thus, while it is currently possible to watch the recorded programs on the handheld they must be manually transcoded to achieve the desired parameters. The current parameters that have been found to be most successful are 320x240 resolution and 10 frames per second. A screenshot of the application can be seen in figure 5.5.

5.4 Other System Examples

Unfortunately, the lack of hardware resources made it impossible to demonstrate conclusively that Scarlet can handle all of the same sample applications as RCSM, One.World and Context Toolkit. This section intends to describe how some of those sample applications would be implemented under Scarlet.

5.4.1 Smart Classroom. This is the example that was first presented in Section 1.2.1. The smart classroom's presentation scenario would most likely utilize

three context providers and one context aggregator. The first context provider would be on the presenter's handheld device and would be a location context provider. The second provider could be anywhere in the environment and it would monitor the state of the lights in the room, this sort of context provider has already been created through an interface with an RCX brick. The third context provider would be a noise sensor, it too could be located anywhere in the room.

The presenter would also have a context aggregator built into the presentation program. It would monitor the three context providers and would provide a new notification stream when all of the conditions are met. The students watching the presentation would then subscribe to the notification stream and receive an update about the location where the slides may be obtained from once the presentation has begun.

There is nothing in this scenario that Scarlet cannot theoretically do. There are some challenges to making this scenario a reality though. The main one is dealing with locating information, a task that most context aware projects are struggling with. Current techniques allow a rough location to be obtained by looking at what wireless access point is being used, however this scenario requires a higher resolution than such a method provides. Until a higher resolution wireless location method is available, Scarlet, and most other context aware systems, will have difficulty implementing the entirety of this service

5.4.2 Smart White Board. The Smart White Board from Context Toolkit is an application that can currently be done with little difficulty. Like the smart classroom, the smart white board relies on location information, specifically it has to know that there are multiple users interacting with it before it starts recording. The primary difference is the use of iButtons for physical location information.

In the Scarlet implementation there would be a pair of context providers that

are connected to the receptacles for the iButtons. Each of these would export a notification method that provides information about the button that is currently in the receptacle and when it is removed. A context consumer could subscribe to this information and when all both context providers indicate the presence of a user, would activate a camera and microphone through operating system native methods.

When a user leaves, by removing their iButton, the notification would be received by the context consumer which would stop recording the conversation. At this point it would save a copy of the film and make it available for the people who just participated in the discussion.

CHAPTER VI

CONCLUSION AND FUTURE WORK

Scarlet was designed to be a cross-platform, scalable, modular and extensible solution for context aware computing. This document began by first providing a brief overview of some of the current technologies available for context-aware computing and their advantages and disadvantages. It then outlined the requirements for a pervasive computing infrastructure and highlighted what aspects of such an infrastructure Scarlet fulfills. Based on the requirements for pervasive computing, the four primary design requirements for Scarlet were then described in detail.

Chapter 3 described the components of the system and showed that by using a transport and programming language independent message passing protocol, much of the cross-platform requirement can be satisfied. This chapter also described in detail the modularity of the system and described how modules could be replaced with different ones depending on requirements. The registry is a module of Scarlet that exemplifies this; it is easy to drop in replacement registry modules to provide better support for handheld devices or devices that don't need to support the full range of functionality.

Chapter 4 provided information about the initial implementation of Scarlet in Python. The memory profile showed that although Scarlet has moderate memory requirements, primarily due to the use of Python and its associated interpreter, they are note beyond the range of current generation handheld computers. Chapter 5 helped to reinforce the scalability issue by providing real examples of applications that run on both handheld and desktop computer systems.

Table 6.1 provides a comparison of One.world, Context Toolkit, RCSM and Scarlet on several key issues; communication, programming language, memory consumption and extensibility. Communication refers to the primary method of commu-

Table 6.1: Pervasive Computing Comparison

	Communication	Language	Memory Consumption	Extensibility
One.world	Java Serialization	Java	High	Moderate
Context Toolkit	XML over HTTP	Any	Moderate	Moderate
RCSM	Proprietary	Any	Low	Low
Scarlet	SOAP over HTTP	Any	Moderate	High

nication used between components in the system. One.world uses Java serialization to communicate between components of the system, Context Toolkit uses XML fragments over HTTP, RCSM uses a proprietary CORBA like protocol called R-GIOP, while Scarlet uses SOAP over HTTP currently, but is easily extensible to other protocols.

