MODELING HYBRID MULTIMEDIA N/W-WEB SERVICES USING RAPIDE ADL

Ahmed Sameh
Department of Computer Engineering, The George Washington University Washington, DC 20052

Keywords: SDP, ADL, Web Services, XML

Abstract: Dynamic delivery of network/web services across platforms/technologies will provide leverage of existing investment, scalability, and promote interoperability. In this research we envision a number of hybrid wireless terminal devices/nodes with at least one device/node bridge (called base station or access point) between the air and a physical wired network hosting a number of server applications in the form of real-time interactive network-web multimedia services located on either the wireless devices or on the nodes of the wired network. Each heterogeneous wireless mobile device/node that inhabit this hybrid platform/technology environment has a specific discovery protocol (Jini, UPnP), and a set of network-web services implemented in various languages (VB, VC++, Java) running on various platforms (XML, JSP). We model these networked enabled devices, applications, and services in Rapide ADL to seek out and find other complementary networked devices, applications, and services needed to properly complete specified wireless multimedia tasks. This federation of wired-wireless heterogeneous environment presents a modern, flexible infrastructure based on wired-wireless technologies and streaming standards. The federation is open for integration of new-networked Internet services and for evolving to provide a complete heterogeneous distributed computing environment.

1 INTRODUCTION

The number of network/web services is expected to increase enormously in the incoming era. Other than traditional services (e.g. printing, scanning and faxing), new network/web services for business/entertainment purposes, such as network based multimedia streaming systems, multi-players 3D games, interactive entertainment, unified messaging systems, mobile commerce, besides light weight services, such as restaurant directories and translators, are becoming available and highly important. For an effective use of these services, users should have means for direct and easy access to them. Service Discovery Protocols (SDP) represent a solution for services discovery and coordination (Richard, 2000) of such interoperating services in a heterogeneous environment with traditionally competing technologies.

The exploding deployment of network enabled wireless mobile devices, along with the expansion of networked and web services have created the need for users to easily manage these devices and services and also to coordinate with one another. SDP enables networked devices, applications, and services to seek out and find other complementary networked devices, applications, and services needed to properly complete specified tasks. A variety of SDPs have been proposed by the market and academia, including Jini, UPnP, SLP, Salutation and Bluetooth (Richard, 2000). For these protocols to co-exist, they should exhibit interoperability features. A number of bridging techniques have been proposed and implemented (Guttman, 1999). Interoperability among distributed object computing architectures such as .NET and RMI is becoming more and more inevitable for wireless mobile devices that speak different languages (WML/Java). The emergence of WML/Java as flexible and well-structured transportation models has made an entry point towards this goal. In a previous work (El-Ashmawi, 2003), we have utilized the flexibility of XML and the simplicity of UDP socket communication to build a generalized model of communication that supports interoperability among existing distributed object computing architectures. The proposed system is composed of a number of broker components that also act as naming services and several client/server objects. All components share the same feature of having built-in support for

Sameh A. (2004).
MODELING HYBRID MULTIMEDIA N/W-WEB SERVICES USING RAPIDE ADL.
In *Proceedings of the First International Conference on E-Business and Telecommunication Networks*, pages 79-87
DOI: 10.5220/0001384100790087

XML parsing and socket messaging. The proposed system is both platform and language independent. A mix of Visual Basic, Visual C++ and Java components prototype that demonstrated interoperability with .NET and RMI was developed and tested. In this paper, we extend this system to the wireless domain of multimedia to support a heterogeneous distributed environment of a number of wireless mobile devices that implement a mix of Java/XML network-web services in a wireless multimedia-networking environment. For the experimental part of this research, we have chosen the domain of wireless multimedia networking for testing our heterogeneous environment. This domain is rich with many network-web services such as: video and audio streaming, video conferencing, multi-players 3d video games, video on demand downloads, video editing and remote browsing, and unified messaging service. We envision a number of heterogeneous wireless mobile devices that inhabit this platform/technology hybrid environment, each with a specific discovery protocol (Jini, UPnP), and a set of certain network-web services implemented in various languages (VB, VC++, and Java)on various platforms (XML, JSP). We model these networked enabled devices, applications, and services in Rapide ADL to seek out and find other complementary networked devices, applications, and services needed to properly complete specified wireless multimedia tasks. Such wireless digital multimedia has a number of challenges such as error resistance, varying transmission speeds, adaptive decoding for limited power and processing capabilities in wireless mobile devices. To experiment with the proposed federation, we model and simulate the bridging in both service discoveries and deployments using Rapide ADL simulation and analysis toolset. We perform a number of simulation tests and use Rapide Poset viewer to analyze the simulator's output Poset tree of events

2 WIRELESS MULTI-MEDIA NETWORKING

Multimedia over wireless networks and/or the Internet has a number of obstacles. For example, with multimedia, and particularly with video, the amount of data that must be moved across the network is huge. The wireless network/ Internet is not always reliable and the bandwidth available for individual wireless devices is frequently insufficient. The complexity of network routing, unexpected bottlenecks, limited devices processing and battery power can cause that data streams of the digitized video to be delayed and/or arrive out of sequence.

