RESOURCE ADAPTIVE AGENT BASED FAULT TOLERANT ARCHITECTURE

by

SHREYAS K SHETTY

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

AUGUST 2007

ACKNOWLEDGEMENTS

I would like to extend my gratitude to my advisor, Dr. Mohan Kumar, for giving me an opportunity to work on this challenging research topic and providing me the right guidance and support through the course of this research. I am grateful to Dr. Ramez Elamsri and Dr. Jeff Lei for serving on my thesis committee.

I am very grateful to my parents and family who have been a constant source of inspiration during my entire academic career. I would like to thank Sukruth and Prathiba for their invaluable help and advice all the way.

July 20, 2007

ABSTRACT

RESOURCE ADAPTIVE AGENT

BASED FAULT TOLERANT

ARCHITECTURE

Publication No. ______

Shreyas Shetty, M.S.

The University of Texas at Arlington, 2007

Supervising Professor: Dr. Mohan Kumar

Pervasive environment consists of increasing number of mobile, tiny and heterogeneous devices communicating through interconnected network. As ubiquitous computing has seeped into various aspects of everyday life, there has been an increasing demand for dependable systems. However providing reliability demands fault tolerance mechanisms that require substantial time and resources. The dynamic nature and the uncertainty associated with pervasive systems coupled with the energy constraints of the devices involved makes fault tolerance a challenging task. In general, the techniques used to provide fault-tolerance are based on having redundancy and duplication of the user tasks. However the additional cost and the low resource availability will prohibit implementation of such fault tolerance methodologies for a pervasive environment [17]. The traditional fault detection and recovery techniques need to be modified to make it applicable in a pervasive environment [16].

Pervasive Information Community Organization (PICO) [20] is a framework consisting of software agents, called delegents that perform services on behalf of users and devices. In the PICO framework computing community of collaborating delegents is formed to carry out application-specific services. PerSON (Service Overlay Network for Pervasive Environments) [23] provides the service overlay network for the implementation of the community computing concept introduced in PICO. PerSON uses the device model proposed in PICO and provides an overlay network which abstracts the details of service creation, discovery and utilization in a pervasive environment. In this thesis we have developed a Resource Adaptive Agent System (RAAS) which is integrated with PerSON to enhance and facilitate the services provided by PerSON. To deal with the dynamic nature and make best use of resources available in a pervasive environment, RAAS adds features like fault tolerance, checkpointing and resource aware distribution of user requests to PerSON. RAAS not only provides reactive measures to failures, but also proactively deals with the probable future failures and if required performs reassignment of user task from the recently saved checkpoint.

Demonstration applications that perform data intensive tasks have been developed and tested on RAAS. For a set of tasks, energy savings of about 40% was achieved by adding the resource adaptiveness feature to PerSON. The energy savings achieved is proportional to the size of tasks and is subject to the devices available in the environment.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Over the last decade, there has been a dramatic increase in the use of computing devices and that have become intrinsically involved in our daily life. The tremendous developments in technologies such as wireless communications, networking, mobile computing, wearable computers, sensors, smart spaces, middleware, software agents, and the like has led to the evolution of pervasive computing platforms as natural successor of mobile computing systems [33] . Pervasive computing aims at creating an embedded environment of network devices to provide unobtrusive connectivity and services to users without the user's explicit awareness [34]. Users in pervasive computing environments can be mobile and have computing sessions spanned over a range of heterogeneous devices.

The dynamicity of a pervasive system increases the possibility of failure. A system is said to be performing correctly if, in response to a certain input, the system behaves in a manner consistent with a given specification [14]. The same definition will hold good for the services provided by the system. Fault tolerance is the ability of a system to perform correctly in the presence of faults [21]. The basic purpose of a fault tolerant system is to increase the availability and the reliability of the system.

1.1 Motivation

The motivation of this thesis stems from a demand for dependable systems with higher reliability properties in a pervasive environment [16]. Time critical applications from the field of health care require continuous operation of systems even in the presence of faults and failures [13]. Having intrinsic resilience to faults and failures helps in providing transparency to users in a pervasive environment. Ideally, a fault tolerant system ensures correct system operations in the presence of faults. This approach typically uses redundancy to detect the components that produce errors and to take further action. However, redundancy would mean use of extra resources which are prized commodities in a pervasive environment. So far only few fault-tolerant techniques have been explored in pervasive computing [15, 18, 32] and these approaches typically require redundancy in terms of resources and time to detect faults. Any system model for fault tolerance in a pervasive environment will need to understand and deal with the challenges posed by ubiquitous systems in terms of the dynamicity, heterogeneity and resource availability. Another approach to provide dependable systems is to build reliable systems through thorough testing and debugging [32]. However, the degree of reliability of a dependable system depends on the thoroughness of the testing done which is not only difficult for a complex system but also costly. In addition, a pervasive computing system typically contains heterogeneous nodes which make testing beforehand very difficult [15].

PerSON , the service overlay network for pervasive environments successfully manages to abstract the details of connecting and utilizing the services provided in the environment but does not provide any kind of fault tolerance. PerSON also does not provide any mechanism for resource aware adaptation to dynamic changes and for service distribution.

1.2 Contributions

The contribution of this thesis includes the development of a resource adaptive agent system (RAAS) which is integrated with PerSON to enhance its features. RAAS adopts a resource adaptive approach for task distribution and completely masks the internal reconfigurations to deal with the faults and failures of the devices, network and services. RAAS proactively deals with probable future failures and provides reactive counteraction to dynamic changes in the environment. RAAS also provides provisions for checkpointing and restarts failed service requests from the last saved checkpoint.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 gives a background of pervasive systems, covers challenges in pervasive computing, agents, classify various kinds of faults and failures and discuss fault tolerance in pervasive environment. In Chapter 3, we discuss PICO framework and the architecture of PerSON in detail. In Chapter 4 we discuss the failure and resource model adopted by RAAS and the various algorithms implemented in RAAS. In Chapter 5, we present the architecture and implementation of different aspects of the RAAS. Chapter 6 contains the discussion of the results. Finally, Chapter 7 draws conclusions and discusses future work.

CHAPTER 2

BACKGROUND

In the early 90's Mark Weiser, who coined the phrase "ubiquitous computing", envisioned an environment embedded with advanced electronics, connected and constantly available for users even when the user was not explicitly aware of the technology [1]. Transparency of the actual computing and the technologies involved in computing became the principal goal of pervasive computing [15]. In this chapter, we first introduce pervasive computing and discuss its history in brief. We then look at the challenges in pervasive systems and subsequently discuss the various faults and failures likely to occur in pervasive environments. We then provide a brief discussion about agent based systems in pervasive environment from fault tolerance point of view. Finally, we discuss the need for fault tolerant systems and related work in pervasive systems.

2.1 Pervasive Computing

Pervasive computing is the new trend in computing environments with information and application services available anytime and anywhere. As Mark Weiser prophesied pervasive computing is the idea of integrating computers seamlessly into the world such that these technologies "weave themselves into the fabric of everyday life until they are indistinguishable from their environment" [1]. Pervasive computing applications allow users to access information anytime using any kind of devices. The devices in a pervasive environment invariably have high resource constraints and many a times have to rely on short-range and low-power mobile ad hoc networks to exchange information. Current computer networks and the Internet are increasingly becoming heterogeneous and the applications commonly operate across different types of hosts with different resource capabilities using wired and wireless networks. Pervasive computing is the ability to extend applications to such heterogeneous devices.

There are five key players in a typical pervasive environment: end users, application developers, service providers, network operators, and equipment providers [2]. The relationship between them is as shown in following figure.

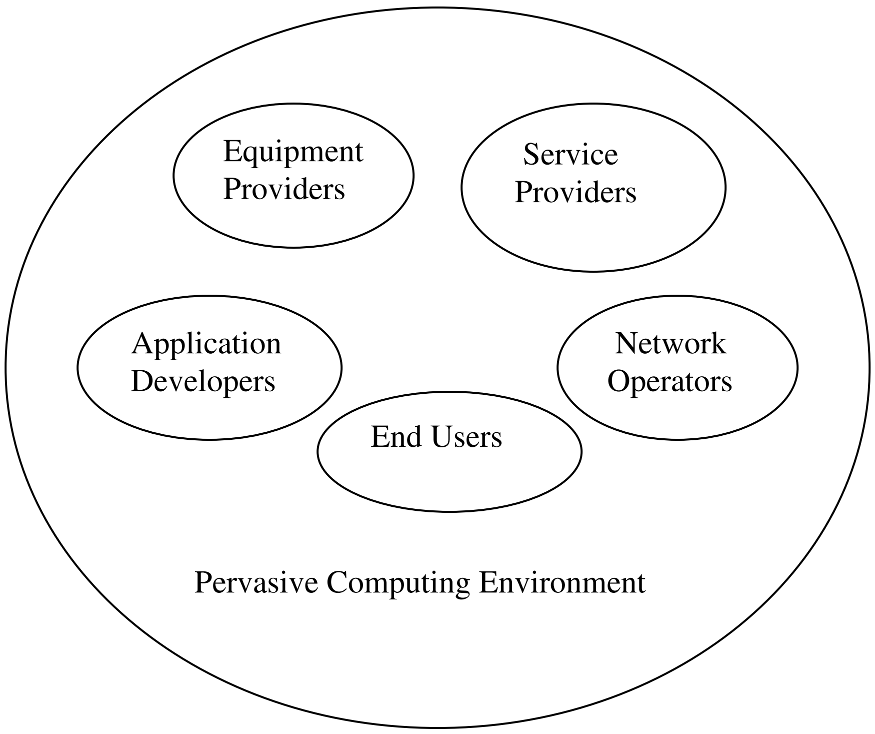

Figure 2.1 Key Players in Computing Environment

According to Liu et al. [3] pervasive computing system should have the following features:

- Pervasive: must be everywhere, with every portal reaching into the same information base.
- Embedded: live in our world, sensing and affecting the way we live.
- Nomadic or Mobile: allow users and computations to move around freely, based on their needs and requirements.
- Adaptable: provide flexibility and spontaneity, in response to changes in user requirements and operating conditions.
- Efficiently powerful: free itself from constraints imposed by bounded hardware resources, addressing system constraints instead which are imposed by user demands and available power or communication bandwidth.
- Intentional: enable people to name and use services and software objects based on intent.
- Eternal: never shutdown or reboot; must be available all the time.

2.1.1 History

Mark Weiser wrote some of the earliest papers on ubiquitous computing [1, 5, 19], defining ubiquitous computing. Wieser recognized that including computing into everyday scenarios would require understandings of social, cultural and psychological phenomena. Weiser was influenced by many fields outside computer science, including philosophy, phenomenology, anthropology, psychology, post-Modernism, sociology of science and dystopian Philip K. Dick novel Ubik [29]. The computing history has progressed in terms of four phases of interaction between human and computer [30] as described below. The first phase was that of mainframes, where one machine was shared among many human users. The second phase, known as personal computing, saw one on one interaction between computer and human user. The third phase is that of internet computing where one user accesses and utilizes the services hosted on a world-wide network. The fourth and current phase is ubiquitous or pervasive computing, in which many devices will serve many people in a personalized way on a global network [30]. Currently, the state of art is not as mature as Weiser hoped, but research in pervasive computing is very much ongoing and considerable amount of development is taking place.

Mahadev Satyanarayanan in his paper, “Pervasive Computing: Vision and Challenges” [4], identifies four new research areas brought about by pervasive computing; effective use of smart spaces, invisibility, localized scalability, and masking uneven conditioning. In the paper, Satyanarayanan points out that a smart space is brought about by embedding computing capabilities in building infrastructure which brings together two different worlds, enabling sensing and control of one world by the other. He adds that, in practice, the ideal proposed by Weiser [1] of complete disappearance of pervasive computing technology from a user’s consciousness or in other words total invisibility might not be completely possible and a reasonable approximation to it will be achieving minimal user distraction. If a pervasive computing environment continuously meets user expectations and rarely presents him with surprises and allows him to interact almost at a subconscious level would suffice the goal of achieving transparency. Achieving transparency would require smart spaces to grow in terms of complexities and also will require increased interactions between a user's personal computing space and his surroundings. This has severe bandwidth, energy and distraction implications for a wireless mobile user. He also mentions that pervasive computing environment will need development of techniques for masking uneven conditioning and heterogeneity of environments.

