Dynamic Service Discovery for Mobile Computing: Intelligent Agents Meet Jini in the Aether [*]

Harry Chen
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
hchen4@cs.umbc.edu

Anupam Joshi
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
joshi@cs.umbc.edu

Tim Finin
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
finin@cs.umbc.edu

ABSTRACT
The emergence of ad-hoc pervasive connectivity for devices based on Bluetooth-like systems provides a new way to create applications for mobile systems. We seek to realize ubiquitous computing systems based on the cooperation of autonomous, dynamic and adaptive components (hardware as well as software) which are located in vicinity of one another. In this paper we present this vision. We also describe a prototype system we have developed that implements parts of this vision – in particular a system that combines agent oriented and service oriented approaches and provides dynamic service discovery. We point out why existing systems such as Jini are not suited for this task, and how our system improves on them.

General Terms
Design, Experimentation

Categories and Subject Descriptors
J.m [**Computer Applications**]: Miscellaneous

Keywords
service discovery, Jini, multi-agent system

1. INTRODUCTION
In the past year or two, the research community has seen plenty of hype associated with wirelesss, pervasive, mobile and ubiquitous computing. Mobile Commerce (M-Commerce) in particular was declared as the "killer app" driving the wireless revolution. Cell phones and wirelessly connected PDAs essentially became mobile storefronts for e-tailers. We are all familiar with the ads of people buying stuff via their cellphones from the beach. The drawbacks of this idea are not hard to identify, as an increasing number of recent critical commentaries show. This approach essentially replaces the desktop computers with the palmtop devices allowing users to order goods and access information in an anywhere and any time fashion. This client-proxy-server approach has been developed by the academia over the last five or six years in contexts such as web access from mobile platforms (for instance [2, 3, 12, 11, 13, 15, 17]) or transaction support for database access [8].

Variants of this approach are now emerging from several commercial companies in the form of systems that allow phone based "microbrowsers" or PDAs to access domain specific internet content, such as headline news, stocks, sports etc., from designated portals. The WAP consortium (`http://www.wap.com/`) is leading efforts among telecommunication companies and service providers to evolve a standard mechanism and markup language to support this. There have also been efforts, mostly from startups and e-tailers, to allow people to buy goods using the phone microbrowsers or PDAs. In some sense, one can think of this as a *supermarket approach*, where a few identified service providers exist and the traffic in services is one-way.

Viewed in a broader perspective however, M-Commerce in particular, and Mobile Computing in general, have yet to be fully articulated or explored, especially in the context of emerging mobile ad-hoc networks. Staying with the M-Commerce idea, but let's consider to move away from the prevailing mobile storefront vision. In the future, instead of just interacting with the "buy–it–yourself" web storefronts, consumers will be able to interact with service providers in a personalized way via automated service-oriented eMarket models. The selling of tangible goods will be one such service. Other services would include data, information, software components or even processing time/storage on networked machines. There will be no explicit clients and servers – but peers, often other persons/machines in ones vicinity, which can be both consumers and providers of dif-

[*]Paper published in Cluster Computing volume 4, issue 4 Special Issue on Internet Scalability: Advances in Parallel, Distributed, and Mobile Sytems

ferent services. M-Commerce will thus be transformed into what we refer to as *Me-Commerce*, or more generally, into *Me-Services* – a personalized view of services dynamically available from fixed providers on the wired side, as well as wirelessly connected peers in the vicinity.

In this paper we describe our initial research efforts to realize ubiquitous computing systems based on the cooperation of autonomous, self-describing, highly interactive and adaptive components that are located in "vicinity" of one another. In such dynamic environments, service discovery becomes a critical research problem. Hardware and software components will automatically become aware of each other and establish basic communication. They will also be able to discover and exchange services with each other. In Sections 2 and 3, we describe our vision of the future mobile computing environment. The next section 4 discusses the weakness in the current service discovery schemes, while section 5 evaluates the Jini service discovery mechanism in a mobile computing environment in light of these deficiencies. In Section 6 we describe the design goals and the architecture of the Ronin Agent Framework. In Section 7 we present a restaurant recommendation system called Agents2Go that we have developed based on the Ronin Framework. Our research findings are summarized in Section 8.

2. SOJOURNS INTO THE FUTURE

We envision our protagonist (Jane) driving down I95. A palmtop and cellular phone are in her purse, and the car has its own onboard computer. All these devices are networked using Bluetooth. The car's onboard computer connects with computers in nearby cars using Bluetooth, and exchanges information necessary to coordinate their driving in a Distributed Traffic Management Environment. Sometimes when the driver is not sure of things (where the nearest gas station with a car wash is, for example), the computer can ask the other cars around it. Every five miles or so down the highway, while the car is within range of an *electronic-sign* or a rest area, it is connected to broadband high speed wireless LAN. While driving, Jane receives a page on the palmtop alerting her that her office is sending her a fax. Her palmtop dials out on the phone to retrieve the fax and requests that it be converted to an audio stream that can be played on her car's sound system. The server cannot honor this request, so the palm activates a proxy agent when the car passes the next e-sign. This agent locates and negotiates with fax-to-audio conversion services in the vicinity to effect the conversion, and migrates to the next e-sign the car will pass. When the car is in range, the proxy agent streams the audio to the car. After listening, Jane decides that this is something she will need to read carefully. It is getting close to 7 in the evening, so Jane decides to call it a day. She tells her palmtop to book her a room at the next Chariott hotel and have the fax print there. She wants to have dinner at a Mexican restaurant before going to the hotel. At the next e-sign, the Palm device contacts the proxy agents for the local hotel/motel and restaurant associations. It negotiates to get Jane the best possible deal on the room she wants (non smoking, king). It also finds out that the hotel and the Mexican restaurant are at some distance, but that there is a Tex-Mex restaurant close to the hotel which serves Jane's favorite dish. The Tex-Mex restaurant also agrees to print her fax at no extra charge, and the agent reserves her a table, downloads directions into her car's computer and routes the fax to the restaurant's printer.

