Allia: Alliance-based Service Discovery for Ad-Hoc Environments

Olga Ratsimor
Department of Computer Science and Electrical Engineering,
University of Maryland Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
oratsi2@cs.umbc.edu

Dipanjan Chakraborty
Department of Computer Science and Electrical Engineering,
University of Maryland Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
dchakr1@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

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

ABSTRACT

Static directory based service discovery is unsuitable for m-commerce in ad-hoc environments. In this paper, we present Allia: a peer-to-peer caching based and policy-driven agent-service discovery framework to facilitate cross-platform service discovery in ad-hoc environments for mobile electronic commerce applications. Our approach removes the problems associated with structured compound formation of agent communities in mobile commerce environment and achieves high degree of flexibility in adapting itself to the changes of the ad-hoc environment. Our framework takes into consideration device capabilities and limitations, user preferences regarding usage of the devices, application specifics with respect to mobile commerce and adapts accordingly. We also present an initial implementation of Allia over ThinkPads and iPAQs by extending the LEAP Agent Platform and using Bluetooth as the underlying network protocol.

Categories and Subject Descriptors

H.4.3 [**Communications Applications**]: Information browsing;
H.3.3 [**Information Search and Retrieval**]: Retrieval models, Search process, Selection process.

General Terms

Algorithms, Design, Experimentation.

Keywords

Mobile service discovery, advertising, caching, device profiles, device preferences, agents.

WMC'02, September 28, 2002, Atlanta, Georgia, USA.

1. INTRODUCTION

Recent years have witnessed a tremendous growth in the field of mobile computing, software agents and electronic commerce. E-commerce systems often use the agent oriented abstraction in viewing their components and the interaction between clients and servers. Of late, there has also been the push to make e-commerce systems accessible from wirelessly connected thin clients. In both academic and leading edge industry research, often the connectivity comes from not the traditional infrastructure based wireless networks (e.g. cell phones), but from short range ad-hoc networks such as Bluetooth (or even 802.11). The merger of these three fields has created a new class of problems that must be solved in order to make commerce in pervasive computing environments a reality.

As in the more traditional wired environments, one such problem is *service discovery*. For wired networks, many service discovery techniques have been proposed (e.g. Jini, UDDI, SLP). Agent platforms used in e-commerce [5, 4, 19] have also been designed to facilitate flexible service/agent discovery within an agent community. In all these platforms, an agent that belongs to a certain platform registers itself and its services to some service/agent management/registration component like FIPA *Directory Facilitator* (*DF*) and *Agent Management System* (*AMS*)[19]. Agents searching for services query this component (*DF*/*AMS*) to discover other agents/services. These manager entities (*DF* and *AMS*) are assumed to reside on relatively powerful high-end static machines and the communication links between agents and these components (*AMS*/*DF*) are assumed to be wired or stable. Most service discovery mechanisms for wired networks (e.g. UDDI, Jini) make very similar assumptions about some kind of a directory server, often centralized.

Ad-hoc networks, a key component of future mobile electronic commerce environments, further compound the problem of discovering services, since they can no longer be discovered using a static platform structure as discussed above. This is because all the machines/nodes hosting services or directory are mobile and hence can move out of the vicinity at any time. Moreover, mobile devices may be extremely resource-poor and never be able to act as directory servers. We now bring out some recent efforts in the agents community to address such problems. Recently, the Foundation for Intelligent Physical Agents (FIPA), which is an industry led standards body in the agents area, introduced the idea of forming agent compounds between FIPA agents operating in ad-hoc environments. An agent in such an architecture could be part of a fragment (part of a platform) or a compound (a platform formed between multiple mobile nodes where one of the nodes that is resource-rich enough would host an *DF*/*AMS*). The main aim behind compound formation between agents is to facilitate flexible agent interoperability and discovery of other agent-services in such an environment.

An obvious solution for forming compounds is the one in which some node in the ad-hoc environment that is resource-rich maintains information about services and agents present in it's neighborhood. All other peer nodes would utilize the services of this resource-rich node in discovering services. This resource-rich node would provide the mandatory *AMS* and *DF* services, thus acting as an agent platform. Such an approach is probably adequate for the near term m-commerce systems. For instance, a fixed computer in a strip mall, or even a vending machine, could provide the resource rich platform that will host the directory server (or the *AMS*/*DF* servers). However, we anticipate that m-commerce in the future will also have a peer-to-peer component. This could be as simple as people exchanging MP3s from their players, to more complex transactions that involve payments and commitments. In other words, the m-commerce system of today, which is essentially a case of few service providers and many consumers (essentially a supermarket), will get transformed to a bazaar, where many devices in our vicinity will provide a variety of services. In such environments, there is an overhead associated in "electing" a resource-rich node to host an agent platform. Moreover, a new device moving into this structured compound would have to discover this agent platform and register it's services with it. Another major drawback with this approach is that if the resource-rich node moves out of the network, all other peer nodes would have to reconfigure themselves to the peer-to-peer mode. Highly dynamic environments where nodes keep moving in and out of the network would further aggravate this problem. For enabling service and agent discovery across multiple hops, "leaders" in different compounds have to federate with each other. In the context of ad-hoc networks, this would require those "leaders" to be in direct connectivity range of each other and this might not always be the case.

1.1 Design Overview

In this paper, we introduce a different approach towards handling the problem. We introduce the concept of peer-to-peer caching of services between nodes in an ad-hoc environment. The main goal of platform formation is to provide an agent better access to services in the vicinity (to facilitate interoperability). Our solution describes a policy-based distributed architecture towards achieving best-effort service discovery in ad-hoc networks for mobile commerce environment.

