HIEARCHICAL OVERLAY FOR SERVICE COMPOSITION IN PERVASIVE ENVIRONMENTS

by

APARNA KAILAS

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

MAY 2007

ACKNOWLEDGEMENTS

I sincerely thank my thesis supervisor Dr. Mohan Kumar for his guidance, encouragement and support throughout the course of this thesis work. I would also like to thank him for providing me the opportunity to be a part of the Pervasive Information Community Organization (PICO) research group. I thank the thesis defense committee members Dr. Ramesh Yerraballi and Mr. Mike O'Dell for their valuable inputs.

I am grateful to my husband, Venu and my brother, Deepak for their encouragement and support to be all that I can be and to my mother, who always believed in me, even when I didn't believe myself.

February 13, 2007

ABSTRACT

HIEARCHICAL OVERLAY FOR SERVICE COMPOSITION

IN PERVASIVE ENVIRONMENTS

Publication No. _____

Aparna Kailas, MS

The University of Texas at Arlington, 2007

Supervising Professor: Dr. Mohan Kumar

The objective of pervasive computing is to allow users to perform their tasks in a transparent way regardless of device features. Resources on devices should be exploited to provide services in order to perform user tasks. When there is no exact match for the user task in the environment, the capabilities of available devices should be combined to perform the user task. Seamless Service Composition (SeSCo) abstracts device capabilities as services and leverages existing work on graph algorithms to perform service composition. A lightweight framework, PerSON (Service Overlay Network for

Pervasive Environments) was developed to provide a service overlay network for the Pervasive Information Community Organization (PICO) middleware.

In this thesis SeSCo service composition mechanism is implemented on top of PerSON. The ability of PerSON to construct an ad hoc service overlay network is exploited to create a hierarchical service overlay using message exchanges that facilitate the hierarchical order. This thesis develops a mechanism for propagation of services in the hierarchy. Services at each level are aggregated with services at lower levels of the hierarchy and requests are resolved by constructing a path in the service aggregation. Demonstration applications that perform string encryption and decryption are implemented using the PICO framework. The application problem comprises a matrix multiplication operation, and the encryption and decryption of matrix data.

SeSCo is extended to include device resources in the service path, so that service composition will find the set of services hosted by the least resource constrained devices among all feasible candidates. A combination of device resources is used to define a suitable length for all services hosted by the device in the service aggregation. A proof of concept example scenario that uses battery energy has been implemented to demonstrate resource aware service composition.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

Mark Weiser introduced pervasive computing as an environment saturated with computing and communication ability and yet so integrated with users that it disappears into the background [16]. A user in a pervasive environment should be able to perform tasks without worrying about the computing involved.

1.1 Motivation

With the proliferation and multiplicity of devices that perform similar tasks, there is a need to dynamically bind one of several or a combination of devices offering the same service in the environment as opposed to specifying a host that can provide a service. A device should be able to search for and combine capabilities in the environment effectively.

Service Oriented Computing (SOC) which is commonly used in the web services domain defines device capabilities as services [1]. SOC can be applied to the pervasive computing environment [3] to provide support for seamlessly combining different device capabilities. Device features being exported as services; they can be advertised, discovered and utilized in a uniform way. The user tasks will require a number of resources possibly spread over the network. With service oriented architectures, user requirements can be fulfilled by locating services that match the need.

Some service discovery mechanisms, for example the discovery mechanism used in Bluetooth [5, 14, 18] try to find an exact match for the required service, but there might not always be an exact match in the environment. Pervasive environments are subject to short switched on periods and device mobility. The availability of services is therefore dynamic. In such cases, user queries should not rely on pre-existent services but rather use services that can be combined dynamically using existing services in the environment. Service Composition is the process of creating customized complex services from existing basic services [1].

In the current environment, collaboration of devices requires significant human intervention to configure devices. The service composition mechanism should be able to minimize the user role as much as possible. The users express their requirements to the service composition system, and from there on the service composition mechanism should take care of identifying resources and configuring their interoperability.

Current service composition efforts use a task graph oriented approach [8, 9, 13]. A task graph is intuitive and can leverage existing work in graph algorithms. The service request is given in the form of a graph with the nodes representing the services required to fulfill the request and the links between the nodes representing application data flow. The service composition mechanisms however attempt to discover exact matches to the requested services (nodes in the graph) [8, 13]. There is no effort at interleaving different device services to satisfy the service request.

As devices get smaller and more mobile, their computing power, storage capacity and amount of energy that can be stored are constrained. The current task graph oriented

approaches use alternatives merely to handle service unavailability [8, 9, 13]. When the environment is well conditioned, service composition should have the ability to choose devices based on criteria like battery power and bandwidth, to avoid straining the resources of already constrained devices. When dynamic resource availability is considered, choosing the least resource constrained device implies that the required service will be available longer, reducing the chances of failure.

1.2 Objective

Seamless Service Composition (SeSCo) addresses the problem of service composition by adopting a service oriented computing approach [3]. SeSCo abstracts device features as services and uses graph algorithms to compose user requests. This thesis seeks to implement SeSCo on the middleware framework, Pervasive Information Community Organization (PICO) [7]. SeSCo is implemented over an existing implementation of PICO over a lightweight framework, Service Overlay Network for Pervasive Environments (PerSON) [14]. PerSON has been developed to provide a service overlay network for PICO. This thesis extends SeSCo to handle resource availability changes in devices, thus enabling service composition choices that seek to conserve device resources.

1.3 Contributions

- Implementation of Seamless service composition (SeSCo)
  - Implementation of Hierarchical Service Overlay using message exchanges between devices - an improvised version of the earlier proposed Latch Protocol.
  - Developed a mechanism for propagation of services up the device hierarchy.

  - Implementation of service aggregation at each device of level 1 and above.
  - Implementation of query resolution with the request being propagated up the hierarchy in case of failure at local device.
- Integration of Seamless Service composition over PerSON (Service Overlay Network for Pervasive Environments).
- A prototype for a string encryption and decryption application is implemented using PICO services for demonstration of SeSCo.
- Resource aware service composition is designed and developed as an extension to Seamless Service Composition. A proof of concept application that uses battery strength is used to demonstrate resource aware service composition

The efficiency obtained by merging messages with route information in PerSON is exploited here. The route information allows the shortest path to the service to be determined, along with alternate routes in case of failure. The hierarchical service overlay network is joined by every incoming device. Proxies to handle service discovery and composition are determined dynamically by the latch protocol. Proxies allow resource constrained devices to offload service discovery and composition work to less resource constrained devices. Service composition consists of choosing suitable services that executed one after the other to accomplish the user task. While choosing these services, device resources are considered so that services on the least resource constrained devices are chosen. Resource aware service composition proposes regular messages that let a device have a resource view of those devices for which it is the

proxy. The resource view enables the device make optimal service choices while composing for a request.

1.4 Thesis Organization

The challenges in pervasive environment, service oriented computing, service composition, details of some of the existing frameworks and their limitations are discussed in Chapter 2. The architecture, service discovery and connection process and message formats of the PerSON framework are described in Chapter 3. The overview of PICO middleware and Seamless Service Composition (SeSCo) [3] are discussed in Chapter 4. The implementation details of SeSCo and its enhancements, PerSON improvements and additions to support SeSCo, the proposed resource aware service composition and the demonstration application are discussed in Chapter 5. The thesis is concluded in Chapter 6, along with some ideas for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

Pervasive computing envisions technology so integrated with users that it disappears into the background [16]. The possibility of technology disappearing in the background has increased with the proliferation of a variety of devices along with high availability of communication infrastructure [12]. To accomplish the objective of disappearing technology, heterogeneous devices should be able to communicate with each other efficiently with minimal user interaction. The practical realization of pervasive computing is hindered by a number of challenges [2, 12].

2.1 Challenges in pervasive environments

2.1.1 Heterogeneity

There may be different types of devices, with the resources varying due to limitations of weight and size. While static devices might have infinite resources say in terms of energy, many battery-operated resources will have to conserve energy. The necessity to conserve energy will affect device capabilities, restricting them to those operations that use as less energy as possible.

2.1.2 Network Quality

With the advent of new networking technologies like Wireless LAN, Bluetooth, Infrared etc., there is no unique way that devices connect to each other. The presence of different kinds of networks results in variations in network quality. Network quality or

available bandwidth should be considered while choosing services for fulfilling the service request.

2.1.3 Device mobility

Devices in a pervasive environment have varying mobility rate. The resulting environment will be dynamic with the services availability varying as devices move in and out of the environment.

2.1.4 Reliability

Many of the devices in the pervasive environment will be battery operated rather than being connected to a static power supply. Thus the devices will have short switched on period, and are vulnerable to system overloading. So service composition should consider power availability when deciding which services to use in fulfilling the service request.

Heterogeneous devices will be required to interact seamlessly, despite wide differences in hardware and software capabilities [2, 12]. A transparent mechanism to discover and use device capabilities is necessary. Service oriented computing, which defines device capabilities as services can be applied to pervasive computing, thus providing support to identify and use device capabilities as services to handle a user task [3].

2.2 Service Oriented Computing

The basic premise of Service Oriented Computing is to use services as the building blocks to facilitate the development of applications [1].

In the web domain, services are offered by service providers; service providers implement the services, supply the service descriptions and provide technical and

business support to users of the services. The services provide a distributed computing scenario that facilitates intra and cross enterprise applications. Service descriptions advertise service capabilities, behavior and QOS parameters like cost and performance metrics (for instance, response time), which are used by service aggregators in selection and composition of services.

The service composition layer uses individual services to compose more composite services. The services are then published and can be used as to build much more composite services. Some composite services may provide services to the composition function by [15]:

- Controlling execution of the services and the data flow among them.
- Monitoring events of the component services and publish higher level composite events
- Ensuring that the composite service has inputs that match those of its component services and works according to a defined set of business rules.
- Deriving QOS including cost.

The SOA includes a management layer. The management layer manages deployment of services and applications, provides statistics that will help evaluate the service, delivers notifications when a particular decision point is reached or a certain activity is completed and manages open service marketplaces.

Open service marketplaces while functioning as much of the vertical domains do, allow business to be conducted electronically and aggregate distributed service demand/supply. These marketplaces advertise products and services, provide business

transaction and negotiation, related financial services, service certification in terms of QOS, rate services based on parameters like turnaround time and negotiate and enforce Service level agreements.

An URL identifies web services which use open Internet standards. Interactions are SOAP calls using XML data content. Interface descriptions are typically written using WSDL (Web Services Definition Language). Directory services that contain these descriptions are defined using UDDI (Universal Description Discovery and Integration) standard. The directory services let enterprises discover required services and their details. Service composition uses these web services to compose other services. Typically BPEL4WS (Business process execution language for web services) is used to describe these composite services [1].

2.3 Service Composition

When a required service is not available in the environment, the available services in the environment should be used to compose the required service. Service Composition creates new services using existing services [1]. Service composition mechanisms have been classified as [11]:

1. *Proactive Composition and Reactive Composition*

Pro-active composition refers to offline composition of available services. The compound services are pre-composed before the user makes a request. The services used in such a context are usually stable and run on a resource-rich platform. Proactive composition is used on a large scale on the internet.

Reactive refers to dynamic composition, created only when a request comes in from a user. Reactive composition requires some sort of entity to take the responsibility of composing the request. The use of a central entity is particularly useful when the entities are typically unstable.

2. Mandatory-composite services and Optional-composite services

Mandatory services refer to those subcomponents that are required for execution of the composite service, and a correct result can be obtained only if all the subcomponents execute successfully. Example is a service that calculates averages of stock values.

Optional-composite services do not require participation of all the subcomponents. Examples are connectors between service components that can be used to optimize data transfer through a variable bandwidth line.

Existing web service composition mechanisms compose services offered on the wired network infrastructure. These usually rely on a centralized server that discovers individual services, registers them and composes them into composite services when required. The web services domain is comparatively more stable with no constraints on resource requirements. These however cannot be directly used in a pervasive computing environment [5].

In pervasive computing, services need to be provided considering challenges of resource mobility, availability and the other factors listed above. Devices with a short switched-on period together with mobility, and ad-hoc connections make the network connection highly unreliable.

Some wireless service discovery schemes are limited by the range in which they can operate; for instance, Bluetooth has a range of 10m. These types of discovery mechanisms restrict devices to those in the vicinity of the device. As a result the search for possible matches that can be found for a request becomes restrictive. The devices in the geographical proximity can be used, although multiple hops away in network terms. With the proliferation and multiplicity of devices that perform similar tasks, we need to move from specifying a specific host to provide a service, to dynamically binding to one of the several devices offering the same service in the environment. Dynamic binding will also allow user queries to not rely on pre-existent services but rather use services that can be combined dynamically using existing services in the environment. Centralized mechanisms are popular in wired environments. The problem with centralized mechanisms is that when the centralized entity fails, the whole infrastructure fails. Along with mobility changes that change the topology, and dynamically changing resource availability, a centralized system cannot be adaptive to these failures. To handle the above challenges, distributed schemes are positioned to perform better than centralized schemes in such environments [3, 5, 8, 9].

2.4 Existing Service Composition mechanisms

2.4.1 Anamika[5]

Anamika proposes a semi-distributed architecture with reactive service composition. A broker capable of executing on any node is used in this scheme. The individual broker is selected based on various parameters like resource availability, proximity etc. The broker composes the request and is responsible for handling the execution of the

composed request and the various error conditions that could arise during composition and execution.

The *service discovery* layer encompasses the protocol used to discover services in the environment. Service discovery is primarily peer-to-peer service discovery and service descriptions are cached. Each device has a service manager that lets local services register themselves and advertises these services to other nodes. Services are described using *DAML+OIL.* The advertisements from other devices are cached

The *service composition layer* is responsible for managing the discovery and integration of services into complex services. This architecture proposes two schemes for broken selection, described later. The broker arbitration and delegation module supports one of the two.

The *service execution layer* is responsible for the execution of the composed service. The layer takes care of optimizing the cost of executing a service. A Fault Recovery module is part of the service execution layer that is responsible for handling service unavailability and node failures.

Dynamic Broker Selection

When a request comes in for composition, a broker is chosen to carry out the composition and execution of the request. The selection is based on parameters like power of the platform, number of services in the vicinity, stability etc. In some cases, the originator itself might be the broker. The broker sends the request to its neighbors,

which then forward the request. When the broker cannot discover all the services, an error is returned.

During execution of the request, the broker sends back checkpoints to the client. The client maintains a cache of the result so far. If the broker fails, the client detects the failure using the absence of checkpoints. The client reconstructs the unexecuted part of the query into a new request and repeats the process again.

Distributed Brokering Technique

Another technique used to improve the dynamic brokering scheme is to distribute the brokering to different entities. The module selects a broker for the initial set. The criterion is similar to that used for the dynamic brokering scheme. Here the emphasis is the vicinity of services required immediately. Hence, if the requester finds service in its vicinity, the broker does limited composition and execution of the services available. Once that is done, the broker returns the partial set to the requester and selects another broker to do a subset of the remaining composition. The current broker then returns the final composition result to the client. The checkpoints in this scheme are propagated from different brokers. Each broker keeps track of the source and the source knows which broker is currently handling the request. If the client does not get any results for a reasonable amount of time, a composition request is issued with the remaining composition.

2.4.2 Task Graph Based Application Framework for Mobile Ad Hoc Networks[8]

This is a *Task Graph* (TG) based framework for application development and execution in Mobile Ad-hoc Networks (MANETs). The task is decoupled from the host allowing an application to use any of the multiple hosts available in the network. Hierarchical composition is supported, with devices grouped together and treated as a single unit, enabling new composite services to be offered.