Programming language refers to what languages may be used to interface with the system. One.world requires the use of Java because it uses Java object serialization for communication. Each of the other systems operates with any programming language, but some work may be required to create the bindings for the language. Interfacing with Context Toolkit only requires processing XML and transmitting it over HTTP, the existence of XML parsers and HTTP libraries for many languages greatly simplifies this task. Scarlet is slightly more complex as it requires a SOAP interpreter in addition to XML parser. RCSM is the most complex as it requires modification of the CA-IDL compiler to provide stubs for new programming languages.

The next comparison point is memory consumption of the system. One.world has fairly high memory requirements due to the need for a Java run-time environment in the system. There are moderate memory requirements for Context Toolkit and Scarlet, primarily this is due to the need to process XML data and send it using HTTP. RCSM has the lowest resource consumption because it is coded in C and has some of the functionality off-loaded to a customized FPGA chip.

The final comparison criteria is related to the overall extensibility of the system, both in terms of the ability to add new functionality, such as support for new protocols, and the ability to add new sensors and context providers to the system. One.world is moderately easy to extend because new programs need only be programmed in Java and utilize the packages provided by the framework. However, reading actual data from sensors in One.world can be a bit tricky because of the virtualization and restrictions imposed by running a Java Virtual Machine. Context Toolkit makes it fairly easy to add a new context provider to the system, but the user must write the code to parse the XML fragments that are passed. Both of these systems require modification to both provider and consumer sides of the system for new providers to function. RCSM has the least extensibility because adding a new context provider involves modifications to the CA-IDL compiler, the creation of new bindings, and possible modifications to R-GIOP. Scarlet can be easily extended thanks to its modular architecture. By using the APIs for context providers and context consumers the creation of new context provider takes only a few lines of code and clients can be created that dynamically invoke new remote providers. There is no modification required anywhere in the architecture for a new provider to be deployed.

Based on this information we see that the only category of the four comparisons where Scarlet is not the optimal solution is memory consumption. However, this is due primarily to the fact that Scarlet is written in Python which has a high memory footprint due to its interpreted nature.

6.1 Future Work

As stated at the beginning of this thesis, Scarlet is not meant to be used in a vacuum. It is meant to be the first component of a larger system that for pervasive computing. There are still many components that need to be developed for this vision to become a reality. Some of these components are an efficient system for remote data

access and a system to provide more flexibility inherent to the system.

There also is work that can be done within Scarlet to improve the system. Some of it is simple implementation issues with cleaning up the code, while others present larger issues that are necessary to make Scarlet a robust environment, such as fault tolerance and adaptability.

As mentioned in section 4.4, there are steps that should be taken to reduce the memory usage of Scarlet. One easy step to do this is to allow the Scarlet base to load providers internal to the system, and thus avoid instantiating another instance of the Python interpreter. Another step is to address the efficiency of the consumer API and develop an optimized library for more efficient languages like C. This would allow the creation of more compact producers and consumers and would be very helpful for handheld and embedded devices.

Interoperability with more toolkits should be ensured also. Currently the Scarlet runtime follows all of the standards of the W3C as of it's creation, but that does not guarantee interoperability with other systems as there are many SOAP implementations that are not fully compatible with the SOAP standard[6].

Finally there should be wider vision beyond Scarlet, that towards a complete pervasive infrastructure. A good place to look next is toward uneven conditioning and allowing services to automatically adapt to changing conditions. Together with Scarlet such a system can make great strides toward recognizing a pervasive infrastructure.

BIBLIOGRAPHY

[1] Larry Arnstein, Robert Grimm, Chia-Yang Hung, Jong Hee Kang, Anthony LaMarca, Gary Look, Stefan B. Sigurdsson, Jing Su, and Gaetano Borriello. Systems support for ubiquitous computing: A case study of two implementations of labscape. In *Proceedings of the 2002 International Conference on Pervasive Computing*, Zurich, Switzerland, August 2002.

[2] Don Box, Davin Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol. http://www.w3.org/TR/SOAP, May 2000. visited June 12th, 2003.

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language 1.0. http://www.w3.org/TR/REC-xml, October 2000. visited June 2nd, 2003.

[4] Peter J. Brown, John D. Bovey, and Xian Chen. Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications*, 2(1):1–9, March 1997.

[5] Erik Christensen, Fancisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language. http://www.w3.org/TR/wsdl, March 2001. visited June 12th, 2003.

[6] Robert Cunnings. SOAP builders interoperability lab. White Mesa Software. http://www.whitemesa.com/interop.htm, June 2003. visited June 12th, 2003.

[7] Dallas Semiconductor. iButton - contact memory, digital temperature data loggers, java-powered and secure ecash tokens. http://www.ibutton.com/. vistied June 20th, 2003.

