Technical Report TR-CS-01-21

An Agent-Based Approach to Ubiquitous Information Access

Dipanjan Chakraborty, Filip Perich, Anupam Joshi
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, MD 21250
{dchakr1, fperic1, ajoshi}@cs.umbc.edu

Abstract

Ubiquitous information access from different mobile devices is a challenge to computer science researchers. Research in this area has focused on eliminating different wireless link related problems like bandwidth, disconnection existing in providing information to the mobile systems. This information is also assumed to be available through some services existing in the high bandwidth fixed network infrastructure. We investigate the situations where the information required is not readily available on the network and it needs to be obtained by dynamically locating the required data and then possibly initiating a series of computations to obtain the information. This paper presents a layered architecture for addressing this problem generally and also our initial implementation of this architecture.

1 Introduction

Recent years have seen the proliferation of various types of wireless handheld devices such as Palm Pilots and Cell phones. Realizing the dream of ubiquitous information access necessitates providing the information or data to these personal digital assistants (PDAs) in a uniform and unhindered manner. Typical mobile devices are resource poor compared to their wired counterparts. Usually they are capable of getting data from different service providers using the wireless networking infrastructure. However, certain mobile devices are also unable to compute information that needs some compute power or resources on the mobile platform. They have to depend on different services to get the data they want.

The level of computing and networking hardware has also seen significant changes in the past few years. We have seen the emergence of palmtop, wearable and embeddedable computers, GSM phones and smartcards based SIMs. These devices are infact capable of providing a platform to do some local computation. We have also seen proliferation of Bluetooth-like systems, which provide short range, moderate bandwidth connections at extremely low costs. Wireless LANS (IEEE 802.11b) and satellite WANS have been deployed in the industry and academia. Apart from these comparatively low bandwidth wireless networks, a high bandwidth wired Internet infrastructure always exists.

Different e-commerce applications are being developed[23, 24, 25] by taking advantage of wireless networks. Cell phones or PDAs have essentially become mobile storefronts for e-tailers. We are familiar with ads of people buying commodities via their cell phones from the beach. This approach essentially treats palmtop devices as consumers of goods or information. The information or goods come from servers on the wired side. This approach is based on a client-proxy-server model [3, 26, 2, 11, 17, 9] and has been developed by the acedemia over the last five-six years in contexts such as web access from mobile platforms.

The underlying assumption behind such extension of e-commerce is that the information being provided to the mobile units is usually coming from a fixed single service provider from the wired Internet. This in some sense is like a *supermarket* approach, where a few identified service providers exist. There are many consumers and the traffic is necessarily one-way. However, it is possible that the information required by a mobile user may not be available per se but needs to be computed dynamically using various resources (software, data or hardware services) distributed over the network. Research advances in the area of software copmonents[16, 6] and the availability of dynamic service location platforms have provided the necessary groundwork to achieve this purpose. The componentization of software allows for independently manufactured software units to interact easily with each other. Also, thanks to inexpensive disk storage, there is an increasing amount of raw data available on the network. For example, on-line trading services provide a wealth of stock information for customers to research. We see a paradigm where independent software and data resources could "plug and play", with computations performed on the fly, to obtain required information.

Consider a situation where you are traveling in a car with a GPS enabled navigation system to provide location information. You want to find out the route to the local stadium, where a baseball game is being

played. The navigation system has the necessary resources to provide you with a route to the stadium. However you observe that the game would start in another 10 minutes and you don't want to miss it's beginning. The traffic conditions on the road are really bad. Under these conditions you might be interested in taking the least crowded way to the stadium. Or in other words, you want to compute the optimal route to the stadium taking into consideration the dynamic traffic condition on the roads.

The GPS system in your car has the capability of providing the current location information of the car. The navigation system inside cars these days usually have all the ancillary data required to provide you with a route to the local stadium. It definitely would not be able to provide you with the optimal route taking into consideration such dynamically changing parameters like the traffic condition on different roads leading to the stadium. Even if we found out one service (an information provider) willing to give automatic traffic updates of traffic conditions on different roads, the system in the car may or may not have the necessary compute power or software component to calculate the route. To further emphasize the utility of our system, consider the situation when you want to get to the local stadium from the point where you are at present through the least crowded roads. Your car does not have a navigation system nor is it GPS enabled. You have a PDA with a GPS chip capable of giving you the location information. Thus you can't count on the support of the navigation system to provide you with the route to the local stadium also. If we technically analyze the query, we observe that to compute the optimal route to the stadium with respect to traffic conditions, we need the following data:

- Dynamic traffic information on all roads leading to the stadium from where the car is
- Road map of the area
- GPS locations of the car and the stadium
- a computation platform to compute the optimal route to the stadium with respect to the traffic information on different roads

The standard network based solution one can think of is to implement a single powerful centralized service provider to compute the result of such a query. The service provider would be able to obtain dynamic traffic updates from different streets with the help of sensors or other hardware/software components. It would have a database of road maps of the nearby area. It would also have the software component required to compute the optimal route to the destination with respect to the traffic conditions. Apart from the usual problem of scalability in such a solution, we also assume the existence of such a service capable of doing such a specific task available in the network. However, in the standard Web environment this is a tough proposition. Traditionally, Web has been an information delivery vehicle. We will find different services capable of doing small generic tasks, like providing the road map of the nearby locality or traffic information on certain streets. Thus it will provide you with distinct heterogeneous services to get information. But it clearly lacks the ability of cooperation between heterogeneous software (and hardware) components to collaboratively compute information and provide it to the mobile user. Distribution of computation has been limited to sending executable content to the requesting client in form of java byte codes or other scripting codes. It does not consider cases where the client may not have enough power to run the executable content. Recently, some solutions, which essentially provide a light, weight script suitable for mobile clients have been proposed. However, this is focused on phone type devices and ignores the cases where mobile clients possesses more computational power. Thereby if resources are available at the mobile client, it may be under utilized.

Considering the above issues, there is a need for a model where the distribution of computation is based on availability of resources at the mobile client. This distribution may also depend on other wireless link related issues like bandwidth, delay or quality of service factors. This idea of adhoc teams of entities that are dynamically formed to pursue individual and collective goals can be used to create the software infrastructure needed by the next generation of mobile applications. These will use the emerging $3^{rd}$and 4th generation broadband wireless systems as well as short-range narrowband systems such as Bluetooth[21]. Realizing this paradigm in a mobile information access system has a few problems.

1. The mobile clients may not have adequate computational power on board and may need to borrow computation power from networked resources.
2. Efficiency demands that some of the computation be performed on the mobile host. This is especially true for components that need a lot of user interaction. Executing these on the network will generate a lot of traffic.
3. The wide variety of mobile platforms available in the market, argue against statically defining the location where a particular component should be executed, because some clients may not be able to handle even the slightest computation demands.

In this paper, we describe our initial work towards realizing a system, which supports dynamic discovery of services and negotiation between different software and hardware entities in the fixed network. These entities collaborate with each other using this system to dynamically compute information, which is not apriori available on the mobile device. The services that collaborate with each other may be present in the fixed network infrastructure or the service providers could themselves be mobile. The scenario where the services are mobile is feasible in the context of emerging mobile ad-hoc networks. In this paper, we concentrate in addressing the problems where the services are fixed and available some where in the fixed network infrastructure.

In order for an entity to cooperate with others in its vicinity, it first needs to discover other entities as it moves into a new location. This is the problem of service discovery. State of the art systems such as Jini,[1] Salutation,[22] UPnP,[22] IETF's draft Service Location Protocol,[13] E-Speak,[14] Ninja [5] provide for networked entities to advertise their functionality. However, these systems describe services entirely in syntactic terms as interface descriptions. This not only limits interoperability, but also forces a client to know a-priori how to describe a service it needs in terms of an interface. Moreover, these service discovery techniques return "exact" matches and can only handle equality constraints. This leads to loss of expressive power in the component description. Within these newly emerged service discovery techniques, Jini provides a flexible and robust infrastructure for distributed components to find each other. However, the Jini service discovery is not without its drawbacks. Even though it provides a robust platform independent service discovery infrastructure, it does service discovery based on string interface matching only. In this paper, we describe our enhancement to the Jini Lookup Service Reggie, which does a much broader 'semantic' level service discovery based on XML descriptions of the software and hardware components.

Another important aspect of cooperation amongst heterogeneous services is to define a common infrastructure for them to collaborate with each other. Clearly, there is a need for an agent-oriented abstraction of the software and hardware components. An agent-based infrastructure would help us in generically defining each component in the system. It would also help in defining the common communication protocol that these heterogeneous entities would follow in order to communicate with each other. In this paper, we describe the Ronin Agent Framework. It's a simple Jini-based agent development framework that is designed to aid in the development of next generation smart distributed mobile applications. Ronin Agent Framework helps in providing an agent-based wrapper over any software or hardware component. It also defines a uniform communication protocol for heterogeneous agents to communicate with each other irrespective of the underlying communication protocol being followed.