A network of mobile devices is described as a graph. Each device is associated with a set of attributes – static as in resolution of a digital camera or dynamic as in the location of the camera. A node is a representation of a device or a collection of devices that has certain attributes and offers a service. An edge is an association between two nodes that satisfies attributes for performing a task. A complex task is divided into sub-tasks. An atomic task is an indivisible task.

The task graph is described as (V_T, E_T), where V_T is the set of nodes required to fulfill the task and E_T is the set of edges that describe the execution flow. An instantiation of the task graph is a path in the device graph. Each node in the task graph is the instance of the node required to fulfill the task. When the instance of the node becomes unavailable, node re-instantiation chooses another suitable device.

This framework assumes the existence of a controller node that takes care of instantiating and re-instantiating services. The application instantiation process consists of:

Discovery: The controller broadcasts a TASK-QUERY with the required nodes. A device that is receiving the query for the first time, broadcasts the query. A device that can satisfy any of the nodes returns a TASK-QUERY-ACK to the controller.

Selection: The controller selects one among the possible instances. The criteria used here is to minimize the average path length. A TASK-INSTANTIATE message goes from the controller to the selected nodes.

Connection: Each selected node is notified of its parent and child. A request-reply message passes between all parents and their children. When all the nodes have been contacted, the controller is informed.

Mobility management of devices is managed by periodic exchange between the selected nodes and the controller. TASK-CONTROLLER-HELLO is sent from the controller to all selected nodes and possible nodes till the selection is complete. Each node sends TASK-DEVICE-HELLO to the controller. Each node also sends a TASK-NEIGHBOR-HELLO to its parent and child nodes. Any device receiving the hello packets has to acknowledge to the sender. If the sender does not receive a reply within a designated T seconds, then the recipient is deemed unreachable.

The type of node that is unavailable determines the action to take. If the node is:

Controller (Case B): Sender assumes that the application has been aborted and goes to an idle state

Node instance (Case C): Controller attempts to re-instantiate the instance and restarts the discovery phase.

Neighbor node instance (Case A): The controller is informed so that the node can be re-instantiated if needed.

Possible node instance: Controller drops the node from the list of possible instances.

The following metrics are used to describe the task graph instantiation:

Average Dilation is the average length of the paths over all edges of the task graph. Lower dilation implies lesser hops.

Time to Instantiate a task graph is the time from the start of the discovery phase to the end of the selection phase.

Time to Re-instantiate is the time taken to rediscover and re-select a new device when an existing instance becomes unreachable.

Effective Throughput is the ratio of average number of application data units received at the destinations and the number of application data units actually sent.

2.4.3 A distributed scheme for autonomous service composition[9]

This proposes an autonomous distributed service path selection scheme for building directed service graphs for composition. A distributed media overlay network based on a peer-to-peer network of physical overlay nodes (ONodes) organized as a Media Overlay Network (MONet) is used. Each of these physical overlay nodes consists of one or more media ports.

Description Scheme

The description scheme describes the hierarchical relationship between elements of a media endpoint by classifying an element into four object classes:

- *User* - user level, for example preferred language
- *App* - application level, for example available codecs
- *Dev* - device level, for example video resolution
- *Ifc* - network interface level, for example bit rates.

A media description may consist of many instances of each class. A media description with one object of each class e.g., {application, device, network interface} is called irresolvable. Media descriptions have to be broken down into irresolvable forms in the service path. An element may exist in more than one class but in general an element is classified according to the lowest class in which it is a constraint.

A MediaPort (MP) has a set of irresolvable endpoint descriptions to describe the input and another set to describe the output ports. As long as one mediaport can receive the output of another mediaport, they can be chained together in the service path. A node is complete if, during the execution of the service graph routing algorithm, there is a completed path from the source for all outputs. A node has to be complete before it can be considered for the service graph.

Transformations and media adaptation is required to eliminate mismatches between what the client can accept and the media available. As the service path is built, all

alternates are checked to eliminate these mismatches for each end point. The process is repeated through several iterations until you get a complete service path for all sub-flows or no service path is found. All possible matches are ordered based on the number of mismatches, to determine the optimal path.

Three heuristics are used to construct these service graphs:

1. Compatible is used to determine if an input port can receive a media flow in its current state.

2. Adapt generates description of the result of applying the media port on the media flow.

3. Useful determines desirability of an operation. A positive result is returned if the differences post-operation is less than that started with.

Using these, the service path construction algorithm determines if a media port should be in the service path or not.

Service Discovery

Service discovery is peer-to-peer based. The service discovery process consists of finding services at each successive hop in the routing algorithm. Three permutations were tried: global directory service, limited scope broadcast, and directed path search.

Global directory

In the global directory media discovery approach, a centralized directory of MPs is assumed, for instance based on UDDI.

Scope-limited flooding

Scope limited flooding is a peer-to-peer search technique by which a peer floods a scope limited search query to all of its neighbors on the MONet. The query is then propagated to its neighbors. The search query has an associated time to live (TTL) that determines if the search query should be propagated.

Path directed search

Media routing works within the constraint of end-to-end delay, especially real time application. So only those paths that bring the media stream closer to the media client are considered.

Service Graph Routing

The service graph routing is a state based depth first search. Nodes once selected, select their successors based on heuristics that consider if a node has all the required inputs (complete), if the successor produces any useful transformation towards the end media content. Each path that results from choosing an initial node is a candidate path, irrespective of whether the path results in a complete media port or not. The final path is chosen based on lowest latency.

2.4.4 Spidernet: An Integrated Peer-to-Peer Service Composition Framework[13]

The Spidernet system is a quality aware service composition middleware deployed in wide-area networks. Spidernet creates a P2P service overlay, where each peer provides a number of service components. Each service is described in terms of the quality

parameters of its input and output. The service also has associated quality parameters such as delay that are used to decide service component matches.

The service request is given in the form of a function graph with the nodes representing the services required to fulfill the request and the links between the nodes representing application data flow. The attributes on the nodes indicate the required QoS. The function graph also indicates dependencies and commutation links. A commutation link indicates that the composition order can be exchanged.

Service Discovery is decentralized and is based on the Pastry distributed hash table (DHT) system. The function name is mapped to a key in the hash table and the service component's static meta-data (like location, input QoS, output QoS) are stored against the key. All functionally duplicate service components share the same key, and the DHT system stores the meta-data list of service components on the same DHT assigned peer.

When a peer wants to discover a list of service components for a given function name, the same secure hash function is used on the name and a query message is sent. The query is then routed to the assigned peer by the DHT system. The peer then returns the list to the requesting peer.

For a service composition request, the application sender invokes the Bounded composition probing (BCP) protocol. The protocol consists of the source first generating a probe, which includes the function graph and the user's QoS/resource requirements. A probing budget, that defines how many probes can be used for a composition, and a probing quota, which defines the number of duplicated service

components to probe for a required service, dictates how extensive the probe is going to be. Each peer processes a probe independently using local information like QoS and resource availability and allocates required resources temporarily. The allocation of resources is soft and expires after a certain timeout. The peer decides the next hop nodes based on the function graph. Depending on the probing budget and quota, the most suitable candidates for the next hop function are selected, the probe is updated with the peer's resources and QOS states and sent out. When all the nodes in the function graph are matched, the probing messages are sent to the source. The source then initiates a service session with the selected peers.

Spidernet adopts a proactive approach to failure recovery. Backup graphs are maintained for the service session, and in case of failure, Spidernet switches to one of the backup service graphs. These backup graphs are maintained by sending periodic probes to maintain liveness and resource/QoS conditions.

2.5 Observations on existing frameworks

Service composition done a priori is suitable for the web domain. Due to the dynamic nature of the environment, services might not be available all the time. Available services in the environment should be used to dynamically construct complex services. A centralized service composition mechanism will not be effective as the topology is always changing, and the network connectivity is not constant.

Some service composition approaches discussed in Section 2.4 use a task graph based approach. A task graph based approach can leverage existing work in graph algorithms.

The schemes discussed in 2.4 use a request template and try to find an exact match for each of the nodes in the template. An absence of matches for any of the nodes is flagged as an error. There is no effort at building a composite service for each of the nodes using available services.

The scheme described in section 2.4.2 is controller-based and hence prone to problems of central node of failure. There are also a lot of liveness messages sent between the nodes, which will reduce available network bandwidth. In low bandwidth conditions there is the possibility that data passing between nodes will be affected due to these periodic messages. These could potentially be regarded as node failures and lead to a multitude of re-instantiations.

In Anamika, the broker selection process uses a lot of resources. If the requester is a mobile phone with a limited battery life, thrusting on it the work of broker arbitration might run the battery down. The caching of results is also not effective in an environment where devices are memory constrained, for instance requiring a mobile phone to cache results while parsing a word document into a printable form. The selection of the broker considers resource availability. The scheme however deals with ad hoc networks and does not consider wired infrastructure which might have more stable devices.

In the scheme described in section 2.4.3 each node discovers the next node in each of the paths. All these paths are considered as candidates, and then the final path is selected. Whichever node chooses the final path, whether the requesting node or some other device, might be resource-constrained and unable to save and consequently

choose one of these paths. There is also no mechanism to narrow down the space of possible paths based on any criteria.

Spidernet broadcasts the request to the device peers, each of which composes the request independently. Such a scheme wastes resources, particularly in the worst case when all the paths returned from the peers are the same. The type of devices and their resource availability is also not considered in this scheme. A cell phone might not have the resources to handle service composition.

Any scheme in the pervasive computing environment will have to handle device mobility. Some of the schemes use periodic messages/checkpoints to handle mobility. While periodic messages are guaranteed to detect device unavailability, the scheme should also ensure that the periodic messages would not use up network bandwidth in a low bandwidth condition.

CHAPTER 3

ARCHITECTURE OF PerSON

This chapter discusses the details of the Service Overlay Network for Pervasive Environments (PerSON) [14] framework. The PerSON stack is discussed in 3.1 and the physical network connections in 3.2. The overlay network in PerSON is discussed in 3.3 and the service connections created as a result of the overlay network are discussed in 3.4 and the message format is discussed in 3.5.

The PerSON framework creates a service overlay network, among devices with different computing capabilities. PerSON uses the ability to connect to different types of networks, for instance a laptop able to connect to a wireless LAN as well as to a Bluetooth device. The service discovery process can be used without considering how the devices connect to each other.

The service overlay of PerSON is shown in Figure 3.1. As shown, the laptop, cell phone and PDA are connected through different physical networks. The laptop and cell phone connect through Bluetooth and the laptop and PDA through wireless LAN. In PerSON, the service overlay network facilitates the cell phone to access a service that is available on the PDA. The laptop acts as the conduit for any exchanges between the cell phone and the PDA. Service discovery and composition are abstracted by the PerSON layer.

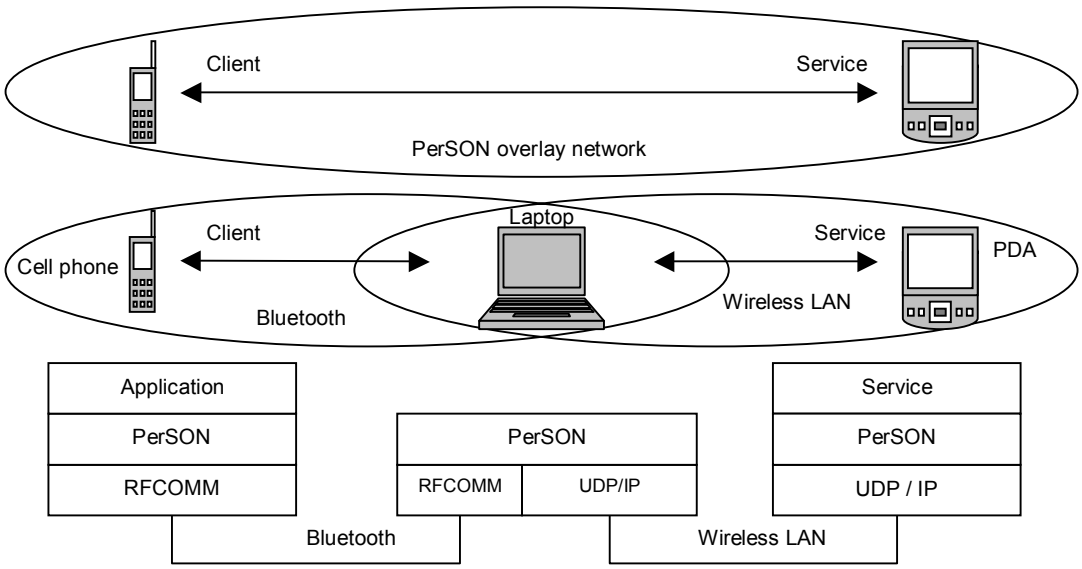

Figure 3.1 PerSON architecture[14]

3.1 PerSON stack

Figure 3.2 shows the different layers of the PerSON stack.

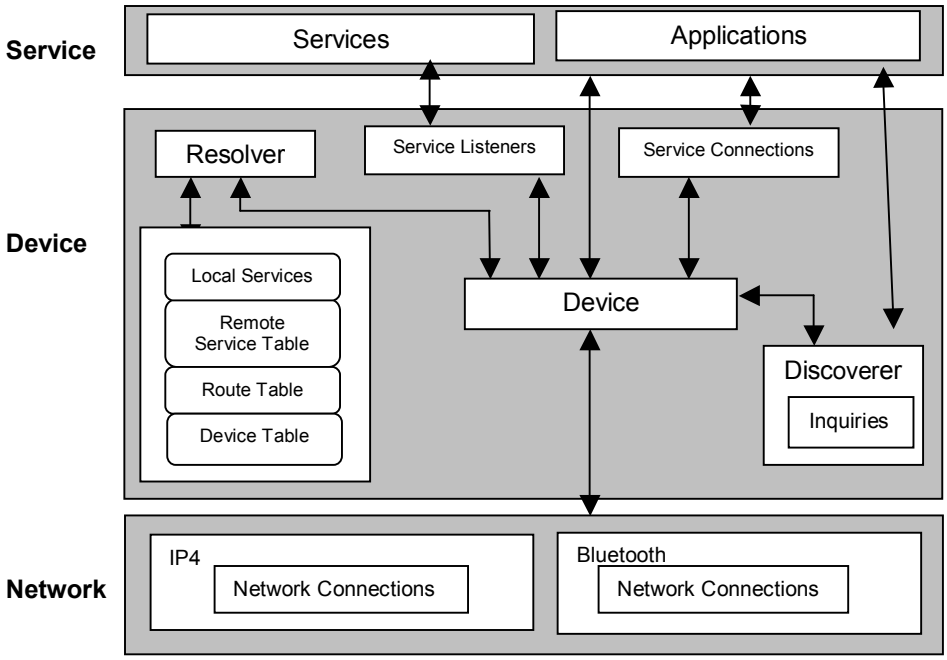

Figure 3.2 PerSON stack

The PerSON stack is implemented by every device that is part of the service overlay. The stack includes the network, device and service layers.

3.1.1 Network layer

The networks that the device can connect to are implemented by this layer. Each device has a unique device identifier that is used to identify the device during service discovery, composition and execution. The unicast and broadcast capabilities of the network layer are used to exchange messages with the other devices.

3.1.2 Device layer

The device layer uses the network layer to communicate with other devices for service discovery, composition and execution. The device layer receives all messages and sends the messages onto the entity handling the message or on to the child devices connected to the device.