2.2 Challenges in Pervasive Systems

Pervasive computing characteristically allows users, machines, data, applications, and physical spaces to interact seamlessly with one another. The idea of realizing seamless interaction faces a number of challenges with the existing technologies.

2.2.1 Device and Network Heterogeneity

A pervasive computing environment may consist of wide variety of devices such as wired devices, wireless devices, handheld devices, sensors operating in different network technologies. Each of these devices will be required to interact with each other despite the differences in hardware and software specifications and capabilities of heterogeneous networks. This will require an infrastructure or middleware which will manage the integration of the devices into a coherent system and will allow arbitrary device interactions. Due to the complexity arising from the interaction of ubiquitous services in a heterogeneous environment, probability of faults and occurrence of failures increases.

2.2.2 Device Mobility

The device mobility primarily results in the need for maintenance of connections between areas of differing network connectivity and different network technologies and also puts a need for handling network disconnections.

2.2.3 Scalability

Scalability poses a challenge for any computing environment. As smart spaces grow in sophistication, the intensity of interactions between a user's personal computing space and its surroundings increases which will have severe bandwidth, energy and distraction implications for a wireless mobile user. But scalability is a blessing in disguise when we consider fault tolerant techniques. The probability of successful execution of a service increases with the increase in the number of participating devices.

2.2.4 Adaptation

Adaptation is required in order to overcome the intrinsically dynamic nature of pervasive computing. With time, mobility of users, changes in devices and revisions of software components can occur, leading to changes in the physical and virtual environments of these entities. There will be dynamic changes in the resources available and the pervasive computing infrastructure should be able to handle and exploit such dynamism.

2.3 Classification of Faults and Failures

The basic purpose of a fault tolerant system is to increase the availability and the reliability of the system. Reliability of a system is defined as the probability with which the system will perform correctly up to a given point of time [11]. Availability is defined as the probability that a system is operational at a given point of time [11]. A fault tolerant system should be resilient to various kinds of faults and should be able to function in spite of these faults. We will first classify different kind of faults and failures.

2.3.1. Types of failures errors and faults

In general, whenever a service does not behave in the manner specified, a service failure is said to have occurred [14]. The cause of this failure is called error. The error itself is the result of a defect or fault in the system or the service [21]. We can categorize faults in different ways as follows.

Faults can be classified as transient or permanent based on the duration for which the fault occurs. But in our service oriented architecture we are looking at the execution of an individual service request at that instance of time with a time bound, and hence we will deal with both these faults in a similar way. We identify them as a permanent fault for that service request and deal with that fault.

Based on the behavior of the failed component we can classify the fault as omission faults, timing faults and arbitrary faults [22]. The classification is as shown in fig 3.1.

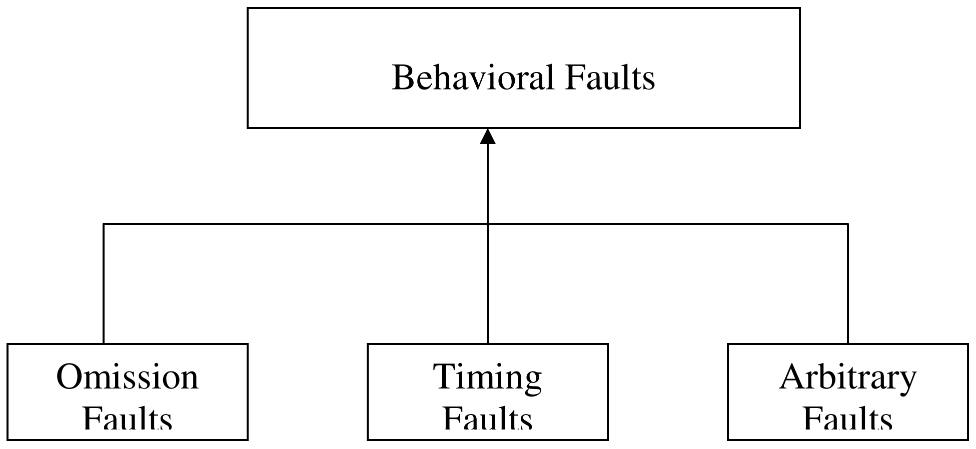

Figure 3.1 Behavioral Faults

2.3.1.1 Omission Faults

Based on the failure of service or the communication channel, there are two kinds of omission failures.

i) Service Omission Failures

Service failure is a norm in an inherently unreliable pervasive computing environment, making service failure the chief omission failure. A service crash or a device crash on which the service is executing can be classified under service omission failures. In a pervasive environment we can encounter service omission failures for various reasons like mobile device moving out of the network range, complete drainage of battery power of the device and so on.

ii) Communication Omission Failures

A pervasive computing environment consists of heterogeneous devices which can be wired as well as wireless. In wireless networks, messages sent between devices can be lost because of the unreliability associated with these types of networks. The loss of messages is known as communication omission failure.

2.3.1.2 Timing Faults

A timing fault occurs when a process or service result is not delivered within the specified time interval. Timing requirement is particularly relevant to critical services executing in the pervasive environment. For example when we consider an accident situation, there is a time bound within which the emergency service has to be contacted and failure to do so results in timing fault.

2.3.1.3 Arbitrary Faults

Arbitrary or Byzantine fault is a fault in which a computing entity does not stop running but the entity does not operate correctly or provides false result [24]. An arbitrary fault of a service is one in which the service arbitrarily omits intended processing or maliciously takes unintended processing steps. Byzantine failures results in service maliciously returning incorrect results making it very hard to detect.

Dealing with different types of faults and failures in a pervasive environment will require an intelligent system that is flexible and will respond in timely fashion to environmental changes. These features can be provided using an agent based system.

2.4 Agent Based Architecture in Pervasive Systems

Computer systems are no longer stand alone, but are interconnected to form distributed and concurrent systems. As the computing requirements have grown, the complexities of the tasks that we are capable of automating and delegating to computing systems have steadily grown. With this trend, our programming abstractions have progressed from machine code, machine independent languages and then objects to the current agent based abstractions [31]. The idea and the significance of agent based systems in ubiquitous computing solicit from the challenges posed by the intrinsic nature of pervasive systems. One major challenge in making pervasive computing a reality is the ability to access large amount of distributed information sources and services in a heterogeneous environment and to respond dynamically to changing circumstances. To deal with the challenge, we need the services of intelligent assistants, also known as intelligent agents [2].

2.4.1 Agents

There are many existing definitions of agents in the literature. However, an increasing number of researchers find the following characterization useful [14]: An agent is an encapsulated computer system which is situated in some environment and is capable of flexible, autonomous action in that environment in order to meet its design objectives. A software agent is a piece of software that autonomously acts to perform tasks on user's behalf [3]. In an agent based design, the user only needs to specify a high-level goal instead of issuing explicit instructions, leaving the how and when decisions to the agent. A software agent has various features that make it different from other traditional components such as autonomy, goal-oriented, collaborative, flexible and mobile [3].

In a distributed agent framework, we conceptualize a dynamic community of agents, which invariably need to interact with one another in order to manage their inter-dependencies. These interactions involve agents cooperating, negotiating and coordinating with one another. A distributed agent framework allows the construction of systems that are flexible and adaptable to available resources within the context.

The significance of agents in pervasive computing is recognized and highlighted in [10] as follows “An important next step for pervasive computing is the integration of intelligent agents that employ knowledge and reasoning to understand the local context and share this information to support intelligent applications and interfaces”.

2.4.2 Characteristics of an agent

The basic characteristics of an agent in pervasive computing systems are:

- Autonomous: Agents are self starting; independent entities which should ideally incorporate all fault tolerance measures within themselves and do not report exceptions or failures to higher level such as user applications.
- Adaptive: Agents must be able to change its behavior to be able to reach its goal in dynamically changing pervasive environment. As agents are situated in the environment itself, it can monitor the environment and respond quickly to changes. Agents should proactively react and adapt to changing circumstances to reach its goal.
- Mobile: As the execution environment and the resources available can change on the fly in a pervasive computing environment, the agents

should be able to move around and change its location based on the requirements

- Interactive: Agents will be acting on behalf of users with different goals and motivations. To successfully interact, they will require the ability to cooperate and coordinate with each other.
- Decentralized: Agents intrinsically are decentralized in nature. The advantages of decentralization are scalability and fault tolerance. Centralized systems can fail more often because of single point of failure, but decentralized systems can survive by spreading the load. Also decentralization allows agents to implement parallel implementation of its tasks

2.5 Fault Tolerance in Pervasive Computing

The dependence of users on well-designed and well-functioning computer systems has led to an increasing demand for dependable systems [15]. But achieving expected levels of reliability, in an environment of heterogeneous and mobile devices, which is common in a pervasive environment, is a challenge [21]. Designing for reliability requires devoting substantial time and resources. The dynamic nature and the unreliability associated with pervasive systems coupled with the energy constraints of the devices involved makes provision for fault tolerance a must in one way but a difficult and challenging task in another. Also, because of the increased heterogeneity and interconnection of diverse wireless networks, the end-to-end performance has become even more dependent on the weakest links and components [11]. Fault tolerance issues have not been well explored so far in pervasive computing. Although the general principles of fault tolerance can be applied to pervasive systems, several important and additional constraints, including user mobility, network reliability, user location, energy constraints and user density require that new design and models must be considered. Even though the fault tolerance can be achieved by adding high redundancy such as duplication of tasks, however, the additional cost and the low resource specifications will prohibit such fault tolerance in practice [17]. In [16], Banavar et al. points out that traditional fault detection and recovery techniques would need to be modified to be applicable for pervasive computing.

In [12] Edwards and Grinter articulated seven challenges facing ubiquitous computing in home, ranging from technological to social. One of the challenges pointed is reliability and fault tolerance. Edwards and Grinter point out reasons for unreliability in computing systems which does not exist in traditional domestic technologies such as television, telephones as follows:

- Differences in development culture
- Differences in technological approaches
- Differences in technological advances and limitations
- Differences in expectations of the market

Bohn et al. [19] discuss dependability requirements of pervasive computing in a healthcare environment. They mention that introducing technical equipment into health care areas imposes a high reliability constraint as there is high (possibly life-threatening) danger of a service failing when needed and stress on the paramount importance of fault tolerance in ubiquitous computing. In [20] Falvin Cristian mentions that there are a growing number of user communities for whom the cost of unpredictable, potentially hazardous failures or system service unavailability can be very costly. To minimize losses due to unpredictable failure behavior, these users will have to depend on fault tolerant systems.

2.5.1 Fault Tolerance in PerSON

When we compare the PerSON framework with other existing frameworks, we find that PerSON shares many common features but also carries a lot of advantages over them. Table 2.1 provides the summary of the features of PerSON compared with the existing frameworks.

Table 2.1 Comparison of PerSON, JXTA, ALASA and Konark [23]

Feature	PerSON	JXTA	ALASA	Konark
Support for resource constrained devices	Yes. Uses binary messages and is light-weight	Partial support. Uses XML messages for JXTA and binary messages for JXME	No support. Uses XML messages.	No support. Uses XML messages
Support for heterogeneous network	Supports TCP/IP and Bluetooth networks	Current implementation supports only TCP/IP networks. Supports different message transport binding	Relies on underlying P2P network	No support. Depends on IP multicasting
Support for multiple network interfaces	Yes.	Yes	No	No
Support for dynamic networks	Yes	Yes	Yes	Yes
Service discovery	Highly Decentralized Only reactive.	Uses distributed service indices. Reactive and proactive	Uses distributed service directories Only reactive	Highly decentralized Reactive and proactive
Service description	Simple Text	XML	XML	XML
Support for service composition	Yes	No	Yes	No
Platform Independence	Yes	Yes	Yes	Yes
Scalability	Locally scalability	Internet Scalability	Internet Scalability	Depends on the scalability of IP multicasting

But PerSON does not provide any kind of fault tolerance and does not provide mechanism for adaptation and service distribution based on the resource constraints.

When we consider a fault tolerant system model, we also need to find answers for the following challenges pointed in [16]:

- Cannot expect to get services from a particular device for a long span of time.
- Multiple devices may concurrently request service from one specific resource.
- The devices in a pervasive environment are themselves suffering from a number of limitations [4] to date, which includes but not limited to inadequate processing capability, restricted battery life, limited memory space, slow expensive connections, frequent line disconnections.
- Service discovery just lets you know how to find the resource but does not deal with resourceful distribution of tasks

To confront the challenges related to fault tolerance in pervasive environment in general and specifically for facilitating and enhancing services provided by PerSON, we have developed a resource adaptive agent system (RAAS) for PerSON.