We envision each participating device will be able to run a lightweight version of the platform components like yellow pages and white pages service. Putting an agent platform on extremely resource-constrained devices might overload such devices. It is essential for each device to have a set of bare minimum platform components like *AMS* and *DF* so that it can host an agent platform and operate even in regions of disconnections or stay independent. An agent in that case does not have to depend on other peer devices to host itself. The *Main Container* of LEAP[5] that contains the *AMS* and the *DF* runs on standard laptops and 3870 iPAQs. Looking at other devices with lesser resources and smaller footprints, we see that they have the capability to host advanced runtime environments (like java virtual machine) and their memory capacity is increasing. For example, most i-mode phones support Java Micro Edition[12], Mobile Information Device Profile[13] or Connected Limited Device Configuration[9] and have capability to store around 300K of static memory space. Some phones have a run time heap of 180K (e.g. Samsung x350 series). While this is less than what the current version of LEAP on Personal Java[10] requires (around 700K of dynamic memory), it is well within the limits that will be feasible on even cell phones shortly considering the rate at which the capabilities of these phones are increasing. Cell phones can be considered one of the least resource-full devices amongst the plethora of heterogeneous mobile devices existing around us. In addition to these platform components, the device will be able to host at least one agent. The main purpose behind formation of compounds was to enable agents on a device to better utilize services/agents in its vicinity. Our approach concentrates on providing a solution to this issue without imposing the dependence that devices sharing a distributed compound will have on each other. The yellow pages service component (*DF*) registers only the services that are hosted on that local platform. The white pages service component (*AMS*) registers only the agents that are hosted on the local platform. Each device has a policy that reflects the device capabilities, user preferences, application specific settings, etc. The policy governs the way this device is going to present itself to the other platforms/devices in its vicinity. It also describes the way in which the device takes advantage of resources/services in other platforms in its vicinity.

In our approach, every node advertises its services to other nodes in its vicinity in accordance with the local policy. These advertisements are broadcasted. On receiving an advertisement, an agent decides based on its policy, whether to cache it or reject it. We introduce the concept of the "alliance" of a node. An *Alliance* of a particular node is a set of nodes whose local service information is cached by this node. Thus, a node explicitly knows the member nodes in its *alliance*. However, each node does not know the alliances in which they are members. Whenever a node leaves a certain vicinity, and enters a new vicinity, it constructs its own *alliance* by listening to advertisements. It also becomes a member of other *alliances* by advertising its local services. Its exit from other *alliances* is passive since the nodes governing these *alliances* detect its absence and remove the node from their *alliances*.

The local policy of a node dictates the ways in which the node wants to advertise itself to peer nodes in its vicinity. The rate of advertisement is also controlled by the policy. The policy can specify algorithms to allow dynamic adjustments of advertisement rates based on mobility of nodes in a neighborhood. Policies also determine the number of members that a node can have in its *alliance* (by limiting the number of remote advertisements this node can cache). An *alliance* of a node can span across multiple hops. The advertisements in that case also are sent across multiple hops.

When an agent needs to discover a certain service, it first looks at its local platform to check whether that service is available. On failure, by looking at its own cache, it checks the members of its *alliance* to discover the service. If the service is still unavailable, the source platform tries to broadcast or multicast the request to other *alliances* in its vicinity. The local policy determines the method (broadcasting or multicasting). Multicasting is used to prevent broadcast storms in the network. We use policy-based multicasting where the node multicasts the request to other nodes in its vicinity where there are greater chances of obtaining the service. A node on receiving a service request chooses, based on its policy to either process it or drop this request.

Our flexible approach towards *alliance* formation does not have the overhead of explicit leader election. Every device in this environment is self-sufficient. However, it utilizes resources/services in the vicinity whenever they are available. Dynamic network topology changes are automatically reflected in the *alliances* that are formed. A node do not need to register or deregister with the neighborhood alliances when it changes its location. Our policy-based approach towards *alliance* formation buys us the advantages of a compound of agents. Using such policies, more traditional compounds like those utilizing explicitly elected leaders can also be achieved. With the help of policy, our architecture can also take user preferences into consideration. This is a major advantage since users of mobile commerce applications need the ability to control the ways in which their own resources are utilized.

2. RELATED WORK

There are a number of platforms, architectures and protocols that provide service discovery and multi-agent communication and collaboration. The Service Location Protocol (SLP) [11] is one such protocol. SLP is a protocol for automatic resource discovery on IP based networks. It bases its discovery mechanism on service attributes, which are essentially different ways of describing a service. SLP service agents are responsible for advertising service handles to the directory agents thus making services available to the user agents. This protocol aslo supports a simple service registration leasing mechanism.

Another service discovery architecture is Jini [3]. Jini is a distributed service-oriented architecture developed by Sun Microsystems. Jini Lookup Service (JLS) maintains dynamic information about the available services in a Jini federation(a collection of Jini services). When a Jini service wants to join a Jini federation, it first discovers one or many Jini Lookup Service from the local or remote networks. The service then uploads its service proxy (i.e. a set of Java classes) to the Jini Lookup Service. The service clients can use this proxy to contact the original service and invoke methods on the service.