The device forwards messages to the resolver so that the device table can be updated with the physical addresses corresponding to the device id, and the time the sending device will still be active. The device chooses the network connection to use for sending outbound messages.

The services hosted by the device are registered. These are maintained in the local services table, with the unique service identifier used as the key. A new service connection is spawned by the device when a request for the service is received. The service connection is used for message and data exchanges between the service and the client.

The discoverer is used to find the required services. When the application requests the discoverer for a service, the discoverer sends out a query message. The query is a broadcast to all devices on all the networks to which this device is connected. The service request is a textual description, used for name matching against local services available on the device. The scope of the service discovery is restricted by the hop count specified by the device when the discoverer broadcasts the query.

The query message is processed by the resolver. The resolver matches the request with services that the device hosts. These services are only those that are currently accepting requests. A string compare is performed on the active service list and any match is sent as a reply messages to the client.

If the service discovery returns a match, the device sends the message to the client. The resolver forms the reply message with the SID of the service that satisfies the request, and sends the reply to the client. At the client side the message is first sent to the resolver, which updates the service information in the service table. The reply message is then sent to the discoverer corresponding to the request, which passes on the SID to the application that sent the request.

3.1.3 Service layer

The user defined services and applications are part of the service layer. The service layer uses the device layer functions. The discoverer is used to find services and service connections are used to connect to a service. Once a service connection is created successfully, the service layer uses the connection for further communication with the service.

3.2 Physical network connections

A device may have the ability to connect to different networks. For example, a laptop able to connect to the Bluetooth network as well as to the wireless LAN. The network is used to broadcast as well as unicast messages. IP and Bluetooth networks are supported by the current implementation of PerSON.

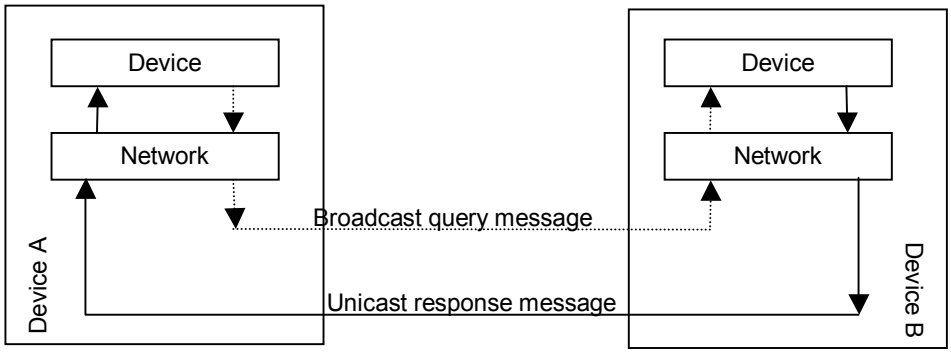

Figure 3.3 IP network connections

Suppose, device A and device B connect through the wireless LAN. Query messages are broadcast on a specified port on the local network. The IP address of the device is included in the query message. The IP address is stored in the device table by the resolver and is used when actually sending the message. The device B receiving the query message uses UDP to send a reply to the query.

3.3 Overlay network

The overlay network in PerSON is created to facilitate service provisioning. Route information is used to facilitate service discovery. The route information is part of every message exchange, so additional route discovery messages are not required.

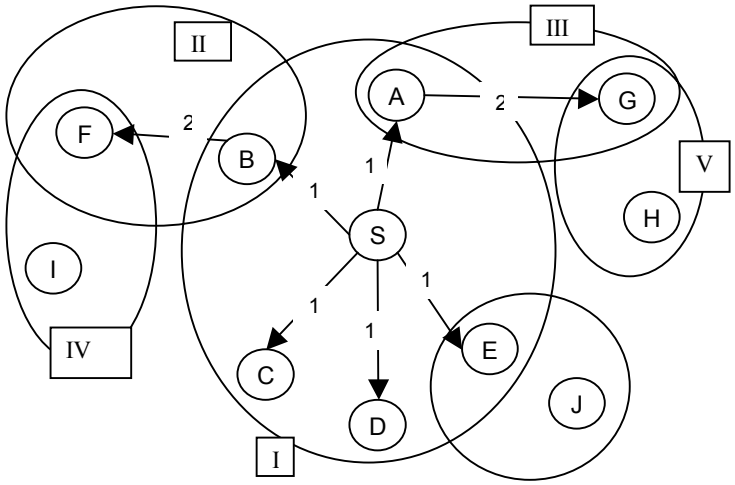

Figure 3.4 Query propagation in PerSON

PerSON uses flooding limited by scope to propagate the service discovery query. The absence of service directories, make PerSON insusceptible to a central point of failure. All devices process the query message, and those that act as a bridge between networks forward the queries in their local networks.

Consider the network in Figure 3.4 that comprises of five different networks. Devices S, A, B, C, D, E all belong to the same network 1. A and G belong to network 3 with A acting as a bridge between 1 and 3. B and F belong to network 2, with B as a bridge between 1 and 2. F and I are part of network 4, with F as bridge between networks 2 and 4. G and H are part of network 5, with G as bridge between networks 3 and 5. E and J are part of network 6; E does not act as bridge between the networks it belongs to. Device S sends a query for a specific service with a scope of 2. The query is broadcast in the local network of S. The devices A, B, C, D, and E all process the query and devices A and B forward the queries to the network they belong to, other than network

1. Device E is not a bridge so it does not forward the query. Devices G and F process the query but do not forward it, as the query is now out of scope.

Whenever a device sends out a query, its device id (DID) and physical network address are added to the message. Every device that receives the query message, updates its device table using the DID as the key. Every time a device needs to send a message to another device, the device is looked up in the device table and the physical address of the device is retrieved to send the message. Each record in the device table contains the DID, available time and the physical address of the neighboring device.

DID	Available Time	Physical Address
16 bytes	8 bytes	N bytes

Figure 3.5 Device table

In the network in Figure 3.4, if device G has a match to the query sent by S, G reverses the route in the query and knows that the reply has to be routed through device A. Each route record contains the list of all intermediate devices to use for each device, with its DID as the key.

DID	RLEN	List of DIDs of intermediate devices
16 bytes	1 byte	RLEN * 16 bytes

Figure 3.6 Route table

Device G stores the route for device S as A-S. When sending the reply, the resolver in device G specifies the whole route in the message. At the device layer, the DID of device S is used to retrieve the first hop, in this case device A and set as the initial destination in the message. Device A on receiving the message, checks if it is the final destination. As the final destination is device S, A forwards the message to device S.

If device G is unsuccessful in sending the message to the next destination device A, an error is returned to the device layer. All routes that include device A are cleared from the device table. The device tries to send the message using an alternative route. If there are no more routes, then the message is dropped.

In effect, device S and G are connected in the service overlay network. When device S receives the result message, S retrieves the DID of device G along with its physical address in the message and stores it in the device table. It saves the route information to G in the route table, reversing the route if G is in a different physical network or storing it as the final destination if G is part of the same physical network. Device S updates the remote service table with the result sent by G.

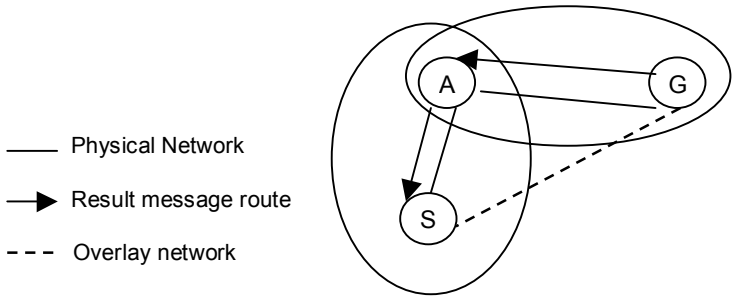

Figure 3.7 Result message route

In PerSON, each record in the service table contains the SID, service available time, the DID of the device which provides the service and a textual service description.

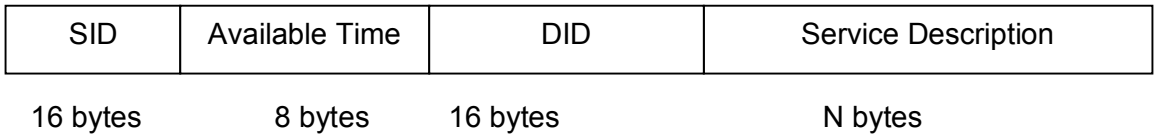

Figure 3.8 Service table

The application on device S that initiated the service query is sent the SID to call, by the discoverer. The application asks the device layer to connect to the SID. The device layer retrieves the DID of the device providing the service. The first available device on the route to the device is obtained from the route table. A request to connect message is sent to the first destination A, to be forwarded to device G.

3.4 Service connection

The creation of a service and the process of discovering and accessing the service are shown in Figure 3.9. The service on device G is specified by SID, textual description, and the available time. The service is registered with the device layer, indicating that the service is ready to receive any requests. The service discovery query messages and connection requests are handled by device layer.

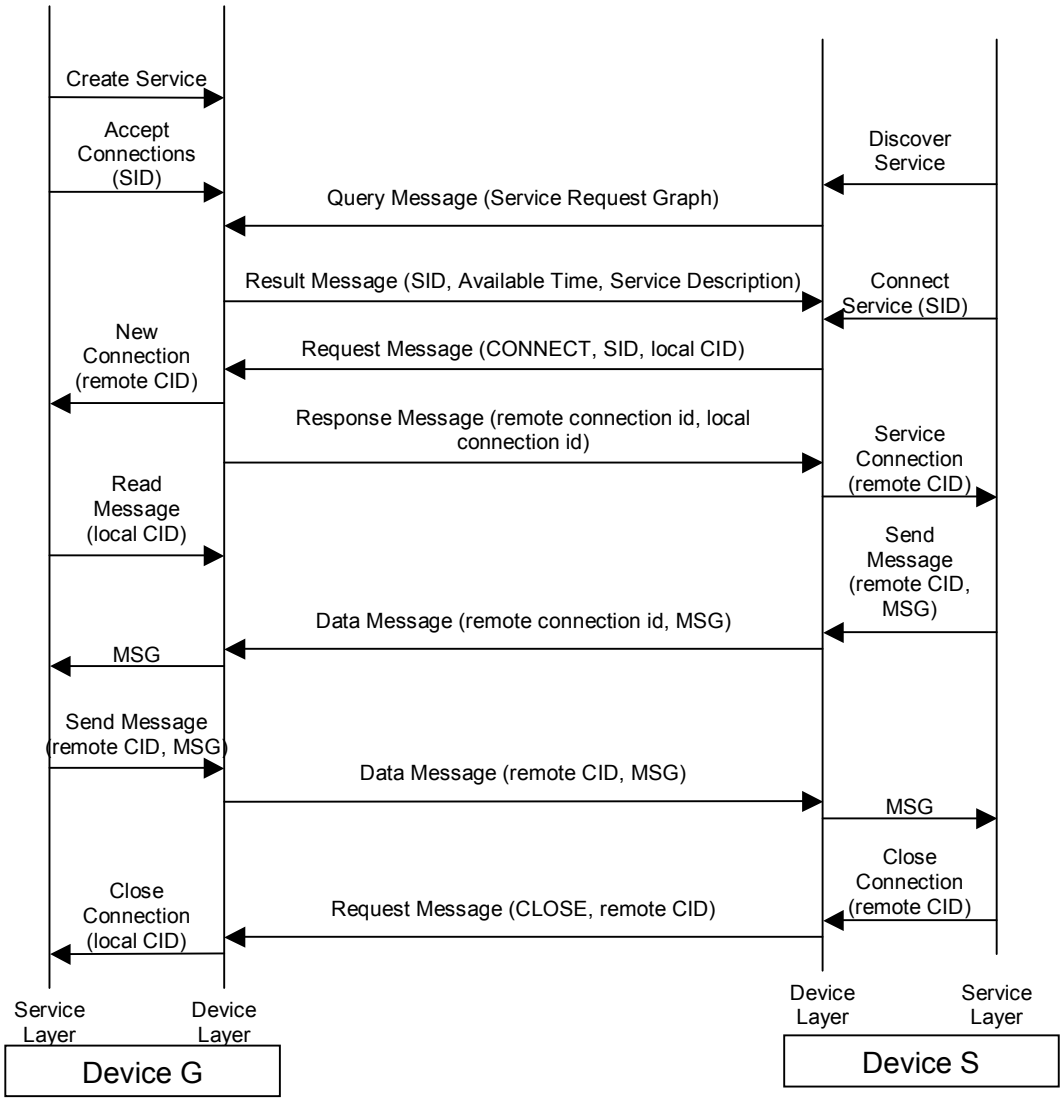

Figure 3.9 Service connection

The application requests the device layer to discover the service request described by a textual description of the service. The service request (query) is broadcast by the network layer. The query is propagated by every device that acts as a bridge between physical networks, limited by scope. When device G receives the query, G matches the query with the local services. If the service is accepting requests, a result message is

sent with description of the service, SID and time for which the service is available. The device layer on device S processes the message and informs the application. The application now asks the device layer to connect to the specified service. A connect request is sent by the device layer, with the local connection identifier used by the application. The device layer on device G forwards the connect request to the service, which creates a service connection and sends a response to device S. On getting the response, the device layer notifies the application, which then starts sending data.

If there is no service that satisfies the request, the device S waits for a timeout and resends the query. This process is repeated until device S receives a result.

3.5 Message format

The messages in PerSON are in the format shown in Table 3.10. The header field has +person+ and the footer –person-. The message type field indicates the type of message and is one byte long.

Table 3.1 PerSON message types

0	Query
1	Result
2	Request
3	Response
4	Data

The hop count field is one byte long and specifies number of hops that the message has passed through to the device. RLEN is the length of the route that the message has to take for the final destination. The route field specifies the DIDs of the devices, including source and destination. As each device forwards the message, it's DID is

added to the route field. The address specifies the address of the next device to send to. For an IPv4 network, the address is 6 bytes long. The first four bytes contain the IP address and the last two, the port number.

Table 3.2 PerSON message format

Header	8 bytes	+person+
Message Type	1 byte	Message type
Hop Count	1 byte	Current number of hops
RLEN	1 byte	Length of the route
Route	RLEN * 16 bytes	List of DID in the route
Address	N bytes	Physical Address of previous device
Available Time	8 bytes	Available time of the device
DLEN	2 bytes	Length of data
Data	DLEN bytes	Data
Footer	8 bytes	-person-

The available time is 8 bytes long and specifies the time for which the sending device is available. The available time is used to clear the device table when the time expires and there are no connections to the device. The DLEN field specifies length of the data and is 2 bytes long. The data field contains the message to be sent.

When a device forwards the query, the hop count and RLEN are incremented by 1 and the DID of the device is added to the route field. When forwarding from one network to another, the message is forwarded to all devices in the network from which the message did not originate.