2 Background

The Ninja project offers a solution for implementing a robust, scalable, distributed Internet infrastructure that permits heterogeneous devices to discover and access the provided services in a secure manner. The components of the architecture consist of the following four principal elements: On the client side are the unit elements, which represent various devices that a person can use to access services that satisfy the person's

objectives. On the server side are the bases, collections of powerful interconnected workstation cluster environments, which are the heart of the service discovery and completion as well as the stage management. In between are the active proxies, which are responsible for adapting and translating data and access protocols to allow and to provide the necessary base-client interaction. Finally, the last principal element is a path that provides an abstraction linking the former three elements for diverse service execution while maintaining the secure access to the information, dynamic service adaptation, and the fusion of multiple devices. The vital aspect of the Ninja project is the Service Discovery Service (SDS), which provides a mechanism allowing services to advertise their presence to the SDS servers, and which provides a mechanism allowing the clients to locate the advertised services across wide area networks. The core part is the SDS server, which collects and stores advertisements from services located in its domain and uses the cached information to answer client service discovery requests. Periodically, based on the determined rate for issuing advertisement messages, each service registers with its closest SDS servers by sending an authenticated XML-based description of provided services over a secure channel. The SDS server contacts the certificate authority (CA) to validate the service authenticity, and the SDS server additionally contacts the capability manager (CM) to obtain the access control list for the given service. Subsequently, upon client request, the SDS server tries to locate the matching service based on the XML description plus it verifies the client's access rights for the services found. In this manner, the SDS server addresses the security concerns by controlling the set of entities that are allowed to discover particular services. It does so by hiding the service presence from unauthorized entities instead of disallowing access to a service that has already been located. Finally, to provide a scalability of service discovery, the SDS servers are hierarchically ordered and responsible for maintaining descriptions of services located only in their children's domains, and the SDS servers employ a hash function to decrease the amount of information that the higher positioned SDS servers must maintain for their children. The hash function correctly predicts when a service is not offered by any SDS server in a particular hierarchy sub-tree; however, due to the loss of information a parent-SDS server may incorrectly predict a service existence in the sub-tree.
The project Portolano focuses on the needs of consumers in order to provide easy-to-use, low-maintenance, portable, ubiquitous, and ultra-reliable task-specific devices. The principal idea of the project is to create a prototype of the consumer computing landscape, which would employ diverse user interfaces, network infrastructures and distributed services. Currently, the research is underway and the Portolano group is discovering and defining the numerous problems and evaluating possible solutions to provide the necessary infrastructure. The problems range from service representation to the client to service discovery and service execution in a secure and reliable manner.

The Service Location Protocol (SLP) is a language-independent protocol for automatic resource discovery on IP networks utilizing an agent-oriented infrastructure. The base of the SLP discovery mechanism lies on predefined service attributes, which can be applied to universally describe both software and hardware services. The architecture consists of three types of agents: User Agent, Service Agent and Discovery Agent. The User Agents are responsible for discovering of available Directory Agents, and acquiring service handles on behalf of end-user applications that request services. The Service Agents are then responsible for advertising the service handles to Directory Agents. Lastly, the Directory Agents are responsible for collecting service handles and maintaining the directory of advertised services.

Jini is a distributed service architecture developed by Sun Microsystems, whose services can be realized to represent hardware devices, software programs or their combination utilizing the Java language. Its overall goal is to transform the traditional network into a flexible, easily administered tool on which human and computational clients can find services in a robust manner. One feature of the Jini system is to form a Jini federation that represents a collection of Jini services, which can then coordinate tasks and information among themselves, such as adding and deleting of services. A key component of Jini is the Jini Lookup Service (JLS), which maintains the dynamic information about the available services in the Jini federation. Each service is responsible for discovering one or more JLS before it can enter a federation by using either a

known location or by using a Jini multicast discovery. Upon JLS discovery, the service also is responsible for uploading its service proxy and for periodically refreshing its registration at the JLS. Similarly, each client is responsible for JLS discovery before it can download the matching service and invoke various methods at the original service location. However, in the Jini architecture the client is solely responsible for knowing the precise name of the Java class representing the service.

Universal Plug and Play (UPnP) extends the original Microsoft Plug and Play peripheral model to support service discovery provided by network devices from numerous vendors. UPnP works and defines standards primarily at the lower-layer network protocol suites, so that the devices can natively, i.e. language and platform indecently, implement these standards. UPnP uses the Simple Service Discovery Protocol (SSDP) for discovery of services over IP networks, which can operate with or without a lookup service in the network. In addition, the SSDP operates on the top of the existing open standard protocols utilizing HTTP over both unicast (HTTPU) and multicast UDP (HTTPMU). When a new service wants to join the network, it transmits an announcement to indicate its presence. If a lookup service is present, it can record such advertisement to be subsequently used to satisfy clients' service discovery requests. Additionally, each service on the network may also observe these advertisements. Lastly, when a client wants to discover a service, it can either contact the service directly through the URL that is stored within the service advertisement, or it can send out a multicast query message, which can be answered by either the directory service or directly by the service.

Salutation is an open standard, communication-, operating-system-, and platform-independent service discovery and session management protocol. The goal of Salutation is to solve the problem of service discovery and utilization among a broad set of appliances and equipment in a wide-area or mobile environment. The architecture provides applications, services and defines a standard method for describing and advertising their capabilities, as well as locating and determining other services and their capabilities. In addition, the Salutation architecture defines an entity called the Salutation Lookup Manager (SLM) that functions as a service broker for services in the network. The SLM can classify the services based on their meaningful functionalities, called Functional Units (FU), and the SLM can be discovered by both a unicast and a broadcast method. The services are discovered by the SLM based on a comparison of the required service types with the service types stored in the SLM directory. Finally, the service discovery process can be performed across multiple Salutation managers, where one SLM represent a client to another SLM.

3 Technical Concept

Our system foresees the next generation of distributed and mobile computation in an agent-oriented infrastructure. In a mobile environment resource availability is very critical to properly run different services and tasks. In a resource poor environment, a given platform may not have the capability of executing the whole task. The task then needs to be broken down into executable components and executed separately. We have to find out services ready to execute the different subtasks. Even though a mobile device may have the necessary computation power to execute a task, it might not have the data available with it. There are various problems one can foresee in such an environment.

- The mobile device may have all the necessary components to fetch the data from a service provider but due to the underlying lossy and low bandwidth connection, it will be difficult to load the data on to the mobile platform.

- Even if the bandwidth factor is negotiated, the resource poor mobile client might not have adequate volatile memory to load all the data and then compute the desired information.

- All these problems become more acute when the device needs information, which is not apriori available from a single service provider. Our system provides an infrastructure to do such dynamic computation by taking all these factors into consideration.

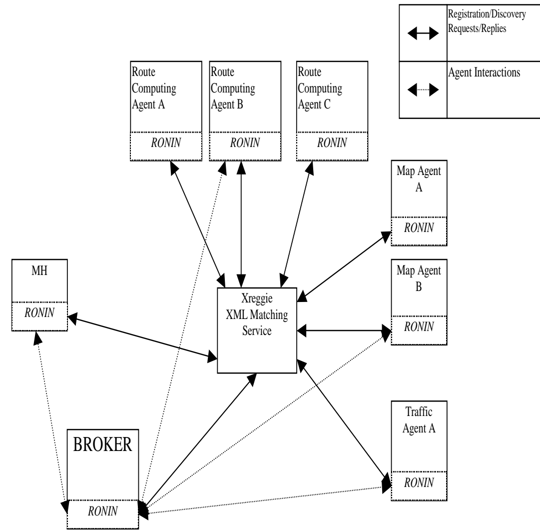

Figure 1: General Picture

Consider the example we discussed in section 1. You are travelling in a car and you want to find out the optimal route to the local stadium, which is having a famous baseball game. You want to find out the optimal route taking into consideration the dynamic traffic conditions on the streets leading to the stadium. The car may or may not have a navigation system. All you have is a GPS enabled mobile device. You submit the query to our agent residing in the mobile device. The agent residing in the mobile device is the client part of our system. We explain the technical concept behind the system by using the above example.

3.1 Service Discovery, Uniform Agent Communication

Our agent in your device figures out that the local platform does not have a road map of the area. Using short-range Bluetooth service discovery techniques, it also figures out that the car does not have a navigation system. The agent figures out that there is no locally available software to compute the optimal route from a source to a destination taking into consideration certain constraints. Unable to find out any solution locally it tries to find out whether there is any 'broker' service in the vicinity that is capable of providing a solution to the query. The agent uses existing service discovery techniques available to it at that point. It figures out that there is a broker service nearby who is willing to take your task and provide you with the answer. All the services are essentially Ronin Agents (discussed in section 4.1). Hence they use an uniform communication interface to communicate with each other. The underlying actual communication infrastructure is encapsulated in the Ronin Agent's communication API. So you submit your query to the broker. The agent also specifies its GPS location to the broker. The client agent also specifies the system properties of the mobile device with the help of an well-formed XML document.