CHAPTER 3

PICO AND PerSON

The resource adaptive agent system (RAAS) proposed in this thesis is embedded into PerSON for enhancing its features. In this chapter we first present the details of PICO framework which is implemented using the service overlay network provided by PerSON. Then we look at the architecture of PerSON in detail.

3.1 PICO

PICO [20] is a framework for pervasive computing developed at the University of Texas at Arlington's pervasive computing research lab. The goal of the project is creation of a framework consisting of dynamic communities of software entities which are goal oriented and collaborate with each other to perform services on the behalf of users and devices. Devices, delegents and communities are the different components of PICO and each one of them is discussed in detail.

In the PICO framework, a device C in the pervasive computing environment is represented by <C_{id}, C_h, F>. Here C_{id} represents the unique identifier of the device whereas C_h is the set of system characteristics and F is the set of functionalities provided by the device. For example, if we consider a mobile device such as a PDA, the system characteristics will include the operating system, processor type, memory battery, wireless card and so on. The PDA can provide functionalities of communication and computing services for a user.

A delegent is a software entity that provides services on behalf of the device. A delegent D is represented by $<D_{id}, F_d>$ where D_{id} is the unique identifier of the delegent and F_d represents the functional description of the delegent. The functional description includes the set of program modules, the rules which controls the behavior of the delegent and the goals of the delegent. For example, a delegent associated with a street lamp may include the program modules for capturing images and detecting events such as an accident. The rules may specify the protocol for state transition and communication. The goal of the delegent is to survey the coverage area.

A community will consist of one or more delegents interacting with each other to accomplish the common goal. The delegent may reside on the same device or different devices and the community may be formed statically or dynamically defined by the rules specified in the delegent.

The PICO framework stack is implemented over the Java reference implementation of JXTA (Figure 3.1). HTTP and TCP message transport bindings are supported by the reference implementation of JXTA. The implementation of the PICO framework over JXTA and prototype for the enhanced response system is described in [6].

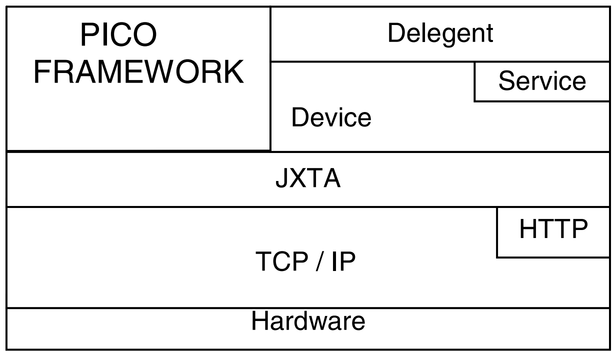

Figure 3.1 PICO Framework Stack over JXTA

A default peer group is created when the first device layer in the community is started. Any other device in the local network which starts after the instantiation of the first peer group will search for the existing peer group and joins the group. After creating a new group or joining an existing group, the device layer searches for the available rendezvous service on the local network. The device layer gets connected to the other devices through the service layer which enables the communication through message exchanges between the devices. The received messages are processed by a message handler of the respective device. The messages intended for the services are passed to the delegent layer's message handler.

The service layer helps the device layer to create and publish services. In order to create a service, the service layer creates and publishes an advertisement for the service. The service layer also creates an input pipe and listens for any incoming service requests. When the service request is received by the service layer from other peers, service layer accepts the connection and creates an input pipe and an output pipe and returns them to the device layer. The device layer uses the pipes for further communication.

A service or an application is created by implementing the interface provided by the delegent layer. When a service is started, the service will request the device layer to create a service with the specified name. The service also specifies the message handler to be invoked whenever a message is received. When an application needs to find a service on the network, the application will call the service layer to look for the service. Upon shutdown, services enter the exit state, which breaks the infinite execution of the state machine and requests the device layer to close the service. The device layer closes all the open pipes and requests the service layer to close the server pipe.

3.2 PerSON

The implementation of PerSON [23] is used to provide the overlay network for the PICO framework. PerSON is a framework for constructing the service overlay network that abstracts the underlying network details and provides connections between applications and services. The framework is not dependent on a specific development language or operating system. The reference implementation of PerSON is developed using Java. The J2SE version of PerSON can be executed in powerful devices like desktops and laptops. The J2ME version can be executed in resource constrained devices like PDA's and cell phones. Consider three devices (Figure 3.2) which are connected in two different physical networks.

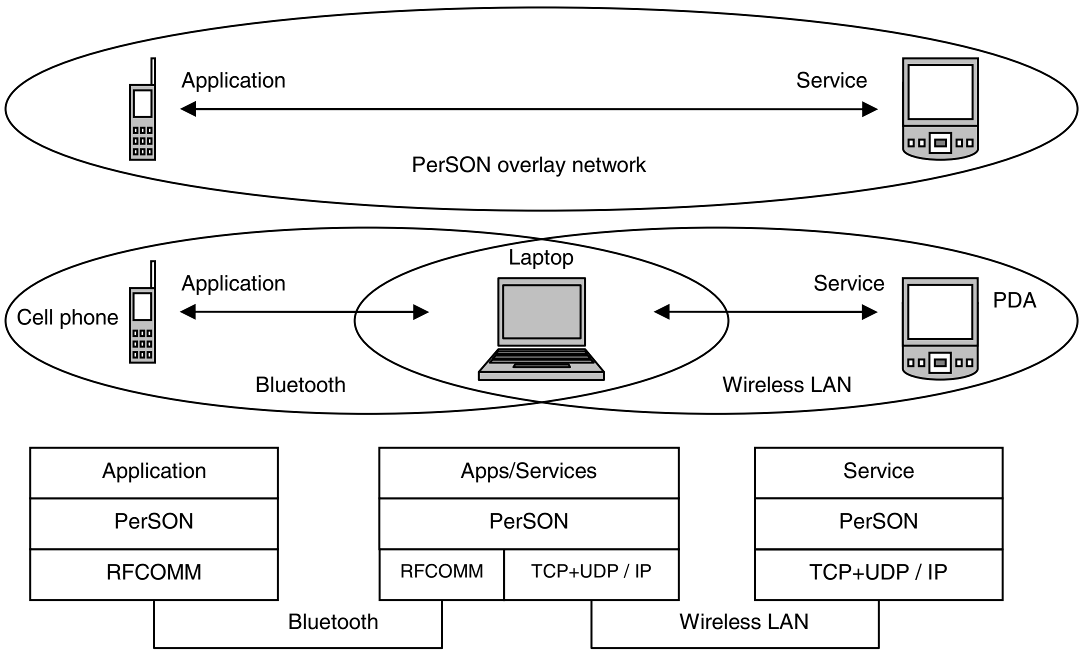

Figure 3.2 Abstract Representation of PerSON Architecture [23]

Here the laptop and the cell phone are connected via the Bluetooth network whereas the PDA and the laptop are connected using TCP/IP. In such a scenario the PDA will never be able to access the services provided by the cell phone. But PerSON masks the heterogeneity of the networks and enables communication between PDA and cell phone using the laptop. The details of the underlying network complexity are hidden by the overlay network. The details of discovering the available services and establishing the service connections are also abstracted from the applications and services.

The PerSON stack consists three layers as shown in the following figure.

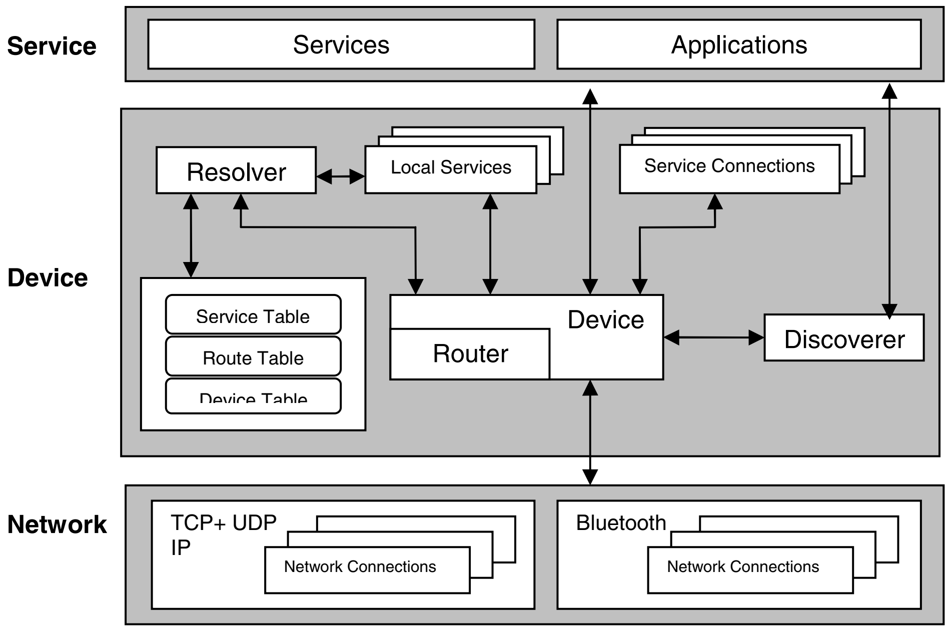

Figure 3.3 PerSON Stack[23]

3.2.1 Network Layer

The primary function of network layer is to abstract the underlying network to the devices connected to PerSON. The network specific address of the device is masked by PerSON and the device will be identified by a unique device identifier (DID). The network connections are made using the available transport protocols on each physical network. The device layer utilizes the unicast and broadcast functions of the network layer to exchange messages with the neighboring devices.

3.2.2 Device Layer

The device layer lies on top of the network layer and utilizes the services provided by the network layer. The device layer takes care of service creation, service discovery and utilization on behalf of application user service. The device layer manages all the service connections of the device currently active and performs the multiplexing and de-multiplexing of service connections with respect to the respective user service. The device layer maintains a device table which is updated with every incoming message with the physical address of the neighboring device from which the message is received. The device layer also takes care of choosing the appropriate network connection to send the outgoing messages.

The two main components of device layer are *Discoverer* and *Resolver*. The basic function of a *discoverer* is to locate a given service. The services hosted by a device are registered in the local services. Each service is identified by a unique service identifier (SID). A new service connection is spawned when the device receives a service request. The connection is utilized by the service layer to communicate with the application that requested the service. The *discoverer* will broadcast a query message to other devices to find a service and the broadcast message will be restricted in terms of hop count to limit the scope of discovery.

The *resolver* processes the query messages received by a device. The resolver tries to match the requested service in the local registry and if found then a result message is sent back by the *resolver* to the device that requested the service. The result message will contain the SID of the discovered service and the complete route to the device that hosts the service. The *resolver* of the service requesting device will process the result message and the service information is updated in the service table and the route information is updated in the route table.

The third component of a device layer is the *router*. The *router* is used only when the device has access to multiple networks and is willing to act as a bridge between those networks. When a message received is not intended for the device, the router forwards the message to the next hop in the route.

3.2.3 Service Layer

Service layer forms the topmost layer of PerSON stack. The service layer contains the user defined services and applications. New services are created by utilizing the functions provided by the device layer. The applications in the service layer request the device layer to discover other services and connect to the required services. Once a service connection is established by the device layer, an application and a service can communicate using the connection.

The service creation and service connection instantiation process is described in figure 3.4. Here we consider two devices, device S and device X where S is hosting a service and device X runs an application which requires the service from device S. The service running on S is identified by the SID and the service description and is registered in the local registry by the device layer. The application on X requests its device layer to discover the service specified by a simple service description. The query message is broadcast using the network layer and is propagated by other devices till the message reaches S. The device S responds back and the response message will include the SID. A connection request message from device layer of device X which includes a new connection identifier (CID_S) is sent to S. The request message is received by the device layer on S, which will create a new service connection and a response message with the new connection identifier (CID_X) is sent back to X. The CID_X is also provided to the service on device S. Once the device layer on S receives the success message, CID_S is returned to the application. Once the service connection is created, the application and the service can exchange the application specific data messages. In the event of service not accepting incoming connections, an error message is sent. The connection request is then reinitiated after a timeout interval. A close message is sent to X, once the application has finished accessing the service.

