Technical Report TR-00-06, School of Computer Science, Carleton University, October 2000.

Comparison of Bandwidth Usage: Service Location Protocol and Jini

Javier Govea and Michel Barbeau
School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, ON K1S 5B6 Canada
{jgovea,barbeau}@scs.carleton.ca

September 30, 2000

Abstract

Recently there has been an increase in the development of technologies for resource discovery, since for example, resources such as printers, mail boxes, memory space, and disk space are available in every network, ready to be used for any host. This has been caused, in part, by the growth in the popularity of portable devices such as laptops, PDAs, and cell phones which require configuration each time they attach to a new network segment. Since the configuration of such devices is tedious and sometimes complicated, there have been some attempts in past years to solve this problem, such as the DHCP approach.

This paper focuses on the bandwidth analysis of two new approaches for dealing with resource discovery: the Service Location Protocol (SLP) and Jini. This work is particularly important since the communication among the devices is often wireless, whereas bandwidth is a limited resource. We present equations for characterizing the usage of bandwidth made by SLP and Jini, based on their specification, on previous work and experiments.

Keywords - PERVASIVE COMPUTING, RESOURCE DISCOVERY, SERVICE LOCATION PROTOCOL, JINI, BANDWIDTH USAGE ANALYSIS.

1 Introduction

Pervasive computing or ubiquitous computing is a new concept in computer science. Several attempts have been made to define what pervasive computing is. For example, Birnbaum defined it as acomputer hidden in so called information appliances such as washing machines and furnaces [**?**]. More recently Ciarleta and Dima define a layer model for pervasive computing [**?**]. The aim of all this work has been to provide a standard definition. A rough definition of Pervasive Computing could be computers everywhere at anytime. The term Desegregated Computing can be used as well to denote devices with small processors embedded and interconnected, such as monitors, projectors, printers, input devices, PDAs, and phones [**?**]

Currently, resources such as mail boxes, printers, and disk space are readily available at any network. Devices such as PDAs and cell phones are capable of accessing them. The growth in the popularity of this kind of portable devices has increased the need to develop technology for connecting to these resources. Further study in this area will lead us eventually to the definition of pervasive computing stated above: computers everywhere at any time. But more importantly, computers and resources at the right moment.

In this scenario, it is easy to see that communication among the desegregated devices has to be wireless, since many devices are nomadic. Also, it is desirable that the connection be plug-and-play because configuring devices each time that they move is clearly tedious and complicated.

In order for the computers or devices to become pervasive, many software and hardware technologies have to be developed and deployed. Currently there are several technologies available in the market for this purpose. These technologies range from frameworks to protocols and hardware-based technologies. All these technologies and further developments in this area are very important for the full deployment of pervasive computing.

Some examples of frameworks are the Salutation framework [**?**] and Home Audio/Video interoperability (HAVi) [**?**]. The resource discovery protocols, which allow the desegregated devices to find each other and perhaps to talk to each other are also available in the market. For example the Service Location Protocol (SLP) [**?**], Dynamic Host Configuration Protocol (DHCP) [**?**], HP JetSend [**?**], Universal Plug-and-Play (UPnP) [**?**], Bluetooth [**?**], and Jini [**?**]. Some of these technologies are more than a simple discovery protocol. For instance, Jini allows the devices to talk to each other as well. Finally, there are other technologies, mainly hardware-based, that can work with the aforementioned technologies in order to speed up the communication among the desegregated devices. Some examples of these technologies are the Universal Serial Bus [**?**], FireWire (IEEE 1394) [**?**], and Home Phoneline Networking Alliance (HPNA) [**?**].

This paper focuses on the analysis of bandwidth usage by resource discovery protocols of two main technologies: SLP and Jini. SLP is in itself a resource discovery protocol whereas Jini makes use of the Join/Discovery protocol.

As it was mentioned above, the wireless communication among the desegregated devices is desirable. Therefore this sort of analysis is relevant since bandwidth is a limited resource and some of the devices might not have enough power for many or long transmissions.

Based on the SLP and Jini specifications, as an extension of previous work [**?**], and experiments, we developed equations for characterizing the bandwidth usage made by Jini and SLP. These equations are used to simulate the bandwidth used by SLP and Jini, given a number of desegregated devices.

Section 2 gives an introduction to these two technologies. It is followed by a presentation of our bandwidth usage in Section

3. Finally some conclusions are given in section 4.

2 Background

This section gives an overview of how Jini and SLP work.

2.1 Service Location Protocol

The Service Location (SRVLOC) group is an active group in the Internet Engineering Task Force (IETF). They published in 1997 the SLP Version 1 [**?**] and in 1999 the SLP Version 2 [**?**]. SLP is the IETF standard for resource discovery of desegregated devices.

2.1.1 How it works

SLPv2 is a framework for resource discovery that makes use of three entities:

- Service Agent (SA). Services are resources that can be used by a device, a user, a program or another service. Some examples are memory space, printers, and mailboxes. In general any resource available in the network is potentially a service. An SA provides services advertising them by multicast or broadcast. An SA also intercepts and replies to queries about its services with a service access point. An SA registers its service with a DA when it is present.
- Directory Agent (DA). Their primary function is to implement a repository of services where the clients can look for particular services given particular attributes. A DA catches the advertisements from the SAs, collects information from the advertisements of SAs, and replies on behalf of SAs to UAs when they request a particular service.
- User Agent (UA). A UA is a client of the services. It looks for and requires services with particular characteristics by sending queries about services to the DAs or directly to the SAs.

UAs and SAs can discover the DAs by different means, such as:

- Active discovery. SAs and UAs multicast SLP requests.
- Passive discovery. DAs multicast advertisement messages on a periodic basis by announcing their presence.
- UAs and SAs can learn the location of DAs by using the DHCP options for Service Location (SLP DA Option). DHCP servers configured with this option can distribute the DA addresses to the agents that require them.

SLP has the notion of the scope, which is an unadministrative domain. For example, a group can be a department within a company. UAs, SAs and DAs are members of a scope. UAs send a *ServiceRequest* only to SAs and DAs supporting their scope. The purpose of the scope is to provide scalability limiting the network coverage of a request and the number of replies.