The query is now passed to the broker layer in our system. The broker figures out that none of the services existing in the vicinity has the capability of providing a solution to the query all by itself. The broker breaks the query into various subcomponents since one particular agent doesn't have the capability to give a reasonable answer to the question. The next task of the broker is to find out the different service agents willing to do the different subcomponents. The subcomponents to be considered in this case are

- Finding out an agent capable of providing the location information of the stadium

- Finding out an agent capable of giving an index of the traffic in each road leading to the stadium.
- Finding out an agent capable of providing a road map of the different roads leading to the stadium from the point where the car is (obtained from the GPS readings from the mobile device inside the car)
- Finding out a computation agent capable of computing the optimal route from an arbitrary source to an arbitrary destination in a graph taking into considerations certain constraints on the edges of the graph

The broker thus tries to find out the location of the stadium and finds out an agent capable of providing the location of the stadium. We assume that the location information being provided is uniformly understood by each agent in the system. The broker tries to find an agent capable of providing the traffic updates. It might find that no single agent can provide such a data. Rather, multiple geographically distributed agents are willing to give updates of roads near its vicinity. The broker then has to find an agent capable of combining these information into a single information set. However if it finds an agent who can provide all the required information, then its problem is solved. Similarly the broker goes ahead and finds out the agent who can provide a road map of the area from the point where the car is to the point where the stadium is. If either of these services are not available, then the broker would not be able to provide an answer to the query.

All the agents/services are Ronin Agents (discussed in section 4.1) who implement a uniform communication interface. The broker uses Xreggie(discussed in section 4.2), the XML matching lookup service to find out the different map agents existing in the network. Each agent in the environment has to provide an XML file to the Xreggie lookup service. This XML file describes the attributes of the service each agent supports. It also provides a description of the resources on which the agent is running. The broker submits to Xreggie an approximate XML description of the services it wants. Xreggie does an intelligent matching to find out the different services. It provides the broker with all the services it could find. It might well be the case that multiple agents are willing to do the same task for the broker.

The broker now selects one of the agents based on constraints like overloading factor or cpu usage of the local machine on which the agents reside, memory availability, cost of providing the service etc. These attributes are available to the broker through the XML description of the different service agents. Similarly it also finds out another agent giving dynamic traffic updates on various streets nearby.

3.2 Dynamic Composition of Services

After finding out the required service agents, the broker has to decide on a schedule of how to execute the different sub queries. Note here that the problem of breaking an arbitrary task into different executable subcomponents is a classical planning problem [7]. We assume that the broker has preconfigured information of the different subcomponents that are required to execute a particular task. However, we envision that in the near future, there would exist an agent capable of providing the broker with such knowledge. In addition, due to the inherent characteristics of our proposed agent-based architecture, injecting such an agent into the system would require minimal configuration. The broker also has at its disposal, a description of the system on which each agent is running. The broker also knows about the constraints of different services each agent is supporting. E.g. it knows whether a service is mobile or in other words, whether a service is willing to provide a serializable code to be executed locally to another service . The broker computes the execution path of the query by taking all such elements into consideration.

While arriving at a decision, the broker also considers certain dynamically changing factors like bandwidth available, amount of data transfer required and the current memory allocation or processor usage of the local machines where the agents are running. The main task of the broker is to find out an optimal way of executing a task. If it finds that the mobile agent has got sufficient resources available with it, it might also consider sending some executable components to the mobile host. Considering our earlier example in section 1, the

broker thus figures out that the order of execution of the different sub components should follow the following partial ordering:

1. Get the location of the stadium from the agent providing such location information
2. Get a road map from the agent providing road maps
3. Get an index of the amount of traffic on each of the roads from the traffic agent
4. Invoke the service (route computing agent) which can compute the optimal route from one point to another by providing it with the required data you have collected
5. Send the computed information back to the mobile client

We note, that 2 and 3 could be done parallelly since they are independent of each other. However, instead of simply executing all the components one by one, the broker applies its intelligence on the dynamic information it has, to distribute the computation. E.g. It may figure out that the amount of data involved in transferring a road map over the network would be huge. This might well be the case if the car is very far from the stadium. Thus it would require a lot of bandwidth to get the map from the map agent and then send it to the route-computing agent. Rather it would be much more efficient to send an executable code of the route computing agent to the local system where the map agent resides. This would of course be subject to the condition that the route-computing agent is able to provide one such serializable code and the local system of the map agent has enough memory and resources to execute such a component.

Again in another situation, due to the high volume of requests coming to the Traffic Agent, the traffic agent might be overloaded and might not want to serve any more requests locally. Rather it might be happy to give a serializable object, which might be executed locally in some other platform, and the results calculated. The Route Computing Agent and the Map Agent also might have certain similar limitations. E.g. the Map Agent might not have any serialized code available and hence has no mobility. The broker takes all these factors into consideration while deciding on the way to execute the queries. After figuring out a reasonable schedule, the broker orders the different agents involved to execute their own queries. The broker gathers all this information and sends the result back to the car agent.

4 System Architecture

The general architecture of the system is divided into 3 distinct layers, viz. Network layer, Uniform Agent Communication Layer, Dynamic Service Discovery Layer and the Broker layer.

Before going into details about the different layers, we present an overview of the Ronin Agent Framework, which is the underlying agent-development infrastructure we use and the Xreggie: the smart semantic service discovery system.

4.1 Ronin Agent Framework

The Ronin Agent Framework is a Jini-based agent development framework [4] that is designed to aid in the development of next generation smart distributed mobile systems. The Ronin Agent Framework introduces a hybrid architecture, a composition of agent-oriented and service-oriented architectures, for deploying dynamic distributed systems. Ronin contains a number of features that distinguishes it from other comparable frameworks, including an Agent Communication Language (ACL) and network protocol independent communication infrastructure. It also includes an agent description facility that allows agents to discover and find each other, an agent proxy architecture that provides agent mobility behavior and customizable agent communication scheme.

The framework encourages the reuse of non-Java AI applications by promoting the development of Jini-enabled services for these AI applications. For instance, as a part of the Jini-enabled Agent Development

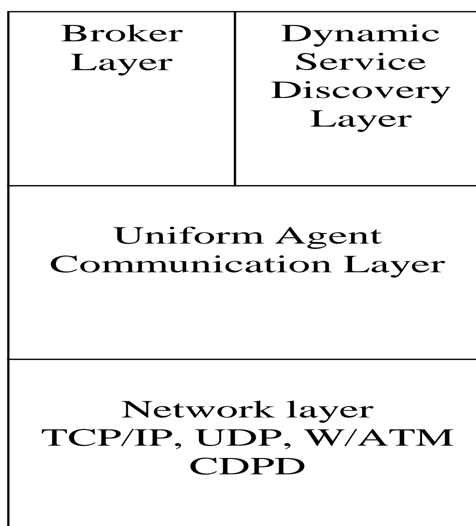

Figure 2: layered Architecture

Toolkit, a Jini service, Jini Prolog Engine Service (JPES), is developed to provide knowledge sharing and inference capabilities for Ronin agents across the network.
The design goal of Ronin is to define an open framework that specifies the infrastructure requirement and the interface guideline for the interaction and communication between agent-oriented Jini components. Ronin framework models Jini services as agents. Each agent consists of two parts:

- Ronin Agent
- Agent Deputy

4.1.1 Ronin Agent

A Ronin Agent is a specialized version of a Jini service that is designed with the agent-oriented abstraction in mind. Unlike a traditional Jini service that only provides services through the predefined service methods, a Ronin agent provides services through agent communication as well. A Ronin Agent can be realized either by hardware or by software. Ronin Agents may differ in their actual implementation; however, each Ronin Agent must have a set of Common Agent Attributes that can be used to describe the generic agent properties and functionalities across different domains. Ronin agents communicate with other agents through message passing using ACLs.

4.1.2 Ronin Agent Attributes

There is a set of attributes associated with each Ronin Agent. These attributes can be further divided into two disjoint subsets. The first set of attributes, Agent Attributes, defines the generic functionality of an agent in a domain independent fashion. Ronin framework defines the types and the semantic meanings of Agent Attributes. The second set of attributes, Agent Domain Attributes, defines the domain specific functionality of an agent. The Ronin framework does not define the Domain Attribute types and their semantic meanings. Both Agent Attributes and Agent Domain Attributes can be registered with a Jini Lookup Service as Jini Entry objects along with the Agent Deputy that represents the agent.