The format of the different messages is as shown below:

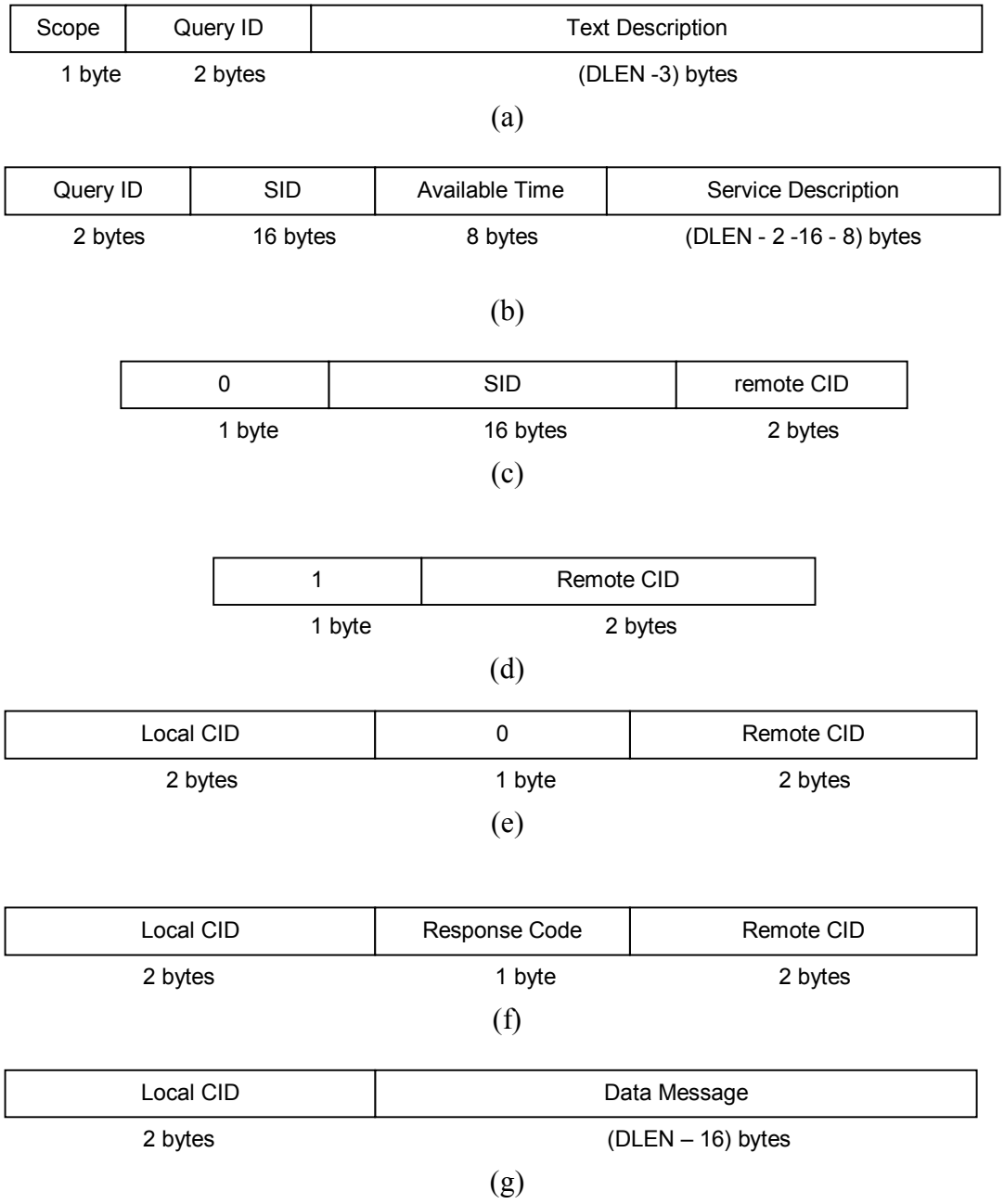

Figure 3.10 Message Formats : (a) Query, (b) Result, (c) Connect Request, (d) Close Request, (e) Success Response, (f) Error Response, (g) Data

The query message specifies the scope of the query, which specifies the number of hops that the query can be forwarded to. It also specifies the query id, and the request. For a

text description of the request, the query is name matched to the local and children services. The result message consists of the query id, followed by an array of service data. The service data includes the service id, available time and a text description of the service.

The request message specifies the service id and the remove connection id, when the client requests connection to a service. When the client wants to close the connection, the request message contains the service connection to close.

The response message indicates the local cid to send the message to and the remote cid that indicates the application that sent the message. The 2nd byte indicates if it is a success or an error message.

The data message indicates the service connection that the data has to be sent to, along with the data.

3.6 PerSON enhancements

When there is more than one match to a request, the client should be able to receive all the matches, so that there are alternatives in case of a failure. In the initial version of PerSON, the at most one feature is implemented. So the discoverer handles a single SID instead of a list of SIDs. The discoverer is now modified to return a list of SIDs to inquiry. The inquiry module has been modified to receive a list and send the list onto the client. Every client could now attempt a connection to the first SID in the list. In case of a failure, the client could iterate through the list of matches, until there is a successful connection to a service.

The current version of PerSON has no distinction between services offered by the device and queries initiated by it. This distinction is now added to improve aesthetics.

CHAPTER 4

SERVICE COMPOSITION

This chapter discusses Seamless Service Composition (SeSCo) [3], which is a task based composition scheme based on a hierarchical service overlay. SeSCo applies Service Oriented Computing [1, 15] that is popular in the web services domain. The basic premise of Service Oriented Computing is to use services as the building blocks to develop applications [1]. Pervasive Information Communities Organization (PICO) middleware is employed to use resources as services. An overview of PICO is described in section 4.1. The rest of the chapter describes the various aspects of the proposed Seamless Service Composition (SeSCo) mechanism [3].

4.1 Overview of Pervasive Information Community Organization (PICO)

PICO [7] is a middleware for pervasive computing. PICO abstracts device resources as services. PICO creates dynamic and static communities of these services. The basic constructs of PICO are:

a. *Devices* - abstract representation of a device. Represented by the tuple $C=\{H,F\}$, where H is the set of characteristics like memory, F is the functional feature of the device like printer features like B/w printing, color printing etc.

b. *Delegents* - software entities corresponding to services that represent device features as in printer features above. These are represented as *{M, R, S}* where

M: set of modules used to build the delegent

R: set of rules that control transition from one module to another.

S: set of services provided

In a streetlamp device, a camera on the streetlamp may include modules to capture images and detect events. The rules specify the transitions that take place on various events and the service it provides may be surveillance.

c. *Communities* - logical organization of a number of delegents to accomplish a complex task. Described as $P = \{U, G, E\}$ where

U: set of members in the community

G: set of goals that the community achieves

E: set of operational characteristics like cost, current load etc.

The PICO middleware has three different versions that can be used on devices. Resource constrained devices use a minimal version of the stack, the second version, though complete, can be used on mobile devices and the complete version is used on resource-rich infrastructure based services. Resource-rich devices can be used to host delegents on behalf of resource-constrained devices.

The PICO service layer includes:

a. an *advertisement manager* to send, collect and respond to advertisements,

b. a *service aggregator* to store received service advertisements and

c. a *composition manager* that computes composite services from the aggregated services.

In static environments, it is enough to be assigned a proxy prior to operation. But in a pervasive environment, as devices move in and out of the community, it is not possible to assign a proxy prior to device setup. The identification of a proxy has to be dynamic. SeSCo creates a hierarchical service overlay to handle the heterogeneity as well as mobility of devices in the environment. Device levels are used to classify the devices, to provide a mechanism for devices to identify suitable proxies and choose the best among those available.

4.2 Hierarchical Service Overlay

There are four different levels of classification for the devices. Table 4.1 shows the different device levels, the corresponding PICO middleware versions and the features.

Table 4.1 Device Classification Chart[3]

Level	PICO middleware version	Features	Examples
0	None	Features exported through delegents on Level-2 or Level-3 devices, No native personalization support	Sensors, legacy printers
1	Minimum	Community member, cannot act as proxy for any other device	Cell phone, mote sensors, smart printers
2	Complete	Community member, can act as proxy for any other device, Resource rich and possibly mobile	Laptops, PDAs
3	Complete	Community member, can act as proxy for any other device, Resource rich and possibly mobile	Servers, PCs

Devices with no additional facilities for software installation are classified as *L0*, for e.g. a legacy printer. There is usually a device of higher level that hosts the services of a *L0* device. *L1* devices can host one or more delegents. These devices are resource

constrained and cannot host proxies for other devices, for example cell phones and sensors. *L2* devices have higher resource availability than *L1*. They are mobile, can host a number of services and act as proxies for other devices, for instance laptops. *L3* devices are the same as *L2* devices but static, for instance PCs. These devices have minimal resource constraints and have a high degree of availability.

Latch Protocol

The Latch protocol defines the process of building the hierarchical overlay given the different device levels. The service layer of the PICO middleware sends out these latch messages. Every device on startup sends out a LATCH_HELLO message in all the networks it is connected to. The message includes the device level. The other devices inspect the LATCH_HELLO message and compare their device level to the level in the LATCH_HELLO message. If the device A is a higher level device, then it can act as a parent to the other device B. Device A sends a LATCH_INVITE to device B. Device B sets the parent and returns a LATCH_ACK. Device A adds the device B as a child. If the device A is a lower level device, it could latch on the other device B. Device A sends a LATCH_REQ to device B. Device B adds device A as a child and returns a LATCH_ACK. Device A on receiving a LATCH_ACK sets the parent as device A. Same level devices send a LATCH_SIBLING message to the other device. Both devices add the other as a sibling. The sequence of messages is as shown in Figure 4.1.

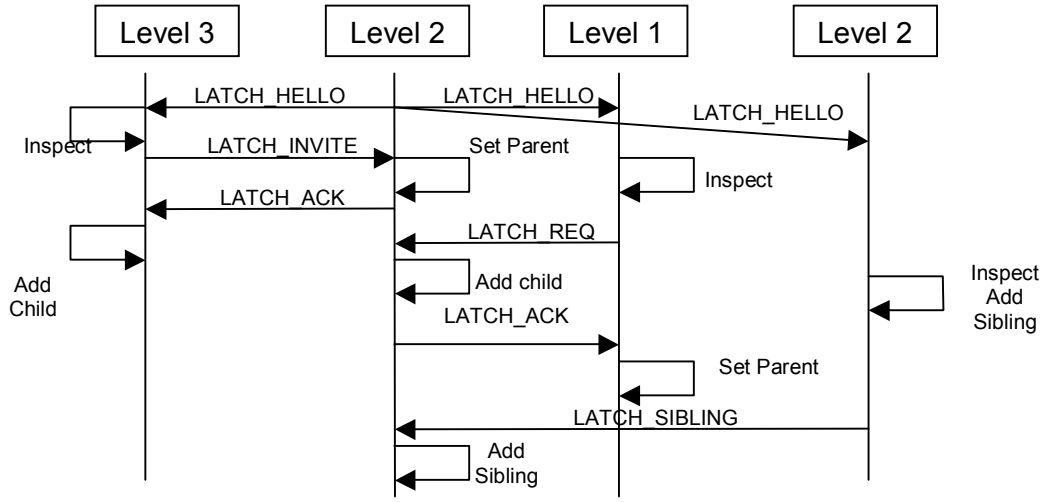

Figure 4.1 Latch Protocol Message Exchange

The liveness of a device in the hierarchy is checked with a periodic LATCH_HELO message. When new services are added, updates are sent up the hierarchy, using LATCH_ADD message and when services are no longer available, updates are sent using LATCH_REM message.

The result is a parent-child-sibling relationship as shown in Figure 4.2. The hierarchy culminates at a level 3 device, with all level 3 devices communicating as siblings.

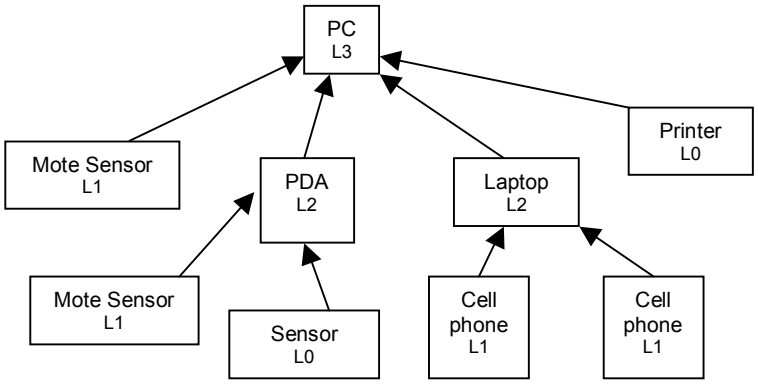

Figure 4.2 Sample Device Hierarchy

Each of the lower level devices uses a higher level parent as a proxy. The higher level parent being less resource constrained than its children helps them in advertisement, service discovery and composition. A device of level 0 has to be associated with a higher level device for its services to be advertised. Any device of level greater than 1 can act as a proxy for a level 0 device. For e.g. a PC with a connected printer can advertise a printing service. By maintaining the hierarchy, differences in device capabilities can be made transparent to applications that operate in the environment.

Service Representation

A service is described as a graph that transforms one form of input to another form. The service graph is a directed graph described as $Gs = \{Vs, Es\}$ where Vs is the set of vertices in the graph and Es is the set of nodes representing parameters in the graph. One can define vertex functions that specify name, location of service, cost of using the service, quality etc, and edge functions that specify type of parameters, size of parameters etc. As services come up on each device, the device sends the service graph to its parent. The parent consolidates its service graphs and those of its children into a consolidated service graph.

4.3 Service Aggregation

As services come up locally the device sends its service graphs to the parent. It also sends any services that the children have in turn passed to the device. The parent aggregates the service graphs based on inputs and outputs. Given two graphs

Gp1 = {*Vp1, Ep1*} where *Vp1* is set of nodes that represent modules that accept an input and transform it into another type and *Ep1* is set of edges representing the data flow from one node to the other.

Gp2 = {*Vp2, Ep2*} where *Vp2* is the set of nodes representing modules that accept an input and transform it into another type and *Ep2* is set of edges representing the data flow from one node to the other. Each of these graphs is transformed as follows:

a. parameters are represented as nodes

b. edges represent services that transform one node to another.

When the graphs have been transformed into the desired graph format, they are aggregated into a consolidated service graph.

Figure 4.3 shows a number of service graphs and the resulting aggregate graph. Service A converts *a* to *j* and *j* to *n*. Service B converts *d* to *k* and *k* to *c*. The other services C, D, E, F transform their input parameters similar to A and B. While adding a service to the consolidated graph the inputs and outputs of the individual service graphs are represented as nodes. An edge between two nodes represents the service. Consider service A, which converts from *a* to *n*. The consolidated graph has a node *a* and another node *n,* and the edge between these nodes is the service A.

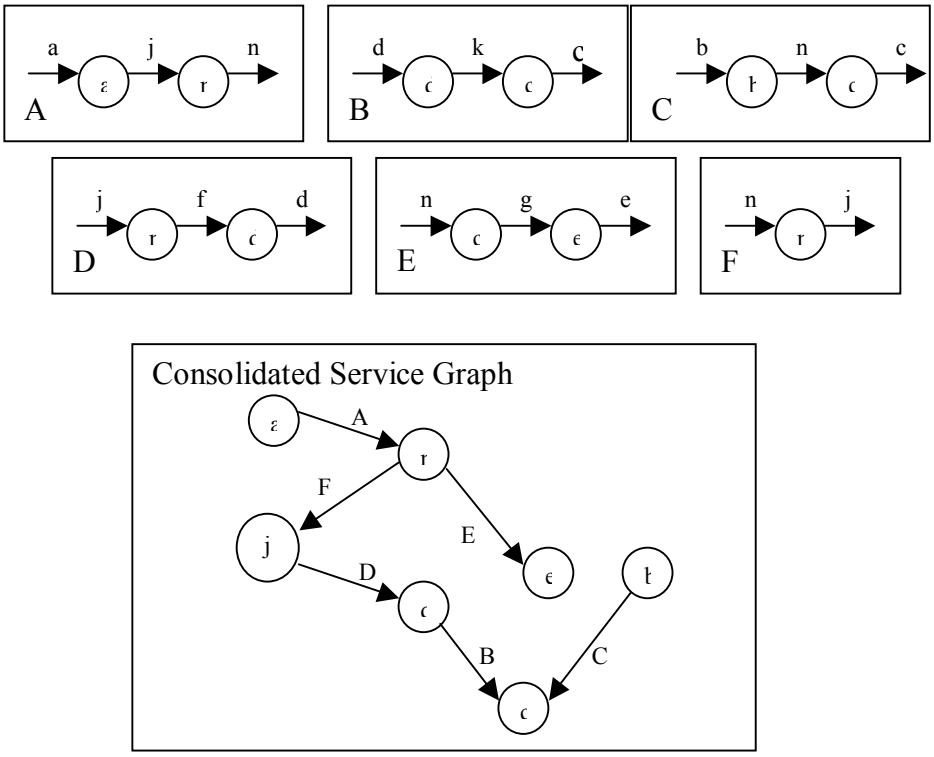

Figure 4.3 Service Aggregation

4.4 Service Composition

To describe service composition, the following are defined in the hierarchical service overlay:

Service Zone: This includes all services available through a device and its children.