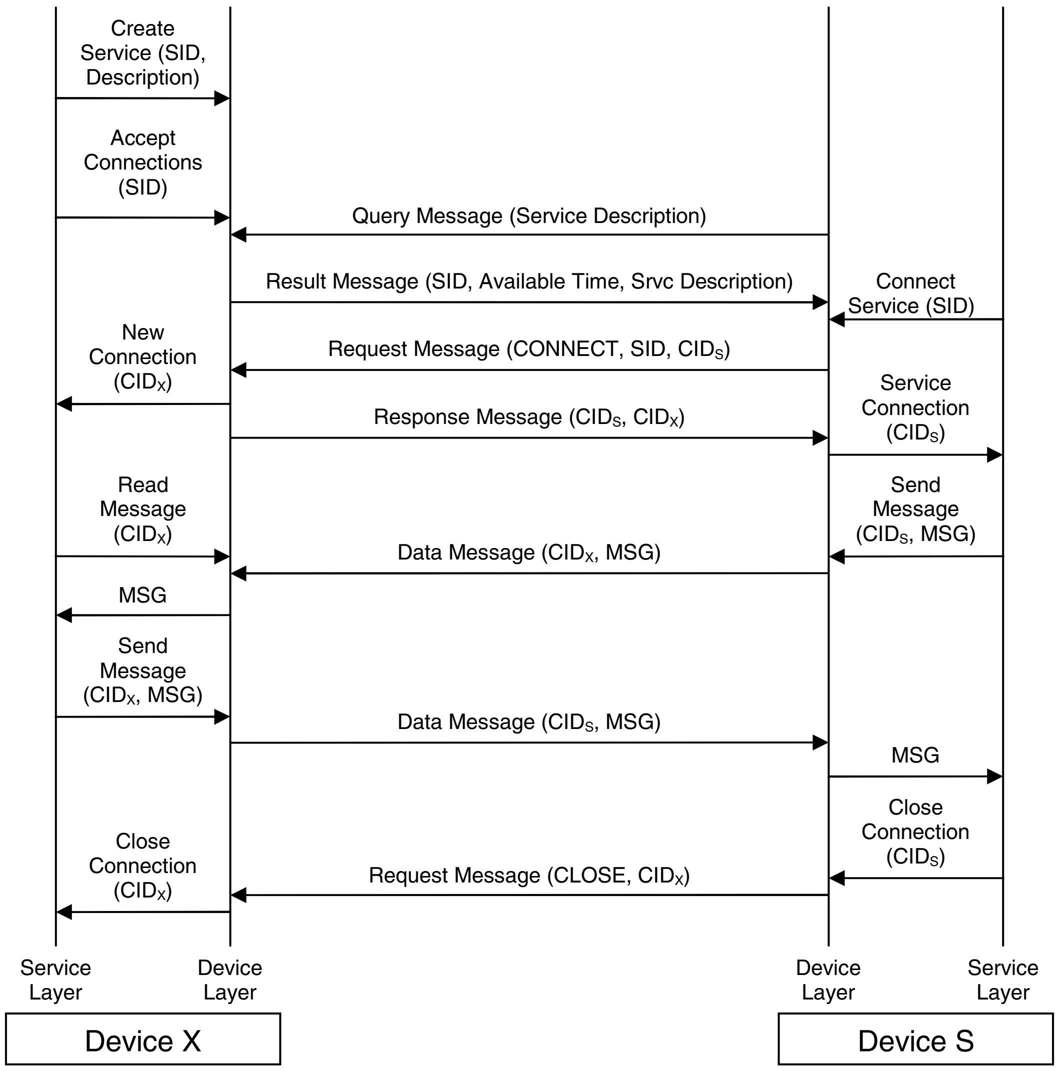
Create Service (SID, Description)
Accept Connections (SID)
Query Message (Service Description)
Result Message (SID, Available Time, Srvc Description)
Connect Service (SID)
New Connection (CID$_X$)
Request Message (CONNECT, SID, CID$_S$)
Service Connection (CID$_S$)
Response Message (CID$_S$, CID$_X$)
Read Message (CID$_X$)
Send Message (CID$_S$, MSG)
Data Message (CID$_X$, MSG)
MSG
Send Message (CID$_X$, MSG)
Data Message (CID$_S$, MSG)
MSG
Close Connection (CID$_S$)
Close Connection (CID$_X$)
Request Message (CLOSE, CID$_X$)
Service Layer
Device Layer
Device Layer
Service Layer
Device X
Device S

Figure 3.4 Service Connection [23]

PerSON takes care of masking the underlying network technology and interconnecting the services but does not make any resource aware decisions or provide other services to deal with any changes which might occur after two services are connected. To do so we will need an intelligent system on top of PerSON to keep tap on current environment situation and to make any kind of dynamic reconfigurations and this brings about the need for agent based system.

CHAPTER 4

RAAS SYSTEM MODEL AND ALGORITHMS

This thesis proposes a resource adaptive agent system (RAAS) for providing fault tolerance features in PerSON. RAAS not only deals with the fault related changes occurring in pervasive environments but also proactively tries to deal with possible future failures. As a resource aware component, RAAS tries to distribute the tasks as efficiently as possible and also facilitates the feature of checkpointing and parallel distribution of tasks. We will initially discuss the resource model, Byzantine failure model used by RAAS and then discuss the algorithms used for the implementation of various aspects of RAAS.

4.1 Resource Model

RAAS models the pervasive network environment as a collection of autonomic devices which provide services to each other. We use the model proposed in the PICO [23].

Devices in the overlay network are represented by the tuple C = < Cid,H, F >, where Cid is the device identifier, H = $\{h_1, h_2, .. . , h_n\}$ is the set of characteristics of the device, and F = $\{f_1,f_2 …., f_m\}$ is the set of functionalities available through the device. The device identifier is a mechanism to address the device which can include the IP address, MAC address etc. The functionalities or the services are identified by the service identifier (SID).

The RAAS mainly consists of agent manager service, device manager service, agent task manager service and reassign manager service. All devices using the service overlay network provided by PerSON will have the services of RAAS running. The agent of the overlay network at any particular instance of time will be chosen based on the device profile value (α) of devices. The device profile value of a device is the resource capability of the device and is calculated based on residual battery power, CPU utilization and memory usage at that instance of time [26]**.** Each of the devices involved will have a utility value specified in the device specification document. The utility value will determine the device willingness to provide any service and also plays a role in calculating the resource capability of the device. One of the most important resources in a mobile environment is battery life of a device. The time a wireless device remains usable to the user is constrained by limited battery capacity and thus remaining battery capacity plays a major role in choosing the device to perform a requested service and also in getting chosen as the agent. By considering the CPU and the memory value we ensure the distribution of load on all the available devices. The CPU load and the memory usage of a device can vary based on the number of jobs being performed by a device and the nature of those jobs. Assigning new task to a device which is already performing at its full capacity will not only cause delays in the service execution but will also affect the rate of the power consumption in the device. If the CPU and the memory in a battery operated device is utilized at its full capacity for a period of time, the device consumes more power as compared to the consumption over the same time with lower CPU load [26]. For any service request from a user we elect an agent based on the α value, which is the device with highest resource capabilities in the overlay network, at that instance of time. We then assign the user request to the chosen agent.

4.2 Byzantine Failure Model

The agent service will duplicate the high priority service requests from user or applications based on the availability of the worker devices. The agent service will implicitly check for Byzantine failures whenever there is a duplication of service. To detect the Byzantine failures we use the model proposed in [27] and modify it to implement in our component. The model proposed ensures the detection of these failures with minimum communication overhead. Here we detect Byzantine failures with only t +1 replicas instead of the normal $2t + 1$ replicas. We use the checkpointing scheme to further enhance the efficiency of the protocol.

Consider a service request which is replicated in m worker devices denoted $W_1,W_2, \ldots,W_m$. We assume that a worker device is either correct or corrupt, where a corrupt process is one that might completely halt its execution, or even deviate arbitrarily from its specification. A process is correct if it is not corrupt. Now let us assume that the number of corrupt worker devices is bounded by t. Thus, by setting $m = 3t + 1$, the agent can reach agreement despite t intrusions by running a Byzantine agreement protocol, such as in [28]. Now in the model, to ensure that a set of workers includes at least one correct worker, the agent sends the same computation task to $t + 1$ workers. Here m could be more than $t + 1$, but for every computation task we only need

$t + 1$ active workers. Once a fault is detected, the total number of workers required to deal with the failure is $2t+1$. Here each checkpoint sent to the agent by the device manager of the worker device is assigned a unique sequence number. The agent can identify the corresponding valid checkpoint for each checkpoint interval as follows.

Since we have $t + 1$ workers running a service request, there is always at least one correct worker running. If during a checkpoint interval some workers behave in an observable corrupt manner, then the agent will receive checkpoints different from the valid checkpoint. If the agent is unable to reach a consensus with a valid checkpoint from the t different worker devices, the agent will ask up to an additional t workers to execute the checkpoint interval, leading to a total of $2t+1$ workers. As at most t can be corrupt, at least $t + 1$ checkpoints reported to the workers will be the same. The agent will pick this checkpoint as the valid checkpoint and discard the faulty worker devices. By using the checkpointing mechanism, we will need to restart any task only from the previous valid checkpoint. The Byzantine failure model is implemented in the agent task manager and it performs the Byzantine failure analysis as when each checkpoint is recorded.

4.3 Algorithms

Any application requiring a service will contact the device manager of the device on which the application is running. The device manager will contact the agent manager of other devices involved in the overlay network and then run agent election algorithm before offloading the service request. The device manager then forwards the service request to the agent and keeps track of the request by running the service request algorithm. The agent service assigns the service request to agent task manager. The agent task manager takes care of the service request by running agent service algorithm. The task manager then works with the reassign manager to deal with different kind of failures.

4.3.1. Agent Election Algorithm

The agent election algorithm is run by the device manager when it receives a service request from the application running on the device. The device manager chooses the most resourceful device as the agent to take care of the service request.

```
Notations:

agentDevProf : the agent device profile object

localDevProf : the instantiation of the singleton device profile
               object

tOut : timeout value for receiving current agent message


Procedure agentCommunicator
Begin

      localDevValue := localDevProf.getValue ()
      t1 := initial timeout value for receiving α values.
      Choose agent with the highest α value.
      if no device responds before timeout occurs
            setCurrentAgent (localDevProf.Device, localDevValue)

      else
            agentDevValue :=  agentDevProf.getValue()
            if localDevValue > agentDevValue

                  setCurrentAgent(localDevProf.Device,
                  localDevValue)
            else
                   setCurrentAgent(agentDevProf.Device,
                   agentDevValue)


End Proc
```

4.3.2. Service Request Algorithm

Any application which requires a service will just connect to its device manager service and specify its request details. The device manager forwards the application request to the agent and awaits further communication from the agent. The device manager also keeps track of the agent status to reconfigure the request processing in the event of agent failure. The device manager will also reassign the task in the case of agent as well as the worker device failures.