4.1.3 Ronin Agent Deputy

The functionality of an Agent Deputy is similar to the functionality of a proxy object in the Jini programming paradigm. An Agent Deputy acts as a front-end interface for the other agents in the system to communicate with the Ronin Agent it represents. Each Ronin agent may have zero or more agent deputies in the network. Conceptually an agent deputy can be thought of as a lightweight copy of the owner agent in the network. The main duty of the Agent Deputy is

- **Mediate agent communication:** The agent deputy accepts information on behalf of the owner agent and delivers the messages to their owners. They may mediate on the content of the information also. The agent deputies are considered as trusted representatives of the owner agent.
- **Provide local agent representation:** The Agent Deputy acts as a local representative for the owner Ronin Agent in a different system. This local proxy thus can answer some questions on behalf of the owner Ronin Agent.

4.1.4 Ronin Communication Infrastructure

Ronin does not define how an Agent Deputy should communicate with its owner Ronin Agent. However, it does define the interface for the communication between the Ronin Agent and Agent Deputy. Specifically, each Agent Deputy must implement a method, *deliverMessage (Envelope)*. The semantic meaning of the method is that after invoking this method, Agent Deputy will try its best to deliver an Envelope object, which contains the actual message, to the Ronin Agent it represents. A developer is free to implement any transport protocol and delivery scheme between the owner Ronin Agent and it's Agent Deputy. For example, DeputyArthur, a concrete Agent Deputy implementation, is implemented to communicate with the owner Ronin Agent through the Distributed Event model, which is based on the Remote Method Invocation infrastructure. Another concrete Agent Deputy implementation, is implemented to communicate with the owner Ronin Agent through the generic network socket model, and is designed to handle the customized store-and-forward message delivery scheme. Agent Deputies can be easily designed and implemented to handle other complex delivery scheme such as notification mailbox etc.

4.1.5 Envelope Objects

The messages that are interchanged between Ronin Agents are embedded within Envelope objects during the delivery process. This meta-level approach allows Ronin Agents to interchange messages with arbitrary content message types under a uniform communication infrastructure metaprotocol. Each envelope contains the following components:

- **Recipient:** The intended receiver of the content message
- **Sender:** The message sender
- **Language:**The ACL identifier of the content message
- **Ontology:** The ontology identifier of the content message
- **Deputy Locator:** The locator is used to locate the Agent Deputy of the message sender, making the two-way communication possible between the receiver and the sender. The framework does not define how a Deputy Locator should locate an Agent Deputy in the system. This makes the framework open and implementation independent.

4.2 Xreggie: Enhanced Jini Lookup Service

The Jini service discovery infrastructure provides a good base foundation for developing a system with components distributed in the network that need to discover each other. But Jini does not fully address the problem of semantic service discovery. Some of the disadvantages of Jini are:

- Jini is still not adequate because it essentially operates at a syntactic level. In other words, a client needs to specify the exact function/interface needed in order to discover a service. We aimed to build a dynamic service discovery mechanism where the lookup process can operate at a semantic level. Namely, the client should be able to provide a service description at a higher level of abstraction than the interface description.
- Jini does not handle parameters like cost of executing services. We wanted to specify constraints on these features (e.g. cost $<$ max) – recall that Jini can only handle equality relationships.
- Jini is unable to provide services, which match a service description at a certain level of approximation. We wanted to allow "similarity" matches and return services that were similar to, but not the same as, the requested service. For example, the system may return a service providing a 3-month stock chart when the client asked for a service providing 6-month stock chart. This is useful when exact matches cannot be found for the service requested by the user.

We address these problems in the Xreggie system. It shows how Jini (and similar systems) can be taken beyond their simple syntax based service matching approaches. The idea is to modify the Jini lookup service to match services using XML attributes instead of interfaces and java attribute objects. Each component (service) provides an 'XML' entry describing its functionality and requirements when it registers into the Jini Lookup Service. When a client wants to use a service, it creates a XML DOM object describing the service it needs along with the constraints. It then finds the appropriate service using the XML match function. The XML match handles constraints such as requirements, cost, mobility etc. For instance, the client can ask that the service cost less than some amount, or that it be mobile. In order to use XML in the match, the entities in the system need to agree on an 'ontology'. In our system, we assume that both services and clients have previously agreed on an ontology and knows what each entry in the XML description means.

4.2.1 XML Representation of Components

Each component or service in the system is described using well-formed XML (Extensible Markup Language). The semantic tags of XML make service descriptions meaningful. XML files can also be converted to DOM objects and hence can be easily merged into the Java-Jini system. Each service in the system conforms to a standard XML DTD definition. each service uses an XML file to describe its attributes. Whenever a client wants to find out a service, it also fills out an XML document approximately with the information it wants. Inside an XML document a service gets the opportunity of providing a lot of details about itself and also about the system on which it runs. We have features for the service to provide information about

- **Capability:** Definition of what this component or service is capable of doing
- **System Features:** The service may specify the details of the system on which it is running and also has the option of providing the minimal system requirements
- **Cost:** The total cost of using this component. This also varies depending on whether the object is executed remotely or locally with respect to the client
- **Mobility:** Whether this component is static and cant be serialized; whether it is mobile

4.2.2 Xreggie Lookup Server

In order to make the XML based match work, we have made modifications to Reggie, the Sun's implementation of Jini Lookup Service. Reggie provides all the functions and methods defined in the Jini Core package, besides some extra functionality provided by Sun. We modified it in the following way:

- A Xmlentry class is added in the net.jini.lookup.entry package, which takes a XML DOM object as argument in its constructor.
- An XmlMatch interface is added in the com.sun.jini.reggie package, which defines the various functions needed in the XML match function
- An XmlMatchImpl class is plugged in the com.sun.jini.reggie package, which implements the functions defined in the XmlMatch interface. It uses the Java Xml parser to convert the XML description to a Java DOM object and does the comparison with the two XML DOM objects from client and server side.
- The RegistrarImpl class in the com.sun.jini.reggie package is modified, which lets the enhanced Jini Lookup Service (XReggie) know when and how to do the XML attribute match instead of doing the ordinary Name/Type syntax-level match.
- The CreateLookup class in the com.sun.jini.reggie package is changed, which lets the enhanced Lookup Service (XReggie) know where to find the information of XML parser package
- The command line to start Jini Lookup Service has also been adjusted to contain some extra options.

When ever a service registers with the Xreggie lookup server, it provides an XML DOM object to the lookup server with the help of an 'XML' entry attribute. This DOM object is stored in the lookup server and updated whenever the service registers again. When a client wants to find a service, it also provides and XML DOM object to the lookup server which has an 'approximate' information about the service it wants to find. Xreggie finds service matches based on various factors like name, attribute type and service description. The matching logic inside Xreggie first tries to find out named matches. If it fails, then it looks at the other attributes of the XML file and tries to obtain a similar match. The Xreggie system assumes that the client and service agrees upon a common ontology to use. If the ontologies are not same, then service matching becomes very difficult and goes into the domain of AI. Research efforts are going on standardizing a ontology to be used by services[19].

4.3 Layered Middleware Architecture

4.3.1 Network Layer

The Network layer is the lower most layer in our architecture. This layer represents the underlying communication infrastructure being used to communicate between different Ronin Agents in our system. Some distinctive features of the network layer are:

- The network layer encapsulates the whole underlying transport module needed to transfer data from one system to another. It is not analogous to the standard network layer in OSI or TCP/IP protocol suite. Rather our network layer comprehensively encapsulates the various different network transport protocols like TCP/IP, UDP, W/ATM, Bluetooth or CDPD.
- The Ronin Agent development framework allows the use of different protocols for transferring its messages. This is particularly useful since our network layer can encapsulate any type of communication mechanism. The network layer provides the upper layer with the required abstraction to switch between different networks as and when required. The network layer thus helps in seamless integration of

different network protocols in our system. In our particular implementation of a stock brokering service, we have chosen to implement the network layer using TCP/IP sockets.

- Instead of using standard network protocols like TCP/IP, the network layer also allows higher-level message transport mechanisms. Thus the transport modules could also be written based on the Distributed Event and Notification system, or any other communication protocol.

4.3.2 Uniform Agent Communication Layer

This layer manages the delivery of information to the agent. Since all the entities interacting in the environment are Ronin Agents, they implement a common way of handling and delivering messages. Thus, even if the underlying transport mechanism used to actually send the messages are different, the agents can send and receive messages in a uniform way. The Agent Deputy part of the Ronin Agent Framework is responsible for handling messages in this layer. An Agent Deputy imitates the functionality of a Proxy object in the Jini programming paradigm. An Agent Deputy acts as a front-end interface for the other agents in the system to communicate with the Ronin Agent it represents. As discussed in section 4.1, the remote sender Agent uses a standard interface to communicate with any Agent Deputy. The communication mechanism between the Agent Deputy and the owner agent is however not specified. It is implementation dependent. This proxy-based uniform agent communication mechanism has a few distinctive advantages in an agent-oriented infrastructure. Figure 3 explains this communication mechanism.