Search Zone: This is essentially the service zone of a device. But if the device is L0 or L1, then the search zone is the same as the parent's search zone. Composition starts from the search zone of the device extending upwards to higher layers of the hierarchy. Figure 4.4 shows the service and search zones within the hierarchy in Figure 4.2.

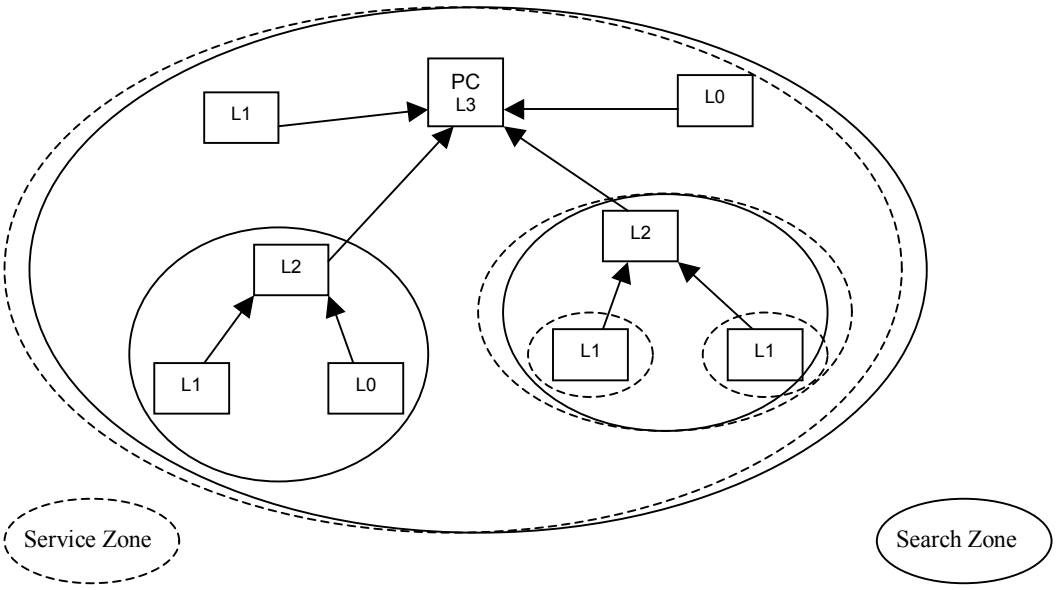

Figure 4.4 Search and Service zones in the hierarchical service overlay

In SeSCo, we assume that the user task or application requirement is given in the form of a request graph. The request graph is either part of the application or the application is capable of mapping the request onto a graph. The request graph denotes the individual services required to accomplish a task. The request graph is also a directed attributed graph similar to the service graph. So it is represented as $G_R = \{Vr, Er\}$ where Vr is the set of services that are needed to satisfy the request. The set of services is the sequence in which the services need to be executed to fulfill the request. Each node could also specify cost, quality parameters, device requirements etc. Er is the set of edges representing the lines of data communication between the services. Each edge could also specify quality parameters like required data rate for an audio stream.

Every device first tries to find an exact match, either with its local services or with the services that its children host. When there is an exact match, the matching service is used for the service in the request graph. When there is no exact match, a composite service needs to be constructed to achieve the same result as is required in the request graph. In the service graph, a node *n1* is found matching the input of the service in the request graph, and another node *n2* that produces the same output as the service. The shortest path from *n1* to *n2* is used to provide the same service as the service under consideration in the request graph. Figure 4.5 illustrates this using the consolidated service graph in Figure 4.3. To compose for the request, a path from *a* to *j* is first composed followed by the path from *j* to *c*.

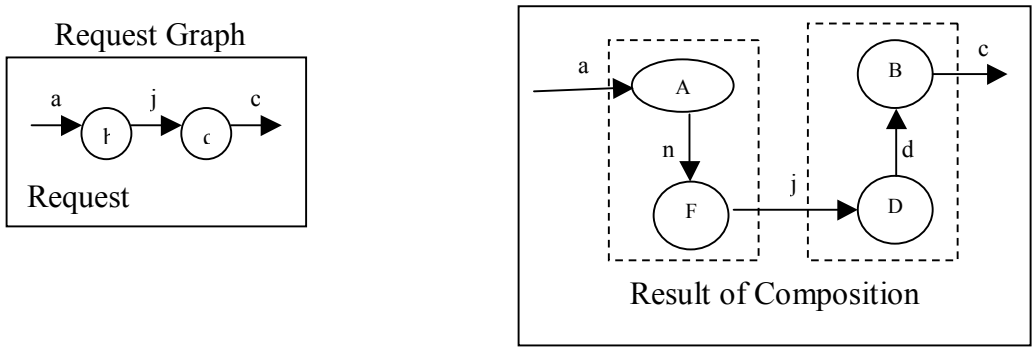

Figure 4.5 Service Composition for a Request

If one or more services cannot be composed in the local service zone, the search zone is expanded to include the parent's service zone. The search zone is expanded to the higher layers of the hierarchy as required. When a level 3 device is not able to compose the request either, the request is passed on to its siblings. The siblings are also level 3 devices and hence will know about all the services in their service zone.

Distance between the requesting application and those that provide the service be as less as possible. The hierarchical structure has a level 3 device at most two hops away. The hierarchical overlay supports finding a service as close as possible to the requesting application, as the service zone recursively expands from the requester's service zone to a level 3 device's service zone. The expansion of the service zone ensures that the service providers are as close as possible to the requester.

The node attribute function associates attributes of the service with each node. The vertex attributes of the request graph can be matched against those in the consolidated service graph and the composed service can be picked to ensure the best possible quality.

Updating Aggregation

The liveness of the devices in the hierarchy is checked with a periodic LATCH_HELLO message. A parent detects the absence of a child by the missing LATCH_HELLO messages. The services offered by the device are removed from the service graph. The missing services are propagated up the hierarchy by sending a LATCH_REM message to the parent. When a new device comes into the network, it sends its services to the parent. The new services are propagated up the hierarchy by sending a LATCH_ADD message.

CHAPTER 5

PROTOTYPE IMPLEMENTATION

PerSON (Service Overlay Network for Pervasive Environments) creates a service overlay network on top of different physical and logical networks, facilitating connections between clients and service providers [14]. The service discovery process in PerSON broadcasts the request to all devices in every network that the device is connected to. Any device that has an exact match to the name of the request sends a reply. For example, a service "UI" would not be a match for a request for "UIService". Support for service composition is provided by SeSCo developed on top of the existing implementation of PerSON. The reference implementation extends existing work of PerSON developed in Java. The J2SE version of PerSON is used on laptops and desktops. The J2ME version can be executed in mobile phones and PDAs. This chapter details the implementation of SeSCo, its enhancements and the PerSON additions to support SeSCo.

5.1 PICO over PerSON

PerSON provides the overlay network for PICO. The system architecture of PICO over PerSON is shown below in Figure 5.1.

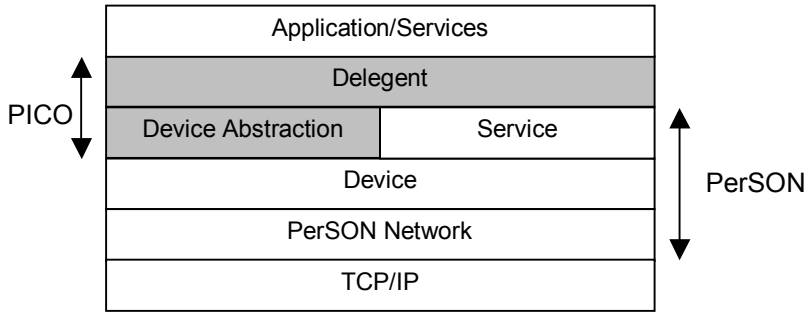

Figure 5.1 System Architecture

The device abstraction is implemented on top of PerSON's device layer and the delegents represent services, applications and device capabilities. Each device has an associated *pc.xml* that has a set of parameters for the device – the type of device, the device id, level of the device and the IP address. When the device abstraction is initialized, it starts the device and passes the device id to use in all communication and the IP address when initializing the IP network. The associated level of the device indicates whether it can act as a proxy for devices which are relatively more resource constrained in the hierarchical service overlay discussed in Chapter 4.

5.2 Modified PerSON Architecture

Figure 5.2 shows the modified PerSON architecture. The shaded portions show the modifications done on the original PerSON architecture. The route table is now sorted according to the path length. The router forwards messages either to the parent or its children depending on the destination of the message. The service aggregation and service composition modules are part of the service composition mechanism implemented on PerSON. The changes are described in detail in the later sections.

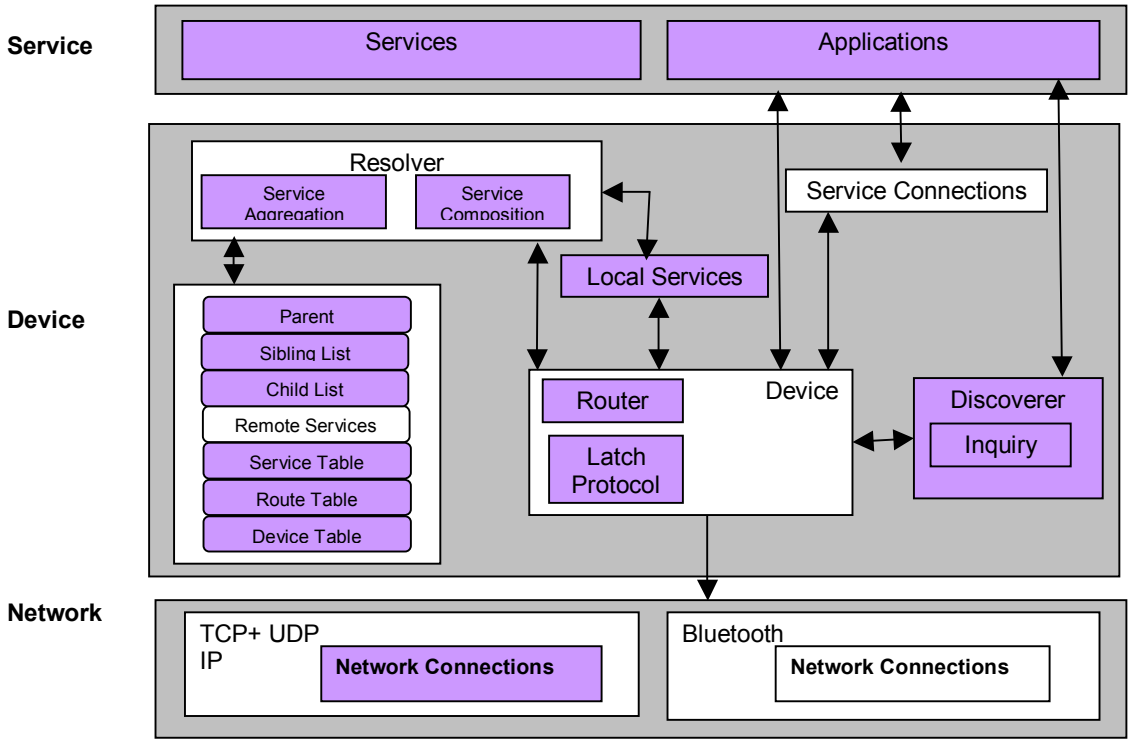

Figure 5.2 PerSON Architecture

5.3 Hierarchical Service Overlay

There are four different levels of classification for the devices. The device level assignment follows Table 4.1 in Chapter 4. The file *pc.xml* file specifies the level of the device, which is sent to the device by the device abstraction on start up. The latch protocol defines the message exchange between the devices in the community to form the hierarchical service overlay.

Latch Protocol

The device sends a LATCH_HELLO message on initialization. The general format is "LATCH_HELLO: <device Level>". The resolver on a receiving device retrieves the

device level from the message and compares its level to the level in the LATCH_HELLO message. The device sets the parent, children or siblings as the case may be, and sends appropriate latch responses to form a hierarchical overlay.

Consider a device A that initiates the latch protocol. A higher level device B receives the message and processes it. Device B can act as a parent to device A, as its level is higher than A. B sends a LATCH_INVITE message to device A. Device A sets the parent to the highest level device that it receives a LATCH_INVITE from. If the highest level device is B, device A sets the parent to B. Device A then sends LATCH_ACK to device B. Device B adds A as a child. Each of the lower level devices uses the higher level parent as a proxy. Hence the necessity to latch onto the most stable and hence the highest level device that it can find.

Consider a device C, which is a lower level device than A. On receiving the LATCH_HELLO message from A, device C checks if it already has a parent. If C does not have a parent, it sets A as its parent and sends a LATCH_REQ message to A. Device A adds C as a child and returns a LATCH_ACK to C. If device C has a parent, then it ignores the LATCH_HELLO message from A.

A device D at the same level as A sends a LATCH_SIBLING after adding A as its sibling. A too adds device D as its sibling. The formats of the LATCH_INVITE, LATCH_REQ and LATCH_ACK messages are similar to the LATCH_HELLO message.

The sending of the LATCH_ACK message signifies the end of the latch protocol. A Hierarchy Devices structure is used to maintain information about the device,

particularly whether the device is alive. The proposed version of SeSCo does not keep track of the device liveness. Here we define a device as alive if it is active. The receipient of the LATCH_ACK message sets the sending device as the parent/child and marks it alive. The alive field is used when the latch protocol has to be re-initiated due to a service failure, or a query message response failure.

SeSCo proposes sending of LATCH_HELLO messages at regular intervals to check liveness of the devices in the hierarchy. The implementation does not use this mechanism. The liveness of the device is important only when a query has to be processed.

The latch protocol is reinitiated to check liveness when there is no reply to the query within an application defined period of time. The device sets the parent and all the children as dead and re-initiates the latch protocol. If the parent is alive, the steps in the latch protocol are carried out and the device resends the query. If the device does not receive a reply from the parent within a period of time, the parent is set to null and latch protocol is reinitiated. The timed LATCH_HELLO message takes stock of the devices in the hierarchy. A child receiving a LATCH_HELLO from its parent will send a LATCH_REQ. The parent sets the child as alive. At the end of timeout, the parent checks its list of children and weeds out those that have not replied.

When the device is not active anymore, say because its battery ran out of power, a LATCH_REMOVE is sent to its parent and children before shutting down. At the receipt of a LATCH_REMOVE a parent device removes the device from its list of

children. At the child, the parent is set to null and the latch protocol initiated again. The format of the remove message is similar to the hello message.