3. BACKGROUND AND RATIONALE

The basic assumption behind our work is that at the level of computing and networking hardware, we will see dramatic changes in the next few years. More specifically, we predict the emergence of (i) palmtop and wearable/embeddable computers, (ii) Bluetooth-like systems which will provide short range, moderate bandwidth connections at extremely low costs, and (iii) widely deployed, easily accessible wireless LANs and satellite WANs. We assume that there will be a mixture of traditional low bandwidth systems and next generation high speed ones. These developments will lead to wireless networks that will scale all the way from ad hoc local area networks to satellite WANs, and link together supercomputers, "walkstations" and embedded controllers. There is ongoing research in creating the hardware and low level networking protocols that are needed to realize this vision.

The industry's present vision of the use of Bluetooth-like devices is fairly narrow – essentially as point-to-point communicators to replace cables, similar to IrDA for computers and printers. However, both industry and academia have of late realized that these devices can also be used in creating ad hoc networks – e.g. when police and emergency personnel respond to a call, their computers would automatically get networked when they come in each others' vicinity. There is significant ongoing work in solving various network layer problems associated with mobile ad hoc systems. The point of departure for us is the creation of a software infrastructure that can actually exploit ubiquity arising from ad hoc networking, and enable a new class of applications.

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 problem of "service discovery"[1] has recently been explored in the context of distributed systems and elsewhere. State of the art systems such as Jini[1], Salutation[20], Universal Plug and Play (UPnP)[16], as well as IETF's draft Service Location Protocol[18, 23] provide for networked entities to advertise their functionality. Within these newly emerged distributed systems, our evaluation [6] shows that Jini provides a more flexible and robust service discovery infrastructure for building distributed components comparing to other systems.

The Jini discovery infrastructure supports both unicast and multicast service discovery protocol. This infrastructure allows services to be found in a uniform way, either on a local or a remote network. The service advertisement takes the form of interface descriptions. This simple form of the advertisement mechanism can be easily employed to provide high-level of abstractions for both software and hardware entities in the network. This is how a Jini-enabled printer can talked to a Jini-enabled digital camera. It is clear that the Jini discovery infrastructure provides a good base foundation for developing a system with distributed components need to discover each other in the network. However, the Jini discovery process is tightly bounded to the Java class

[1]We use the term service here in a broad sense

interface description advertisement. The simplicity of this mechanism is also the major weakness of the Jini service discovery architecture.

4. DEFECIENCIES IN EXISTING SERVICE DISCOVERY ARCHITECTURES

Architectures and systems like Service Location Protocol (SLP), Jini, Universal Plug and Play (UPnP) and Salutation have recently been developed to explore the service discovery issues in the context of distributed systems. While many of the architectures provide good base foundations for developing systems with distributed components in the network, we argue that they are not sufficient for building the *Me-Services* environment due to the following:

- **Lack of Rich Representations:**

  Services in the *Me–Services* world are heterogeneous in nature. These services are defined in terms of the their functionalities and capabilities. The functionality and capability descriptions of these services are used by the service clients to discover the desired services. The existing service discovery infrastructures lack expressive languages, representations and tools that are sufficient for representing a broad range of service descriptions and are also useful for reasoning about the functionalities and the capabilities of the services [4]. In the Jini architecture, service functionalities and capabilities are described in Java object interface types [1]. Service capability matchings are processed at the syntax-level only using objects.

- **Lack of Constraint Specification and Inexact Matching:**

  Most of the discovery protocols only allow service requests to match exactly with service descriptions, and have simplistic notion of contraints. For instance, the default registry that comes with Jini allows a service client to find a printing service that supports B/W printing, but will not return a color printing service if it can't find a B/W one. The protocols do exact syntactic matching while finding out a service. Thus they lack the power to give a close match even if it was available. Similarly, the protocol will allow finding a printer at a given location or with a given sized print queue, but the registry is not powerful enough to find a geographically closest printing service that has the shortest print queue.

- **Lack of Ontology Support:**

  Services need to interact with clients and other services in the vicinity. Service descriptions and information need to be understood and agreed among various parties. In another words, well-defined common ontology must be present before any effective service discovery process can take place in dynamic ad-hoc network systems.

  We found that common ontology infrastructures are often either missing from or not well represented in the existing service discovery architectures. Architectures like Service Location Protocol, Jini and Salutation do provide some sort of mechanisms to capture ontology among services. However, these mechanisms like Java class interfaces and ad-hoc data structures are unlikely to be widely adapted and become standards. In the Universal Play and Plug (UPnP) architecture, service descriptions are represented in XML (eXtensible Markup Language), which provides a good base foundation for developing extensible and well-formed ontology infrastructure [16]. However, service descriptions in UPnP does not play a role in the service discovery process [19]. Newer XML based semantic standards, such as RDF or DAML [7] are perhaps better suited for service description.

5. EVALUATING THE JINI SERVICE DISCOVERY IN A MOBILE ENVIRONMENT

Jini provides an infrastructure for providing services in a network, and for creating spontaneous interactions between programs use these services [1]. Services can join and leave the network in a robust fashion. The Jini service leasing mechanism allows clients to be well informed of the availablity of visible services.

Our studies show that the service discovery infrastructure is one of the shortcomings of Jini [6]. The weaknesses are the following: 1) it is difficult to discover services that require specific attribute value that are depended on the dynamic content of the environment, 2) it is difficult for cross-domain services to discover and interoperate when service descriptions are expressed in the object-level and syntax-level.