SLP allows a configuration without DAs. It addresses two modes of operation:

- Without DAs. UAs send, using IP multicast or broadcast, *ServiceRequest*s to SAs. SAs are listening to a well-known port and, when they find a match between a requested service and the service they offer, they reply to the UAs using unicast.
- With DAs. UAs, SAs, and DAs are members of scope. UAs and SAs communicate with DAs supporting their scope using unicast UDP or TCP. DAs are listening on a well-known port. SAs register their offer of service with DAs in their scope. UAs send queries to the DAs supporting their scope. If a DA finds a match between a requested service and a registered service, it replies to the UA.

Figure 1 illustrates a common scenario when working with SLP. The operation mode is with DAs using an active discovery. It is assumed that the SA, UA, and DA are members of the same scope. This scenario works as follows: the SA and UA discover the DA by multicasting a *ServiceRequest*. The *ServiceRequest* message sets the parameter *Type* to *service:Directory-Agent*. Every DA that listens to this message responds with a unicast message called *DAAdvertisment* that contains the URL of the DA.

Then the SA registers with the DA in its scope. The parameters of the *ServiceRegistration* message are a *URL*, *Type* and *Attributes* of the service. The SA gets back a *ServiceAcknowledgment*, which is an error code. On the other hand, after the discovering process the UA can look for a particular service by sending a *ServiceRequest* to the DA specifying the *Type* and *Predicate* over attributes of the service needed. If the DA matches the *Service Request* with a registered service, it replies with a URL of the service requested. Otherwise it remains silent. Afterwards, the UA can establish a connection with the SA.

In the abscence of DAs, SLP defines the multicast convergence algorithm for dealing with the number of multicast or broadcast requests from the UAs when they are looking for a particular service. This algorithm specifies that a *ServiceRequest* is transmitted no more than four times over a period of 15 seconds. The message *Service Request* contains a field called *Previous Responder List*. The list contains the IP addresses of the SAs that have returned *Service Reply*s so far in the execution of the multicast convergence algorithm. An SA receiving a *Service Request* with its own IP address within the *Previous Responder List* of the message ignores the request and remains silent.

The service URL, used to advertise services, contains the location of the service (IP address, port number and an optional path). To define the attributes of the services, SLP makes use of Service Templates [**?**]. The templates specify the attributes and their default values. It is a value/pair list. SAs advertise services based on the attribute defined in the service templates and the UAs request them using these same definitions. The service template of a particular service is registered with the Internet Assigned Number Authority (IANA); this ensures interoperability between vendors.

Currently there is an SLP API [**?**]. This API allows clients and services to access the SLP functionality and it features C and Java bindings. The API contains interfaces for the clients, for example for the discovery process. It also contains interfaces for servers such as for advertising. Finally, the OpenSLP project, an open source implementation of SLP for commercial and non-commercial applications, is under development by the SRVLOC working group [**?**].

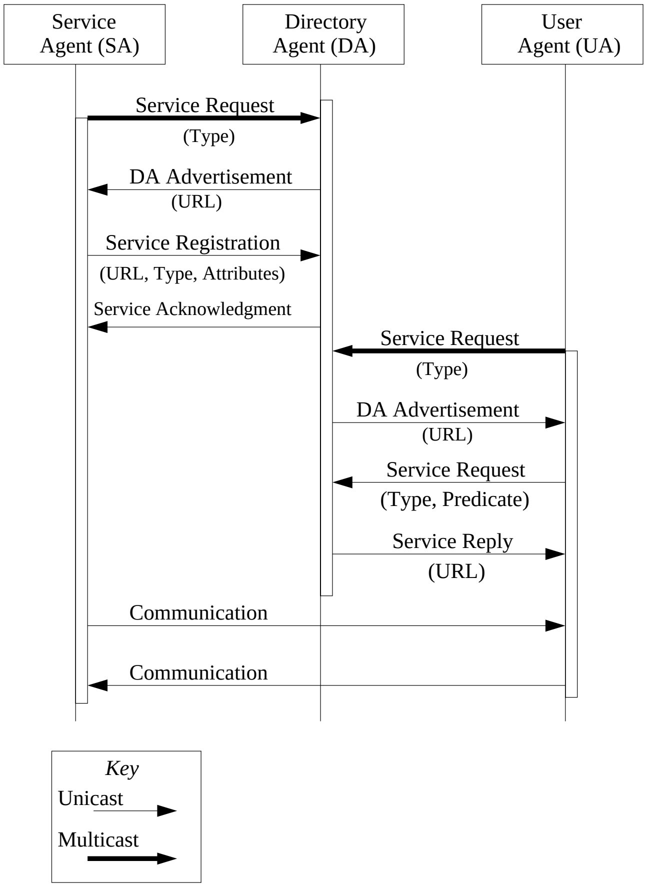

Figure 1: SLP Operation

2.1.2 Additional Features

SLP has also been designed to provide some security, support for browsing operations, and operations over IPv6.

The security mechanism addressed by SLP is the authentication of the sender through digital signatures that can be generated by any cryptographic technique such as public key cryptography. This works as follow: An SA can include a digital signature in its registration message. Then a DA can verify the signature before registering or deregistering the SA and when a *ServiceReplay* message is sent, the message includes the signature. In this way, the UA can verify the signature of the service before accepting it. The *DAAdvertisement* message generated by the DAs can also include a digital signature.

SLP includes several features that can be used to support service browsers. For example, SLP defines the *Service Type Request* and *Attribute Request* messages. The *Service Type Request* is used by the UAs to discover all the service types available and the *Attribute Request* messages are used by the UAs to obtain all the attributes of a particular service, which is sent as parameter. The parameter can be either a service type or a service instance. The response to these messages includes all the information needed to build an SLP browser.

Finally, although there is no current standard for the operation of SLP over IPv6, there has been a lot of work in this field so far. Some of the changes needed are the service URLs and the addresses used for the multicast operations, which need to support the IPv6 specification. In general, all the addresses involved in SLP need to support IPv6.