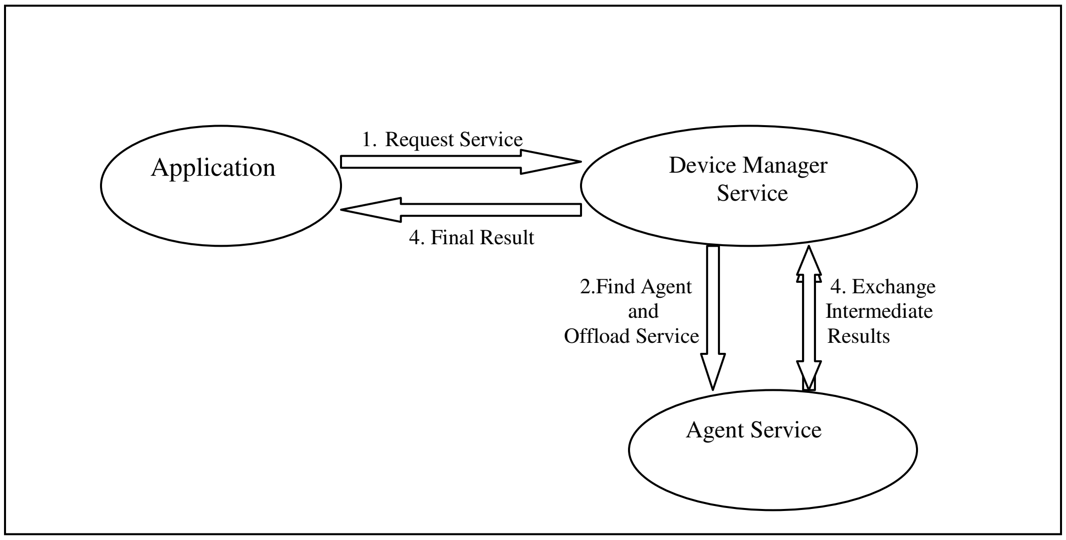

Figure 4.1 Abstract Representation of Servicing an Application Service Request

```
Notations:

agentDevProf : the agent device profile object

Reassign Manager : This service will be called to handle failure
               of services and devices



Procedure deviceMgrServReqHandler
Begin

      agentSID := agentDevProf.getSID ()

      connect(agentSID)

      Request agent to service the application request

      WAIT STATE: Wait for incoming data

      if data is from agent

            if task is complete

                  Send the final result to application
            else

                  Store the intermediate result
                  Update the necessary request specific data
                  structures
                  Go back to WAIT STATE

      else

            Data is received from mobile host; indicating agent
            has failed
            Call Reassign Manager
            Update Agent Status
            Go back to WAIT STATE


End Proc
```

4.3.3. Agent Service Algorithm

Once the agent receives request from any device to process its service request a request handler called agent task manager will be created and that service request will be assigned to the handler. The agent task manager will then perform service discovery and choose the worker devices most suitable to handle the service request. After the task is distributed, the agent waits for the worker devices to respond with the intermediate and final results. The agent also looks to duplicate the job based on its priority and the availability of the worker devices. While the agent waits for the result, the reassign manager is called upon. In the case of duplication of jobs, the agent also handles Byzantine failures. Based on the soft deadline target set for the service request, the agent task manager will trigger service failure event for the reassign manager to take action if the worker device fails to respond within the deadline time.

```
Notations:

ServiceRequest : the service task object

Reassign Manager : This service will be called to handle failure
               of services and devices

workerDevList : The list of worker devices available for that
               service


Procedure AgentServReqHandler
Begin

      workerDevList = ServiceRequest.find()

      if workerDevList == 0
            return ERROR "No devices found"
      else
            Sort the workerDevList based on resource
            capabilities

            Create ServiceRequestObject based on workerDevList

            Assign the ServiceRequestObject to worker Devices
            in workerDevList

            Based on the job priority and number of devices left
            in workerDevList, duplicate the job

            Call Reassign Manager with the ServiceRequestObject

            Receive the intermediate results from worker devices
            and also communicate the same to the Service
            Requester Device Manager.


            Periodically check the soft deadline of the task to
            flag service failure event

            Once the request is completed, trigger the Request
            Completed event and exit
```

4.3.4. Reassign Manager Algorithm

The reassign manager works independently based on the Service request object received from the Agent Service manager or the Device manager. The reassign manager performs a periodic health check on the currently active worker devices based on the soft deadline set for the respective service request. The reassign manager also takes proactive measures to guard against potential future failures. For example if the residual battery power goes below a threshold level, reassign manager will reassign the service request in the current state to the most eligible device in the worker device list. In the event of any other kind of failures, be it the device failure discovered by the reassign manager or a service failure discovered by the agent manager or the network failure, the reassign manager will perform the reassignment task to restart the failed service request on some other device.

```
Notations:

ServiceRequest : the service task object

workerDevList : The list of worker devices available for that
               service

thValue : Threshold value for the low battery level

tDeadline : Time deadline for the service request


Procedure ReassignManager
Begin
            Based on tDeadline perform periodic health check on
            active Worker Devices

            if any failure event occurs

               if workerDeviceList ! = 0

                        Check the Worker Device List to choose
                        the most resourceful unused worker
                        device
                        Assign the failed service request in
                        latest committed checkpoint state to the
                        chosen worker device
                        Change the ServiceRequest object
                        appropriately

                else await completion of a active worker device

                        Assign the failed service request in
                        latest committed checkpoint state to the
                        chosen worker device
                        Change the ServiceRequest object
                        appropriately

            else repeat the above steps till the successful
                 completion of Service Request


End Proc
```

In a pervasive computing environment, as the services leave and join the network, the availability of specific service cannot be guaranteed over time. In this chapter we discussed the various algorithms implemented in RAAS to deal with different types of faults and failures introduced by the dynamicity of the environment. We also introduced some of the main components of the RAAS which implements the above algorithms. In the next chapter we discuss the architecture of RAAS.

CHAPTER 5

ARCHITECTURE OF RAAS

The resource model and the algorithms discussed in the previous chapter are implemented in RAAS. We describe the functionalities of services provided by RAAS in detail in this chapter. The proposed component is embedded into PerSON and facilitates successful execution of service requests using PerSON, with proactive and reactive measures against different types of failures.

5.1 RAAS Architecture

Once a service request is received from a user application, the device manager of the device runs an agent election algorithm involving the agent services of devices in the network. The device with the maximum resources is elected as the agent. The device chosen will continue to act as the active agent and will continue to accept new services until one of the following events occur

- A device with higher capabilities enters the overlay network and is elected as an agent.
- At any instance, one of the existing devices has more resource capability than the current agent. This could happen due to slower resource drainage than the agent.

- The device elected as agent has its resources currently tied up due to active services running on the agent.

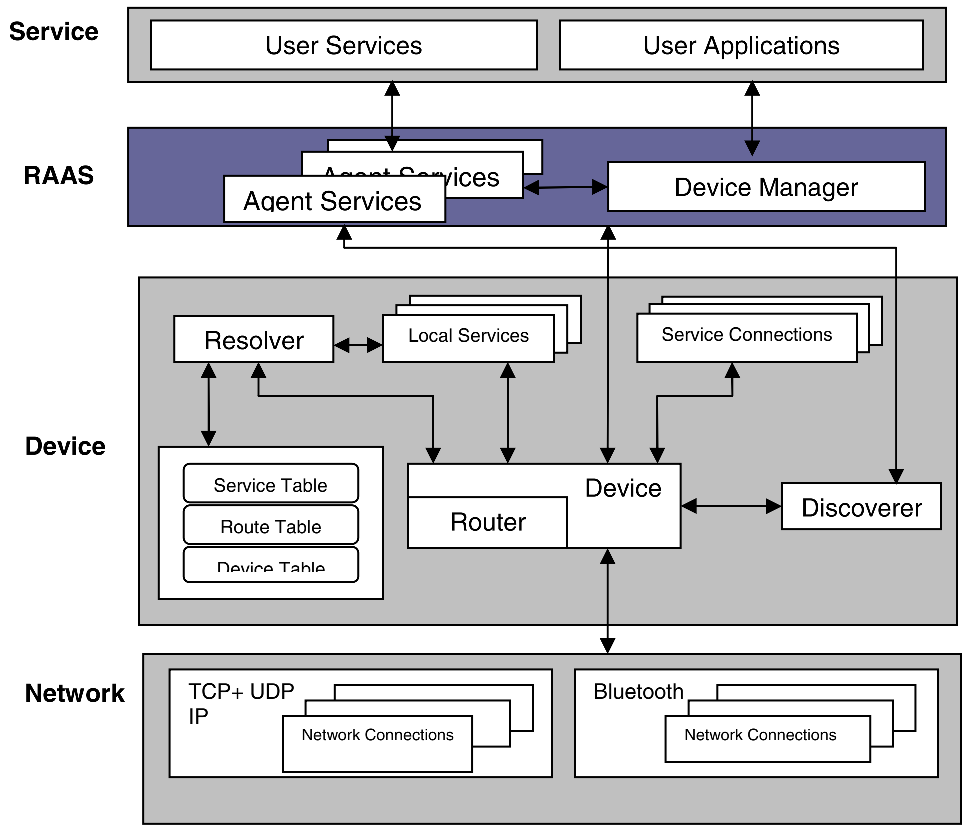

Figure 5.1 PerSON with RAAS

The RAAS consists of the following main services:

- *Agent Manager*: This service performs the high level management of different functional services of RAAS, running them as and when required.

- *Agent Communicator*: Agent Communicator handles the communication of the device profile of the mobile devices in the network. Agent Communicator service will be run in all the mobile devices in the overlay network created by PerSON.
- *Service Manager*: Service Manager runs on the device which has been currently chosen as the agent. Service manager will overlook and manage all tasks and services run by the agent device.
- *Agent Task Manager*: Agent Task Manager will be created for each task request received by the agent service manager. The Agent Task Manager will handle the data structures required to maintain each request and perform the distribution of the user requests among the various devices.
- *Service Discovery Manager*: Service Discovery Manager will work with PerSON to assist Agent Task Manager in identifying the devices available which are called worker devices. The service discovery manager sorts the devices based on the resource capabilities to perform the particular task.
- *Device Manager*: This service will be active for each device involved in the network. The device manager forms the interface between the user application and PerSON. The device manager also provides resilience against a case of agent failure while processing the service request.
- *Reassign Manager*: Reassign Manager will be called upon by the Agent Task Manager and Device Manager to perform the reassignment of a

specific sub-task or service. This happens when the original device (s) assigned with the task, fails to do its job successfully.

The architecture of the RAAS is shown in figure 5.2.

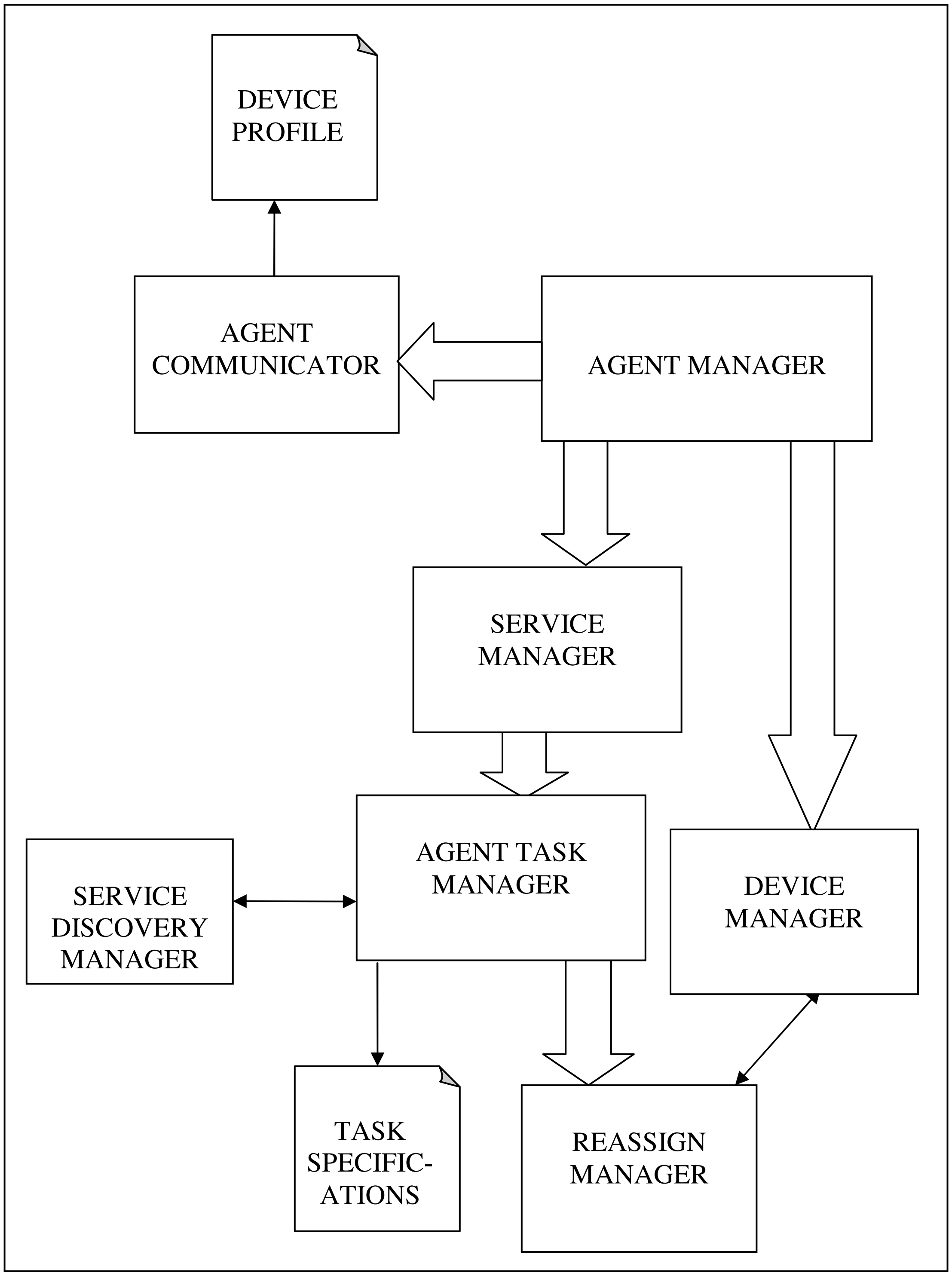

Figure 5.2 RAAS Architecture

In the following section we describe the functionality and the inner workings of each of the service component of RAAS in detail.