The existing Jini architecture requires service advertisements to be expressed in the form of Java interface descriptions. For example, a printing service may advertise itself as a generic printing service by implementing an interface called `Printer`. If the printing service wants to advertise itself as a color printer, then it may add an extra service attribute entry class called `TypeOfPrinter` with the string value `''Color Printer''`. A client that is looking for a color printer will ask the Jini Lookup Service to match for an advertised service that implements the `Printer` interface and has an attribute entry class `TypeOfPrinter` with the string value `''Color Printer''`.

When the Jini Lookup Service performs matching, the matching is done based on comparing the Java class interface types and the string values of the service attribute entries. This syntax level matching creates the obvious semantic interoperability problem. For example, two services are in the printer domain. One service implements an interface called `PrinterInterface`, and the other implements the `Printer` interface. Both of these have the same functionality. Yet if a client is looking for a printing service using the `Printer` interface, then the Jini Lookup Service would not match the service that uses the `PrinterInterface`. Similarly, when service descriptions are expressed in the object-level and syntax-level, it is difficult for cross-domain services to discover and interoperate with each other. For example, a digital camera client is looking for a printing service that is capable of printing 30 pictures with resolution 1024x768 pixels in 30 minutes. It will be difficult for the Jini Lookup Service to find a matching printing service if the printing service domain uses dots per inch (dpi) instead of pixels as the unit for measuring printing resolutions.

Another problem is that the Jini Lookup Service cannot perform inexact matching. For example, consider a printer service that has advertised an attribute entry `TypeOfPrinter` with value `''Color Printer''`. When a client is looking for a printer service that has an attribute entry `TypeOfPrinter` with value `''Black and White''`, it is intuitively obvious that the Color Printer should be returned as a match, especially if no B/W printers are found. However, the Jini Lookup Service is not capable of doing such type of reasoning.

Furthermore, due to the limitation of the Jini matching strategy, it is difficult to discover services that require specific dynamic attribute values. For example, a client can easily discover a color printer service, but it is difficult to discover a printer service that is geographically the closest, since the notion of a "least of all X" type constraint is not present in Jini's matching approach.

One way to enhance the Jini discovery mechanism is to re-implement or expand the existing Jini lookup process, for example, by adding XML support as a rich service description and lookup mechanism and by permitting fuzzy matches and complex constraints. We have created such as system called XReggie [14]. Another approach, which we will describe in detail in this paper, is to apply the agent-oriented design model to the Jini service designs. In particular, clients looking for services look to find broker agents using simple Jini type approaches, and can then negotiate with these brokers using richer representations. The brokers can also use significantly more "intelligence" in the matching process than Jini does.

6. RONIN AGENT FRAMEWORK

The Ronin Agent Framework introduces a hybrid approach to developing dynamic distributed systems based on the composition of the agent-oriented and service-oriented programming design. [4] The framework not only creates a flexible and robust development solution for creating multi-agent systems, but it also enhances the existing Jini service discovery, allowing services to be discovered based on the domain-independent agent attributes. The proxy-oriented Ronin Agent Deputy design allows remote hardware and software services to be used as if they were local services.

6.1 Ronin Design Goals

The Ronin Agent Framework is designed to aid the development of dynamic distributed/mobile systems, taking the advantages of the Jini architecture and the agent technology. The design goals of the framework are the following:

- **Enhancing the service discovery mechinsim:** By incorporating agent attributes into the service discovery process, agents should be able to find each other as well as domain specific brokers .
- **Making knowledge sharing possible:** The framework should provide an infrastructure that allows distributed components in a network to share and exchange knowledge easily.
- **Supporting InterAgent Communication using ACLs:** Agent Communication Languages (ACLs) [10, 9] such as KQML or FIPA-ACL provide a great communication infrastructure for heterogeneous agents to exchange content-rich knowledge about their environment. Using ACL communication, the protocols and the conversations among agents can be formally modeled and precisely defined.
- **Promoting distributed AI tools:** AI tools, such as knowledge representation systems, inference engines, logic programming tools, are very useful to the development of the future *Me–Services* environment, in particular in creating sophisticated service discovery systems. These powerful AI tools should be made available to the light-weight distributed components in the network.
- **Handling mobility related constraints:** Mobile networks present well known problems of low bandwidth and elective disconnections. The system should hide these problems as much as possible.
- **Enhancing the Jini service development process:** It is important not to create a solution that is more complicated than the original problem. The Ronin framework should enhance and simplify the deployment process, overcoming the weakness of the existing Jini development environment.

6.2 Ronin Architecture Overview

Ronin is written in Java. The framework is built on the existing Jini technology (Jini Technology Starter Kit, version 1.1 beta 2). The Ronin package provides an agent-oriented development abstraction to the existing service-oriented Jini development; enhances the Jini service development processing and provides a Jini Prolog Engine Service, which provides reasoning, knowledge representation and logic programming functionalities.

The key components of Ronin are the following: 1) Ronin Agent 2) Ronin Agent Deputy 3) Ronin Agent Attributes and Domain Attributes 4) Jini Prolog Engine Service (JPES). In the following subsections, we will describe these key components and their related components in detail. Specifically, we will concentrate our discussion on the Ronin agent architecture, the agent communication infrastructure, the agent description facility and the logic programming facility.

6.2.1 Ronin Agent

A Ronin agent is a metaphor for defining an integerated collection of Java classes and interfaces that funcations according to certain pre-defined agent behavior. It is not the class signatures that define a Ronin agent; it is the combined behavior of the classes that defines an agent.