2.2 Jini Technology

Jini [**?**] is a coordination framework written in Java and developed by Sun Microsystems. In the development of Jini, many people from the industry and academia were involved. However, it is important to mention the work done by David Gelernter at Yale University and Nick Carriero who built the Linda coordination model using Tuple Spaces [**?**]. The evolution of Linda led to the JavaSpaces technology that eventually evolved into Jini.

2.2.1 How it works

Jini has the concept of service which is analogous to the SA concept in SLP. Jini defines five key concepts: discovery, lookup, leasing, remove events, and transactions [**?**].

- Discovery. The discovery process is very similar to the SLP discovery process. The main purpose of this process is to locate lookup services which, in turn, contain references to services available in the network.
- Lookup. Lookup services are analogous to the DAs in SLP. Lookup services store an object, which is a proxy of the service, for every service registered with them. It is also possible that lookup services include proxies of other lookup services, allowing in this way a hierarchical lookup. Further more, the lookup services can contain objects that encapsulate other naming or directory services not supported by Java or Jini. In this way, Jini implements a mechanism for creating bridges between the Jini lookup services and other types of lookup services such as the SLP DAs. The Jini services are clustered in

communities called groups, which are analogous to the scopes in SLP. The groups have names and it is possible for lookup services to be part of more than one group.

- Leasing. The access to any service in Jini is lease-based. The lease is granted by the service provider and it guarantees the access to the service for a fixed period of time. Leases are negotiated between the service provider and the client using a simple protocol. The clients ask for a lease time and the service provider replies with the time granted. The time granted ranges from not granting the service to providing the time requested. If the service is not renewed before the lease time expires the resource is freed. In Jini, services can use third party services to manage their leases on their behalf. Thus service providers can focus on implementing their services and somebody else will take care of the leasing.

- Remote Events. One of the strengths of Jini is its support for distributed events. Events may occur within an object and other objects may register with it their interest in such events. Then, when the event occurs, they are notified. This is done via the Java Remote Method Invocation (RMI) [**?**].

- Transactions. A transaction is a set of wrapped operations. The operations can be part of one service or part of several services in the network. Using Jini Transactions either the whole set of operations succeeds or fails; there is no partial success. This sort of operation is particular useful when dealing with distributed databases. The two-phase commit (2PC) protocol is widely used by distributed databases for dealing with a set of operations that need to be performed as one operation. The Jini Transaction model only defines the interface for coordinating the 2PC protocol. The implementation of the interfaces is left to the user.

One possible scenario, and perhaps the most common, demonstrating how Jini works is shown in Figure 2. Note that this figure is very similar to Figure 1 which illustrates the operation of SLP. Jini based its operation on three protocols called discovery, join, and lookup protocol [**?**].

The discovery protocol is used by service providers and clients to discover the lookup services in the network. The discovery protocol makes use of three different protocols that can be used for discovering lookup services in different situations.

- Multicast request protocol. It is similar to the active discovery in SLP. Its operation is shown in Figure 2. In this scenario the discoverer, which is the entity that wishes to contact the lookup service, multicasts a UDP request. The lookup service is listening to a well-known port for an incoming request. The request contains, besides other fields, the groups in which the discoverer is interested and a list of lookup services that are already known by the discoverer. When the request is received by the lookup service, it replies with a message via unicast TCP.

- Announcement request protocol. It is similar to the passive discovery in SLP. The lookup service multicasts on a periodic basis a UDP message advertising its presence. The announce message contains a list of groups of which the lookup service is a member. When the discoverers, listening to a well-known port for announce messages, receive the announce message, they reply with a unicast TCP message. This reply is similar to the request used by discoverers in the multicast request protocol. Finally, the lookup service replies back with a unicast TCP message.

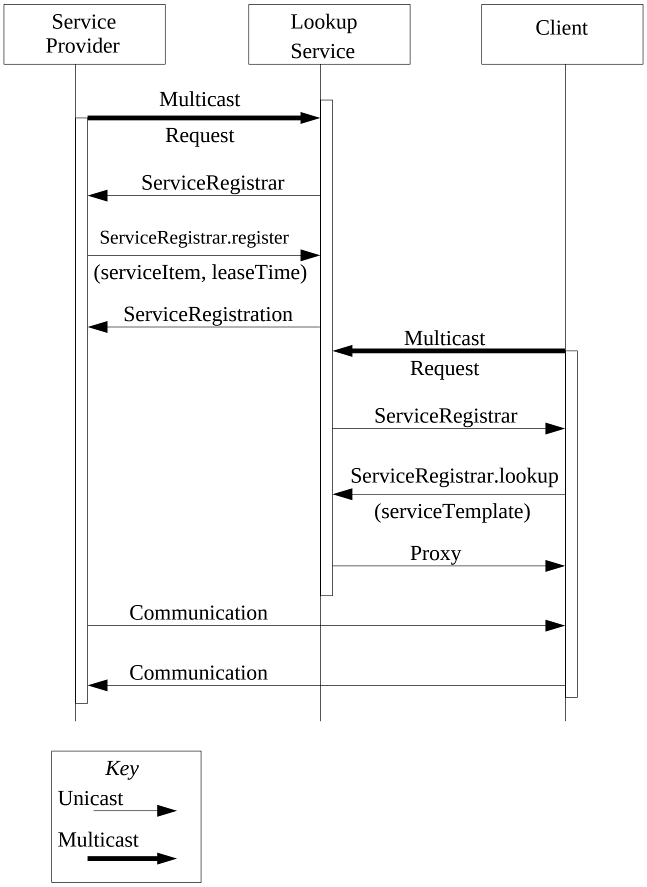

Figure 2: Jini Operation

- Unicast discovery protocol. This protocol is used by the discoverers when they know the address of the lookup service. In this situation, the discoverer delivers a request message via TCP unicast to the lookup service. The lookup service replies with a unicast TCP message.

Note that the last message in either of the previous protocols is always a message from the lookup service to the discoverers, which is sent via unicast TCP. This last message contains the proxy of the lookup service. This proxy is serialized according to the Java Object Serialization specification [**?**]. The proxy is an object that implements the *ServiceRegistrar* interface. This interface defines the methods needed by the discoverers to talk to the lookup service.