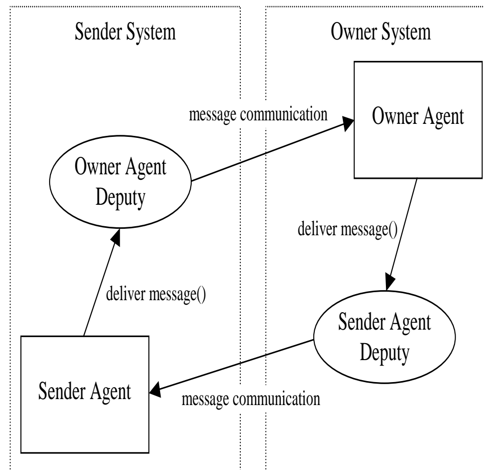

Figure 3: Ronin Agent Deputy Communication Mechanism

- When a mobile host represents a Ronin agent, it's agent deputy sits on the wired network. Similarly, the Agent Deputy of a Ronin Agent sitting on the wired network could reside in the mobile host. Ronin does not define how an Agent Deputy should communicate with it owner Ronin Agent. However, it does define the interface for the communication between the Ronin Agent and Agent Deputy. Thus different heterogeneous agents can use the same agent communication primitive to communicate with each other.

- Ronin is particularly suitable for our purposes because it is designed in such a way that the mode of transportation could be easily changed at run time. Thus, When a mobile host moves between various types of networks, the transport module appropriate for that particular network could be chosen dynamically. Also, the ability to use different transport modules would definitely help in mobile environments where wide variety of mobile clients are available.
- The Ronin Agent Deputy inherits many unique features from the original design of the remote proxy object. In addition to that, there are a number of positive design side effects of using the Ronin Deputy-based infrastructure as the agent communication layer.
  1. The deputy design adds a mobile agent behavior to the owner agent. Each Owner Agent may have more than one Agent Deputies. Once these deputies have been deployed in the network, the Owner Agent may move off to another location. It is the task of the Agent Deputy to establish contact with its owner mobile agent.
  2. The design allows the Agent Deputy to act like a *store-and-forward agent* [1], capable of delivering messages based on various policies for reliability. For example, the Agent Deputy could try to deliver a new message once and if that event fails, not try again. The Deputy might implement an acknowledge-based reliable delivery scheme. Either way, it's the duty of the agent deputy to ensure that the messages reach its Owner Agent. The Deputy could also act as a message mailbox where the owner agent could store messages for a certain amount of time when it is doing some higher priority work.
  3. The Agent Deputy could act as a message filter or a transcoding proxy for the Owner Agent[26]. Eg. If the Owner Agent moves into a low bandwidth network, the Agent Deputy could transcode the content sent to the Owner Agent. It could also implement security by intercepting messages that failed to pass the security check policy.
  4. The Agent Deputy could act as a message multiplexer or demultiplexer. Upon receiving a new message, the Agent Deputy could multiplex the message by delivering the message to the deputies of other agents. The deputy could also combine several messages and send a digested version of the message to its owner agent.
  5. The Agent Deputy could also act as a translator of messages coming from agents following other ACLs. A Ronin Agent may be capable of processing a restricted set of ACLs and ontologies. Thus different *translator* agents could be deployed in the network to translate the message to an understandable format.

4.3.3 Broker Layer

The broker is the core engine, which finds the individual services required in computing required information and manages the distribution of computation. This layer has the knowledge base of what services should interact to produce the required information to the end-user. The broker layer essentially comprises of broker soft wares distributed all over the network being capable of solving problems within a particular domain. Thus each broker is essentially an agent in the system advertising services. The broker is responsible for task disintegration, hierarchical service screening, and calculation of the optimal computation path and distribution of computation. We explain each of these in detail.

- **Task Disintegration:** Once a broker receives a particular task to be executed, first it tries to figure out whether the task can be broken down into multiple subcomponents. As we discussed in section 3, the generic problem of chopping down an arbitrary task into subtasks is very complex and required complex AI techniques. In the future, it is possible to have an agent do this work for the broker. That agent may be implemented by a third party and may employ complex knowledge representation techniques to solve the problem. Due to the agent-based architecture of our system, such an agent

can be easily converted into a Ronin Agent and plugged into our system. Right now, we maintain a database mapping a request or query to the various subcomponents that can be possibly run parallelly or in some order to execute the query.

- **Hierarchical Service Screening** Once the broker has found out the various subtasks required to execute a query, its next job is to find out the different services existing in the network, which are capable of executing the different subcomponents. The broker uses the Xreggie system to initially find out such services. The broker has to supply an XML description of the different services it wishes to access. The broker employs the concept of hierarchical service screening here. Initially, when the request comes to the broker, it has an XML description of the resources available at the mobile client. If it finds out that the mobile client is resource poor and would not be able to execute any component locally, it tries to find out services, which are only implemented as *remote*. The broker uses the XML match functionality of the Xreggie lookup service to screen some services at the lower layer. Once some services have been found, the broker has an XML description of each service available with it. If there are multiple agents offering the same service, the broker starts the second screening phase where it rejects all services, which demand more return for executing a job. The return type is considered as 'cost' in the present implementation. This type of screening may continue on various other attributes like current resource usage of the local system where the agent is residing, the average wait time for getting a job done by a particular service etc. Finally the broker comes up with a set of services that are capable of doing the subtasks.

- **Calculation of Optimal Computation Path and Distribution of Computation** Once the various services have been found out, the broker tries to find out an optimal way of executing the query. The main aim of the broker is to reduce the cost of the query. Cost here may be the monetary cost involved in utilizing the services. But more appropriately it may mean

  - The amount of data transfer over the network
  - Amount of bandwidth utilized
  - Amount of CPU resources used
  - Average response time

  The Xreggie lookup server supplies the broker with an XML description of the different services it requires to execute the query. The XML description contains attributes specifying the above properties as well as other parameters too. The broker parses the XML document to retrieve all the desired values and constructs a decision tree. The leaves of this decision tree uniquely specify a computation path to be followed. E.g. The Broker may decide to get the raw data required to calculate certain information from the data provider agent and then send the data over to another agent, which does some computation on the data. However due to low bandwidth or the high volume of data transfer over the network, it might be cost effective to send a serialized code of the agent doing the computation to the source of the data. This is of course subject to the condition that such a computation code is available and the agent providing the data has sufficient resources to execute the serialized object. Thus we see two distinct computation paths possible in this case. Again, if the mobile host is not resource poor and the information it wants requires a lot of user interaction, then it might be best to move all the components to the mobile host and make them execute locally on the mobile platform. The broker uses the decision tree to arrive at such a solution.
  The Broker hands the distribution of computation once the computation path has been determined. It invokes the services of the different agents, which are executing the subcomponents. The collaboration involved here between the agents is essentially centralized and handled by the broker itself. The broker sometimes might also act as a mediator agent between different executing subcomponents. The broker may decide to execute some components on the mobile host or any *other platform* other than the main

service provider itself. To execute a particular component on a platform, the broker sends the following information to the agent.

1. A serializable service object
2. The method name to be invoked on service object
3. Arguments to be used for the method. The agent then performs the necessary computation and returns the results to the broker. There are two points worth mentioning here - 1)The agent providing the computation platform need not have any knowledge (like the service interface) about the service it is using. It simply uses the Java reflection mechanism to invoke a method on the service object. 2) The broker would assign an agent to perform this computation only after it decides (from XML attributes specified by the agent) that the agent has the resources to do so. Any information (data or software component) that is to be delivered to the agent is done so via the uniform agent communication layer.

4.3.4 Dynamic Service Discovery Layer

This layer represents the mechanisms through which services are discovered and the mechanisms through which services express them. Thus this layer enables the core function of dynamic service location. It is important that the platform support dynamic interaction between services and the clients of the services, with no or little prior knowledge of each other. We implement this layer using the 'Xreggie' system.
The Xreggie system has been discussed in detail in section 4.2. A description of the capabilities and constraints of all services existing in the local network is kept inside the Xreggie lookup server. A service description is cached and purged if the service fails to re-register within a certain amount of time. The lookup server forms the core of the service discovery layer. We however follow a common ontology to specify services and requirements in the system. The reason for doing this has been explained in section 4.2. However such a system is prone to the crashing of the centralized lookup server. However, replicating the lookup servers can reduce the chances of such failures. Thus each service can discover the other services in the system by little or almost no information about the others.
Individual services are registered onto the network with capabilities and constraints expressed as XML attributes. Each service in the system conforms to the same XML DTD definition. To generically describe a service we considered the following attributes.

- Service Name
- Service capability : Contains a description of what the service can do
- Service Requirements: Service requirements include the resource requirements of a particular service. We considered the CPU type, CPU speed, Memory required and other softwares required for the service.
- Service Mobility: whether the service can provide a serialized code to be executed else where in any other system
- Service Inputs: The input types and arguments required to invoke the service
- Output Types: The format in which the service returns a result
- Service Machine Characteristics: It includes a description of the different attributes of the system on which the service is running.