A Ronin agent has two sets of behaviors: the generic agent behaviors and the domain specific behaviors. The generic agent behaviors are the actions that are common to all Ronin agents. For example, all Ronin agents need to advertise a set of agent attributes that describe their functionality and capability to the Lookup Service. On the other hand, the domain behaviors differ from one agent to another. These behaviors uniquely define an agent in the system. For example, the domain behavior of a Ronin agent that manages the printing service is very different from the domain behavior of a Ronin agent that provides restraurant recommendation services.

Ronin provides facilities (classes and interfaces) that define the generic agent behaviors of an agent. These facilities define the infrastructures for 1) the Ronin Agent Deputy realization 2) the agent capabilities and funcationality descriptions and 3) interagent communication. All Ronin agents are required to have at least one Agent Deputy. An Agent Deputy acts as a proxy for the Ronin agent to other agents in the network.

Ronin agents are required to provide descriptions about their capabilities and funcationalities using the Ronin Agent Description Facility. Agents are described in terms of their agent attributes (those attribute classes implement the `AgentAttribute` interface) and their domain attributes (those attribute classes implement the `DomainAttribute` interface). The descriptions of an agent are advertised to the Lookup Service the same way as any other Jini service entries.

Ronin agents communicate with other agents using an Agent Communication Language. However, Ronin does not have any restriction on the ACL that agents can use. Ronin leaves the decision of adapting the kinds of ACL standard to the agent developers.

In contrast to the generic agent behavior facilities that are specified and provided by Ronin, it does not specify how the agent domain behaviors should be implemented. Ronin leaves the decision of crafting agent domain behaviors to the agent developers.

6.2.2 Ronin Agent Deputy

A Ronin Agent Deputy is an object that acts as a proxy for the Ronin Agent in the network. The main duties of an Agent Deputy are 1) mediating agent communication on the behalf of the owner agent and 2) providing a local agent representation of the owner agent in the network. Figure 1 shows the interactions between two Ronin agents and the Agent Deputy.

A concrete Agent Deputy implementation is a serializable class that implements the `AgentDeputy` interface (see Figure 2). All of the concrete Agent Deputy implementation classes are required to confront to the functional specification of the `AgentDeputy` interface methods.

The funcational specification of the `deliveryMessage()` method defines that this method is to be invoked by a message-sending agent when it requests the Agent Deputy of the message-receiving agent to delivery a message to its owner agent. The message is wrapped within an `Envelope` object.

The message delivery scheme can be implemented either as a synchronous or an asynchronous process. For example, a synchronous delivery scheme can be implemented using socket communication procedures between the owner Ronin agent and the Agent Deputy. An asynchronous delivery scheme can be implemented as a set of Remote Method Invocation (RMI) calls [21], based on the Remote Event Model [1, 22] between the owner Ronin agent and the Agent Deputy.

```
public interface DeputyLocator {
   AgentDeputy getAgentDeputy();
}
```

Figure 3: `DeputyLocator` interface

The `AgentDeputy` interface class also provides an event notification mechanism that allows message-sending agents to receive notifications regards the status of individual message delivery process. (More details on the message delivery notification mechanism can be found in [4].)

The Ronin framework provides two abstract Agent Deputy implementation classes, `AbstractDeputy` and `AbstractAdminDeputy`. Both classes implement the `AgentDeputy` interface. The `AbstractAdminDeputy` class extends the `AbstractDeputy` class and implements the `net.jini.admin.Administratable` interface, which provides common ways to export particular administrative funcationality. [1]

The `AbstractDeputy` class provides a basic implementation of the `AgentDeputy` interface. In addition it defines a flexible design pattern that allows agent developers to customize the transport mechanism for delivering the `Envelope` object from the Agent Deputy to the owner agent. The transport mechanism that is used by an Agent Deputy is encapsulated within the logic of the `Transport` object implementation (see Figure 4). Every instance of the `AbstractDeputy` class contains an `Transport` object. An agent developer can configure the `AbstractDeputy` class to use any `Transport` implementation of his/her choice. This achieves the goal of creating Agent Deputies with functionality that is independent from the low-level message delivery mechanism.

The benefits of using this particular pattern can be observed from the the following example. At runtime when the `deliverMessage()` method is invoked on the `AbtractAgentDeputy` object, the deputy delegates the message transport task to the contained `Transport` object. The `Transport` object might decide the appropriate network communication (using RMI, network socket calls etc.) and the protocol (HTTP, HTTPS etc.) to use to deliver the message. It might also decide to provide more sophisticated message delivery managements, such as message filtering, message store-and-forward, message multiplexing and message demultiplexing [1, 22].

There are two possible ways to locate an Agent Deputy. One way is to locate an Agent Deputy through the use of the Jini Lookup and Discovery Protocol, and then to download the Agent Deputy from the Jini Lookup Service. The second way is to extract an Agent Deputy by calling the `getAgentDeputy()` method on the `DeputyLocator` object. This object can be extracted from the `Envelope` object by invoking the `getSenderDeputyLocator()` method.

The `Envelope` object is a container class that contains the message that is to be delivered to the message-receiving Ronin agent. In addition it also has a reference to a `DeputyLocator` object that can be used to extract the `AgentDeputy` object of the message-sending agent.