Yet another solution to service discovery is Universal Plug and Play (UPnP) [16], supported by Microsoft. This architecture is designed to extend the original Microsoft Plug and Play peripheral model. UPnP works primarily at lower layer network protocols suite (i.e. TCP/IP). UPnP uses the Simple Service Discovery Protocol (SSDP) for discovery of services on Internet Protocol based networks. When a service joins a network, it sends out an advertisement message, notifying the world about its presence. The advertisement message contains a Universal Resource Locator (URL) that identifies the advertising service and a URL to a file that provides a description of the advertising service. When a service client wants to discover a service, it can either contact the service directly through the URL that is provided in the service advertisement, or it can send out a multicast query request.

Another in-house framework that was developed for dynamic distributed systems is the Ronin Agent Framework [6]. Ronin is a Jini-based distributed agent development framework. It offers a simple but powerful agent description facility that allows agents to find each other and an infrastructure. In Ronin, agents are described using the Common Agent Attributes and the Domain Agent Attributes. The Common Agent Attributes defines the generic functionalities and capabilities of an agent in a domain-independent fashion. The Domain Agent Attributes defines the domain specific functionality of an agent.

One more in-house architecture that investigated how service discovery can be taken beyond their simple syntax-based service matching approaches XReggie [7]. XReggie adds the facilities to describe service functionalities and capabilities using Extended Markup Language (XML). At the heart of the XReggie is the enhanced Jini Lookup Service that provides "service location" for Jini-enabled services. XReggie allows services to advertise their capabilities in a well-structured descriptive format using XML. Service discovery is done using XML descriptions.

Also there are several FIPA compliant platforms. One of such platforms is JADE [4] distributed by TILAB. Another platform is FIPA-OS [1]. Both JADE and FIPA-OS are convenient tools for agent application development however they are intended for a static network infrastructure and are not really meant to handle frequent disconnections and various other aspects of the dynamic nature of ad-hoc and mobile networks. Also both of these frameworks are too heavy to work in resource limited devices that are common in mobile and ad-hoc environments. The limitations of JADE and FIPA-OS were recognized and LEAP [5] and MicroFIPA-OS [14] were developed to extend the functionality and allow mobile devices to participate. MicroFIPA-OS and LEAP have smaller footprint than the original frameworks and fit well on the majority of the mobile devices like cell phones and PDAs. However the lightweightiness came at a price of functionality. The mobile devices running LEAP or MicroFIPA-OS host only small and most critical parts of the platform. This setup forces the devices to heavily depend on the static and resource-rich infrastructure around them to advertise or host agent management systems and directory facilitators. A mobile device is unable to effectively communicate with other mobile devices if the infrastructure is not present. Only when the infrastructure is present the devices are actually able to participate and communicate with other entities on the platform. Though, it is clear that LEAP and MicroFIPA-OS were the steps in the right direction it is also clear that it is not enough. This resulted in FIPA to introduce the concept of structured compounds between agents operating in ad hoc environments to facilitate agent discovery and interoperability.

The problem of finding information has also been addressed in the peer-to-peer systems. Specifically, [8] presents the idea of routing indices which allow nodes to forward queries to the neighbors that are likely to have the needed answers. If the node is unable to generate a reply to the query, it forwards the request to a subset of the neighboring nodes. The subset is identified by evaluating an index table that contains the inventory of the neighboring nodes. The request is routed to the subset of neighbors with the lagest inventory of relevant information.

The above described related work does not take into account the preferences and constraints of users, their devices and highly dynamic topology of ad-hoc networks. Our framework presents an alternative approach towards facilitating agent-service discovery in ad-hoc networks. The policy-driven approach gives users the flexibility to configure the agent platform based on their needs. Moreover, it makes the agent platform adaptable to the differing resource availability on different devices. Our solution is much more flexible than structured compound formation of agents and is adaptable to the changes of the environment both due to mobility and due to changes in user preferences. Also, Allia framework is independent of any description mechanisms and protocols that could be used to describe services. A description matching mechanism during service discovery is not specified or constrained by Allia framework. Services could be Jini compliant and be described and accessed though Jini proxies, or XML could also be used to describe the services. Other alternatives could also be employed.

3. DEVICE ARCHITECTURE AND PLATFORM COMPONENTS

Every mobile device will run a lightweight platform comprising of the following mandatory components. A *Generic Container* component that can host agents (the device must be able to run at least one agent). A *Lightweight yellow pages service* component that registers services on the local device (the *Directory Facilitator (DF)* of the FIPA Agent platform specification). A *Lightweight white pages service* component that registers local agents (the *Agent Management System (AMS)* of the FIPA Agent platform specification. A *Policy Manager* that controls the behavior of the platform. Local policies can be used to control various aspects of the platform like advertisement frequency, caching policy etc. An *Advertisement Manager* which is responsible for advertising the local services that are hosted on a mobile device. Broadcasting is used to transmit these advertisements in the neighborhood. A *Cache Manager* component is responsible for caching service advertisements that have been received from the neighboring devices. A *Forwarding Manager* is responsible for forwarding advertisements and "request for service" messages to other neighbors.

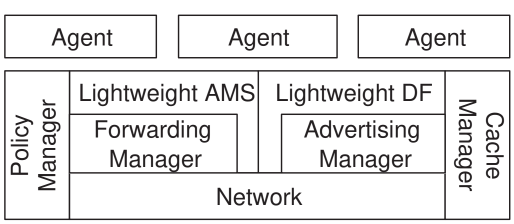

Figure 1: Device Architecture

Figure 1 shows the different components in a single mobile device. We describe the new components briefly in the following section.