PerSON Enhancements

In PerSON, the format of the device record now includes the device level of each device stored in the device table. The changed format is shown in Figure 5.3

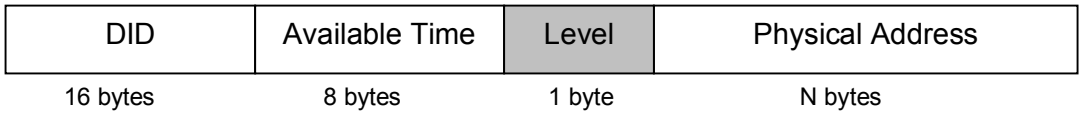

Figure 5.3 Device Record

The earlier version of PerSON broadcasts the service query messages. The current version of PerSON broadcasts LATCH_HELLO and any responses to it are unicast. The query message and other related messages are all unicast. Two new message types have been added to the existing types to handle latch messages. The new message types are shown in Table 5.1.

Table 5.1 PerSON message types for Latch Protocol

5	Latch_Hello	LATCH_HELLO message
6	Latch	LATCH_HELLO response

PerSON constructs an overlay network based on service provisioning [14]. Service discovery is merged with route discovery, by piggybacking the route information in the service discovery message. The earlier version of PerSON does not use every message to keep track of the devices in the network. Every message exchange contributes route

information to the device. In the current version, every message is forwarded to the resolver to update the device and route tables. The time availability of the device is updated when a message is received, and the available time is used to remove the device entries in the route and the device table.

In the previous version, any new route information is added to the route table without considering if the new route is the shortest to the device. In the current version, the route table is sorted. So a message from the device directly, is the first entry in the route table. The sorted route table guarantees that every time a message has to be sent to the device, the route is always the shortest.

When the connection to a device fails, there is no corrective action to ensure that the device and route table reflect the failure. In the current version, the device entries in the route and device tables are removed.

5.4 Service Aggregation

As services come up on each device, the device sends out a Service message to its parent with the task graph of the service. The Service message is a description of the service as a XML file. The service message is handled by the resolver at the parent. A *Service* message type has been added to PerSON to handle these messages. The message format for adding a service is shown in Figure 5.4.

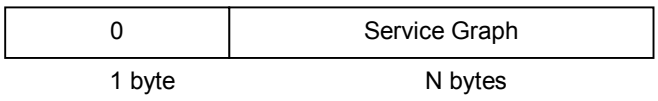

Figure 5.4 Add Service message format

When the device finds a parent, the device sends the consolidated graph describing all the services provided by itself and its children to the new parent node. The parent constructs a consolidated service graph with its local services along with those hosted by its children. The service message is propagated up the hierarchy so that the level 3 device at the top of the hierarchy will have knowledge of all the devices in its service zone.

The aggregated graph is a union of the consolidated service graph and each new graph that is added to it. Algorithm 1 shows the algorithm for service graph aggregation into the consolidated service graph.

Algorithm 1: Service Aggregation Algorithm

Input: G, Consolidated Service Graph Gc

Output: Gc

1: P_{in} = input Parameter (G) , P_{out} = output Parameter (G)

2: If isNode(P_{in} ,Gc) = false then

3: addNode (Gc, P_{in})

4: end if

5: if isNode(P_{out} ,Gc) = false then

6: addNode (Gc, P_{out})

7: end if

8: inNode = getNode(Gc, P_{in})

9: outNode = getNode(Gc, P_{out})

10: if isEdge(inNode,outNode, G) = false

11: edge = addEdge(inNode,outNode, G)

```
12:      attributes(edge) = attributes(G)
13: endif
14: P [] = Intermediate Outputs (G)
15: for every element e in P[]
16:     if isNode(e ,Gc) = false then
17:         addNode(e, Gc)
18:     end if
19:     outNode = getNode( Gc, e)
20:     if isEdge(inNode, outNode, G) = false
21:         edge = addEdge( inNode, outNode, G)
22:         attributes(edge) = attributes(G)
23:     endif
```

When the service is no longer accepting requests, the device sends a remove service message to its parent. The remove service message is also sent when the service starts handling a request. The remove service message is propagated up the hierarchy so that every parent has an updated picture of its service zone. The format of the remove service message is shown in Figure 5.5.

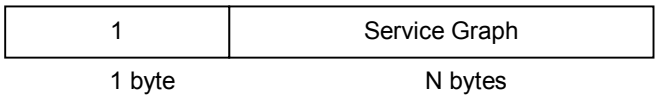

Figure 5.5 Remove Service message format

The parent removes the edges corresponding to the service in the consolidated service graph. If any of the nodes corresponding to the service parameters no longer have any incoming or outgoing edges, these nodes are also removed from the consolidated service graph.

If the device receives a reply, but is unable to connect to the service, it sends an invalid service id message to the parent. The invalid service id message also results in the service being removed from the consolidated service graph. The format of the invalid SID message is shown in Figure 5.6.

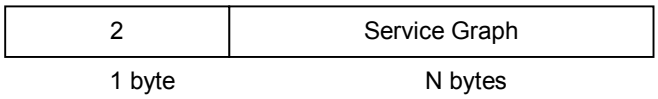

Figure 5.6 Invalid SID message format

5.5 Service Composition

This thesis implements the concept of search and service zones. SeSCo specifies that only a device of level greater than 1 will search within its service zone. This thesis starts the service composition process from the local service zone for every device of level 1 and higher. In case of a level 0 device, the device needs a higher level device to act as a proxy for any communication. So its search zone will start from its parent's service zone.

In SeSCo, we assume that the user task or application requirement is given in the form of a request graph [3]. In this thesis, the request is part of the application. The request graph denotes the individual services required to accomplish a task. The request graph

is also a directed attributed graph similar to the service graph. The definitions of the request graph are specified in Chapter 4.

The format of the query part of the PerSON message is shown in Figure 5.7.

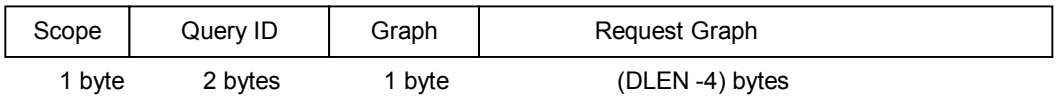

Figure 5.7 Query message – query as Service Graph

An extra field to indicate graphical description or a textual description has been added. The format of the textually described query is similar to Figure 5.7 with *Text* indicated instead of *Graph*. The *Text* query type supports the query format of the earlier version of PerSON.

The requesting device tries to find a feasible path for every node in the query, in its service zone. In case of a failure to find a suitable match, the request is sent to the parent. The parent tries to compose for the query, and sends it to its parent in case of a failure. The request is thus propagated till the top of the hierarchy. When a level 3 device is reached and the device is unable to compose a path, the request is sent to its siblings to search in their service zone.

Any device that composes a path for the request sends a result message to the requesting device. The resolver forms the reply message with the SIDs of the services that fulfill the request, and sends the message to the client. The list of SIDs is the sequence of services that called one after another will satisfy the request. The earlier version of the reply message did not include a list of services, and it did not indicate any

DID as the replying device always hosts the service. The format of the result message is shown in Figure 5.8.

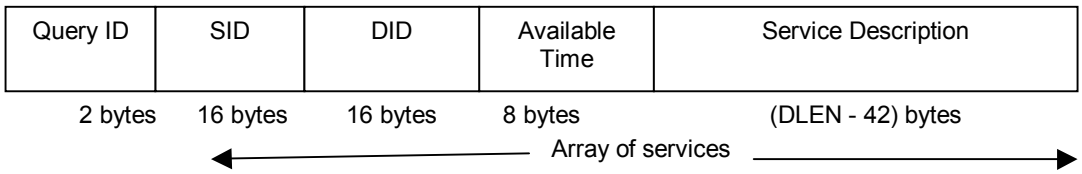

Figure 5.8 Result message format

At the client side the message is first sent to the resolver, which updates the service information in the service table. The message is then sent to the discoverer corresponding to the request, which passes on the list of SIDs to the application that sent the request. The sequence of events follows the sequence diagram shown in Figure 3.9. The application calls each of the services in sequence and passes the result of execution of a service to the next service in the list.

When a requesting device does not receive a reply to its query within a timeout, the parent and children are marked as dead and the latch protocol is reinitiated. The details are discussed in detail as part of the Hierarchical Service Overlay discussion in section 5.3.

PerSON Enhancements

Every service record in the service table now has an associated graph that describes the inputs, outputs and the general flow of data in the service. The earlier version of PerSON specified a name that described the service.

The format of the query message now includes a graph instead of a textual description of the request. Using a graphical description allows for finding better matches to the required service instead of mere text comparison.

The propagation of services up the hierarchy results in situations where the applications might not know how to connect to the service. Every parent will need to pass on connection requests down its hierarchy so that the device providing the service can be reached. Every request for a service that a parent does not provide is now forwarded to its children to be resolved.

Given the hierarchical service overlay, a request might need to be propagated till the top of the hierarchy. When the request reaches the top of the hierarchy the device might not know how to send a reply to the device that originated the query. The reply is sent down the hierarchy to the child device that sent the query to the device. The reply is then forwarded down the hierarchy finally reaching the device that originated the query.

The result message has been modified to include the DID of the device that provides the service. In the earlier version of PerSON, the DID was not required as the device providing the service would send the result. In the current version the device that composes the request might not provide the service. The DID is used to identify the device to contact for the service.

The earlier implementation of PerSON made an implicit assumption that the device originating the query will not host the service required. The current version makes the service connection process transparent to whether this device or some other device hosts the service.

5.6 Demonstration Application

The prototype implements a string encryption and decryption application using matrices, inverse of matrices and matrix multiplication. The process maps every alphabet to a set of two numbers as follows:

a	b	c	d	e	f	G	h	I	j	k	l	m
0	0	0	0	0	0	0	0	0	1	1	1	1
1	2	3	4	5	6	7	8	9	0	1	2	3

n	o	p	q	r	s	T	u	V	w	x	y	z
1	1	1	1	1	1	2	2	2	2	2	2	2
4	5	6	7	8	9	0	1	2	3	4	5	6

Figure 5.9 Alphabet to number mapping

The encryption process maps every alphabet into a set of two numbers as shown above. So a string “serv” will be mapped to “19051822”. The string of codes is converted to a matrix form of

$$\begin{pmatrix} 1 & 0 & 1 & 2 \\ 9 & 5 & 8 & 2 \end{pmatrix}$$

The matrix has its 1st row left shifted three times. The resulting matrix is multiplied with a key, for example, $\begin{pmatrix} 3 & -2 \\ 4 & -3 \end{pmatrix}$ to generate the encrypted message bytes. The encrypted data is saved as a stream of bytes. The flow of data is as shown:

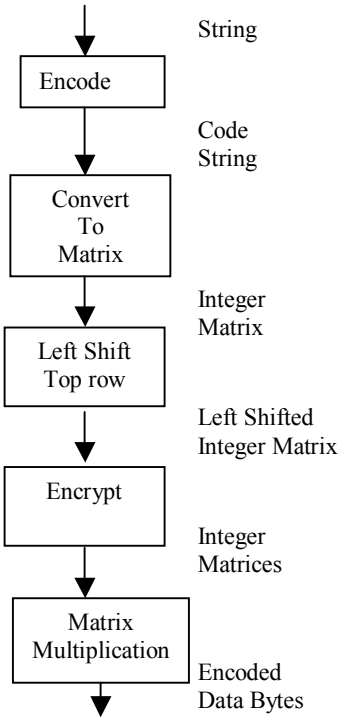

Figure 5.10 Encryption Query

The decryption process converts the stream of encrypted bytes into a matrix form and then multiplies the matrix with the inverse of the key. The resulting matrix has its 1st row right shifted three times and each column is mapped to the alphabet list shown above. The flow of data will be the reverse of Figure 5.10.

Test Environment

A Dell laptop with two USB wireless LAN adapters and two Sharp Zaurus 5500 PDAs are used to form the PICO community. The modules of the encryption and decryption process are distributed among the devices. The laptop has 2.66 GHz Intel Pentium processor with 512 MB of RAM. The operating system is Windows XP professional

and the JVM is J2SE version 1.5. The Sharp Zaurus PDA has 206 MHz Intel StrongARM system on chip processor with 64 MB of RAM. The operating system is embedded Linux and the JVM is Insignia's Jeode, compatible with Personal Java Profile version 1.2. The devices communicate using an ad hoc wireless LAN. One of the IP addresses on the Laptop is a level 3 device, with the other two devices simulating a PDA and a Cell phone. Sharp Zaurus PDAs are level 1 devices and are used as cell phones.

A prototype for an encryption and decryption application is implemented using services developed on the PICO middleware framework. The encryption application puts in a query for encryption of a string. Given a string of alphabets, the string is encrypted into an encoded stream of bytes. The string is first converted into a coded string where each alphabet in the original string is translated into a set of two numbers. The code is converted into a matrix; the top row of the matrix is left shifted three times and then multiplied with the key to get the encrypted stream of bytes. The encrypted data is then stored into a file by the application. The decryption application reads the stored file of encoded bytes. The decrypted bytes are converted into a matrix and multiplied with the inverse of the key. The inverse is stored statically in the service that calls the matrix multiplication service. The decoded matrix has the top row right shifted three times and each column is mapped to the alphabets using Figure 5.9.

Implementation

When a device is started, the different networks supported by PerSON are initiated. The TCP/IP networks open a UDP socket and wait for incoming messages. The device layer

comes up and sends out a LATCH_HELLO message. Other devices in the vicinity send appropriate latch responses and the device is added to the existing device hierarchy. Figure 5.11 shows the test device hierarchy thus formed. The PDA has two of the cell phones latched on to it. The PDA and the third cell phone are latched onto the laptop.

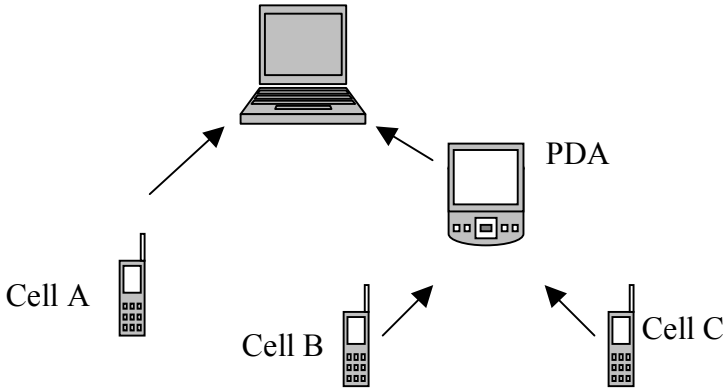

Figure 5.11 Test Hierarchy

As services on each device are started, the service layer requests the device layer to open and accept new service connections. The various functions of the encoding function are distributed among the five devices. Figure 5.12 shows the different services available in the environment. The notation used to describe the services is "<service name> : <service id> : <device that hosts the service>".