The `DeputyLocator` interface has only one method (see Fig-

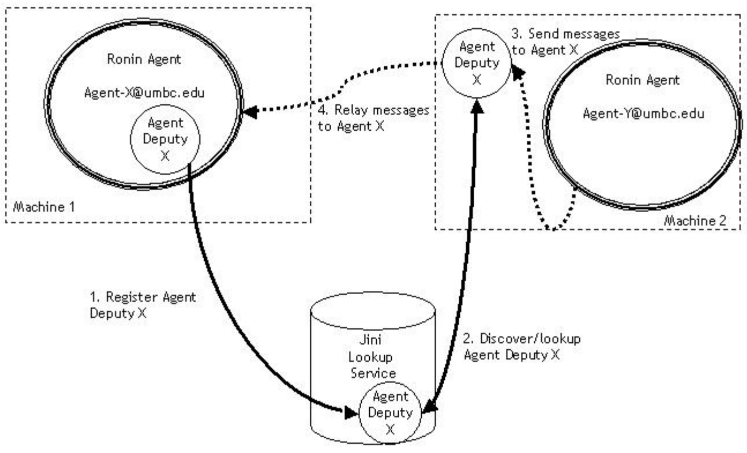


Figure 1: The interactions between the Ronin agents and Agent Deputy

```
public interface AgentDeputy {
   public void deliveryMessage(Envelope e, boolean wait);

   public void addDeliveryNotificationListener(DeliveryNotificationListner
                                                listener);

   public void removeDeliveryNotificationListener(DeliveryNotificationListener
                                                   listener);
}
```

Figure 2: `AgentDeputy` **interface**

```
public abstract class AbstractDeputy implements AgentDeputy, Serializable {
   protected Transport transport;
   protected EventListenerList nofitificationListners;

   public AbstractDeputy(Transport transport){...}
   public void addDeliveryNotificationListener(DeliveryNotificationListener
                                                listner){...};
   public void fireDeliveryNotificationEvent(DeliveryNotificationEvent e){...}
   public Transport getTransport(){...}
   public void removeDeliveryNotificationListener(DeliveryNotificationListener
                                                   listener){...}
}
```

Figure 4: `AbstractDeputy` **abstract class**

ure 3), `getAgentDeputy()`, which returns an instance of the `AgentDeputy` class. The `AbstractDeputy` class does not define the actual implementation of this method. The decision of how to extract an `AgentDeputy` instance is left to the agent developers to decide. The goal of such design is to allow an individual developer to create his/her own solution to the problem of returning a reference to the `AgentDeputy` object to the requestor.

In some systems, it might be desirable for the developer to create a `DeputyLocator` that contains a direct reference to a serialized `AgentDeputy` object. When the `getAgentDeputy()` method is invoked, the reference to the `AgentDeputy` object is simply returned. On the other hand, sometimes it might be desirable for the developer to create a `DeputyLocator` that dynamically discovers or downloads an `AgentDeputy` from the network when the `getAgentDeputy()` method is invoked.

6.2.3 Ronin Agent Attributes and Domain Attributes

The Ronin Agent Attributes provide a basic agent ontology for Ronin agents. These attributes describe the capability and the functionality of an agent in a domain-independent fashion.

The Ronin Agent Attributes consist of five Java classes: `AgentID`, `AgentLang`, `AgentOntology`, `AgentOwner` and `AgentRole`. All of these classes extend `the net.jini.entry.AbstractEntry` from the Jini Technology Core Platform and implement the `AgentAttribute` interface from the Ronin Agent Framework. Figure 5 shows the class diagram of the Ronin Agent Attribute classes.

Ronin defines the semantic meanings of the Ronin Agent Attributes. The `AgentID` attribute contains a unique agent identifier. The format of this identifier is expressed in the email address format. For example, `agent-name@domain.com`.

The `AgentLang` attribute specifies the Agent Communication Language that one agent can understand. This attribute is defined in terms of three string fields: `format`, `language` and `ontology`. The `format` field defines the encoding scheme of the Agent Communication Language. For example, the scheme might simply be the `java.lang.String`. The `language` field specifies the name of the Agent Communication Language (for example, KQML, FIPA etc.). The `ontology` field is an ontology identifier of the Agent Communication Language (for example, `AGENTS2GO-ONT`).

The `AgentOntology` attribute contains the ontology identifer of the Ronin agent. The ontology of the agent is predefined prior to the development of the agent. This attribute is mainly used in the agent lookup process by the Jini Lookup Service.

The `AgentOwner` attribute specifies the name of the institution or the organization that owns the agent. This attribute is defined as a single string field, namely "owner".

The `AgentRole` attribute specifies the acting role of the agent in the system. A Ronin agent can take on one of the six possible acting roles: `BROKER`, `MATCHER`, `MEDIATOR`, `PROBLEM SOLVER`, `PROXY` or `DEFAULT`. The `DEFAULT` role is assumed to be the default role of an agent if no other role values is supplied.

```
public interface JPEServer extends java.rmi.Remote{

   public void loadSourceFile(java.net.URL url)
     throws java.rmi.RemoteException;

   public JPESResult query(JPESQuery query)
     throws java.rmi.RemoteException;

   public JPESResult[] queryAll(JPESQuery query)
     throws java.rmi.RemoteException;

   public JPESResult queryCutFail(JPESQuery query)
     throws java.rmi.RemoteException;
}
```

Figure 6: `JPEServer` interface

These agent attributes are required to be supplied by the individual Ronin agent. When Ronin agents come alive, they register these agent attributes as the Jini service attributes with the Jini Lookup Service.

6.2.4 Jini Prolog Engine Service

Jini Prolog Engine Service [5] is a Jini service that provides the remote Prolog engine accesses to Jini-enabled components in the network. JPES provides a simple solution to bring inference engine and knowledge base facilities to the distributed system.

Like other Jini services, JPES defines a generic Prolog engine service interface, which allows users to perform Prolog operations without caring about the underlying implementation and the location of the actual Prolog engine. It also allows users to load predefined prolog statements from a URL.

The current version of JPES uses the SICStus Prolog implementation as its core Prolog engine. However, the design of the JPES allows the replacement of the core Prolog engine with any other Prolog implementations without affacting the JPES client implementations The main reason we choose to use the SICStus Prolog implementation as the core Prolog engine is because the SICStus Prolog implementation comes with a simple Java JNI API library, namely Jasper, that allows Java programs to access the Prolog engine.