After the lookup services are discovered, the join protocol and lookup protocols can be used. These two protocols are not communication protocols. They only define a set of rules on how well behaved Jini entities should join a Jini community and how to look for a particular entity. These rules define, for instance, what the entities should do in case of failure in joining the community.

The process of finding the lookup services offered by a service provider and then registering with them is called discovery/join in the Jini architecture. The discovery/join process occurs every time that a service is plugged in. And in Jini, the process of discovering lookup services may be delegated to a third party.

For joining a Jini community, service providers can use the *register* method on the lookup service proxy. Service providers register with the lookup service by uploading a proxy. The proxy defines a Java interface of the service, specifying the methods supported by the service provider. One of the parameters of this *register* method is the lease period, which states how long the service will be available. This lease time can be either granted or denied by the lookup service. The lookup services respond by delivering a *ServiceRegistration* object. The *ServiceRegistration* object is more than a simple acknowledgement. For example, it is possible to ask this object whether the lease period has been granted.

Finally, clients can find services by calling the *lookup* method on the lookup service proxy. The parameter is a template specifying a set of attributes desirable in the service looked for. As response, the lookup service delivers the proxy of the service requested if it knows about the service requested. Otherwise it responds with a *Null* value. This proxy is the same that was uploaded by the service provider. After a client downloads a service proxy, it can establish communication with the service provider, since the downloaded proxy specifies how to talk to the service.

The Jini specification mentions that when there are no lookup services available, a client might use a technique called peer lookup. In this scenario, the clients multicast the same request packets, but they receive a direct response from the service providers instead of the lookup service. From these replies, the clients can select the services they are interested in and drop the rest.

2.2.2 Additional Features

Some important points to note about Jini are:

- Two main strengths in Jini are its ability to move code across the network and the use of leases. The leases make it possible to keep an acqurate list of services available in the network, although the lease characteristic is available in SLP as well.
- The services use objects for describing themselves, which is an adventege over the attribute/value pair list used by SLP. The service providers can use JavaBeans for their descriptions.
- As in SLP, the service interfaces must be standardized in order that the client can understand what the services are able to do.
- The security in Jini is left to Java and RMI. Jini does not address any specific means for dealing with security. For example, Jini can make use of the policy files that Java 2 programs use. Anderson and Karlsson make a good proposition for secure services in ad hoc networks in [**?**].

Finally, to be able to run Jini services there are some hardware and software requirements that the devices must support. They must implement the TCP/IP and UDP stack with multicast. Regarding software requirements, the devices must support Java with the Jini extension in order to run a Jini service provider or a Jini client. If the desegregated devices do not have the Jini extension it is possible for them to download it from another device if they support HTTP.

3 Bandwidth Usage Analysis

This section describes a bandwidth usage analysis performed on SLP and Jini. We first discuss SLP, followed by Jini. Our results are based on the SLP specification, the Jini specification, experiments and as an extension of a previous work [**?**]

3.1 SLP Analysis

There are many possible scenarios and different situations when using SLP. We will focus on the one illustrated in Figure 1, which shows a representative situation in SLP.

In that scenario there are three main processes: solicitation, registration, and lookup. The solicitation process is represented by two types of messages: the *ServiceRequest(Type)* and *DAAdvertisment(URL)*. This process is executed by the SA and UA.

The second one is the registration process. This process occurs when the SA registers with the DA. It uses two types of messages the *ServiceRegistration(URL, Types, Attributes)* and the *ServiceAcknowledgement*.

Finally, the lookup process executed by the UA. It uses two kind of messages as well: *ServiceRequest(Type, Predicate)* and *ServiceReply(URL)*.

We compute an upper bound on bandwidth usage, in bytes, by SLP in each of the processes and then we calculate the overall bandwidth using that information.

In the TCP/IP stack, SLP is placed over UDP[1]. SLP messages share a common header and are encapsulated within UDP header, IP header, and network header (e.g. Ethernet header).

In the sequel, variables $EDAADVER$, $ESRVACK$, $ESRVREG$, $ESRVRQST$, $ESRVRPLY$, URL, and PR respectively represent the size of an encapsulated *DAAdvertisment*, an encapsulated *ServiveAcknowlegement*, an encapsulated *ServiceRegistration*, an encapsulated *ServiveRequest*, an encapsulated *ServiceReply*, a URL, and a *Previous Responder* list.

Let N be the number of SAs, M be the number of UAs, P the service registration period, and L the service request period.

[1] We consider SLP over UDP only.

3.1.1 Discovery Process

The discovery of DAs is done through the solicitation process in our scenario. We consider two possible cases without DAs and with DAs.

- Without DAs. In this scenario where active discovery is used, the UAs and SAs broadcast a *ServiceRequest* looking for a DA in their scope. For discovering DAs, the parameter *Type* of the request is set to *service:Directory-Agent*. Since there are no DAs in this scenario, no agent returns a reply. Therefore, let's assume that the *ServiceRequest* messages are sent a constant C times. After doing that, the agents remain silent, assuming that there are no DAs available. We can illustrate this process with the following formulas:

  $$BUWODA_{d_{SA}} = ESRVREQ \times N \times C$$

  and

  $$BUWODA_{d_{UA}} = ESRVREQ \times M \times C$$

  where $BUWODA_{d_{SA}}$ is the bandwidth usage without DAs due to the discovery proces by the SAs and $BUWODA\,_{d_{SA}}$ is the bandwidth usage without DAs due to the discovery process by the UAs. The UAs and SAs looking for a DA broadcast a *ServiceRequest*, with the appropiate parameter, C times. To make a fair comparison with Jini, we will use the same constant in the Jini discovery process. We set C to seven in our simulation, since that is the value used by Jini.