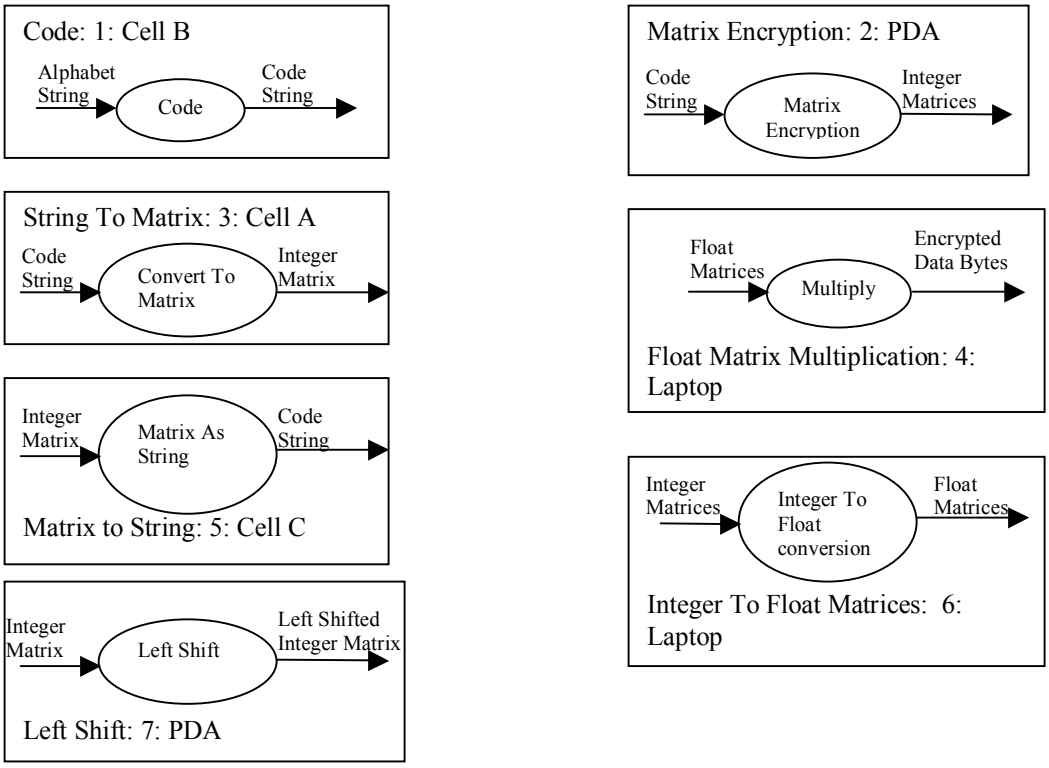

Figure 5.12 Available services

When the encryption application is started, say on Cell B, it sends an encryption query to the parent, PDA. The query is sent using UDP on the IP network to the parent. The parent sees if it can find a path for the request shown in Figure 5.10. The consolidated graph at the PDA is shown in Figure 5.13.

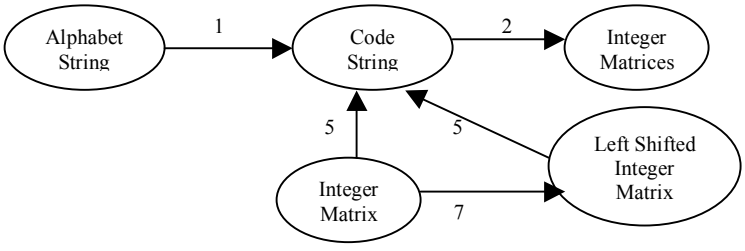

Figure 5.13 Consolidated Service Graph at PDA

As the figure shows, there is a path from alphabet string to code string. The part of the query that left shifts the top row of the matrix and the part that uses the code string to return a set of integer matrices to be multiplied are also available, but there is no path to convert a code string to an Integer Matrix. The rest of the query cannot be composed. The query is propagated to the PDA's parent, the laptop. The consolidated service graph at the laptop is shown in Figure 5.14.

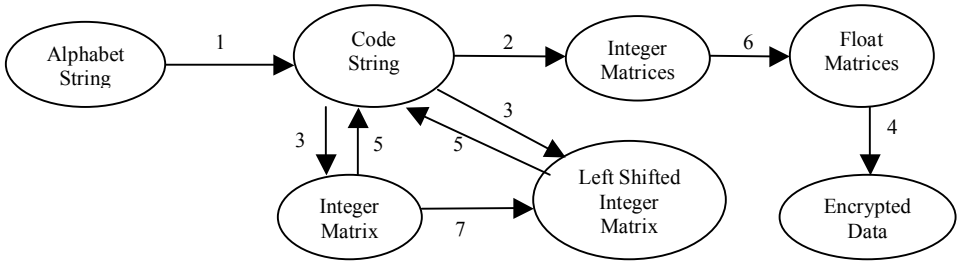

Figure 5.14 Consolidated Service Graph at Laptop

As the figure shows, the query can be composed completely. The laptop sends the list of SIDs and the DIDs of the devices that host these services to the requesting device, Cell B. If the encoding application does not receive a reply within a timeout, the application informs the device, which initiates the latch protocol. The complete path is shown in Figure 5.15.

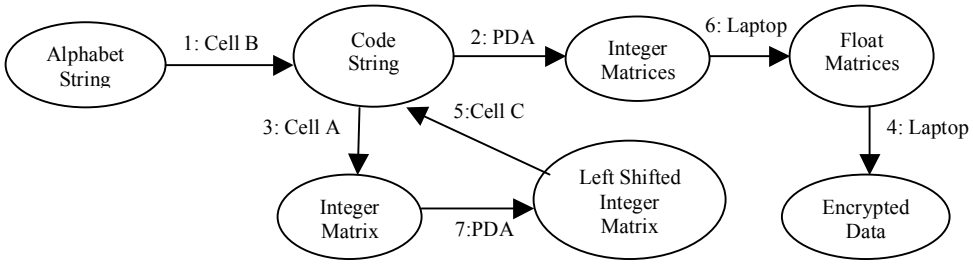

Figure 5.15 Encryption Service path

The reply message is unicast to Cell B using UDP over IP. The encryption application establishes a service connection to the first SID in the list, i.e. Service 1 on Cell B, sending it a string of alphabets. The service 1, *Code* maps the string of alphabets into a set of two numbers. The string of numbers is sent to Service 3, *String to Matrix* on Cell A that converts the string into an integer matrix. The integer matrix is sent to Service 7, *Left Shift* which shifts the top row of the integer matrix three times. The left shifted integer matrix needs to be sent to service 2 *Matrix Encryption* on the PDA, but as it accepts only a code string, the integer matrix is first sent to service 5, *Matrix to String* on Cell C to be converted into a string of numbers and then forwarded to service 2. The integer matrices returned by service 2, *Matrix Encryption* need to be multiplied. The integer matrices are first converted into float matrices using service 6, *Integer to Float Matrices* and then sent to service 4, *Float Matrix Multiplication* to be multiplied. The encrypted bytes are returned to the encryption application, which stores the bytes in a file *code.txt* to be used by the decryption application.

When the encryption application fails to connect to any SID on the list within a timeout, the query is resent to the parent, the PDA. The parent is also informed the particular SID cannot be connected to. The parent removes all the service edges corresponding to the SID. The sequence diagram for the execution of the encryption application is shown in Figure 5.16.

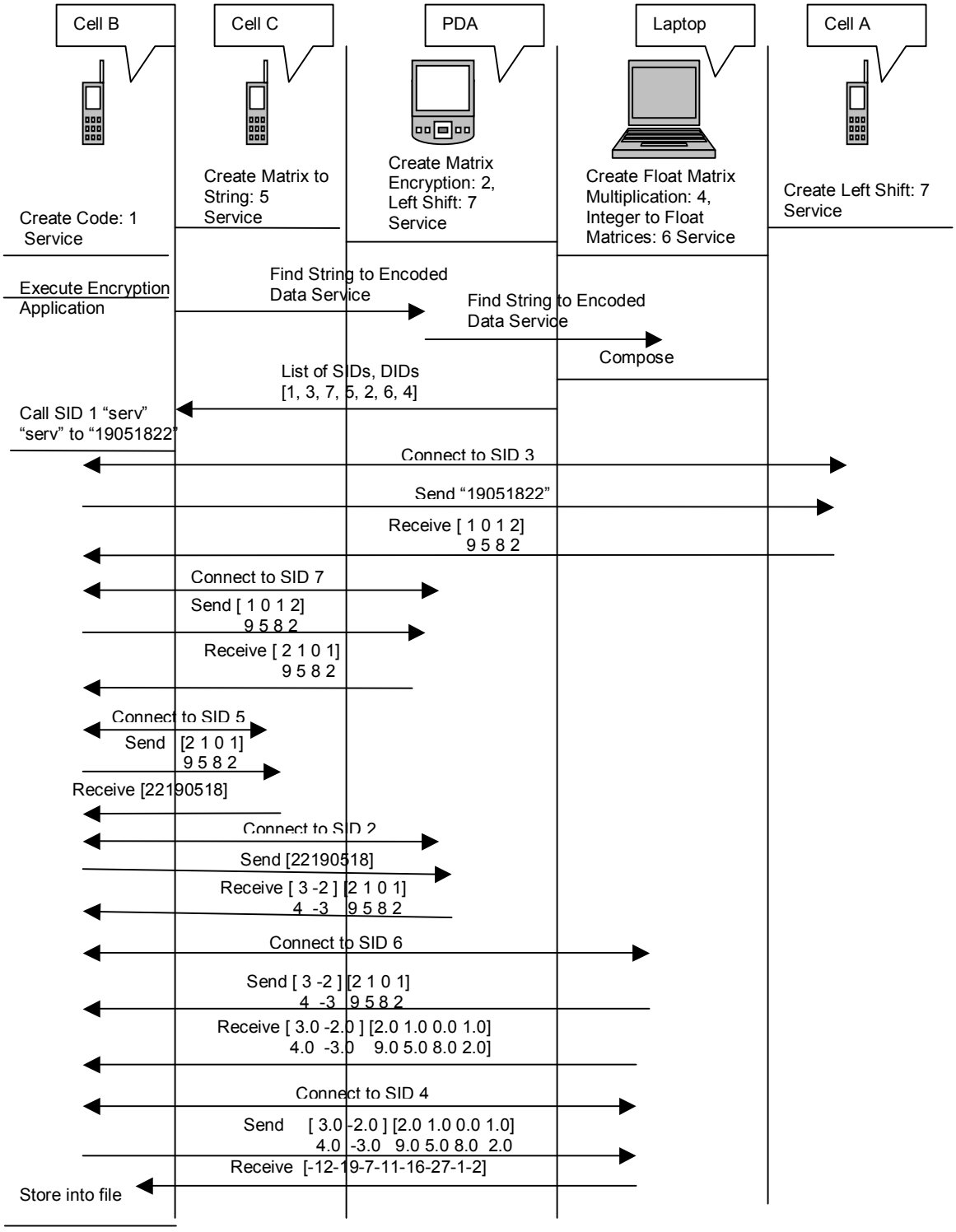

Figure 5.16 Sequence diagram for Encryption Application

When the Decryption application is started, it sends a query to its parent. The parent sees if it can find a path from a node of encrypted data bytes to an alphabet string node. The application reads the encrypted data bytes from the file "code.txt". The device that finds a feasible path returns the list of SIDs to the application. The application connects to the *Decode Byte Array* service and sends the encrypted data bytes to the service. The service returns two matrices - one is the inverse of the key used for encryption and the other is the encrypted data bytes in matrix form. These matrices are sent to the service *Integer to Float Matrices* to convert the integer matrices into float matrices. These float matrices are sent to the *Float Matrix Multiplication* service that multiplies the matrices and returns a stream of bytes. The stream of bytes is converted into matrix form by the service *Byte Stream to Matrix* and sent to the *Right Shift* service. The *Right Shift* service shifts the top row of the matrix thrice. The matrix is sent to the service *Decode* that The maps each column in the matrix to an alphabet, thus forming the string of alphabets. The sequence of service connections in the execution of the Decryption Application is similar to that shown in Figure 5.16.

As each service accepts a connection request, the service is removed from the list of active services maintained in the device layer. A remove service message is sent to the parent. The parent removes the edges corresponding to the service from the consolidated graph. When two requests are started simultaneously, the path in the consolidated graph is returned for the first processed request. As the second request is

processed, the laptop cannot compose the request. The application resends the query after a timeout.

5.7 Resource Aware Service Composition

SeSCo does not take device resources into account when performing service composition. In the consolidated service graph, the service lengths have a default length of 1. The process of finding the shortest path is a breadth first search of the different edges from the input parameter node towards the output parameter node.

This thesis proposes a mechanism to use device resources while performing service composition. The mechanism will help make a better decision thus avoiding burdening devices that are resource constrained.

The process of creating and maintaining the hierarchy remains the same. Each device sends its services on startup and lets its parent know when the service is no longer available. The proposed mechanism enables dynamic assignment of service lengths in the consolidated graph based on the availability of resources. The service composition process performs the shortest path algorithm on the consolidated service graph, than a mere breadth first search.

Each device observes the changing state of its resources like load on CPU, remaining battery percentage and memory consumption. The device evaluates the importance of each resource and appropriately determines a length to be assigned to all its active services. The service length is sent to the parent, which assigns the length to all the services corresponding to the device in its consolidated graph. When a request comes

in, the shortest path algorithm will choose the shortest service path. The device could also choose to not advertise its services beyond a tolerance level.

5.7.1 Parameters of Interest

The dynamic assignment of service lengths lets the service composition choose an optimal service path among those available. The different parameters that could be of interest to the device are as follows:

CPU load: CPU is an important factor that contributes towards a device's ability to accept service requests. Based on the number and nature of current requests, the CPU load can vary with time. For a node performing at peak capacity, any additional work will mean more latency towards the request execution. The CPU load also influences how much power is being consumed on the device. For a higher CPU load, the power consumption is higher.

Memory: A parent assists a child in service advertising, discovery and composition, many a time taking over these tasks for the child (level 0 device attached to a higher level device). Though a higher level device does have significantly more storage space, the limit on memory could introduce delays and CPU load. The increase in CPU load in turn increases battery usage. When there is a choice between two devices, one with lower memory than the other, the optimal choice would be to use the higher memory device, if it is not being used.

Battery power: The pervasive environment will have a number of battery constrained devices. Battery energy will be an important resource to conserve. As a device acts as a parent to many children, the ability to service requests will reduce drastically.

In performing resource aware service composition, these factors could be parameters to determine the length of the service in the consolidated graph. Each of these parameters is multiplied with a suitable factor that determines importance of the parameter to the device. For instance battery power to a laptop might be less important to a laptop than to a PDA, whereas as compared to battery power, CPU load might be less important to a PDA than to a laptop. Based on the computed value, an appropriate length is assigned to all the services hosted by the device.

5.7.2 Implementation

Each device has a file "pc.xml" associated with it. The file specifies parameters like device level and IP address. For resource aware service composition, the file specifies weight for the different parameters, the lengths corresponding to different levels of net resource availability and tolerance levels below which the services are not advertised. The net availability of these resources is determined at regular intervals. When the calculated net availability results in a change in length, the new length is sent to the parent. A latch message, LATCH_RESOURCE is sent to the parent with the format "LATCH_RESOURCE: <length>". The parent sets the length of the edges corresponding to the services hosted by the device to the value sent and forwards the message to the parent. The resource message is propagated till the root device of the hierarchy.

A proof of concept with only battery percentage considered is implemented on the prototype. The weight corresponding to battery is 100. An energy detector thread runs at regular intervals and reads the remaining battery percentage. Different resource

availability levels are specified in “pc.xml” as high and low. The application checks for the values in between as medium level values. The “pc.xml” file also specifies lengths to use for the different resources values. Currently these are lengths of 1 for a high value, 6 for a medium value and 15 for a low value.

5.7.3 Application

The demonstration application is described in Section 5.6. Two instances of the *Integer to Float Matrices* application are added. Cell C, PDA and the laptop now have instances of the service. Consider another service that takes Integer matrices and performs matrix multiplication to return the encrypted data. Let the new service be service 8 provided by the PDA.