Figure 6 shows the `JPEServer` interface class that are used by the JPES clients to access the remote Prolog engine. The `loadSourceFile` method allows the service client to load a set of predefined Prolog statments into the Prolog engine from a URL. The `query`, `queryAll` and `queryCutFail` methods are provided to the clients to make queries to the Prolog engine.

One of the advantages of using JPES with Ronin agents is that it eliminates the requirement of the existence of heavy-weighted Prolog engines on the physical machines where the

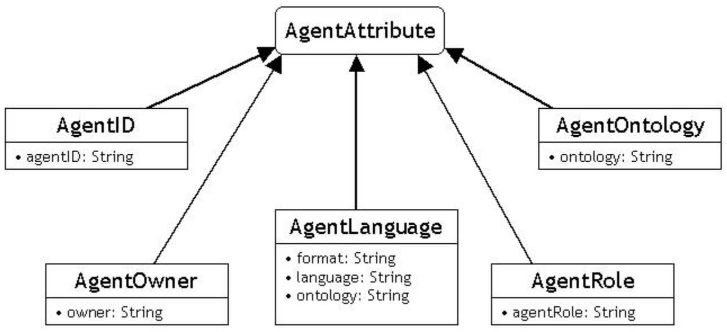


Figure 5: The class diagram of the Agent Attribute classes

agents are executed. In addition, sharing one JPES service between a group of agents provides a simple way for agents to share knowledge.

7. AGENTS2GO: A SMART RESTAURANT RECOMMENDATION SYSTEM

We have defined a futurist scenario to demonstrate the feasibility of using the Ronin Agent Framework to support mobile computing, called Agents2Go.

The Agents2Go project strives to create a smart restaurant recommendation system in the future mobile computing environment. A typical Agents2Go usage scenario can be described as the following: You are on a business trip from a foregin country visiting Baltimore. During the lunch hour, you are walking down the street looking for a restaurant. Since you are not familiar with the Baltimore area and do not have a traveler's guide to the local restaurants, you turn on your PDA or cell phone and ask it to help you to find a restaurant. Given that your device is connected to the local network wirelessly, your agent that lives on the device finds one or more local restaurant recommendation services, and interacts with these services. It provides them with your preferences regarding cuisines, costs, wait times etc., and returns back to you with a list of recommended restaurants. You can then makes reservation at one of the restaurats on the list.

7.1 The Existing Restaurant Recommendation System

At first it might seem to be trivial to realize such scenario. In fact a number of Yellow Page like portal services are available on the Internet today (e.g. zagat.com) that provides similar restaurant recommendation services. Some of these also allow users to make requests through WAP-enabled cell phones and wireless PDAs (e.g. citysearch.com and restaurantrow.com).

However, the restaurant information that is provided by most of the existing services is stored in centralized databases. Further, they provide only static attributes, such as a restaurant's location, cuisine etc. Clients access the information by communicating with centralized server applications. Such systems cannot easily make use of the dynamic location-depedent information as a part of the recommendation process. For example, it is difficult for these services to consider the current waiting times of the restaurants, or the current traffic conditions, or any specials/discounts the restaurants may be offering, when making the recommendations to users.

7.2 Agents2Go Architecture Overview

The Agents2Go restaurant recommendation system is built on top of the Ronin. Agents2Go provides a dynamic and personalized system that is capable of making recommendations to users based on the dynamic location-dependent information.

At its core, the Agents2Go recommendation system is not a centralized server application. Instead it takes a distributed approach, the Agents2Go system is built based on a collection of distributed restaurant agents, restaurant recommendation broker agents and personal agents. Agents are realized using the Ronin Agent Framework. When agents come alive, they discover the local Lookup Services and register their Agent Deputies. Taking the advantages of the Jini Discovery and Lookup Protocols, these agents are capable of dynamically joining and leaving the system in a flexible and robust fashion.

Figure 7 shows an architecture diagram of the Agents2Go system. The restaurant agents represent the restaurants in the community. These agents are responsible for providing both static and dynamic restaurants information to the personal agents and the local restaurant recommendation broker agents. The restaurant agents provide information to other agents by extracting data from their local SQL databases. With this approach, dynamic information of the restaurants such as the current waiting time and the chef's spcial of the day can be updated in the local SQL database. New information is made available to agents in system instantaneously. The tasks that are currently assigned to the restaurant agents are 1) maintaining and updating the restaurant databases, 2) providing information about the restaurants when query requests are received from the recommendation broker agents, 3) providing reservation services to the personal agents and 4) discovering new Jini Lookup Services and registering the Agent Deputies of the restaurant agents with the Jini Lookup Services.

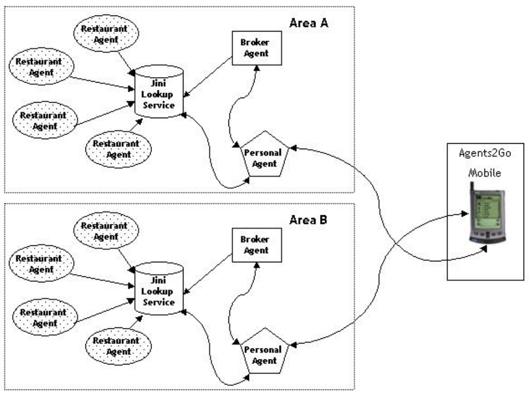


Figure 7: Agents2Go architecture

The recommendation broker agents are responsible for making restaurant recommendations to the human users. The broker agents periodically discovers the existence of the local restaurant agents that are within their contact range. (In the current implementation, the broker agent only pulls the restaurant agents that are registered with the Lookup Service.) The broker agents share the set of restaurant recommendation ontology with the local restaurant agents. When a broker agent is asked to make a recommendation by a users personal agent, but it does not have sufficent information in knowledge base about the local restaurants, it queries the local restaurant agents to retrieve updated information about the restaurants. For example, in the current implementation, the query action is taken by the broker agent when the waiting time of the local restaurants are out of date.