Whenever a client wants to find out a particular service, it specifies some requirements in an XML file following the same ontology. This helps us achieve a more semantic level service discovery in our system.

However, the client might not have all information required to properly recognize the service it wants. The client then only partially fills up the XML file with the information it knows. The rest of the fields are left as 'UNSPECIFIED'. The modified lookup server uses the XML Dom objects produced from the XML descriptions to appropriately find out matching services. Please refer to section 4.2 for details.

5 Implementation and Experiments

5.1 Implementation

As an example implementation of the above architecture, we implemented a set of agents and services to handle complex stock related queries from resource poor mobile clients. We also implemented a vertical broker, which has the necessary power to coordinate between these services, manage the distribution of computation and provide a solution to the client. The system uses the implementation of the Xreggie lookup server to discover the different service agents. The broker and the different service agents are Ronin Agents. All agents in the system use Ronin's uniform communication infrastructure to communicate with each other. All the service and agents are written in Jini programming language. The system is targeted to run on mobile devices that are capable of executing Jini enabled applications. We provide a brief description of the different components of the system.

5.1.1 End User

The end user of the system is essentially a person with a mobile device capable of running Jini in it. Currently high-end mobile devices such as laptops can support fully featured JVM and are capable of executing Jini applications. A lightweight JVM for handheld devices such as palmtop, known as KVM[20] is currently being developed. Thus our assumption of a java/Jini implementation does not cause loss of generality. We use the idea of service proxies in our system. Normally a proxy implements the same interface as the service itself. However the end user is not supposed to know the interface implemented by an arbitrary broker service in advance. Thus in our implementation, we have a proxy which implements the following interface:
public interface UserInterface extends Entry
public void GetUI();
This proxy object once downloaded knows how to communicate with the broker agent. It also presents a GUI to the user describing the services of the broker just discovered (Figure 4).

5.1.2 Service Agents

- **Single Stock Quote Provider Agent:** This Agent is capable of providing stock quotes of a company on that day. It connects to a stock provider agency (yahoo.com in our implementation) and retrieves real-time stock information from it. This Agent does not provide a serializable code to the client requesting the service. Hence all requests have to be served locally and the agent is not mobile. The data returned by this agent is however very small since it contains only a stock quote of a company.

- **Historic Stock Quote Provider Agent:** This Agent is capable of retrieving stock quotes of companies over an extended time period. The back end information provider is yahoo.com. This is of course assuming that the company existed over that time period and that there is some Internet service capable of providing it with the required data. This Agent could well have been the actual service provider in the Internet. We implemented these agent-based front ends for those services in order to keep them untouched as well as show the effectiveness of our system.

- **Average-Stock-Calculator Agent:** This Agent accepts a range of stock values and computes the average of those stock values. This is a simple service. We implemented this service to show the dynamic interaction of different such services in our system. In the future, there could and will be

complex services doing specific tasks like computing the standard deviation of some values. The main aim is to show that such services can easily be injected into our system.

- **Option Pricing Agent:** This Agent provides a service capable of pricing options based on the Black-scholes model.
- **Profit and Loss Agent:** This software agent is capable of computing profit-loss on options.
- **Execution Platform Agent:** This service provides an execution platform for other services to execute their codes. The main concept behind creating the service is that due to system overloading or some other reasons, a service might be unwilling to serve a certain request locally. Rather it might be willing to give a serialized code out to the client to be executed locally. In mobile platforms, downloading and executing codes might be a problem. These types of utility services may help in providing a local execution platform to the broker. The broker invokes this service when it figures out that it doesn't have enough computational power to execute the components in the mobile device, nor is the service provider willing to do the task locally. This agent finds the necessary services using *service ID's* and downloads the necessary components from the service provider. We use the java reflection mechanism and the concept of unique serviceID's to see that the execution platform can be completely ignorant about the components to be executed. The services are executed in the order specified by the broker and the result is returned to the broker.

5.1.3 Service Discovery and Broker

We have implemented the service discovery layer using Xreggie. We have already discussed Xreggie in detail in section 4.2. Our implementation has agents registered in the Xreggie lookup server with capabilities and constraints and system requirements expressed as XML attributes (refer Section 4.3.4). Since only Java objects can be registered as attributes for a 'Jini' service, we have used'XML4J' parser from IBM to parse an XML file to a Java DOM object. Xreggie provides the broker with a dynamic service location protocol, which it uses to find and screen services.
We have implemented a vertical broker capable of handling complex stock related requests. The first task of the broker is to determine the different subcomponents required to execute a task. We have implemented this section by keeping a mapping of the underlying services required to execute a certain job and the partial order in which they should be executed. We didn't use any complex AI techniques to decide which services are required for a task on the fly. This is partially because our broker was handling a tailored set of queries and our design allows the use of an agent capable of doing this for the broker in future. The main aim of the system is to show the dynamic composition of different services and distribution of computation by the broker.
Once the broker has a list of the services that it needs to execute a task, the broker uses Xreggie to find and screen the appropriate services. The broker then has an XML description of the capabilities and constraints of different service components. We use a nested if-then-else statement to implement the decision tree. Based on the different capabilities of the local platforms of the agents (which are obtained from the XML descriptions) a computation path is computed by the broker using the decision tree. The broker in this way makes the decision of where each component is executed. The constraints, which we considered while taking this decision, have been discussed in section 4.3.3. A mechanism is also needed by which the broker knows the resources available at the mobile platform. For this purpose, an XML file describing resource availabilities of the mobile client is transported to the broker when it submits the request.

5.1.4 Agent Deputies

We have implemented two different types of concrete agent deputies in order to facilitate efficient agent communication in different environments. As discussed in section 4.3.2, an agent deputy may be implemented

to offer various different proxy type services to the owner agent. Thus the agent deputy could potentially take into account limitations of mobile devices on wireless network and try to deliver the best result. Ronin defines the interface for the communication between the Ronin Agent and Agent Deputy. The Ronin framework defines a 'Transport' object in its framework. This interface consists of a single method to actually transmit the information, defined as,
public interface Transport
public void transmit(Envelope e);
Each implementation of an agent deputy holds a transport object. whenever a deputy is asked to deliver a message to the corresponding owner agent, it delegates this responsibility to the 'Transport' object it holds. Separating the interface from the object that actually performs the job provides a good abstraction as the 'Transport' module can be changed to suit the need of the agent. The two different implementations of the Agent Deputy are:

- **Simple Reliable Agent Deputy**: This Agent Deputy handles messages in the simplest way possible. The 'Transport' module it encapsulates uses TCP sockets for data transmission. Thus it offers a reliable connection-oriented service to the Agent sending the message. This type of deputy is used in networks, which are less prone to transmission errors like Ethernet LANs or even WANs.
- **Store-and-Forward Agent Deputy:** This Agent Deputy handles some of the wireless link related problems like disconnection. We have implemented a concrete 'Transport' object to handle disconnections of the mobile host. The underlying communication uses network socket model and uses store-forward message delivery scheme. Just before an agent residing on a mobile host is about to be disconnected, it sends a 'store' message to it's deputies on the wired side, which instructs the deputy to queue any messages. The deputy employs a hash organization to keep track of all the queued messages from different sender agents. Later, on re-connecting back to the network, the Owner Agent sends a 'forward' message to its deputies thereby receiving all the collected messages.

5.2 Experiments

We performed several experiments to test our system and check the adaptiveness of our broker with respect to different constraints of the service agents. The main aim of the broker was to find a reasonable path of computation and make the information available to the mobile host. All the agents, Xreggie lookup server and the broker were run in different machines connected through a 100MBPS LAN. Some of the machines ran Linux 2.2.16 while others were run on Solaris platform running SunOS 5.7. In order to simulate the capabilities, resource availability and constraints of different machines, we manipulated the XML files containing the system descriptions. The primary purpose of the experiments was to show the mechanism in which the broker distributed the computation by considering the different system limitations. In the future, we can write scripts, which take the actual system parameters and populate the XML file accordingly. In a similar way, we simulated the system description of a mobile host. It was assumed to be running on a 1.2MBPS LAN.

5.2.1 General System Behavior

Once a mobile host discovers a broker in the system, it's presented with a GUI containing the different services that the broker offer (Figure 4).

Once a query is submitted to the broker, the broker carries out the task disintegration, service discovery, optimal path computation and distribution of computation. It is worth mentioning here that, various models of optimality can be considered here. E.g. we could think of optimizing monetary cost of executing the services (assuming that the agents charge money by offering services), size of software components involved, amount of bandwidth being utilized to transfer data and code over the network, volume of interaction between user and software component over the network or response time to get the data back to the client.