- With DAs. In this second situation, we assume a DA in our scope. Therefore, after the first *ServiceRequest* message sent by the SAs and UAs they get back a *ServiceReply* message. The remaining $C - 1$ *ServiceRequest* messages are discarded by the DA. So,

  $$\begin{gathered} BUWDA_{d_{SA}} = [(ESRVREQ + PR) * C - \\ PR + EDAADVER] \times N \end{gathered}$$

  and

  $$\begin{gathered} BUWDA_{d_{UA}} = [(ESRVREQ + PR) * C - \\ PR + EDAADVER] \times M \end{gathered}$$

  where $BUWDA_{d_{SA}}$ is the bandwidth usage with a DA due to the discovery process by the SAs and $BUWDA\,_{d_{SA}}$ is the bandwidth usage with DA due to the discovery process by the UAs. The SAs and UAs broadcast their requests C times but only one of the request is acknowledged with the *DAAdvertisment* message. The *ServiceRequest* has a field for storing the services known. Since the DA responds once, the *ServiceRequest* message includes the address of the DA in all the requests but the first one.

3.1.2 Registration Process

The registration process is only needed when there is at least one DA in the scope. When there are no Das, this process consumes zero bytes.

In the presence of a single DA, over a duration of T seconds, the amount of traffic, in bytes, generated by the registration process is given by the following expression:

$$BUWDA_r = N \times T/P \times (ESRVREG + ESRVACK)$$

$BUWDA_r$ is the bandwidth usage when there is a DA due to the registration process. The amount of traffic generated by registration is the number of SAs times the frequency of registration times the size of an encapsulated *ServiceRegistration* message times the size of an encapsulated *ServiceAcknowledge* message.

3.1.3 Lookup Process

The request process can involve two scenarios: with DAs and without DAs. Note that this process, in both scenarios, involved two types of messages: *ServiceRequest* and *ServiceReply*s. Therefore, the bandwidth usage for both cases can be represented by the following formulas:

$$BUWODA_l = Req_1 + Rep_1$$

and

$$BUWDA_l = Req_2 + Rep_2$$

$BUWODA_l$ and $BUWDA_l$ are the bandwidth usage with and without a DA due to the lookup process. The Req_1, Rep_1, Req_2, and Rep_2 represent the request and reply messages in each scenario, without and with DAs.

- Without DAs. In this configuration, the multicast convergence algorithm is applied. Let us suppose that x% of the SAs reply after a first request, y% after a second request, and $z\%$ after a third ($x \leq y \leq z \leq 100$). The amount of bandwidth generated by *ServiceRequest* messages is modeled by the following formula:

  $$Req_1 = M \times T/L \times [4 \times$$
  $$ESRVRQST + PR \times N \times (x + y + z)/100]$$

  The amount of bandwidth generated by *ServiceRequest* messages is the number of UAs times the service request frequency times the size of four encapsulated *ServiceRequest* messages. If in the first *ServiceRequest* message, the previous responder list is empty, in the second it contains the addresses of x% of the SAs, in the third y% of the addresses, and in the fourth z% of the addresses.

  After the requests are received by the SA, they reply with *ServiceReply*s messages. Note that every reply contains a single URL. We assume that v% of the SAs return a reply ($z \leq v \leq 100$). The amount of bandwidth incurred to *ServiceReply* messages is given by the following formula:

  $$Rep_1 = M \times T/L \times v/100 \times$$
  $$N \times (ESRVRPLY + URL)$$

The amount of bandwidth due to *ServiceReply* messages is the number of UAs that return a reply times the service request frequency times the number of SAs times the size of an encapsulated *ServiceReply* along with a URL (we assume an average size for URLs).

- With a DA. In this case the multicast convergence algorithm is not applied. The amount of bandwidth due to *ServiceRequest* messages amounts to:

  $$Req_2 = M \times T/L \times ESRVRQST$$

  Every *ServiceRequest* message is sent using unicast to the DA and all URLs matching the required service are packed in a single *ServiceReply* message. We assume that v% of the SAs offer a matching service. The amount of bandwidth therefore corresponds to:

  $$Rep_2 = M \times T/L \times$$
  $$(ESRVRPLY + v/100 \times N \times URL)$$

  The amount of bandwidth due to *ServiceReply* messages is the number of matches times the service request frequency times the size of an encapsulated *ServiceReply* message along with N URLs.

3.1.4 Results

In a configuration without DAs, the overall amount of bandwidth $BUWODA$ generated is expressed by the following equation:

$$BUWODA = BUWODA_{d_{SA}} + BUWODA_{d_{UA}} +$$
$$BUWODA_l \qquad (1)$$

This formula expresses the traffic due to discovery and lookup process with N SAs and M UAs over a period of T seconds. Figure 3 plots the usage of bandwidth if T is fixed to one hour and L to 15 minutes (other fields in SLP messages such as service *Type*, *scope-list*, and *Predicate* are assigned representative values). Percentages x, y, and z are respectively set to 5, 8, and 10.

In this representative situation, usage of bandwidth is more sensitive to the number of UAs than to the number of SAs. In a configuration with a DA, the overall amount of bandwidth $BUWDA$ generated is expressed as the following formula:

$$BUWDA = BUWDA_{d_{SA}} + BUWDA_{d_{UA}} +$$
$$BUWDA_r + BUWDA_l \qquad (2)$$

This equations show the traffic due to discovery, registration and request processes, with N SAs, and M UAs, over a period of T seconds. Figure 4 plots the usage of bandwidth if T is fixed to one hour, L to 15 minutes, and P to five minutes. Percentage v is set to 10. For this particular situation, that may be considered representative, usage of bandwidth without a DA is substantially

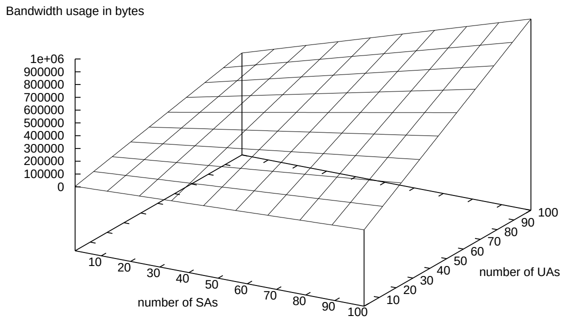

Figure 3: Bandwidth usage of a configuration with SAs and UAs only.