The battery values on the laptop and the PDA are all at full battery percentage. The battery on Cell C is allowed to degrade until it is low. The consolidated service graphs with the appropriate lengths are shown in Figure 5.17.

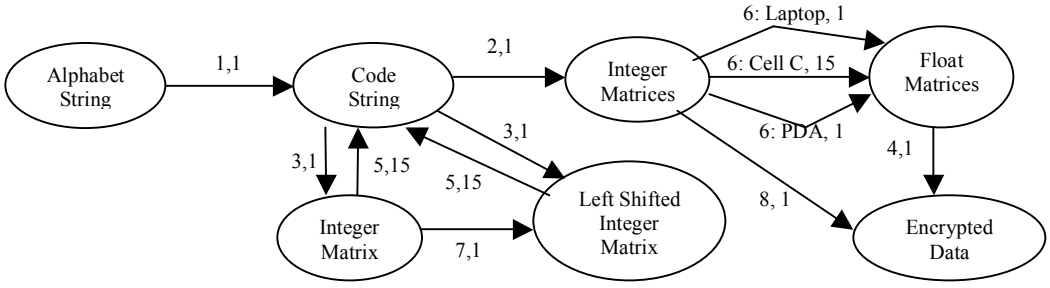

Figure 5.17 Test Case Consolidated service graph on Laptop

The notation used is the <service, service length>. The encryption application is initiated on Cell B. The request is sent to the PDA to be resolved. The PDA cannot

compose the request and the request is sent to the laptop. The shortest path algorithm is run on the consolidated service graph on the laptop. For constructing the path from the node *Integer Matrices* to node *Encrypted Data*, service 8 on the PDA is chosen as it is the shortest.

The battery on the PDA is allowed to degrade to a medium level; the service length of service 8 is set to 6 in the consolidated graph as a result. The consolidated graph with the changed service lengths is shown in Figure 5.18.

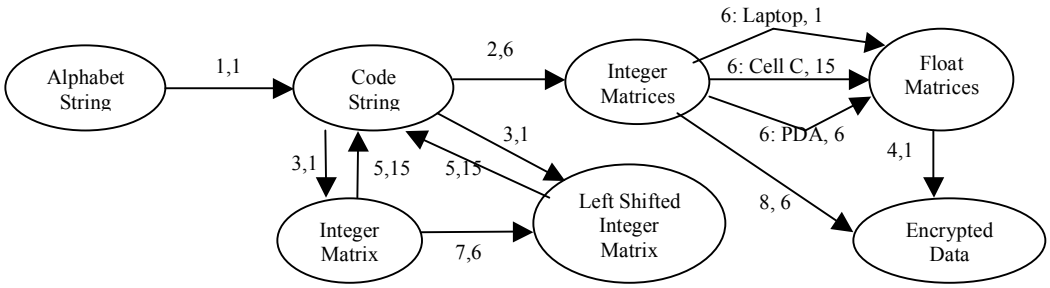

Figure 5.18 Test Case Consolidated service graph with resource degradation

The encryption application is initiated on Cell B. The request is sent to the PDA to be resolved. The PDA cannot compose the request and the request is sent to the laptop. The shortest path algorithm chooses the path from the node *Integer Matrices* to the node *Encrypted Data* consisting of service 6 with service 4, both on the laptop. The combined length of the shortest path is 2 compared to that of the direct path which is 6. While choosing from among the available services for service 6, the service on the laptop has a length of 1, that on Cell C is 15 and that on the PDA is 6. The shortest path algorithm chooses the service on the laptop as it is the lowest among all the choices.

While comparing this scheme with SeSCo without resource awareness, SeSCo would have chosen the direct service 8 from the node *Integer Matrices* to the node *Encrypted Data.* The algorithm being a breadth first search will try to find the route that is the shortest, in this case service 8 of length 1, with the alternate being service 6 followed by service 4 of length 2. The PDA having lesser battery strength than the laptop would have been a poor choice when the alternate path has the lesser battery constrained laptop. Resource aware service composition thus enables SeSCo to choose the least resource constrained devices among all available choices.

5.7.4 Assumptions and Limitations

One of the assumptions while measuring the energy consumed is that all the energy is spent only in execution of task. It assumes that other system applications running in the background do not consume significant energy.

5.7.5 Results

In the scenario described in section 5.7.3, the user wishes to compose an encryption request. There is a choice between using the integer matrices multiplication on the PDA and float matrix multiplication that includes converting integer to float matrices. There are three alternatives available for the integer to float matrices conversion – on the PDA, Cell C and the laptop. Cell C has a low battery condition and the PDA has a medium battery condition.

In resource aware service composition, when the request is sent to the laptop it chooses the combination of Integer to Float Matrices conversion on the Laptop with Float matrices multiplication on the Laptop. In SeSCo, the service composition mechanism

chooses Integer Matrix multiplication on the PDA. The energy degradation on the PDA when using resource aware service composition is measured and compared with the energy degradation when using SeSCo. The PDA continues to work as the parent for Cell B and Cell C. The remaining battery energy on the PDA is periodically recorded for a sequence of five requests composed and executed in the device hierarchy.

The plot shows the percentage remaining battery energy over time as the set of requests is composed and executed when SeSCo is used and when resource aware service composition is used.

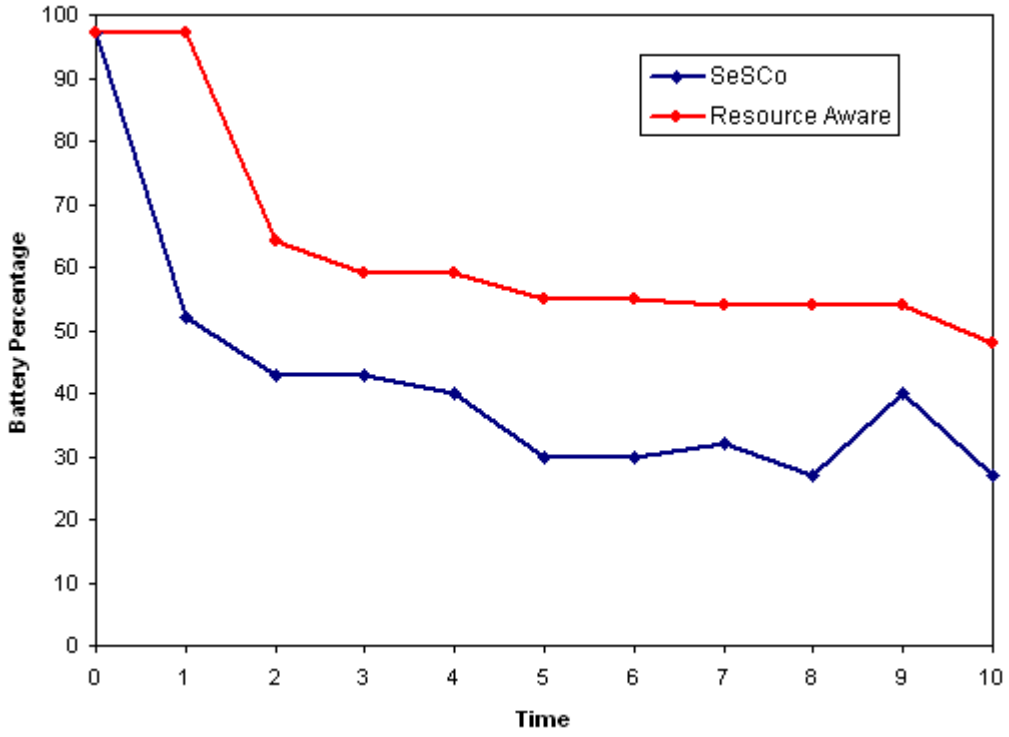

Figure 5.19 Remaining battery energy with time

We see that when the set of tasks have been completed, the remaining battery energy in resource aware service composition is about 50% and that is SeSCo is less than 30%. This gives battery power savings of about 20% when using resource aware service composition over SeSCo.

This plot shows the energy savings of resource aware service composition over SeSCo as time progresses.

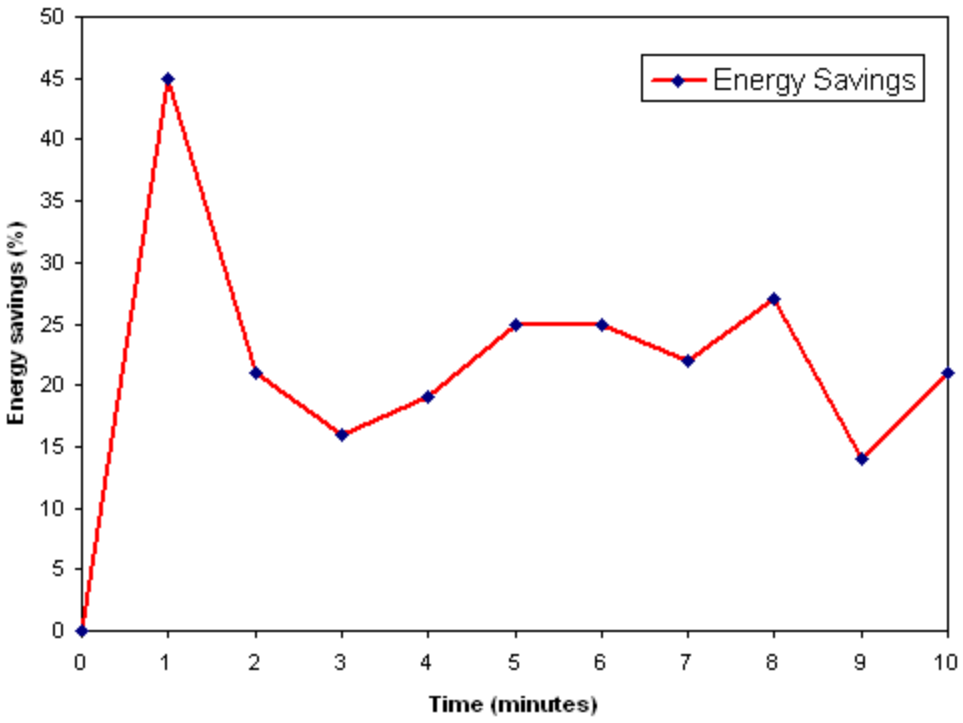

Figure 5.20 Energy savings for Resource Aware Service Composition

As mentioned before, savings of about 20% is achieved using resource aware service composition.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The thesis implements the service composition mechanism SeSCo on PICO using the PerSON framework. The initial version of PerSON implemented a trivial service discovery mechanism, based on textual matching of service descriptions. Service discovery is improved by including graphical descriptions of services and requests to make better matches and compose services in case of no exact matches. This thesis implements the hierarchical service overlay on top of PerSON, new message formats and device and route table changes to support the overlay. The prototype is demonstrated using an encryption and decryption application. The work on SeSCo is extended to monitor resources like battery, CPU and memory. The status of these resources could be considered during service compositions, so that these resources are conserved in already resource constrained devices. The prototype uses the encryption and decryption application and battery values on devices to demonstrate the resource aware service composition.

The future work includes support for offloading service execution from resource constrained devices onto other devices. Adjusting devices levels based on resource availabilities on the device will be implemented on PerSON. Service composition can also be made more optimal by considering how well the service worked for a previous query, thus taking service feedback into consideration for future request resolutions.

REFERENCES

[1] M. P. Papazoglou, D. Georgakopoulos. Service-oriented computing: Introduction, October 2003 Communications of the ACM, Volume 46 Issue 10

[2] Flinn, J., SoYoung Park, Satyanarayanan M. Balancing performance, energy, and quality in pervasive computing. Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on 2-5 July 2002 Page(s): 217 – 226

[3] Swaroop Kalasapur, Mohan Kumar, Behrooz Shirazi. Composition frameworks: Seamless service composition (SeSCo) in pervasive environments, November 2005 Proceedings of the first ACM international workshop on Multimedia service composition MSC '05.

[4] Kalasapur, S.; Kumar, M.; Shirazi, B. Evaluating Service Oriented Architectures (SOA) in Pervasive Computing, Pervasive Computing and Communications, 2006. PerCom 2006. Fourth Annual IEEE International Conference on13-17 March 2006 Page(s): 276 – 285

[5] D. Chakraborty, F. Perich, A. Joshi, T. W. Finin, Y. Yesha. A Reactive Service Composition Architecture for Pervasive Computing Environments. IFIP Conference Proceedings; Vol. 234. Proceedings of the IFIP TC6/WG6.8 Working Conference on Personal Wireless Communications Pages: 53 - 62 , 2002.

[6] S. Kalaspur, M. Kumar, and B.A. Shirazi. Personalized service composition for ubiquitous multimedia delivery. In WoWMoM 2005. Sixth IEEE International

Symposium on a World of Wireless Mobile and Multimedia Networks, 2005, pages 258-263, 2005.

[7] Kumar, M. Shirazi, B.A. Das, S.K. Sung, B.Y. Levine, D. Singhal, M. PICO: a middleware framework for pervasive computing. IEEE Pervasive Computing, July-Sept 2003, Volume 2, Issue 3 Pages 72-79.

[8] W. Ke, P. Basu, and T.D.C. Little, "A Task Graph Based Application Framework for Mobile Ad Hoc Networks," in Proc. IEEE ICC 2002, New York, NY, April-May 2002.

[9] S. Herborn, Y. Lopez, A. Seneviratne. Composition frameworks: A distributed scheme for autonomous service composition. Proceedings of the first ACM international workshop on Multimedia service composition MSC '05

[10] Dipanjan Chakraborty, Anupam Joshi, Yelena Yesha, Timothy Finin, “Service Composition for Mobile Environments”, Conditionally accepted, Journal on Mobile Networking and Applications (MONET), Special issue on Mobile Services

[11] Dipanjan Chakraborty, Anupam Joshi, "Dynamic Service Composition: State-of-the-Art and Research Directions", Technical Report TR-CS-01-19. CSEE. UMBC. December 2001.

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

[13] X.Gu, K.Nahrstedt and B.Yu, “Spidernet: An integrated peer-to-peer service composition framework”, in HPDC 2004: Proceedings of the 13th IEEE

International Symposium on High performance Distributed Computing, 2004, pp.110-119.

[14] K. Senthivel, PerSON - A framework for service overlay network in pervasive environments, Masters Thesis, The University of Texas at Arlington, TX, Spring2006.

[15] M.P. Papazoglou, Service-Oriented Computing: Concepts, Characteristics and Directions, in WISE 2003: Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. Volume, Issue, 10-12 December 2003 Pages: 3 – 12.

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

[17] Z. Maamar, Q. Sheng, B. Benatallah. Selection of Web Services for Composition Using Location of Provider Hosts Criterion. CAiSE 2003 Workshop Proceedings 15th Conference on Advanced Information Systems Engineering. Ubiquitous Mobile Information and Collaboration Systems.

[18] www.bluetooth.com

[19] J. Robinson, I. Wakeman, T.Owen. Scooby: middleware for service composition in pervasive computing ACM International Conference Proceeding Series; Vol. 77 Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing 2004 Pages: 161 - 166

BIOGRAPHICAL INFORMATION

Aparna Kailas received her Bachelor of Engineering in Computer Science and Engineering from Bangalore University, India in 1999. She has been working with Wipro Technologies, India in their automotive electronics division since 1999. She started her Masters in Computer Science and Engineering in 2005. Her research interests include pervasive computing, mobile computing, vehicle networks and applications in the automotive electronics domain.