3.1 Policy Manager

The *Policy Manager* is responsible for enforcing policies to control platform behavior. Policies are specified and registered with the *Policy Manager*. The manager is then responsible for ensuring that all components of the platform are in compliance with the specified policies. Using this mechanism, policies may be used to control the behavior of the platform. Policies can be used to restrict platform functionality and reflect device capability. Policies can specify caching preferences like refresh rate, replacement strategy, etc. Also, policies could describe advertisement preferences like frequency, time-to-live, algorithms, etc. Furthermore, policies could state priorities among services and describe how to share available resources among the services. Policies can specify advertisement and request message forwarding algorithms. Personal user preferences and application specifics can be also expressed though policies. And finally, policies can used to detail security restrictions like access rights and credential verification, etc. Components running on the platform would coordinate all their activities through the *Policy Manager*. Because of its interactions, this manager can monitor the activities of the different components and notify when policies are being violated.

3.2 Cache Manager

The *Cache Manager* handles service advertisements from neighboring devices. These advertisements could describe services that are running on the neighboring nodes or advertisements that are being forwarded by these neighboring nodes. The *Policy Manager* coordinates with the *Cache Manager* to govern various characteristics like entry expiry, replacement strategy, cache size, etc. The cache maintains only hints. Due to the dynamic nature of the network topology, it is not feasible to guarantee that services found in the cache are currently reachable. Whenever there is a hit in the cache for a service that is not currently available, the cache entry is immediately deleted.

3.3 Advertising Manager

The *Advertising Manager* actively broadcasts service descriptions registered to the local *DF*. The *Policy Manager* controls the rate of advertisements. Various policies can be employed to adjust the rate of advertisement. We have described them in section 4.2. For example, if the network is fairly static, the advertisement rate can be slowed down. Also policy could be event driven, events here being the arrival or departure of nodes in an alliance. Advertisements can also be assigned different priorities. The *Advertisement Manager* also controls the *alliance diameter* (explained in section *4.2.2*).

3.4 Forwarding Manager

The *Forwarding Manager* receives *Service Advertisements* and *Request for Service* messages. It decides based on the local policy whether to drop, or to propagate the advertisement. To prevent broadcast storms, this forwarding mechanism can use multicasting to selectively forward advertisements. For example, this can be used to forward advertisements to more active or resource-rich devices in the network.

4. PROTOCOL DETAILS

In this section we describe the discovery, caching, advertising protocols and request routing protocols to be used between mobile devices in ad-hoc networks.

4.1 Peer-to-Peer Caching

Every participating mobile device hosts an agent platform. Each mobile device that is ad-hocly connected to different other heterogeneous devices in the network maintains a cache to store service descriptions of remote services. The cache is controlled by a local *Cache Manager*. Each *Advertisement Manager* in a device, after a finite time interval (discussed later) broadcasts a list of its local services to other peer devices around it. To do so, it uses the local *Forwarding Manager*. These local services correspond to services of local agents that are *registered* to the local *DF*. The *Forwarding Manager* receives all messages from the network then based on the local policy it decides whether to process the message or not. For example, if the policy of the local node is such that it does not want to form its own *alliance*, then it would not accept any advertisement. The *Forwarding Manager* passes the advertisement to the *Cache Manager*. A *Cache Manager* in a remote device on receiving the advertisement, decides based on its policy whether to cache it or not. For example, the policy of the *Cache Manager* could be that it will not store more than a certain number of advertisements. This policy is driven by the resource-richness of the device. The policy could also be user-driven. The user might not want to store any advertisement.

The idea behind passive caching of advertisements rather than active pulling of service descriptions from neighboring nodes has several advantages. One of the advantages is efficiency in detecting the change in the environment. We follow the *push* paradigm for such service advertisements since it enables us to detect in an efficient manner when the ad-hoc environment has changed. When a device receives a new advertisement from a new device, it automatically knows that there is a new platform in the vicinity and could potentially be included in the *alliance*. The pull-based approach of explicitly searching for new platforms in the vicinity would have imposed more burden on the mobile device hosting a platform. As we shall see later, the frequency of advertisement could be adjusted based on the relative stability of the *alliance* and hence, the burden imposed on the mobile device would be less. Another advantage is that advertisement collision are not as frequent as in pull-based paradigm. In a standard pull-based paradigm, a mobile agent platform would have broadcasted a request to all its neighboring platforms asking for a list of local services. The peer platforms would have sent a list of services to the querying platform. Such a solution has a potential danger of the advertisements from different platforms colliding with each other at the receiver's end. Another advantage of this approach is the flexibility in handling advertisements. The device receiving the advertisement has the option to either store or reject the advertisement based on its current policy.

The *Cache Manager* is responsible for handling remote advertisements, storing remote advertisements of services, expunging the cache and also handling requests (from *DF*) to match services present in the cache. This is the way, a platform implicitly makes its services available to other remote devices around it. The *Forwarding Manager*, on receiving an advertisement might also decide to forward it to members of its *alliance* or broadcast the advertisement to all other nodes in its vicinity. This is determined based on whether the advertisement is meant to traverse multiple hops (determined by the sender node of the advertisement). The sender node becomes a member of the *alliances* of the nodes that cache its advertisement. Figure 2 elucidates the concept of *alliance* formation in an ad-hoc environment.