The Agents2Go system is targeted to work with users who have handheld devices that are capable of executing Jini-enabled Java applications. However, due to the computation resource limitation of the existing handheld devices like Palm(III, V, VII), the current mobile application that executes on the Palm device relies on a Ronin agent acting as a proxy that lives in a computer on the wired side. The mobile application on the palm is responsible for rendering GUI display, taking input from the user and mediating the communication between the user and the personal agent.

7.3 Developing the Restaurant Agents

In the current implementation, a restaurant agent consists of the following components: 1) a connnection handle to the database of the restaurant it represents, 2) an agent logic/behavior component, which contains logic for the agent to interact with others in the system, 3) a set of Agent Attributes and Restaurant Domain Attributes, 4) an Agent Deputy, which acts a proxy for the agent in the Ronin agent world.

When a restaurant agent comes alive, it first creats an instance of the Agent Deputy, then initializes its Agent Attributes and Restaurant Domain Attributes accorrding to the restaurant attribute definitions that are given in the agent configuration file. Once the Agent Deputy is properly configured, the agent executes the Jini Discovery Protocol trying to find Jini Lookup Services in the local community. If a Jini Lookup Service is discovered, the restaurant agent registers the Agent Deputy with the Jini Lookup Service along with the Agent Attributes and the Restaurant Domain Attributes. These attributes are registered as the Jini service entries. If the registration process succeed, then the restaurant agent goes into a waiting mode – waiting to receive ACL messages from either the restaurant broker agents or the personal agents.

The restarurant agents can be thought as the representatives of the restaurants in the Agents2Go world. The current design that distributes the individual restaurant information among agents has many advantages over the design of storing restaurant information in a centralized server application. With the distributed agent approach, individual restaurant agent only has to manage a relatively small set of information that is only related the restaurant(s) that it represents. Furthermore, this information can be stored and structured in any way that the restaurant's system desires. As long as all agents in the system share the same ontology and the ACL messages schema, any changes that are made to the internal restaurant systems would not affact the overall system. This reduces the maintainence overheads that are often encounted by the centrailized database-driven systems.

7.4 Developing the Restaurant Recommendation Broker Agents

In the current implementation, a broker agent consists of the following components: 1) a proxy object of a JPES service, 2) an agent logic/behavior component, which contains logics for the agent to interact with others in the system, 3) a set of Agent Attributes and Restaurant Domain Attributes that identifies the agent as a broker agent, 4) an Agent Deputy, which acts a proxy for the agent in the Ronin agent world.

The broker agent uses the Prolog engine that is provided by the JPES as its inference engine and knowledge base. During the initialization stage, the broker agent instructs the JPES service to load a set of restaurant recommendation Prolog rules from a remote URL to setup its inference engine and knowledge base.

During the recommendation stage, when the broker agent is requested by the personal agents to provide restaurant recommendations, the broker agent first converts the incoming request into a set of Prolog facts and rules expressing the preferences (e.g. cuisine is Japanese) and constraints (e.g. cost $<$ \$15) of the user, and then inserts these into the JPES inference engine. Once all rules and facts have been inserted, the broker agent executes the recommendation rules to seek for restaurants that have attributes that satisfy the request contraints. Thus a more intelligent match can be done compared to Jini's limited syntactic match.

At any stage if information about certain restaurants is either missing or out of date, the broker agent will send out request for information to the restaurant agents to update its knowledge base. Once requested information are received, the broker agent converts the information into Prolog facts and inserts them into its knowledge base using the JPES service.

7.5 Developing the the Personal Agent

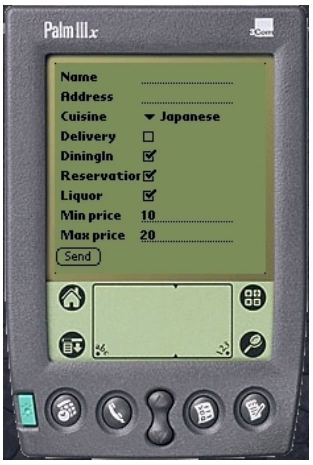
Ideally a personal agent should live on the handheld device of the user, so that personalized services can be constantly available to the user at all time. When a user has activated the personal agent, it dynamically discovers the broker agents in the system. Then the recommendation process begins.

However, due to the computation limitation of the existing handheld devices, the current implementation of the personal agent is seperated into two parts, a mobile application part (Agents2Go Mobile) that executes on the handlehand Palm device and a Ronin personal agents proxy that executes on the wired side. These two components cooperate with each other to fulfill the tasks of the personal agent.

The Agents2Go Mobile is a program that executes on a Palm device that is connected via a CDPD network (see Figure 8). This Palm application is implemented in the C programming language. It is responsible for rendering graphic display on the Palm device and interacting with the user. It is also responsible for transmitting user input requests to the Ronin personal agent on the local network machine. When the Agents2Go Mobile is activated, it makes a wireless network connection with the Ronin personal agent through the OmniSky wireless network. The contact information of the Ronin personal agent on the network is pre-configured as a part of the Agents2Go Mobile program preferrences.

If the connection is made successfully, then the Agents2Go Mobile downloads the restaurant recommendation form data from the the personal agent and renders the form on the screen. This form provides the data model for the Agents2Go Mobile to display the form. The data model is simply expressed as key-value pairs in strings. This allows different brokers to potentially provide different forms that are tailor-made to the recommendation service they provide. For example, the broker would provide only those cuisine types as choices that it knew corresponded to local restaurants.