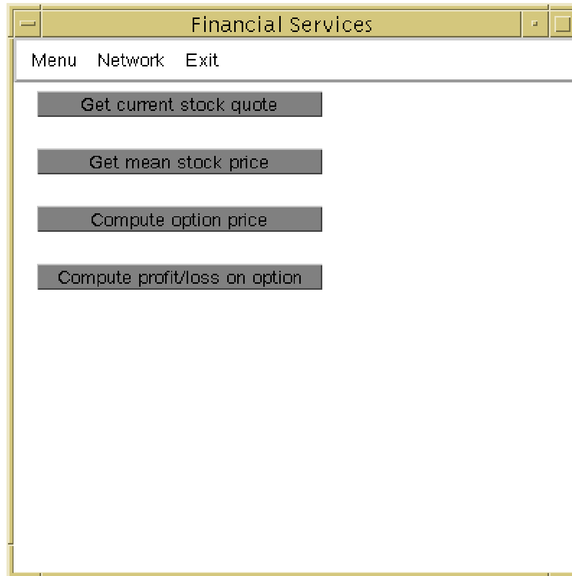

Figure 4: User Interface of the Financial Broker

In our implementation of the broker, it's main objective is to find an optimal solution path by trying to optimize the amount of data transfer over the network by taking into consideration the available bandwidth. It also tries to optimize the average response time required to serve the query. However at the initial stage of hierarchical service screening it also takes into consideration the cost of the service. It considers the mobility of the components at a later stage of service screening. The decision tree calculates an optimal path by balancing these factors. In our implementation, the broker gives higher priority to the amount of data transfer over the network. In our experiments, we have observed that the overall nature of the system is flexible and adaptive to the environment. For a weak mobile client (a client having low processing power and memory), all the computation could be performed at the service site. For moderately capable clients, some of the components could be sent to the mobile host again depending on the amount of data transfer required. The uniform communication layer also acts as another proxy layer. The agent deputies handle certain other limitations like disconnection. In the following sections we report a few experiments showing the different ways in which the broker schedules the different component executions and finally sends the result back to the mobile client.

5.2.2 Experiment 1

- *Problem:* A busy investor wants to find the stock prices of different companies. But he does not have an application running on his palmtop, which can retrieve the stock values. However he has our end-user agent running in his palmtop. He wants our agent to find out services, which can provide him with current stock values of different companies.
- *Description:* In the above example, we had our financial broker service running on a machine and another Single Stock Quote Provider Agent running on another machine. The Xreggie lookup server was also running on the same multicast network. The agent in the palmtop discovered the Xreggie lookup service in the network by using Jini's multicast service discovery protocol. The palm then provided Xreggie with a rough XML description of the type of financial service its wants. Xreggie found our financial broker. The broker's user interface was downloaded to the client machine (Figure

4). The user-interface had options to take in the company symbol for which the stock quotes are sought. We modeled the constraints of the client machine to represent a Palmtop. The necessary XML attributes we considered were the OS type (PalmOS), memory available (RAM available to execute components) and the CPU usage at the moment when the query was sent. These constraints of user's palmtop, which are stored as XML attributes, along with other attributes were also sent to the broker. In this case, the task disintegration phase of the broker was trivial since the broker needed to find out a stock quote provider agent. The broker uses Xreggie to find out all stock quote provider agents. Once the services are found, the broker now uses the XML descriptions of those agents and decides on one particular agent to send the query to. This screening was done at the hierarchical service-screening phase. In this case, the agent that was selected was running on a Linux OS and had a CPU usage of 0.8. In our experiments the broker considered a system to be overloaded if the CPU usage went above 0.5. But in this particular case, the agent was not mobile. Hence even if it was overloaded, it didn't have any serialized code that could be executed elsewhere. Thus the broker had to send the request to the Stock Quote Agent. The result was computed locally and sent back to the broker. The broker returned the result to the mobile host.

5.2.3 Experiment 2

- *Problem:* Noticing the fact that the stock of a company he invested in, is widely fluctuating, an investor wishes to find the average value of a share over six months. However, he doesn't have software on his palm, which is capable of retrieving the different stock quotes over a period of time, and calculate the average. He requests our agent to find out services to which he can delegate the task.

- *Description:* The first few steps leading to finding the broker and exporting the GUI were similar here since both the requests were in the domain of financial services. Hence the same broker was found out in either case, even though the requests were for different services.
  When he requests the information to the broker(Figure 5) as explained in the previous scenario, the broker sees that it primarily needs a data provider service which provides the stock data for last six months. Apart from that the broker also need a computation agent, which would calculate the average of stocks over a given period of time. The broker first finds such service agents using Xreggie.

  In this experiment, we concentrated on testing out the various computation paths possible by optimizing the data transfer over the network and then the response time. For experimental purposes, we changed the system constraints of the different machines to see how the broker is adapting to the changes. We discuss a few conditions and the decision the broker took under those conditions:

  1. we considered the Data Provider Agent to be running on a LinuxOS with a memory of 25K available for external processes to run on this machine(Please refer to Figure 6). The CPU usage of the machine was 0.4. The Data Provider Agent was also not mobile. Similarly for the machine running the Computation Agent, we considered a memory availability of 3K and a CPU usage of 0.8. Thus the system was overloaded. We approximated the bandwidth required for data transfer by the length of time over which the data has been sought. This is a valid approximation since the amount of data grows with the amount of time interval for which you need the data in this case. We considered the amount of data transfer to be high if an average of stock quotes for over a month was sought. In this case the data transfer was considered to be high (since we had requested an average of stock data for the last 6 months). For the mobile platform, we assumed a CPU usage of 0.5 and a memory availability of 22K. Since the amount of data required was significant, the broker figured out that it was unwise to transfer it over the network. Also the computation is fairly processor intensive in this case. As described in Figure 6, it found out that the computation agent was really overloaded, but the data provider and the mobile host are not. But at the same time, since the average is requested over a period of six months, the

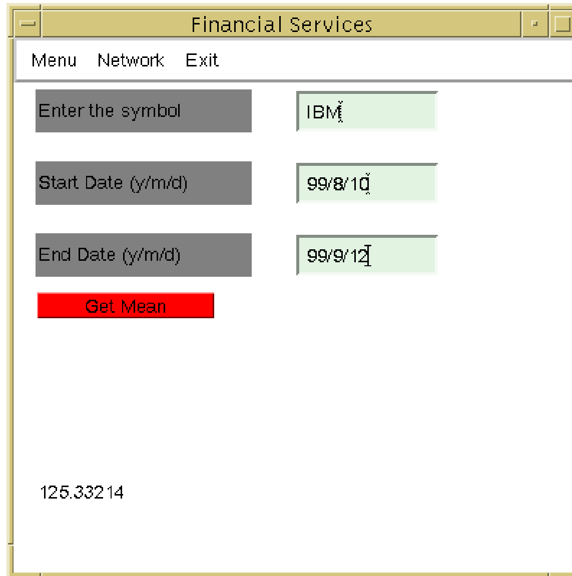

Figure 5: GUI of the Broker Agent for Experiment 2

bandwidth needed to transfer the data to the Mobile client would be high. It also found out that the computation agent is ready to provide a serialized version of its code to compute the average of stock quotes. Thus, the broker instructed the computation agent to provide a serialized code to it. On getting the code, the broker instructed the data provider agent to compute the result locally with the code and return the result. This way of scheduling thus prevented transmission of huge amount of stock data over the network and saves valuable bandwidth. On getting the result, the broker sent the result to the user on the mobile client.

2. In another example(Figure 7), we considered the Data Provider Agent to be having a memory of 10K available for external processes. The CPU usage was considered to be 0.8. Hence now, the Data Provider was overloaded. Similar attributes for the Computation Agent were 25K and 0.4. Thus even though the Computation Agent was mobile, the broker figured out from the XML description of the requirements of the Computation Agent that the serialized code required 12K memory to execute in any remote platform. This much of memory was not available at the local platform of the Data Provider. Also since the Data Provider was overloaded (as evident from its CPU usage), it would take a long response time to actually execute some serialized code in its local platform. Also since the data required was huge, it would anyway require lot of bandwidth. Thus it would be unwise to send the data and the executable component to the mobile platform over the 1.2Mbps link. Hence the broker decided to request the stock data from the Data Provider Agent and transmit it to the Computation Agent and get the result computed locally.

   Thus depending on the various attributes provided by the XML descriptions of the different agents, the broker is capable of intelligently deciding on a sequence of executions.