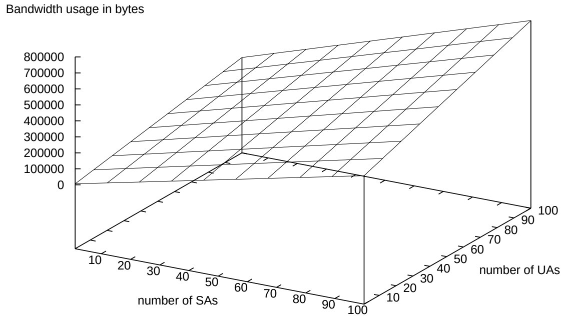

Figure 4: Bandwidth usage of a configuration with a DA, SAs, and UAs.

higher than in the situation with a DA (Figure 3) and it is more sensitive to the number of UAs than the number of SAs.

It is interesting to isolate the condition under which bandwidth usage with a DA is less than bandwidth usage without a DA, that is, the condition such that $BUWDA < BUWODA$. Simple algebraic manipulations and simplifications of Equations 1 and 2 lead to the following conclusion, $BUWDA < BUWODA$ if and only if

$$\begin{gathered}[PR \times (C-1) + EDAADVER](N+M) + \\ \tfrac{NT}{P}(ESRVREG + ESRVACK) < \\ 3 \times SRVREQ + \tfrac{1}{100}[N \times PR \times (x+y+z) + \\ ESRVRPLY \times (vN - 100)]\end{gathered}$$

With the addition of a DA, bandwidth usage is less as long as the amount of traffic genereted by the discovery process (*Previous Responder* list and *DAAdvertisment* messages) and the registration process (*ServiceRegistration* and *ServiceAcknowl-*

edgement messages are less than the amount of traffic generated by the lookup process, which includes the three retransmissions of the *ServiceRequest* message used by the multicast convergence algorithm plus the *ServiceReply* messages. Nevertheless, it tells us that if the number of optional retransmissions by the multicast convergence algorithm can be limited, the bandwidth usage of a configuration without DAs approaches the one with a DA while not having the overhead of a DA.

3.2 Jini Analysis

We will follow the same idea of the previous subsection in our Jini bandwidth usage analysis. We refer to Figure 2 for our analysis of calculating an upper bound of a simple Jini application.

The Jini application that we test is a simple *Hello World* program that can be found in [**?**]. There is a service provider which offers a service of the message *Hello World*. The lookup service used is the one that comes with the Jini distribution, called *reggie*. The client requests the *Hello World* service from the lookup service. This in turn responds with a proxy of the service. Once the proxy is obtained by the client, it displays the message. The display of the message is implemented in the proxy, not in the service, so there is no need of communication, in this particular case, between the service provider and client. We did not test the peer lookup technique.

Jini makes use of RMI to achieve the communication among the entities. RMI uses in turn a TCP connection. First, we define the amount of traffic generated by an RMI communication, and then we apply that formula to the Jini case.

RMI uses a TCP socket for communication. The amount of traffic generated by TCP is modeled as the equation:

$$T_{TCP}(X) = \alpha_{TCP} + \beta_{TCP}(X)X$$

In the above, α_{TCP} represents the overhead of TCP due to connection establishment and release, five packets of 66 bytes each (Ethernet header 26 bytes, TCP header 20 bytes, IP header 20 bytes) for a total of 330 bytes. $\beta_{\ TCP}(X)$ represents the overhead due to segmentation and acknowledgments of each individual data unit. It is defined as:

$$\beta_{TCP}(X) = \frac{2H}{X}\left\lceil \frac{X}{MSS} \right\rceil + 1$$

where H and MSS respectively correspond to the size of headers of TCP and IP encapsulation and maximum segment size (1460 bytes on most systems).

RMI follows a simple protocol for the communication. It uses two types of messages: requests and replies. The objects invoking a remote method send a request over a TCP connection and the remote method returns a value or an exception over the same TCP connection. The RMI messages include header and payload.

The header of the RMI request messages includes, besides other fields, an ID identifying the object and remote method invoked. RMI specifies three different protocols for the communication: StreamProtocol, MultiplexProtocol, and SingleOpProtocol, which is specified in the RMI header. The StreamProtocol only transmits the RMI packets over the TCP connection. The

MultiplexProtocol allows multiple sessions over the same TCP connection. The SingleOpProtocol uses the HTTP protocol on the top of the TCP protocol. It uses Post and it is intended to be used when there are firewalls between the remote objects. The payload of the request contains the parameters of the method invoked remotely. Each parameter is serialized according to the Java Serialization Object specification.

The reply message has an acknowledgment in the header. The payload contains the object, primitive data type or exception returned by the method invoked. The object or primitive data type returned or the exception raised are serialized according to the Java Serialization Object specification. Thus, assuming the use of the StreamProtocol, the RMI communication can be expressed as follows:

$$RMI(REQ_{RMI}(P_1, P_2, ..., P_N), ACK_{RMI}(RtrnValue)) = \alpha_{TCP} +$$
$$\beta_{TCP}(REQ_{RMI}(P_1, P_2, ..., P_N)) \times REQ_{RMI}(P_1, P_2, ..., P_N) +$$
$$\beta_{TCP}(ACK_{RMI}(RtrnValue)) \times ACK_{RMI}(RtrnValue)$$

We can define $RMI_{Req}(P_1, P_2, ..., P_N)$ and $RMI_{Ack}(RtrnValue)$ as follows:

$$REQ_{RMI}(P_1, P_2, ..., P_N) = ReqHdr + P_{1_{SER}} +$$
$$P_{2_{SER}} + ... + P_{N_{SER}}$$

$$ACK_{RMI}(RtrnValue) = AckHdr + RtrnValue_{SER}$$

where $REQ_{RMI}(P_1, P_2, ..., P_N)$ represents an RMI invocation of a method with parameters P_1, P_2, ..., P_N. The $ReqHdr$ includes 7 bytes specifying the protocol plus 34 bytes specifying the object and the method invoked, giving a total of 41 bytes. Then, we add the serialized parameters P_1, P_2, to P_N. $ACK_{RMI}(RtrnValue)$ represents the response of a remote invocation, which includes a 16 byte header plus the serialized return value.