5.1.1 Agent Manager

Agent manager forms the starting point of RAAS which needs to be run on all the devices desiring to be part of PerSON. The agent manager will start up the Agent Communicator for the device and periodically query the Agent Profile data structure to ascertain the current agent in the system. The agent manager will also ensure a negative response for any service request if the device is running short on resources.

5.1.2 Agent Communicator

The Agent Communicator is primarily responsible for the communication of device profile value between devices. When "Device Profile" request message is received, the communicator will return the current resource capability value of the host device. The agent communicator takes responsibility of finding any particular local service running on the device and also communicates the Agent Task Manager SID on request from agent service manager.

The agent communicator exchanges the device profile object which contains the status of the different resources of the device at that instance of time. At this moment we are taking battery level, CPU usage, the memory usage and the distance of the application device from the agent device into consideration for calculating the capability of the device. It can be easily extended to take other criteria based on the requirements of that specific network.

5.1.3 Service Manager

The service manager will be active when the host device is elected as the agent. The main task of the service manager is the management of all the tasks and service requests requested by the device manager on behalf of user applications and user services. For each task or service request received, the service manager will assign the task to an agent task manager handler.

The service manager will also provide mechanisms for offloading a service itself based on the specification of the request received, priority of the request and the availability of a device to perform the service.

5.1.4 Agent Task Manager

The basic functionality of the agent task manager is to ensure the successful execution of the task or service assigned to it which is achieved by implementing the algorithm described in 4.3.3.

When a service is assigned to an Agent Task Manager, the Agent Task Manager contacts the service discovery manager to get a list of worker devices available to perform the assigned task. If a task is eligible to be divided into subtasks, the agent task manager assigns each of the subtasks to available devices; otherwise the agent task manager just assigns the entire task to the devices returned by the service discovery manager. The task is assigned to the worker devices based on the resource capability of the devices. Higher resource capability worker device is chosen first to be assigned with the user request.

The task manager also looks at the task specifications while performing the assignment of tasks. If the task being assigned has a higher priority, the agent task manager will try to duplicate the task assignment in different devices based on the availability. For example if a high priority task is subdivided into task A and B, and if there are two devices available which are willing to provide service A and one device willing to provide the service B, then the agent task manager will duplicate the subtask A on the two available devices.

The actual reassignment of the task to a specific device is done by collaborating with the device manager of that device. Once the task has been assigned, the agent task manager will wait for the result for a certain amount of time. The time agent task manager waits is a function of the soft deadline target specified by the application in the task specification. The agent task manager then contacts the reassignment manager to check if there is any kind of failure with the worker devices currently servicing user request or to check for any potential failure cases and to take proactive measures against the probable failures.

To support checkpointing, Agent task manager provides mechanisms to store the intermediate result associated with a subtask. When a device fails, the computation is restarted from the most recent saved state thus saving time and resources. The agent task manager also periodically interacts with the device manager of the parent device which initially requested the service. The agent task manager will replicate the state of the task-list and the assigned device-list with the associated intermediate results in the parent device by collaborating with the device manager of the service requester device.

This is to ensure that, in the event of agent device failure, the state of the sub task is not lost completely and providing tolerance against failure of worker devices as well as the agent. Though we can safely assume here that the parent device will not fail during the service request but in case of such a failure, the agent task manager provides mechanism to cache the result for some time to check if the parent device comes up again before discarding the result completely.

5.1.5 Reassign Manager

The reassign manager implements the algorithm discussed in section 4.3.4 to provide fault tolerance service to the agent task manager. Once any agent task manager initiates an interaction, the reassign manager will check the status of the active worker devices in the device-list of agent task manager. The reassign manager will check the agent manager of the respective worker devices through heart beat messages to determine the status of the device. The heart beat message will also carry the current resource capability profile status enabling the reassign manager to take some pro-active measures against the probable failure of the device. For example, if the battery level of any device has gone down below a critical threshold value, the reassign manager will try to reassign the last saved checkpoint of the task to another device.

5.1.6 Device Manager

The device manager runs on all the devices in the overlay network and forms communication interface to PerSON for all user applications. The device manager implements the algorithm described in 4.3.2.

Figure 5.3 shows the abstract representation of a user request from the user device perspective.

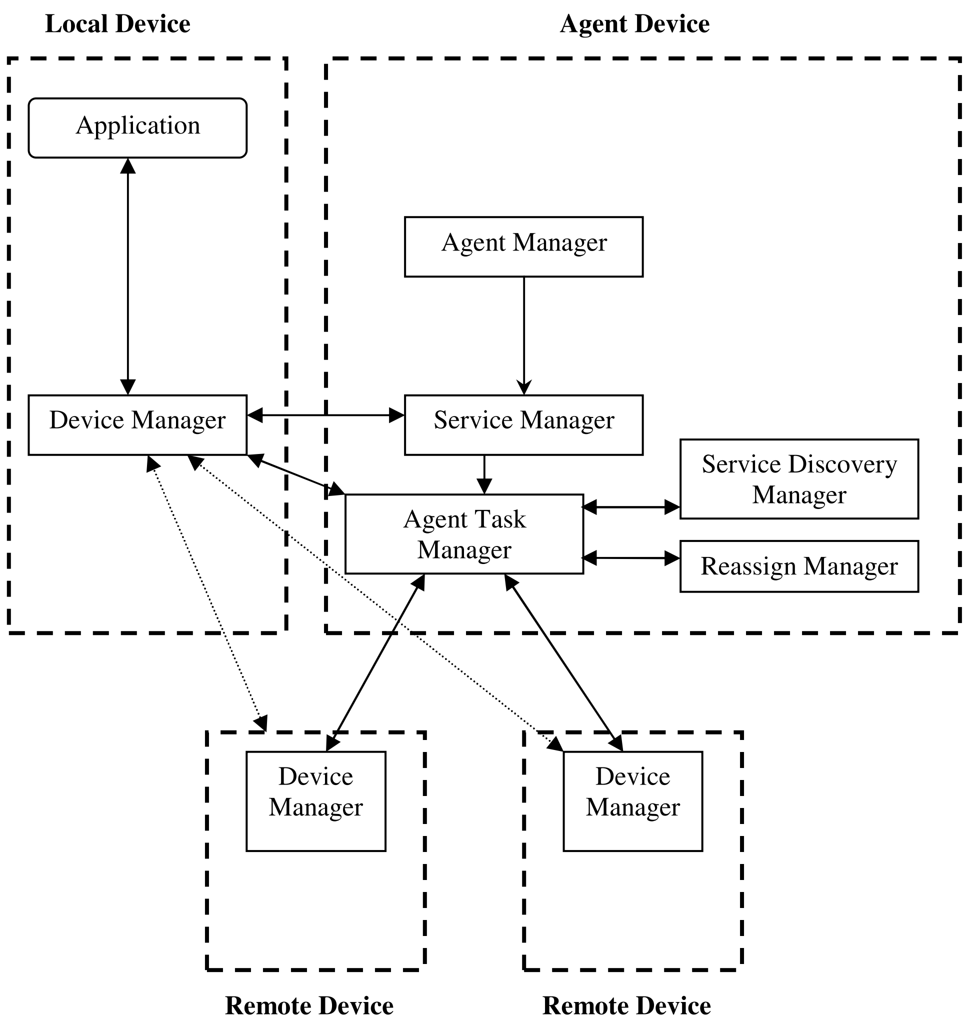

Figure 5.3 User Level Abstract Representation of Service Execution

The device manager finds the current agent for the service request by running the agent election algorithm described in 4.3.1 and queries the service identifier (SID) of the service manager. The device manager service accepts user requests from the device, generates the local task id for the service request and spawns a request handler to deal with the particular request. The device manager communicates with the service manager of the current agent for the remote execution of the service. The device manager wraps the user request with the task specification details before sending the request to the service manager. The request handler of the device manager then offloads the task to the service manager using its SID and also creates the data structures required to store the device list and the intermediate checkpoint results of the service request. The request sent will have details like service SID, the task id, the service specifications, and other details required by the agent to provide the service.

The device manager periodically collaborates with the agent to save the intermediary states of the tasks and the device list. The device list consists of device id's and service id's of the devices which has been assigned the service request by the agent. In the event of failure of the agent, the device manager of the service requester will communicate directly with the device manager of the respective worker devices in the device list to continue the execution of the offloaded tasks. In case of further failure of any device in the device list, the device manager will perform the function of reassigning the task to some other device with the help of reassign manager. The device manager of the service requester will pass on the final result to the user application.

The RAAS architecture discussed above facilitates and enhances the features provided by PerSON. The RAAS primarily consists of agent services, dynamically collaborating to accomplish fault resilience and increase the dependability of the system. It uses resource aware techniques in choosing the service provider and also provides checkpointing facility. RAAS not only deals with failures at application level, but also successfully adapts and deals with the challenges posed by the dynamicity of a pervasive environment.

CHAPTER 6

RESULTS

We look at the performance and experiment results of executing service requests on PerSON embedded with RAAS in varying conditions.

6.1 Device Specifications

The test bed environment consists of devices varying in capacity in terms of CPU processing speed, battery energy, memory and heterogeneity in terms of hardware as well as software.

The resource constrained devices are Sharp Zaurus SL-5500 PDAs. Each of these PDAs contains an Intel StrongARM processor running at 206MHz. They run Jeode Personal Profile for Java with 32MB of RAM. Their operating system is the Qtopia Desktop Environment (QDE) with Linux kernel 2.4.18 (Cacko). Their communications links used by the PDAs are D-Link Air DCF- 50W compact flash cards

The first resourceful device is a Dell Inspiron 5150 laptop with a 3.06 GHz, Pentium 4 Processor and 256 MB RAM. The laptop has a built-in Dell TruMobile 802.11g wireless network card. Its operating system is the Windows XP and has been installed with JDK 5.0 version of the java virtual machine.

The second resourceful device is a IBM ThinkPad with a 1.60 GHz, Pentium M Processor and 760 MB RAM. The laptop has a built-in Intel® PRO/Wireless 2200BG Network Connection wireless network card. Its operating system is Windows XP Home Edition with Service Pack 2 and has been installed with JDK 5.0 version of the java virtual machine.

The devices in the environment form an ad hoc network and communicate with each other using the Wi-Fi (802.11b) protocol.

6.2 Overhead due to RAAS

In this experiment we will analyze the worst case performance of RAAS. In this scenario, we use 2 Laptops where one Laptop provides the MatrixMultiplication service which is requested by the application running on the other laptop. The MatrixMultiplication service performs matrix multiplication of square matrices based on the size and matrices provided to the service. The application requests for MatrixMultiplication service for square matrices starting from size 40, in increments of 40, up to 520. The experiment measures the time consumed to execute the complete task set using the PerSON without RAAS and then we conduct the same experiment on PerSON with RAAS. To simulate the worst case scenario, we have conducted the experiment under the following constraints:

- There will be no failures of any kind during the whole experiment.
- The required service runs only on the resourceful device i.e. a Laptop.

- The agent election algorithm runs with a timeout value of 10 seconds for choosing the most resourceful agent. As a result of which each execution will take 10 seconds extra for agent election.
- The agent service creates all the necessary data structures to support check pointing and failure resilience features though they are never used in the experiment.

The time required to complete each task is recorded and tabulated in table 5.1.

Table 6.1 Time Comparison Results

Matrix Size	PerSON w/o RAAS (secs)	PerSON with RAAS (secs)
40	2.96	17.308
80	3.274	17.448
120	4.907	18.94
160	7.972	22.125
200	12.948	27.172
240	20.82	34.873
280	30.544	44.557
320	44.244	58.087
360	62.38	76.092
400	83.551	96.321
440	110.939	124.653
480	144.478	160.093
520	186.529	200.521

6.2.1 Observations

Running PerSON with RAAS has an overhead and the difference due to overhead decreases proportionally with the increase in the size of the data task set.

Though the percentage increase in time due to RAAS is negligible when the task size is large, but there is a significant difference for smaller task set.