6 conclusions and future work

To realize the paradigm of pervasive computing, in addition to ubiquitous delivery of already available information, it is important to consider cases where the information required by mobile unit is not readily

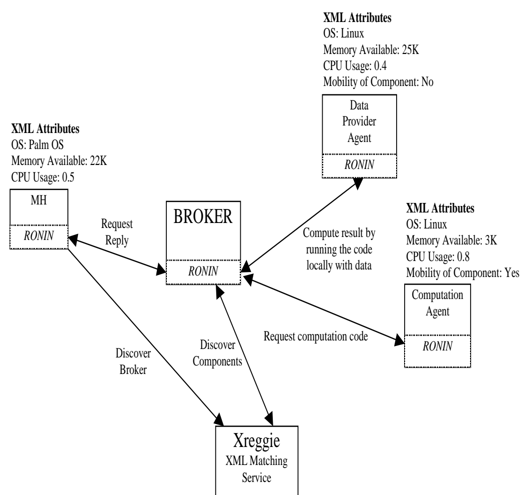

Figure 6: Decision Making by the Broker:Case 1

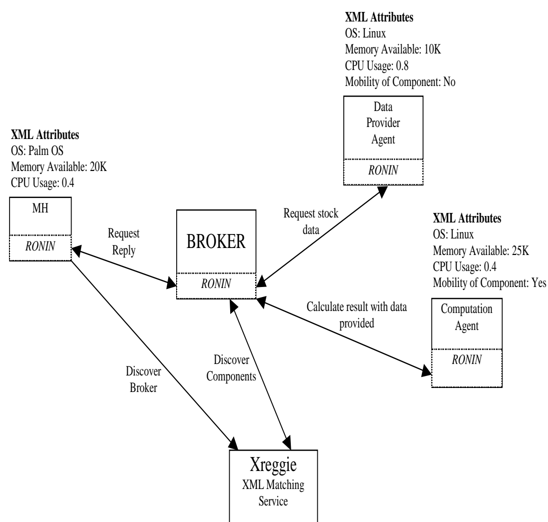

Figure 7: Decision Making by the Broker:Case 2

available, but needs to be computed dynamically. We have explored the difficulties in realizing such a scenario. Discovering services in a semantic way is a major hurdle in addressing the problem. In this paper, we have described the Xreggie system, which is an extension of the Jini based service discovery system. This system does a much more semantic level service discovery using XML instead of simple interface or attribute matching.

We have also explored the significance of defining an agent-based framework to comprehensively define a generic infrastructure to enable such collaboration and cooperation and dynamic computation of information. We have explained the Ronin Agent Framework, which helps us in defining services in an agent-oriented manner, thus enabling a broad range of applications and opening up a broad range of possibilities. The uniform communication infrastructure of the Ronin Framework helps us in communicating between heterogeneous agents without worrying about the actual communication layer being used.

An implementation of this architecture using the Ronin Agent Framework and the Xreggie system has been presented. We have presented a 'financial information system' that we have implemented using the system. The financial information system enables computation of different stock related requests. The answers to these queries are not available apriori in the network and cannot be computed by a single service in the network. Our 'financial broker' helps in collaborating amongst different financial services. It determines the different subtasks required to execute a query, finds the available services using the Xreggie system, computes the optimal path for the computation and manages the distribution of computation amongst different services. We have presented experiments showing the different decisions that our system is capable of making depending on the different queries made to the system, the constraints of the mobile platform and capabilities of the system on which the agents/services are running.

One point worth mentioning is that we have assumed XML to be a language which is capable of adequately describing the agents, services, system capabilities and constraints of platforms on which the services might run. The underlying assumption is that there are agreed upon standards to define such attributes and parameters. More than that, all the services/agents must agree on a common ontology to properly understand and interact with each other. Efforts at standardizing agent communication have been going on for quite some time[18, 12, 10, 8]. In the event where different agents follow different ontologies, the interaction becomes much difficult. The standard solution is to incorporate a translator agent. The agent-oriented architecture in our system enables plugging in such translator agents in future in order to facilitate communication between agents following different ontologies.

The broker that we implemented solves a set of problems within the financial services domain. In our implementation the broker has a mapping of the individual services required in solving the particular problem. We have implemented this mapping statically as a proof of concept. Clearly, one could think of a more horizontal solution where the broker would use complex AI features to find services required dynamically. Also knowledge recommender agent systems such as PYTHIA[15] could be used to intelligently manage the distribution of computation between mobile platform and the stationary machines. However, these systems use previously seen computations as a basis for predicting the execution profile of a computation about to be scheduled.

Currently, the system described in this paper addresses the problem of dynamic on-the-fly computation when the services are all present in the wired side of the network. Also, the presence of the broker is considered essential to the success of the system in delivering such information, which requires collaboration between multiple services. We are exploring the possibility of realizing such scenarios where the broker may not necessarily be available to the mobile host and the services/agents themselves might be residing in other mobile platforms.

References

[1] Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification*. Addison-Wesley, Reading, MA, USA, 1999.

[2] Harini Bharadvaj, A. Joshi, and Sansanee Auephanwiriyakyl. An active transcoding proxy to support mobile web access. In *Proc. IEEE Symposium on Reliable Distributed Systems*, October 1998.

[3] C. Brooks, M. S. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as http stream transducers. In *Proc. WWW-4, Boston, http://www.w3.org/pub/Conferences/WWW4/Papers/56Application-Specific*, May 1996.

[4] Harry Chen. Developing a Dynamic Distributed Intelligent Agent Framework Based on the Jini Architecture. Master's thesis, University of Maryland Baltimore County, January 2000.

[5] UC Berkeley Computer Science Division. `http://ninja.cs.berkeley.edu`.

[6] T. Drashansky, S. Weerawarana, A. Joshi, R. Weerasinghe, and E. Houstis. Software architecture of ubiquitous scientific computing environments. *ACM-Baltzer Journal on Mobile Networks and Applications*, 1, 1997.

[7] K. Erol, J. Hendler, and D. Nau. Htn planning: Complexity and expressivity. In *Proc. AAAI.*, 1994.

[8] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In Jeff Bradshaw, editor, *Software Agents*. MIT Press, 1997.

[9] Tim Finin, Anupama Potluri, Chelliah Thirunavukkarasu, Don McKay, and Robin McEntire. On agent domains, agent names and proxy agents. In Tim Finin and James Mayfield, editors, *Proceedings of the CIKM '95 Workshop on Intelligent Information Agents*, Baltimore, Maryland, 1995.

[10] T. Finin et. al. *Draft Specification of the KQML Agent-Communication Language*. DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1993.

[11] A. Fox, I. Goldberg, S.D. Gribble, D.C. Lee, A. Polito, and E.A. Brewer. Experience with top gun wingman: A proxy-based graphical web browser for the usr palmpilot. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, 1998.

[12] R. Fritzson et. al. KQML- A Language and Protocol for Knowledge and Information Exchange. In *Proc. 13th Intl. Distributed Artificial Intelligence Workshop*, July 1994.

[13] E. Guttman, C. Perkins, J. Veizades, and M. Day. "rfc2068: Service location protocol, version 2". ftp://ftp.isi.edu/in-notes/rfc2608.txt, 1999.

[14] Hewlett-Packard. *E-Speak Architectural Specification*, version beta 2.2 edition, December 1999.

[15] E. Houstis, S. Weerawarana, A. Joshi, and J. R. Rice. The PYTHIA project. In S. K. Aityan et al., editor, *Neural, Parallel, and Scientific Computations*, pages 215–218. Dynamic Pub., 1995.

[16] A. Joshi, S. Weerawarana, T. T. Drashansky, and E. N. Houstis. Sciencepad: An intelligent electronic notepad for ubiquitous scientific computing. In *Proc. of Intl. Conf. on Intelligent Information Management Systems*, pages 107–110, June 1995.

[17] Anupam Joshi. On proxy agents, mobility and web access. *ACM/Baltzer Journal of Mobile Networks and Applications*, 2000.

[18] K. Kuwabara, T. Ishida, and N Osato. AgenTalk: Coordination Protocol Description for Multiagent Systems . In *Proc. ICMAS*, 1995.

[19] DARPA Agent Markup Language. World Wide Web, `http://www.daml.org`.

[20] KVM HOme Page. World Wide Web, `http://java.sun.com/products/kvm`.

[21] Bluetooth White Paper. World Wide Web, `http://www.bluetooth.com/developer/whitepaper`.

[22] Rekesh John. UPnP, Jini and Salutaion - A look at some popular coordination framework for future network devices. Technical report, California Software Labs, 1999. Available online from.

[23] J. C. Schlimmer and L. A. Hermens. Software Agents: Completing Patterns and Constructing User Interfaces. *Journal of Artificial Intelligence Research*, 1(61-89), 1993.

[24] L. Z. Varga et. al. Integrating Intelligent Systems into a Cooperating Community for Electricity Distribution Management. *International Journal of Expert Systems with Applications*, 7(4), 1994.

[25] S. Weerawarana, A. Joshi, E. N. Houstis, and A. C. Catlin. Using NCSA Mosaic to build notebook interfaces for CS&E applications. *Concurrency: Practice and Experience*, 9(7), 1997.

[26] B. Zenel. *A Proxy Based Filtering Mechanism for The Mobile Environment.* PhD thesis, Department of Computer Science, Columbia University, N/A.