As in the SLP case, we consider the three processes involved in the Jini operation: discovery, registration, and lookup. Some of these processes use the Java RMI specification, whereas others use only UDP or TCP connections.

3.2.1 Discovery Process

Figure 2, the discovery process, is represented by the *MulticastRequest* message and the *ServiceRegistrar* object. The *Multicast Request* is a UDP datagram, whereas the *ServiceRegistrar* is a TCP communication. $EJINIREQ$ represents the encapsulated multicast request, which includes the Ethernet, IP, and UDP header. $TSRVREG$ is the transmission of the lookup service proxy, the *ServiceRegistrar* object. Since this transmission is done via TCP, $TSRVREG$ is defined as follows:

$$TSRVREG = T_{TCP}(SrvReg)$$

where $SrvReg$ is the serialized *ServiceRegistrar*. The serialized *ServiceRegistrar* used by the *reggie* lookup service is 701 bytes long. Therefore the bandwidth usage by the service providers and client during the discovery is:

$$BUJINI_{d_{SER}} = [(EJINIREQ + HEARDFROM)$$
$$\times C - HEARDFROM + TSRVREG + DL] \times N$$

$$BUJINI_{d_{CLI}} = [(EJINIREQ + HEARDFROM)$$
$$\times C - HEARDFROM + TSRVREG + DL] \times M$$

where $BUJINI_{d_{SER}}$ is the bandwidth usage in Jini due to the discovery process by the services and $BUJINI_{d_{CLI}}$ is the bandwidth usage due to the discovery process by the clients. The multicast request packet includes a field called *Heard From* in the payload for storing a list of the lookup services that have responded. But the $EJINIREQ$ variable does not include this field. We add the variable $HEARDFROM$ to represents this field. After the reply of the lookup service, all the following multicast request packets include the ID of the lookup service in the *Heard From* field. We assume that the lookup service always responds after the first request message, therefore $HEARDFROM$ is sent together with all the multicast request messages but the first one, in which the *Heard From* list is empty. The DL variable represents the amount, in bytes, of the downloaded code needed by the service providers or the clients to run the proxies or the Jini classes. This code is downloaded during the discovery process.

3.2.2 Registration Process

The service providers register with the lookup service for a lease period of time P and then the lease is renewed. In Figure 2, the registration process is represented by the invocation of the method *register* on the *ServiceRegistrar* object with the item that has been registered and a lease time as parameters. The response is the *ServiceRegistration* object. For a period of time T, with N service providers, the registration process can be defined as follows:

$$BUJINI_r = (TREG + (T/P - 1) \times$$
$$TRENEWLEASE) \times N$$

where $BUJINI_r$ is the bandwidth usage due to the registration process in Jini, $TREG$ is the transmission required for the registration of the service, $TRENEWLEASE$ is the transmission required for renewing the service lease. Since this process is done via RMI, these variables can be defined as follow:

$$TREG = RMI(REQ_{RMI}(serviceItem, leaseTime),$$
$$ACK_{RMI}(serviceRegistration))$$

and

$$TRENEWLEASE = RMI(REQ_{RMI}(leaseTime),$$
$$ACK_{RMI}())$$

The *serviceItem* is an object of type *ServiceItem* that includes the proxy of the service provider, attributes, and service ID, whereas the *leaseTime* is a long value. The lookup service returns an object that implements the *ServiceRegistration* interface. Note that the renew process does not return any value. The service provider can call the method *getExpiration()* to know if the lease was granted. We assume that the lease is always granted.

3.2.3 Lookup Process

Clients look for services by asking for the service proxies to the lookup service. In Figure 2, the lookup process is represented by the invocation of the method *lookup* on the *ServiceRegistrar* object. The response is the proxy of the service requested. With M clients discovering at a ratio of T/L independent services, the lookup process can be defined as follow:

$$BUJINI_l = TLOOKUP \times M \times T/L$$

where $BUJINI_l$ is the bandwidth usage due to the lookup process, $TLOOKUP$ is the bandwidth usage due to the lookup queries. This process is done via RMI. Therefore, we can define it as follows:

$$\begin{gathered} TLOOKUP = RMI(REQ_{RMI}(serviceTemplate), \\ ACK_{RMI}(anObject)) \end{gathered}$$

The *serviceTemplate* parameter is an object that specifies the attributes, type and service ID of the service needed. The object returned by the invocation of the *lookup* method is the proxy of the service requested. If there are many services known by the lookup service that match the lookup request, the lookup service returns only one, which is chosen randomly. If there is no matching, it returns an object with a *Null* value. We assume that the service is always found.

3.2.4 Results

Based on the equations presented in the previous sections the overrall amount of communication used by Jini $BUJINI$ is:

$$\begin{gathered} BUJINI = BUJINI_{d_{SER}} + BUJINI_{d_{CLI}} + \\ BUJINI_{Reg} + BUJINI_l \end{gathered} \tag{3}$$

Figure 5 plots the usage of bandwidth of Jini based on the last equation. Variables N, M, T, P, and L are set as in Subsection 3.1. The other parameters are set according to the *Hello World* example mentioned above. In particular, the variable DL is set to 0, which means that the service providers and client do not need download code for running the lookup service proxy.

The Jini architecture is similar to the SLP architecture with DA. However Figure 5 shows a higher use of the bandwidth, even though no code was downloaded. Figure 6 plots Equation 3 when we take in account the code downloaded. In our experiments, the size of the code was approximately 60 Kbytes, we set DL to 60000.

Using Jini, it is possible to have communication between the service providers and the clients through RMI. However, the *Hello World* service is implemented in the proxy and therefore there is no need of communication between the client and service

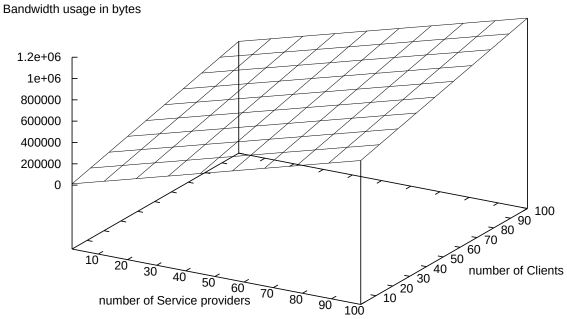

Figure 5: Bandwidth usage of JINI without downloading code.