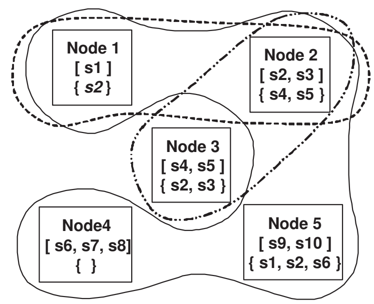

Figure 2: Alliance Formation. Alliance of Node1: Node2[S2]. Alliance of Node2: Node3[S4,S5]. Alliance of Node3: Node2[S2,S3]. Alliance of Node4: ∅. Alliance of Node5: Node1[S1], Node2[S2], Node4[S6].

The cache expunge policy depends on the policy specified by the *Policy Manager*. Each advertisement contains a lifetime (specified by a time-to-live field in the advertisement structure). When a new advertisement is received by a *Cache Manager*, the *Cache Manager* decides to either accept it or reject it. An advertisement is accepted only when there is sufficient space in the cache to hold this advertisement or when an old advertisement can be removed from the cache.

4.2 Policy-based Advertising

We introduce the concept of policy-based advertising of services. The policy may potentially reflect a combination of the user's preferences, resources present in the device, and application-related preferences. Various parameters of the advertising are controlled based on the local policy. Thus, each platform in this environment may follow a different policy in advertising. In this section, we describe the different aspects of the advertisement.

4.2.1 Advertisement Frequency Control

A very important aspect of advertisement is to decide the frequency with which the advertisements are going to be sent out to the platforms in the vicinity. One of the possibilities is to keep the advertisement rate at a constant frequency. A local *Advertisement Manager* sends out a list of the local services that the registered at the *DF* at a constant time interval. This time interval need not be uniform across all platforms. Each platform might have its own frequency of advertising. The immediate advantage of such a design is that a platform having inter-advertisement time interval of t would be detected within $t + \delta$ where δ is a network delay in message propagation. However, such a design choice has its disadvantage of network overhead. It is possible that an advertisment frequency is too high for a relatively stable ad-hoc environment. On the other hand, in a highly dynamic ad-hoc environment, a low advertisement frequency might not be sufficient to form an *alliance*.

Another possibility is to allow the advertisement to be set out with a varying frequency. Each *Advertisement Manager* in a platform may follow a Multiplicative Increase Linear Decrease (MILD) algorithm or a Binary Exponential Back-off (BEB) Mechanism to vary its advertisement frequency so as to adapt the advertisement rate. There will be a threshold value for the inter-advertisement time. In this design, a node will send out advertisements at a high frequency at the start. It will follow the MILD or BEB algorithm to adjust the frequency based on whether it is receiving any new advertisement or not. If it receives a new advertisement, it knows that there is a new platform in the vicinity and hence goes back to the starting frequency (in case of BEB) to advertise its local services to the newly arrived platform in the alliance. Thus if the environment around a platform changes rapidly, the advertisement frequency would be higher. Contrarily, if the environment is relatively stable, the advertisement frequency would be low and hence would impose less network overhead. If in either algorithm, the threshold value of the inter-advertisement time is reached, the algorithm starts from the beginning. This type of advertisement policy would be adaptive to the mobility and changes in the ad-hoc environment.

Yet another possibility is an adaptive mobility-based advertisement frequency rate. In this approach, each *Advertisement Manager* sends out an advertisement only when it receives a new advertisement. A new advertisement signifies the existence of a new platform in the *alliance*. Thus, it sends a list of its own local services immediately in order to be included in the *alliance* of the new node. This type of advertisement policy is highly adaptive to the mobility and change of the environment. The overhead imposed on the network is also adaptive to the change in the environment. In a highly dynamic ad-hoc *alliance* formation environment, the network load (due to advertisements) would be relatively higher than the network load in a relatively stable environment (since there are no new advertisements). However, problem with this policy is that a new platform might remain undetected for a long time in a relatively stable environment where the advertisement frequency is low. A combination of variable frequency techniques to alternate advertisement frequencies along with this scheme would be able to avoid that problem.

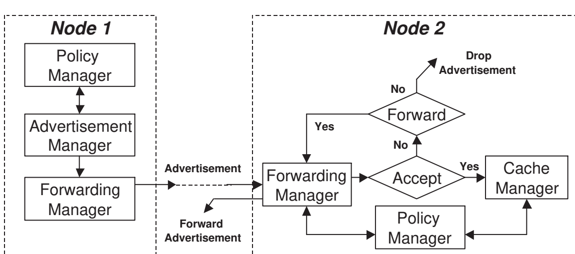

Figure 3: Advertisement Flow between Nodes

The advertisement scheme to be followed by a platform is handled by the local *Policy Manager*. This provides the required flexibility to handle user preferences in using up the resources of its own mobile device. Figure 3 shows the control flow of sending and receiving advertisements.

4.2.2 Alliance Diameter Control

Each advertisement contains a *hop-count* field that would specify the number of hops the advertisement may be propagated to by receiving platforms. This hop-count field determines the potential *diameter of the alliance*. Any platform within the *diameter* would be able to cache the local service informaion of the source platform. In a highly dynamic environment, a small *diameter* is more desirable. This is because, if the *diameter* were large, stability of a platform in an *alliance* would be less. Hence, the information of services in the cache would become stale quickly. For implementation purposes, we will be limiting the hop-count to one. Hence, a platform will only advertise its services to its one-hop neighbors and the *alliance diameter* would be one.