To overcome the overhead problem for smaller tasks the application can choose to bypass the fault resilience feature completely and get the best performance possible. The application can also change the timeout value for choosing the agent by setting it to a lower value or by asking RAAS to choose the first agent who responds.

6.3 Resource Adaptiveness of RAAS

In this experiment we validate the resource adaptation feature added by RAAS to PerSON. We setup a test bed with two resourceful devices i.e. the laptops and one resource constrained device i.e. the PDA. The application requesting for MatrixMultiplication service is run on laptop. The service is run on the other laptop and the PDA. The application requests for matrix multiplication of square matrices starting from size 40, in increments of 40, up to 520. We measure the time consumed to execute the complete task set using the PerSON framework without RAAS and then we conduct the same experiment for PerSON with RAAS. The remaining battery energy on the PDA and service provider laptop is periodically recorded until the set of tasks is completed. To test the resource adaptiveness, we have conducted the experiment under the following constraints:

- There will be no failures of any kind during the whole experiment.
- The required service is running on one resource constrained device i.e. the PDA and one resourceful device i.e. the Laptop.

- The agent election algorithm runs with a timeout of 10 seconds and as result of which, each execution will take 10 seconds extra.
- The agent service creates all the necessary data structures to support check pointing and failure resilience features though they are never used in the experiment.

The time required to complete each of the task set is recorded and tabulated in table 6.2

Table 6.2 Time Comparison with Resource Adaptation Results

Matrix Size	PerSON w/o RAAS (secs) Device Chosen: Laptop (L) or PDA (P)	PerSON with RAAS (secs) Device Chosen: Laptop (L) or PDA (P)
40	3.526 (P)	17.308 (L)
80	3.886 (P)	17.448 (L)
120	5.538 (P)	18.94 (L)
160	8.372 (P)	22.125 (L)
200	13.589 (P)	27.172 (L)
240	19.038 (L)	34.873 (L)
280	33.959 (P)	44.557 (L)
320	43.042 (L)	58.087 (L)
360	60.657 (L)	76.092 (L)
400	84.561 (P)	96.321 (L)
440	110.939 (P)	124.653 (L)
480	143.236 (L)	160.093 (L)
520	186.529 (P)	200.521 (L)

The following plot shows the percentage of remaining battery energy over time on the PDA when all the tasks are executed with and without RAAS on PerSON.

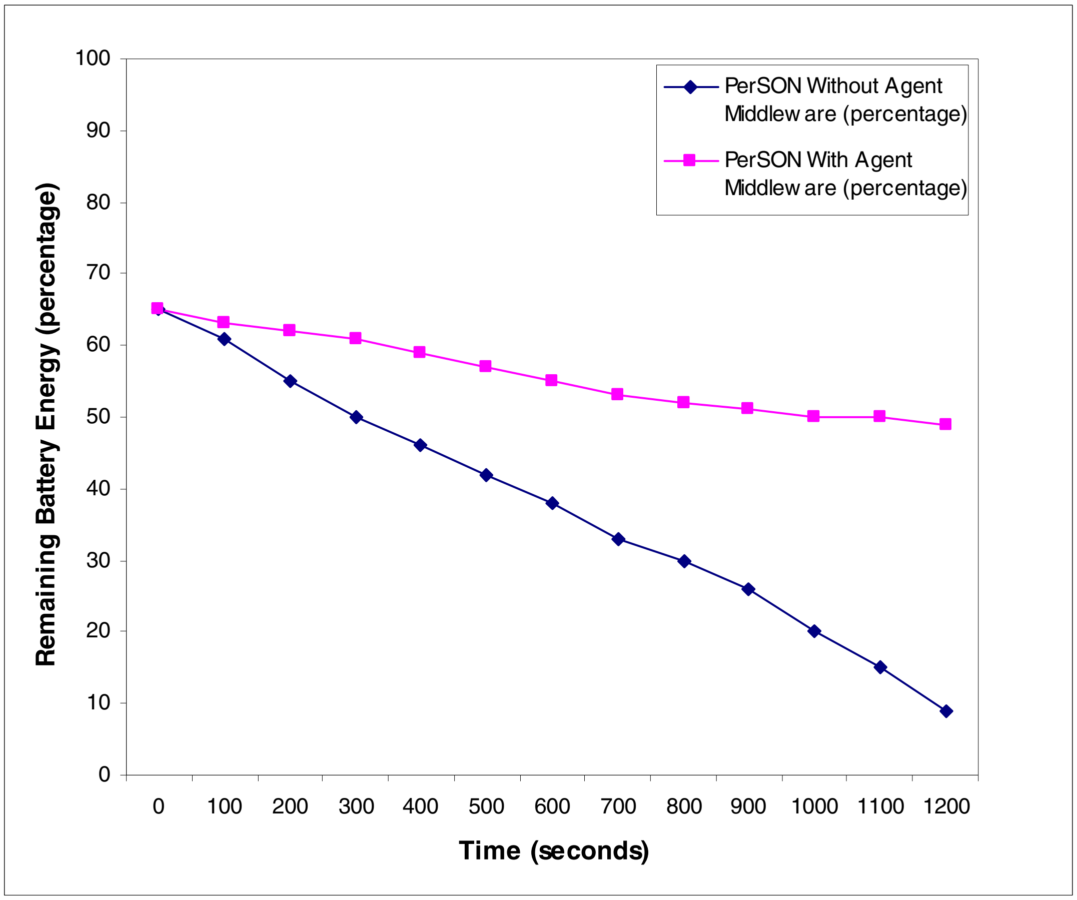

Figure 6.1 Remaining Battery Energy in PDA with Time

The plot in figure 6.2 shows the percentage of remaining battery energy over time on the Laptop when all the tasks are executed with and without RAAS on PerSON.

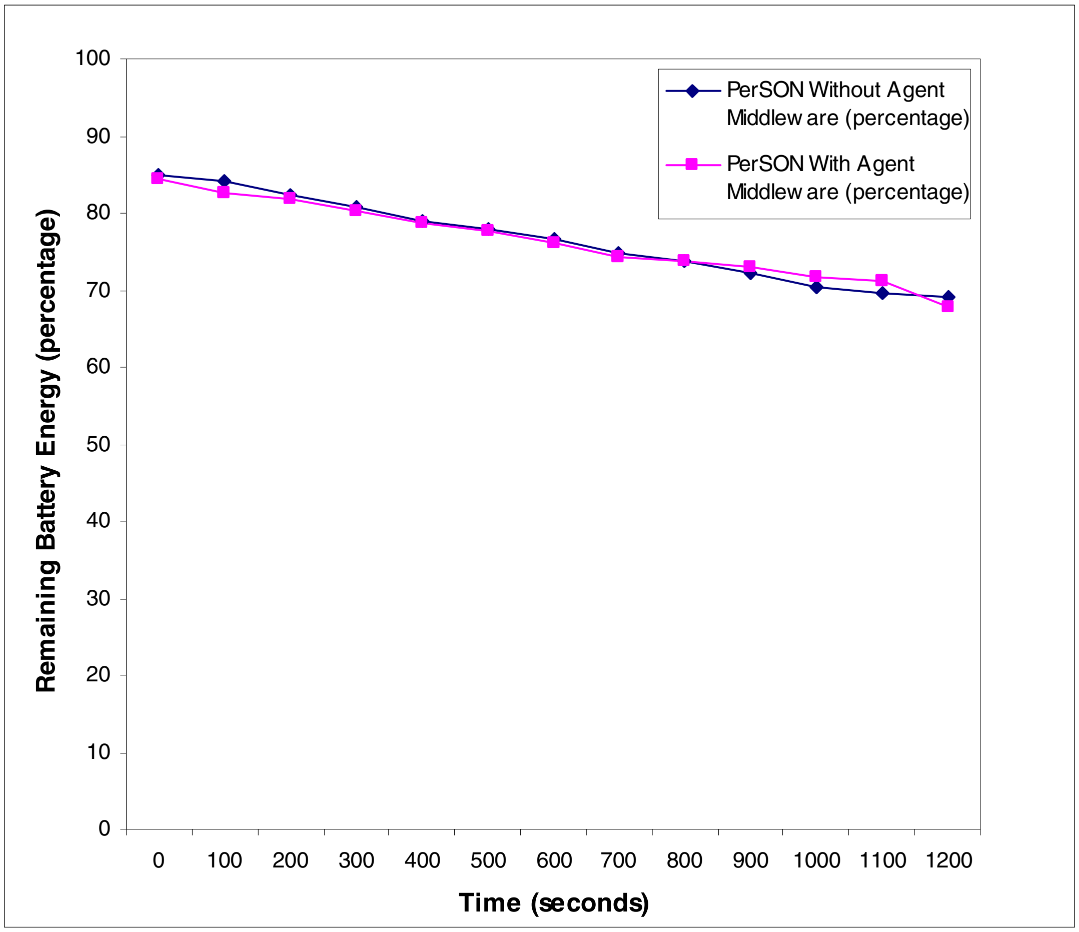

Figure 6.2 Remaining Battery Energy in Laptop with Time

6.3.1 Observations

The resource adaptive property of the RAAS is clearly highlighted from this experiment. When we ran the experiment without RAAS, PerSON randomly selected PDA or the laptop as its service provider while in the second case; RAAS running on top of PerSON ensured that the higher resourceful device is always chosen as the service provider. We can also observe that difference in time required to finish the task has reduced because of the choice of laptop over the PDA. The time difference which remains in executing the service is mainly due to the 10 seconds timeout set for agent election as discussed in the previous experiment.

Though the laptop was chosen as the service provider, we notice that there is no difference in battery energy lost when we compare it with the scenario where the laptop was not chosen as the service provider. We can note from figure 5.2 that the difference is practically negligible and there is no energy lost.

In the case of the PDA we see a significant difference in the remaining battery energy at the end of the experiment. The battery energy lost in the second case is primarily due to the energy dissipated by the PDA during idling. The only other significant task performed by the PDA during the experiment for second case is replying for agent service enquiry and the device profile status. As seen from the following plot, there is a significant energy savings for the PDA at the end of the experiment. We observe that as the size of the task, the energy savings that can be achieved using the resource adaptive feature of RAAS also increases. From the plot in figure 5.3 we see that, the remaining battery energy in PerSON without RAAS is 9% and with RAAS is nearly 50% which gives battery power savings of about 40% by the resource adaptiveness feature of RAAS.

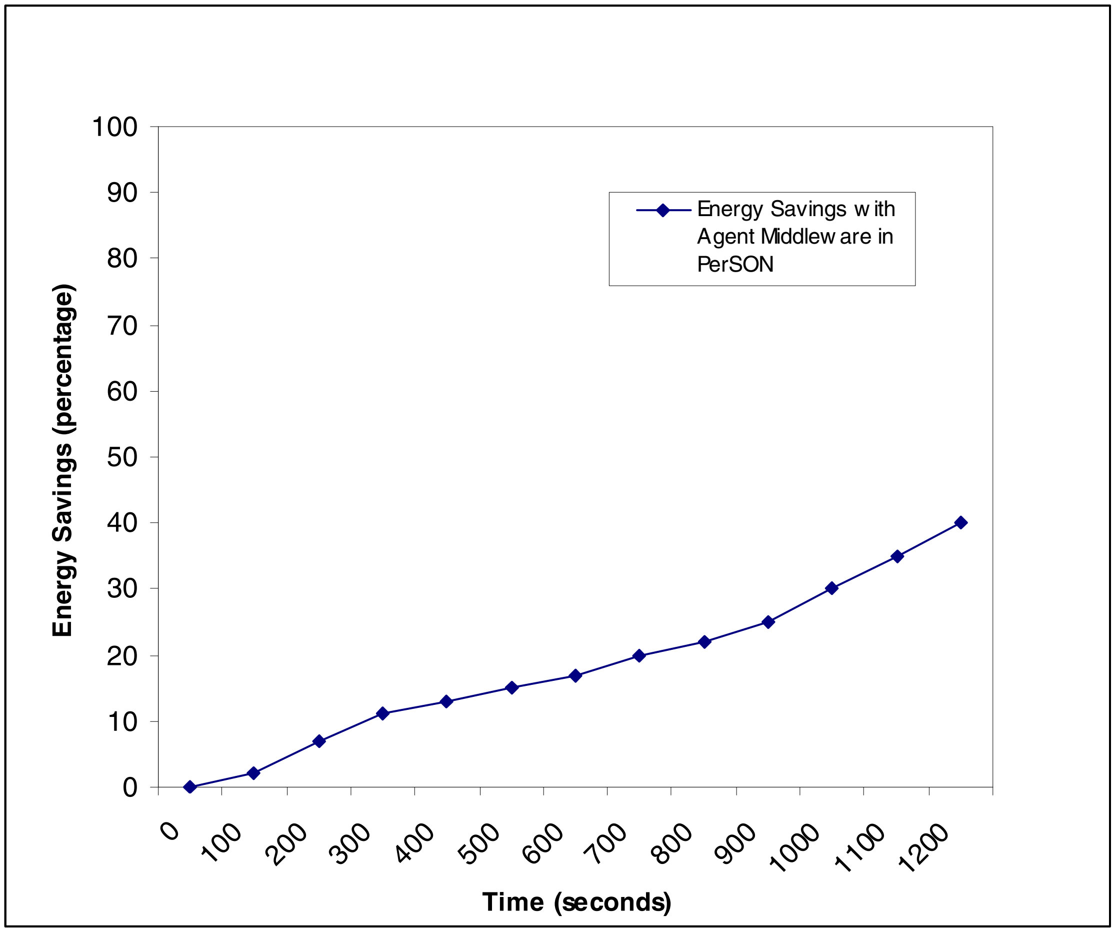

Figure 6.3 Energy Savings in PDA with Time

6.4 Fault Resilience of RAAS

In this experiment we test the fault resilience feature added by RAAS to PerSON. We setup a test bed with one resourceful device i.e. the laptop and three resource constrained devices i.e. the PDA's. One of the PDA's will run the application which requests for the MatrixMultiplication service run on the other PDA's. The application requests for matrix multiplication of square matrix of size 520.The experiment measures the time consumed to execute the complete task set using the