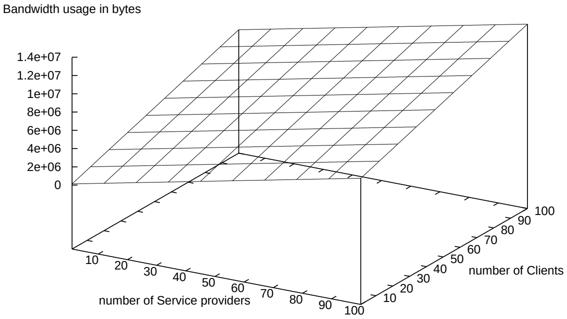

Figure 6: Bandwidth usage of JINI downloading code.

provider. We chose this example in order to make a fair comparison with SLP, since SLP does not address the communication between SAs and UAs.

There is also some traffic due to interactions with RMI activation daemons. These interactions are required to start Jini lookup services and service providers on demand. A RMI daemon normally resides on each node on which a lookup service and service provider are running. Hence, the RMI activation daemon overhead does not necessarily produce network traffic, just local traffic.

4 Conclusion

The resource discovery protocols are key for the full deployment of pervasive computing. Since many desegregated devices in pervasive computing are wireless, the bandwidth is an important resource that must be used efficiently. SLP and Jini are two important technologies for resource discovery that were discussed in this paper. We developed two main equations for characterizing the bandwidth used by SLP and Jini. In the case of SLP, two architectures were discussed, without DAs and with DAs. The important point in the SLP analysis is that, given both configurations, we identified under which conditions the bandwidth usage is higher in one of them.

The Jini configuration is similar to the SLP configuration with a DA but Jini makes higher use of bandwidth than the SLP architecture with a DA. It is important to mention that Jini might need more bandwidth when the services providers or clients need to download code for running the proxies or the Jini classes. We showed that in this case the bandwidth usage is more than 10 times higher than when there is no need of downloading code.

The advantages of SLP are that it is simple to implement, OS independent and uses lower bandwidth in the case of the configuration with DAs. However, SLP is only a string-based protocol for discovery purposes, which does not address communication among the desegregated devices. On the other hand, Jini is flexible in implementing any service and it is OS independent because of the JVM. Jini has as its main strength the ability to move code, although this ability can be regarded as a drawback since moving a small piece of code can involve a lot of traffic in the network. Regarding security, SLP makes use of digital signatures whereas Jini can use policy files. Jini has heavy requirements, and the bandwidth usage is higher than SLP although it is a more sophisticated system.

Finally, two other important technologies available in the market are the Bluetooth and Universal Plug-and-Play. The first one uses the Simple Discovery protocol for resource discovery, whereas the second one uses the Simple Service Discovery Protocol. Further work might include these technologies in bandwidth usage analysis.

References

[1] Ieee std. 1394a-2000, standard for a high performance serial bus, amendment 1.

[2] Linda group. http://www.cs.yale.edu/Linda/linda.html.

[3] Openslp. http://www.OpenSLP.org.

[4] The Home Network Phone Alliance. Simple, high-speed ethernet technology for the home. A White Paper, June 1998.

[5] F. Andersson and M. Karlsson. Secure jini services in ad hoc networks. Master's thesis, Royal Institute of Technology (KTH), February 2000.

[6] M. Barbeau. Bandwidth usage analysis of service location protocol. In *Proceedings of the 2000 International Conference on Parallel Processing Workshops*, August 2000.

[7] J. Birnbaum. Pervasive infomation systems. *Communications of the ACM*, 40(2):40–41, February 1997.

[8] L. Ciarleta and A. Dima. A conceptual model for pervasive computing. In *Proceedings of the 2000 International Conference on Parallel Processing Workshops*, Toronto, August 2000.

[9] Salutation Consortium. Salutation architecture version 2.0c. ftp://ftp.salutation.org/salute/Sa20e1a21.pdf, June 1999.

[10] Microsoft Corporation. Universal plug and play: Background. http://www.upnp.org/resources/upnpbkgnd.htm, 2000.

[11] R. Droms. Dynamic host configuration protocol. IETF RFC 2131, March 1997.

[12] W. Keith Edwards. *Core JINI*. Prentice Hall, Upper Saddle River, NJ, 1999.

[13] D. Farnsworth and B. Miller. Bluetooth specification version 1.0b, Service Discovery Protocol (SDP). http://www.bluetooth.com/link/spec/bluetooth_e.pdf, June 1999.

[14] E. Guttman, C. Perkins, and J. Kempf. Service templates and services: Schemes. IETF RFC 2609, June 1999.

[15] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. IETF RFC 2608. June 1998.

[16] USB Implementers Forum Inc. Usb 2.0 backgrounder. http://www.usb.org/developers/usb20/backgrounder.html.

[17] J. Kempf and E. Guttman. An api for service location. IETF RFC 2614, June 1999.

[18] Sun Microsystems. Java object serialization specification, revision 1.4. http://www.sun.com/research/forest/opj/docs/guide/serialization/, November 1998.

[19] Sun Microsystems. Java remote method invocation specification, revision 1.50. http://www.sun.com/research/forest/opj/docs/guide/rmi/, October 1998.

[20] Sun Microsystems. Jini architectural overview. Technical White Paper, January 1999.

[21] Sun Microsystems. Jini specifications 1.0.1. http://www.sun.com/jini/specs/index.html, November 1999.

[22] The HAVi Organization. The havi specification v1.0. http://www.havi.org/techinfo/index.html, January 2000.

[23] S. Shafer. Invited talk. In *Proceedings of the 2000 International Conference on Parallel Processing Workshops*, August 2000.

[24] HP JetSend Communication Technology. Protocol specification, document version 1.5. http://www.jetsend.hp.com/servlet/js/developer/ registered/disclaimer.shtml, May 1999.

[25] J. Veizades, E.Guttman, C. Perkins, and S. Kaplan. Service location protocol. IETF RFC 2165, June 1997.