A platform on receiving an advertisement will perform a set of tasks. First, if the local *Policy Manager* allows, the platform will accept the advertisement and store it in the cache. Next it will decrement the hop-count field of the advertisement. If after decrementing the hop-count the value is zero, then the advertisement is discarded. Else if the *Policy Manager* permits, the advertisement is forwarded to the one-hop neighbors.

Due to resource consumptions and other user preferences, the policy in a platform might specify not to forward any advertisement message. In this case, the advertisement would be silently discarded by the platform. When a platform decides to forward an advertisement, it sends the advertisement to the *Forwarding Manager*. The *Forwarding Manager* manages the re-broadcasting of the advertisement its one-hop neighbors.

4.2.3 Advertisement Structure

There are two types of advertisement structures in our design, viz. a *Service Advertisement* and a *Platform Advertisement*. A *Service Advertisement* is used to send out an advertisement describing the services that are registered to the local *DF* of the platform. This type of advertisement is sent out only when a node receives a new advertisement or somehow detects that there is a new platform in the vicinity that may potentially be included in the *alliance*. This is determined by the local *Policy Manager*. This advertisement contains: service description(s), agent identifier(s), platform identifier, network address (the actual hardware interface address through which the device can be accessed in an ad-hoc network) and finally some accessory information like a time-to-live field that is used to convey information about how long the platform is going to be in the vicinity, resources of the local platform (e.g. cache size) or some important policy information (e.g. I don't forward any advertisements or requests) that might help the receiver node to decide whether to forward any requests / advertisements to this node later on. A *Platform Advertisement* is used as a *heartbeat* message to notify the members of an *alliance* about the presence of a platform in its vicinity. Clearly, it is not required for a platform to advertise all its services (and hence send a large volume of data) every subsequent time interval in the event that the platform is relatively stable in the environment. It only needs to send the whole information when a new platform (device) is in the vicinity. For the rest of the time, it suffices to send only the information about the presence of the platform in the vicinity of the other platforms. The *Cache Managers* in the other platforms can update the cache information accordingly. The

Platform Advertisement contains only the *platform identifier* and the *network address* of the sender platform. This reduces the network overhead in terms of message size.

4.3 Request Routing

The main goal of our system is to make a best-effort service discovery amongst agent platforms in a mobile commerce environment. Each platform or node in the system has a *DF* (*Directory Facilitator*) and an *AMS* (*Agent Management System*) [19]. When there is a request from any agent to communicate with or discover an agent/service, the request first comes to the local *DF*. If the *DF* does not have the service, the *DF* forwards the request to the *Cache Manager*. If the service description is found in the local cache, the information about the location of the agent (platform identifier, network address) is taken as hint and provided to the requesting agent. The requesting agent contacts the agent/service using standard Agent Communication Language [18]. The *Cache Manager* forwards the request to the *Forwarding Manager* in case of two events. First, if the hint provided by the local *Cache Manager* turns out to be false. This happens when the agent fails to communicate with the device that has the service. Second, a match is not found in the local cache.

The *Forwarding Manager* forwards the request to selected agent platforms in its vicinity. Some of those platforms may be a member of the local *alliance*. It also sets a hop-count field for the request to limit the number of hops or number of adjacent *alliances* through which the request is going to traverse. The value of the hop-count field is decided by the local *Policy Manager*. The standard policy that is used by the *Policy Manager* is to increment the hop-count from 1 to MAX-HOP-COUNT (maximum number of hops the request might be allowed to traverse) before it discovers the required service. There are two different ways of sending the request to neighboring platforms in the vicinity. The request could be broadcasted to all the neighboring devices. In this case, the request will reach all the nodes in the vicinity. Some of these nodes might not be resource-rich enough to even forward the request any further or some of them might not be members of any other *alliance*. The request could be multicasted to a set of particular neighbors. The *Policy Manager* would specify the method (broadcasting or multicasting). The multicasting could be used to prevent broadcast storms. The device could multicast the request to its most active neighbors. It could identify them by looking into the local cache and evaluating advertisement patterns like"Accessory Information". For example, the "Accessory Information" could contain the cache size of the sender. This would provide a measure of the resource-richness of that node. "Accessory Information" could also contain the number of platforms/nodes are there in the vicinity of the sender node. If the sender node is a member of another large *alliance* of powerful nodes, it might be advantageous to send a "Request for Service" to that node since it will in turn broadcast/multicast it to all other nodes in its vicinity. The strength of our design lies in the fact that all these decisions would be strategically influenced by the local *Policy Manager*. Since the *Policy Manager* takes user preferences and local resources into consideration, the solution could be adapted to reflect these demands and limitations.

Each device upon receiving a request message can chose to drop the message or process it. The local policy is used to make this decision. If the device chooses to process the request message, it checks its *DF* to see if it is hosting the desired service. If the service is local then the service description is sent to the requesting node. If the device is not hosting the needed service, it could check the local cache. If the cache contains info about the service then the device replies with hint. Otherwise, the request is sent to the *Forwarding Manager*. The *Forwarding Manager* decides to forward the request to other nodes in its vicinity depending on the local policy. Figure 4 shows the control flow between the different components in processing the request. In the event that a service was discovered, the reply is reverse-routed back to the source node. The intermediate nodes may or may not choose to record this as a successful discovery to maintain hit-rate statistics or utility measure of itself. This utility measure could be sent to remote platforms with advertisements to give the other nodes an idea of its efficiency.