[8] Anind K. Dey. *Providing Archtecture Support for Building Context-Aware Applications*. PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.

[9] Anind K. Dey and Gregory D. Abowd. The context toolkit: Aiding the development of context-aware applications. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, Limerick, Ireland, June 2000.

[10] Tim Dierks and Christopher Allen. IETF RFC 2246: The TLS protocol. ftp://ftp.rfc-editor.org/in-notes/rfc2246.txt, January 1999. visited June 16th, 2003.

[11] Ralph Droms. IETF RFC 1541: Dynamic host configuration protocol. ftp://ftp.rfc-editor.org/in-notes/rfc1531.txt, October 1993. visited June 16th, 2003.

[12] Ralph Droms. IETF RFC 2131: Dynamic host configuration protocol. ftp://ftp.rfc-editor.org/in-notes/rfc2131.txt, March 1997. visited June 16th, 2003.

[13] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. IETF RFC 2616: Hypertext transfer protocol – HTTP/1.1. ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt, June 1999. visited June 12th, 2003.

[14] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. Grid services for distributed system integration. *Computer*, 35(6), June 2002.

[15] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The physiology of the grid: An open grid services architecture for distributed systems information. In *Global Grid Forum 5*, Edinburgh, Scotland, June 2002. Global Grid Forum.

[16] Futures Lab, Mathematics and Computer Science Division, Argonne National Laboratory. Access grid. http://www.accessgrid.org/, June 2003. visited June 10th, 2003.

[17] Robert Grimm. *System Support for Pervasive Applications*. PhD thesis, University of Washington, December 2002.

[18] Robert Grimm, Tom Anderson, Brian Bershad, and David Wetherall. A system architecture for pervasive computing. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 177–182, Kolding, Denmark, September 2000.

[19] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, and David Wetherall. *Programming for pervasive computing environments*. Technical Report UW-CSE-01-06-01, University of Washington, Department of Computer Science and Engineering, June 2001.

[20] John Klensin et al. IETF RFC 2821: Simple mail transfer protocol. ftp://ftp.rfc-editor.org/in-notes/rfc2821.txt, April 2001. visited June 17th, 2003.

[21] Chris Larson. OpenZaurus. http://www.openzaurus.org/, May 2003. visited June 1st, 2003.

[22] Simon Latapie. VideoLan. http://www.videolan.org/, July 2003. visited July 1st, 2003.

[23] MapQuest.Com, Inc. Mapquest. http://www.mapquest.com/, June 2003. visite June 29th, 2003.

[24] Paul Mockapetris. IETF RFC 883: Domain names - implementation and sepcification. ftp://ftp.rfc-editor.org/in-notes/rfc883.txt, November 1983. visited June 14th, 2003.

[25] Jon Postel and Joyce Reynolds. IETF RFC 959: File transfer protocol. ftp://ftp.rfc-editor.org/in-notes/rfc959.txt, October 1985. visited June 17th, 2003.

[26] Kekoa Proudfoot. RCX internals. http://graphics.stanford.edu/~kekoa/rcx/, 1999. visited June 17th, 2003.

[27] Python XML Special Interest Group. Python/XML libraries. http://pyxml.sourceforge.net/, June 2003. visited June 6th, 2003.

[28] Isaac Richards. MythTV. http://www.mythtv.org/, June 2003. visited June 29th, 2003.

[29] Marshall Rose. IETF RFC 3080: The blocks extensible exchange protocol core. ftp://ftp.rfc-editor.org/in-notes/rfc3080.txt, April 2001. visited June 12th, 2003.

[30] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, pages 10–17, August 2001.

[31] Bill Schilit and Marvin Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8(5):22–32, 1994.

[32] Julian S. Taylor. The java problem. http://internalmemos.com/memos/memodetails.php?memo_id=1321, 2001. visited on June 16th, 2003.

[33] Cayce Ullman, Brian Matthews, Gregory Warnes, and Christopher Blunck. SOAPpy. http://pywebsvcs.sourceforge.net/, June 2003. visited June 6th, 2003.

[34] Guido von Rossum. The python programming language. http://www.python.org/, May 2003. visited May 30th, 2003.

[35] Yu Wang. Situation-aware middleware for application software development in ubiquitous computing environments. Presentation to Department of Computer Science, Illinois Institute of Technology, April 2003.

[36] Mark Weiser. The computer for the 21st century. *Scientific American*, September 1991.

[37] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, and Sandeep K.S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–49, July–September 2002.