Before any recommandation request is made, the user first authenticates himself/herself with the Agents2Go Mobile. If the authentication process succeed, then the Agents2Go Mobile instructs the running personal agent to start the Jini discovery process to find an local restaurant broker agent in



Name	
Address	
Cuisine	▾ Japanese
Delivery	☐
DiningIn	✓
Reservatior	✓
Liquor	✓
Min price	10
Max price	20

Figure 8: A snapshot of the Agents2Go Mobile Palm Application

the community. From this point on, any requests that are submitted to the Agents2Go Mobile from the user is directly delivered to the personal agent.

After a personal agent has received a discovery request from the Agents2Go Mobile over the network, it starts the standard Jini Discovery and Lookup Protocols. It first tries to discover a close-by Jini Lookup Service, and then it tries to lookup a restaurant recommendation agent using the Ronin Agent Attributes and the Restaurant Domain Attributes.

If there is an appending recommendation request from the Agents2Go Mobile, the personal agent starts a restaurant recommendation ACL conversation with the available recommendation broker agent. It then returns the recommendation results back to the Agents2Go Mobile. All ACL communication messages are expessed in KQML, and the content messages are expressed in Prolog-like rules.

8. CONCLUSION

Computing is no longer a discrete activity bound to a desktop; mobile computing is fast becoming a part of everyday life. The emerging mobile networking technologies will create a world where persons and devices will wirelessly interact with other devices all around them, in addition to interacting with tethered networked entities. In order to develop a mobile e-services system which will use such networks, these hardware/software components need to be aware of others components in their vicinity, and interact and interoperate with them.

The Ronin Agent Framework, which we present in this paper, introduces a hybrid architecture, a composition of service-oriented and agent-oriented architectures, that provides a simple and uniform scheme for deploying highly dynamic distributed "intelligent" components in a mobile world that easily discover and communicate with others components. We also present the Agents2Go system which uses Ronin to create a restaurant recommender system for the mobile scenario that provides for discovery and intelligent matching in a dynamic environment.

9. ACKNOWLEDGEMENTS

This work was funded in part by NSF awards IIS 9875433 and CCR 0070802, and an award from DARPA under the DAML program.

10. REFERENCES

[1] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini specification.* Addison-Wesley, Reading, MA, USA, 1999.

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

[3] E. Brewer, R. Katz, Y. Chawathe, A. Fox, S. Gribble, T. Hodes, G. Nguyen, T. Henderson, E. Amir, H. Balakrishnan, A. Fox, V. Padmanabhan, and S. Seshan. A network architecture for heterogeneous mobile computing. *IEEE Personal Communications Magazine*, 5(5):8–24, 1998.

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

[5] H. Chen. Jini prolog engine service (jpes), 2000. Available online from `http://gentoo.cs.umbc.edu/jpes/`.

[6] H. Chen, D. Chakraborty, et al. Service discovery in the future electronic market. In *Proc. Workshop in Knowledge Based Electronic Market*. AAAI, AAAI Press, 2000.

[7] J. Davis. The semantic web. *Daily Insight, INSIGHTS Online News*, Sep. 2000. `http://www.business2.com/content/insights/dailyinsights/2000/09/27/1996%9`.

[8] M. Dunham, A. Helal, and S. Balakrishnan. A mobile transaction model that captures both the data and movement behavior. *ACM/Baltzer Journal of Mobile Networks and Applications*, 2(2):149–162, 1997.

[9] T. Finin, Y. Labrou, and J. Mayfield. Kqml as an agent communication language. In J. Bradshaw, editor, *Software Agents*. MIT Press, 1997.

[10] FIPA, Geneva, Switzerland. *FIPA ACL Message Structure Specification*, edition 2000/08/01 edition, Aug. 2000.

[11] R. 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 `http://www.cswl.com/whitepaper/tech/upnp.html`.

[12] A. Joshi. On proxy agents, mobility and web access. *ACM/Baltzer Journal of Mobile Networks and Applications*, 2000. (accepted for publication, also availbe as UMBC CS TR 99-02).

[13] A. Joshi, S. Weerawarana, and E. N. Houstis. Disconnected Browsing of Distributed Information. In *Proc. Seventh IEEE Intl. Workshop on Research Issues in Data Engineering*, pages 101–108. IEEE, April 1997.

[14] R. H. Katz, E. A. Brewer, E. Amir, H. Balakrishnan, A. Fox, S. Gribble, T. Hodes, D. Jiang, G. T. Nguyen, V. Padmanabhan, and M. Stemm. The bay area research wireless access network (barwan). In *Proceedings Spring COMPCON Conference*, 1996.

[15] M. Liljeberg, M. Kojo, and K. Raatikainen. Enhanced services for world-wide web in mobile wan environment. http://www.cs.Helsinki.FI/research/mowgli/mowgli-papers.html, 1996.

[16] Microsoft Corporation. *Universal Plug and Play Device Architecture Reference Specification*, version 0.9 edition, 1999.

[17] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R.Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles.*

[18] C. Perkins. Rfc 2002: Ip mobility support. Technical report, IBM, 1996.

[19] The Salutation Consortium Inc. *Salutation Architecture Specification (Part-1)*, version 2.1 edition, 1999.

[20] Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303, USA. *Java Remote Method Invocation Specification*, revision 1.5, jdk 1.2 edition, Oct. 1998.

[21] Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303, USA. *Jini Distributed Event Specification*, revision 1.0 edition, Jan. 1999.

[22] SVRLOC Working Group. *SLP White Paper.* SVRLOC Working Group, May 1997.

[23] L. Xu. Using Jini and XML to build a component based distributed system. Technical report, University of Maryland Baltimore County, 2000.