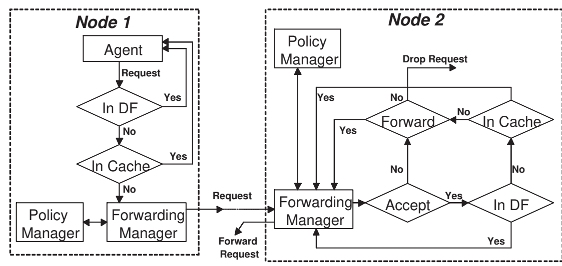

Figure 4: Control Flow for a Request

5. IMPLEMENTATION

To substantiate our design we have come up with a preliminary implementation of our protocol. The platform used to demonstrate our scheme is a JADE [4] based Lightweight and Extensible Agent Platform called LEAP [5]. We have extended the implementation of LEAP (we call *eLEAP*) to support agent discovery over Bluetooth [15] using our alliance-based discovery protocol. We selected LEAP since LEAP is light-weight enough to run on selected mobile devices. However, the current implementation supports connectivity over TCP/IP over GSM or IEEE 802.11 wireless LAN.

We use a set of laptops and Windows CE iPAQs as a test set of devices on which our architecture is implemented. The laptops use Ericsson Bluetooth Development Kit as hardware components and the iPAQs use the embedded Bluetooth in them. Every device (laptops and iPAQs) has a *Main Container* that has a *DF* and an *AMS* running. In our system, the functionality of the *Main Container* has been extended to incorporate a peer-to-peer caching mechanism for the discovery of non-local services in an ad-hoc network. The new components in the system include a *Cache Manager* component, an *Advertisement Manager* component and a *Service Discovery* component. The device architecture showing the various components of the current implementation is shown in Figure 5.

We have implemented the prototype version on top of Bluetooth [15] using AXIS OpenBT [20] Bluetooth stack for communication between two devices and service discovery. One big factor of choosing OpenBT amongst other famous stacks like BlueDrekar [2] is because it is an open source stack. OpenBT stack provides a set of 8 device files (ttyBT[0-6] and ttyBTC) for applications to use. Stack control is done using the blocking ioctl calls.

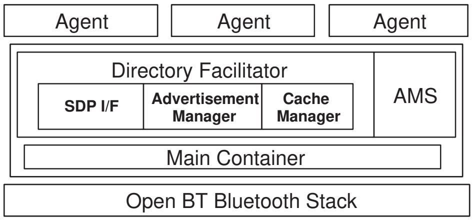

Figure 5: eLEAP Device Architecture

Currently, we use a simple policy to control the advertisement rate, advertisement acceptance and request forwarding. We use a Multiplicative Increase Linear Decrease (MILD) algorithm for controlling the advertisement rate. We accept an advertisement if there is a free entry in our cache. We forward a request if it generated a cache miss in a local node. The advertisement diameter in our current implementation is restricted to 1 hop.

5.1 Advertisement Manager

This component takes care of periodically advertising its local services to all of its one-hop neighbors in the Bluetooth network. Current Implementation considers the *alliance diameter* as 1. Each advertisement packet has a unique *Advertisement Identifier*, a *DF agent description*, the device *cache size* and the *advertisement's time to live*. We use MILD algorithm to determine the time interval between successive advertisements. The rate of incoming advertisements will determine the value of this time interval. To send periodic advertisements each mobile device does an HCI inquiry [17] and opens a RFCOMM connection [15] with all the devices found during inquiry. It then closes the connection after sending the advertisement messages. All incoming advertisements are cached based on the caching policy of the platform. In our case, the resourcefulness (measured as a function of memory, processor speed etc.) of the device is taken into account to determine the cache size. Due to lack of space, we are unable to present the function. Figure 6 shows an advertisement packet format.

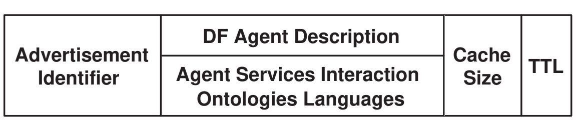

Figure 6: Advertisement Packet

5.2 Cache Manager

The *Cache Manager* maintains a cache entry for services that are present one hop away. The number of entrees in the *Cache Manager* determines the *alliance* formed by a node. When an agent searches for a service, it first looks up the *DF* table. If no such service is available in the local *DF*, then the cache is checked if the service is available in the node's *alliance*. If there is no such entry in the cache, then a Bluetooth Service Discovery is initiated to look for services in other nodes that might not have been cached. When a service is discovered, the *DF agent description* of the service is sent to the agent that is performing the lookup. Further communication can be established using the *agent identifier* of the service provider.

5.3 Service Discovery Component

We have used the Bluetooth SDP [15] to discover services on remote platforms. Every local *DF* registers its local services to the local *Bluetooth Service Discovery Manager (SDM)*. An SDM takes request from the *DF* service on the platform and performs a Bluetooth service discovery. When a matching service is not found in the local *DF* or cache, the *SDM* first discovers the devices that are in the vicinity of the source and then initiates a request to discover the service in the remote *SDM*. The remote device may or may not be a member of the *alliance* of the requester node.

The request is passed in the form of an agent description which consists of the agent-id, service-type and the attributes of the services. Since Bluetooth discovery is done through UUIDs we perform a mapping of these agent service types into UUIDs. We assume that the UUID mapping is same on all the Bluetooth devices running agents. We maintain a limited size-mapping table "MAP-UUID" to map the services into 16 bits UUIDs. We are currently using 16 bit mappings. However, it can be enhanced to use the 32 bit or 128 bit UUIDs by modifying the mapping function and SDP request Protocol Data Unit. We use the cache size as a parameter to judge the resource availability of a device.