PerSON framework without RAAS and then we conduct the same experiment on PerSON with RAAS in a faulty environment. During the experiment, to test the fault resilience, we have conducted the experiment under the following constraints:

- The devices involved can experience failures such as going out of range or go down because of battery energy.
- The agent election algorithm runs with a timeout value as 10 seconds for choosing the most resourceful agent and as result of which, each execution will take 10 seconds extra.
- The agent service creates all the necessary data structures to support check pointing even though they are never used in the experiment.

The time required to complete the task set is recorded and tabulated in table 6.3

.

Table 6.3 Time Comparison under Faulty Conditions

Case	Kind of Failure	PerSON w/o RAAS (secs)	PerSON with RAAS (secs)
1	No failure	186.529	200.521
2	Service provider goes out of range around 90th second after being assigned the service	X (The task is never completed)	320.982
3	Service provider goes down around 120th second after being assigned the service	X (The task is never completed)	310.243

6.4.1 Observations

When the device which provides the service fails, the task is never completed in PerSON even though there are other devices providing the same service at that instance of time. But RAAS reacts to the failure as soon as a failure is detected and then reassigns the service request to the next most capable device in the network. In the case of failure, RAAS takes more than 200.521 + 90 seconds, as the agent might not detect the failure as soon as failure occurs. The detection of the failure depends on when the device, on which the service is currently executed, is polled for the device profile value. In the 3rd case, even though the device goes down at the 120th second, it still takes less time to complete the task then the 2nd case, highlighting the proactive failure resilience feature provided by the RAAS. The agent predicts that the device failure before the device actually fails and reassigns the task to the other device. Again in this case the time in seconds might vary based on how soon the agent detects the probable failure.

6.5 Checkpointing feature of RAAS

In this experiment we test the check-pointing feature added by RAAS to PerSON in a failure prone ad-hoc environment. We setup a test bed with one resourceful device i.e. the laptop and three resource constrained devices i.e. the PDA's. The two PDA's run the MatrixMultiplication service whose service will be utilized by the application running on the third PDA. The application requests for matrix multiplication of square matrix of size 520. We measure the time consumed to execute the complete task set using the PerSON framework without RAAS and then we conduct the same experiment using PerSON with RAAS. We have conducted the experiment under the following constraints:

- The devices involved can experience failures due to mobility or because of low battery energy.
- The agent election algorithm runs with a timeout value of 10 seconds for choosing the most resourceful agent.

The time required to complete the task set is recorded and tabulated in table 6.4.

Table 6.4 Time Comparison with Checkpointing under Faulty Conditions

Case	Kind of Failure	PerSON w/o RAAS (secs)	PerSON with RAAS (w/o checkpointing) (secs)	PerSON with RAAS (with checkpointing) (secs)
1	No failure	186.529	200.521	201.103
2	Service provider goes down around 90th second after being assigned the service	X (The task is never completed)	320.982	238.114

6.4.1 Observations

When we compare the results to the previous experiment, we note that the time savings and thus resource savings due to checkpointing service is very significant. Here to complete the same task, we see that it nearly takes 90 seconds less when we restart the services from the saved state rather than completely restarting the services. The time taken for completion will be effected by the frequency with which checkpoints are saved and the frequency with which the service provider is polled for its current resource status.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

The thesis presented the architecture and implementation of RAAS, an architecture for introducing the fault tolerance component on an existing middleware framework for pervasive computing. Here we presented the design and evaluation of a light weight agent component for provisioning resource aware task distribution as well as resilience against various faults and failures. RAAS also provides features for parallel distribution of user tasks and flexibility to configure the agent services based on the needs of a service request.

Implementation results show that the RAAS achieves significant energy savings through resource aware decisions when compared to the performance evaluation of the middleware without RAAS features. Experiments also show that the overhead introduced by RAAS decreases as the data size of user request increases, even in the ideal scenario of no faults and failures.

The future work includes incorporating support for service composition and a model for *trust* in the RAAS. Service composition will facilitate in achieving fault tolerance even when direct service match is not available.

REFERENCES

[1] Weiser, M. (1991) The Computer for the 21st Century. Scientific American, 265(3), 94-104

[2] Ulema, M., Waldman, M., Kozbe, B. (2006) A Framework for Personal Mobile Agents in Wireless Pervasive Computing Environment. Wireless Pervasive Computing, 2006 1st International Symposium, 16-18

[3] Liu, R.; Chen, F., Yang, H., Chu, W.C., Yu-Bin Lai. (2004) Agent-based Web services evolution for pervasive computing. 11th Asia-Pacific Software Engineering Conference

[4] Satyanarayanan, M. (2001) Pervasive Computing: Vision and Challenges. IEEE Personal Communications, 8(4), 10-17.

[5] Mark Weiser. (1994) "The world is not a desktop". Interactions, pp. 7-8

[6] Chaozhen Guo, Wu Dong, Jia Wu. (2001) Research on Multi-Agent for General Group Decision Support System. Computer Supported Cooperative Work in Design. The Sixth International Conference, (7), 308-312.

[7] L. Chen, T. Finin, A. Joshi. (2004) An Ontology for Context-Aware Pervasive Computing Environments. Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review, Cambridge University Press, 197-207.

[8] N. R. Jennings. (2000) On Agent-Based Software Engineering. Artificial Intelligence, (117), 277-296

[9] Felix C. Gartner. (1999) Fundamentals of fault tolerant distributed computing in asynchronous environments. ACM Computing Surveys, 31(1), 1-26

[10] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J.B.Sussman, and D. Zukowski. (2000). Challenges: An application model for pervasive computing. Mobile Computing and Networking, 266-274.

[11] Upkar Varshney and Alisha D. Malloy. (2006). Multilevel fault tolerance in infrastructure-oriented wireless networks: framework and performance evaluation. International Journal of Network Management, 16(5), 351-374

[12] W. Keith Edwards and Rebecca Grinter. (2001). At Home with Ubiquitous Computing: Seven Challenges. Proceedings of the Conference on Ubiquitous Computing 256-272.

[13] J. Bohn, F. Gartner, and H. Vogt. (2003). Dependability Issues of Pervasive Computing in a Healthcare Environment . Proceedings from the First International Conference on Security in Pervasive Computing

[14] Flavin Cristian. (1991). Understanding Fault-Tolerant Distributed Systems. Commmunications of the ACM, 34(2), 56–78.

[15] Doreen Cheng, Henry Song, and Alan Messer. (2006). Reliability, Diagnosis – Challenges to Pervasive Computing. 8th Annual Conference on Ubiquitous Computing.

[16] Sharmin, M., Ahmed, S., Ahamed, S.I. (2005). SAFE-RD (secure, adaptive, fault tolerant, and efficient resource discovery) in pervasive computing environments Information Technology: Coding and Computing. International Conference on Volume 2, Issue , 4-6, 271 – 276

[17] S. S. Yau, F. Karim, Y. Wang, B. Wang, S. K.S. Gupta. (2002). Reconfigurable Context-Sensitive Middleware for Pervasive Computing, IEEE Pervasive Computing, IEEE Computer Society Press, 33-40

[18] Shiva Chetan, Anand Ranganathan, and Roy H. Campbell. (2005). Towards Fault Tolerant Pervasive Computing. In IEEE Technology and Society . Volume: 24, No. 1, pp 38-44.

[19] Mark Weiser. (1993). Some Computer Science Problems in Ubiquitous Computing. Communications of the ACM.

[20] Kumar, M., Shirazi, B., Das, S.K., Singhal, M., Sung, B.Y., and Levine, D., (2003) PICO: A Middleware framework for Pervasive Computing. IEEE Pervasive Computing, 2(3), 72-79.

[21] Li Jiang, Da-You Liu, Bo Yang. (2004). Smart home research - Machine Learning and Cybernetics. Proceedings of 2004 International Conference on Volume 2, Issue ,26-29, 659-663.

[22] Coulouris,G.F., Dollimore,J., Kindberg,T. (2005). Distributed Systems, Concepts and Design. 4th edition Addison-Wesley.

[23] K. Senthivel. (2006). PerSON - A framework for service overlay network in pervasive environments. Masters Thesis, The University of Texas at Arlington, TX

[24] Lamport, Shostak and Pease. (1995). The Byzantine Generals Problem, in Advances in Ultra-Dependable Distributed Systems. N.Suri, C.J.Walter, and M.M.Huue(Eds.). IEEEComputer Society Press.

[25] Kalasapur, S., Senthivel, K. & Kumar, M. (2006) Service Oriented Pervasive Computing for Emergency Response Systems. Pervasive Computing and

Communications Workshops 2006. Ubicare'06. Proceedings of the Fourth Annual IEEE International Conference on, 517 – 521.

[26] L. Benini and G. de Micheli. (2000). System-level power optimization: techniques and tools, ACM Trans. Des. Autom. Electron. Syst., vol. 5, no. 2, 115-192.

[27] Adnan Agbaria, Roy Friedman. (2005). A Replication- and Checkpoint-Based Approach for Anomaly-Based Intrusion Detection and Recovery. Conference: International Conference on Distributed Computing Systems - ICDCS(Workshop). 137-143.

[28] M. Castro and B. Liskov. (1999). Practical Byzantine Fault Tolerance. In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, 173–186.

[29] Ubiquitous computing http://en.wikipedia.org/wiki/Ubiquitous_computing

[30] Pervasive Computing: The Next Chapter on the Internet http://www.phptr.com/articles/article.asp?p=165227&rl=1

[31] An Introduction to MultiAgent Systems. http://www.csc.liv.ac.uk/~mjw/pubs/imas/

[32] Fault tolerance techniques for distributed systems. http://www.128.ibm.com/developerworks/rational/library/114.html

[33] A. M. Memon. (2004). Developing Testing Techniques for Event-driven Pervasive Computing Applications. Workshop on Building Software for Pervasive Computing.

[34] Sajal K. Das (Editor-in-Chief), Marco Conti (Associate Editor-in-Chief), Behrooz Shirazi (Editor-in-Chief, Special Issues). Pervasive and Mobile Computing.

http://www.elsevier.com/wps/find/journaldescription.cws_home/704220/description#description

[35] Research Activities of the Complex Systems Modelling Laboratory. (2005) Complex Systems Modeling and Cognition Lab Eurocontrol and EPHE Joint Research Lab. 4th EUROCONTROL Innovative Research Workshop & Exhibition - Workshop Proceedings

BIOGRAPHICAL INFORMATION

Shreyas K Shetty received his Bachelor's degree in Engineering at Bangalore Institute of Technology, India in 2003. He worked as a software engineer at Oracle India Pvt Ltd before starting his Masters of Science in Computer Science and Engineering at the University of Texas at Arlington in August 2005. His research interests include pervasive and wireless networks and embedded technologies in mobile devices.