Current implementation of eLEAP has a footprint of 632K and during runtime uses a heap size of 817K and loads only 11 extra classes than standard LEAP does.

5.4 Experimental Results

We tested our system to emulate an airport scenario by running two agents, *AirlineInfo* having a *Booking* and *Schedule* services and *TourInfo* having *HotelReservation* and *CarRental* services, on a device called *ABCAirlines*. We have another device called *XYZPassenger* having an agent *FindInfo* looking for some services. Initially the device *XYZPassenger* is not in range with *ABCAirlines* (on the way to airport). When agent *FindInfo* requests for services *Booking*, *Schedule* and *HotelReservation* it does not receive the *Booking* service but receives the *Schedule* and *HotelReservation* service from an another agent *AllAirlinesInfo* which provides some services on behalf of *AirlineInfo* and *TourInfo*. *FindInfo* finds the services in its cache due to previous advertisements by agent *AllAirlinesInfo*. When device *XYZPassenger* comes in range with *ABCAirlines* (after entering the airport) it again requests for the missing service i.e. *Booking* service and this time receives the service from the *AirlineInfo* agent.

6. CONCLUSIONS

In this paper, we have presented a peer-to-peer caching based and policy-driven approach towards flexible *alliance* formation amongst agents in an ad-hoc environment. An agent actively forms its own *alliance* and passively joins other *alliances* in an environment and hence facilitates agent-service discovery. An agent knows the members of its own *alliance* but it is not aware of the *alliances* where it is a member of. Our protocol follows a policy-driven approach for advertisements, caching and request forwarding thus letting user preferences to be taken into consideration. An *alliance* helps in agent-service discovery in an ad-hoc environment. However, it removes the leader election and other problems associated with structured compound formation to facilitate service discovery amongst agents in ad-hoc networks. The dynamic nature of *alliance* formation and policy-based advertising achieves high degree of adaptability in a changing environment. We have also presented an initial implementation of our protocol using ThinkPads and iPAQs. We have extended LEAP to incorporate our protocol using Bluetooth as the underlying ad-hoc networking protocol. We have used openBT stack of Bluetooth for implementation purposes. Current implementation has an advertisement diameter of 1 hop. However, it can efficiently demonstrate the flexibility of our protocol in practical ad-hoc environments.

7. ACKNOWLEDGMENTS

This work was supported in part by NFS awards IIS 9875433 CCR 0070802 and the Defense Advanced Research Projects Agency under contract F30602-00-2-0 591 AO K528.

8. ADDITIONAL AUTHORS

Sovrin Tolia (email: `stolia1@cs.umbc.edu`),
Deepali Kushraj(email: `deepali1@cs.umbc.edu`),
Anugeetha Kunjithapatham(email:`anu1@cs.umbc.edu`),
Gaurav Gupta (email: `gagupta1@cs.umbc.edu`).

9. REFERENCES

[1] FIPA-OS Website. Emorphia Limited. 27 Dec. 2001. World Wide Web, http://fipa-os.sourceforge.net.

[2] IBM alphaworks. BlueDrekar protocol driver. World Wide Web, http://www.alphaworks.ibm.com/tech/bluedrekar.

[3] Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Schei and Jim Waldo. The Jini specication. Addison-Wesley, Reading, MA, USA, 1999.

[4] F. Belligemine and G. Rimassa. Jade-a pa-compliant agent framework. In Proc. PAAM '99. London, pages 97-108, 1999.

[5] F. Bergenti and A. Poggi. Leap: A pa platform for handheld and mobile devices. Presented at ATAL, 2001.

[6] Dipanjan Chakraborty, Filip Perich, Sasikanth Avancha, and Anupam Joshi. An agent discovery architecture using ronin and dreggie. In First GSFC/JPL Workshop on Radical Agent Concepts (WRAC). NASA Goddard Space Flight Center. Maryland. USA., January 2002.

[7] Harry Chen, Dipanjan Chakraborty, Liang Xu, Anupam Joshi, and Tim Finin. Service discovery in the future electronic market. In Working notes of Seventeenth National Conference on Articial Intelligence, Eleventh Innovative Applications of AI Conference. Austin. TX, july 2000.

[8] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. ICDCS 2002, October 2001.

[9] Connected Limited Device Conguration. http://java.sun.com/products/cldc/.

[10] PersonalJava Application Environment. http://java.sun.com/products/personaljava/.

[11] E. Guttman, C. Perkins, and J. Veizades. RFC 2165: Service location protocol, 1997.

[12] Micro Edition Java 2 Platform. http://java.sun.com/j2me/.

[13] Mobile Information Device Prole (MIDP). http://java.sun.com/products/midp/.

[14] University of Helsinki. MicroFIPA-OS Small footprint extension to FIPA-OS. 2001. Available at: http://fipa-os.sourceforge.net/.

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

[16] UPnP White Paper. World Wide Web, http://upnp.org/resources.htm.

[17] Bluetooth Specication. World Wide Web, http://www.bluetooth.com/dev/specifications.asp

[18] FIPA ACL Message Structure Specication. World Wide Web, http://www.fipa.org/specs/fipa00061/.

[19] FIPA Agent Management Specication. World Wide Web, http://www.fipa.org/specs/fipa00023.

[20] AXIS OpenBT Stack. World Wide Web, http://developer.axis.com/software/bluetooth.