On the service development side, there are currently two competing schools that promote two independent solutions: XML-based solution, and Java-based solution. We deploy these two solutions in our experimental heterogeneous environment (section 5).

In the XML-based solution (El-Ashmawi, 2003) a Web service is a form of RPC that uses XML and HTTP to make functions/services available over the Internet. They are based on three technologies: Universal Description, Discovery and Integration (UDDI), Simple Object Access Protocol (SOAP), and Web Service Description Language (WSDL). Their main purpose is to provide loosely coupled, course-grained interoperation among applications in a heterogeneous environment. B2B is currently a popular special case of Web services. It goes through phases of: service definition, implementation, testing, discovery, and finally deployment. UDDI is a directory for storing information about web services. In it, each web service interface is described by WSDL. UDDI communicates via SOAP. SOAP is a communication protocol via Internet. It is used for communication between applications, and it is based on XML. WSDL is used to locate, and describe web services. It is written in XML.

In the Java-based solution (El-Ashmawi, 2003) on the client tier, we either have a thick client- J2ME-based application using for example Kjava, CLDC, or MIDP, or a thin client- microbrowser-based: CHTML/i-Mode. On the web tier (wireless portal server) JSP, Servlets, JAXP are used. On the middle Ware Tier: application server, Entity Beans, Session Beans, Message Beans, RMI, RMI over IIOP, JNDI. IMAP server, LDAP server are used. On the backend tier: database, EIS, JDBC, are used. Wireless Java technologies such as J2me GUI (XML parser) on client's handheld devices, JSP/XML (JAXP) wireless protocol, JSP/WML and JSP/HTML wireless protocols, EJB for the application server are used. J2me-enabled handsets can communicate directly with HTML servers.

3 MODELING HETEROGENEOUS SERVICE DISCOVERY PROTOCOLS

SDPs are re-shaping the way software and network resources are configured, deployed, and advertised, all in favor of mobile wireless users. They easily enable wireless users to find and locate needed services effectively. They also give these users automatic access to needed devices and services,

without the need for manual configuration. For example, a device would transparently connect to the service address and automatically download needed drivers. It is of special importance for mobile devices, since it facilities query and search mechanisms that allow users to accurately select specific types of services. A client can choose a specific service, with specific location and characteristic, in wireless multimedia for example device-2-device phone, videoconference, video on demand, interactive TV, video rental services, video news distribution, multi-player 3D games, interactive 3D digital video, unified messaging system, integration of Web with traditional broadcast media, access to digital libraries, electronic guide and navigator, dating services, promotional services, collaborative work, telemetry services, Webcams, digital content delivery, and other innovative video and multimedia applications. Unlike classical lookup protocols, most SDPs require no human administration. They are automatically configured using multicast messages on known multicast groups or using centralized databases that are self-configured. This reduces the hassles of building and maintaining a network since they cut short most of the configuration efforts. Among the most prominent SDPs are: Jini developed by Sun Microsystems, and Microsoft UpnP (Dabrowski, 2001).

In a previous work (El-Kharboutly, 2002), we have built on the idea of a Jini network proxy described in Jini Device Architecture and based on the efforts of Eric Guttman in (Guttman, 1999) to build a Jini-UPnP Bridge that is an entity that enables services that support UPnP protocol to be reachable by Jini clients. For Jini clients, Jini-UPnP is a transparent layer that they are unaware of. The UPnP services that are advertised via the bridge are treated as native Jini services. The proposed Jini-UPnP Bridge has been modeled as a special network node that can communicate with other network nodes in both Jini and UPnP protocols. It mainly acts as a *Service User* (i.e. Control Point) in UPnP environment and a *Service Manager* (Service) in Jini environment. It waits for announcements made by UPnP devices and services that are willing to advertise their presence to the Jini clients and acts as a representative, almost a mirror for them in the Jini environment. In this work, we have modified the work done in (El-Kharboutly, 2002) to allow for a two-way bridging instead of a one-way bridge.

The main idea of the modification of the Jini-UPnP bridge is to prepare an appropriate entry for UPnP services, in the Jini Lookup Service. This involves primarily setting the appropriate attributes required and creating a service object as part of Jini service's registration. UPnP services that are willing to advertise their presence to Jini clients are not required to have a JVM installed. They are mainly required to have a Jini driver Factory (Guttman, 1999). A Jini driver factory is a (*.jar) file that bares a manifest for the advertised service. A Java Archive File (*.jar) file is used to bundle multiple files into a single archive file. Typically a JAR file contains the class files and auxiliary resources associated with applications. The bridging process is modified through the following steps: -The Jini-UPnP bridge searches the UPnP reachable entities to find devices and services that have Jini driver Factory or waits till it receives announcements made by Jini driver Factory services. Once a Jini driver Factory service is found, the Jini-UPnP bridge obtains a complete description of the service including attributes, GUI URL and control URL. -The URL of the Jini driver factory is composed by extending the control URL with a unique identifier. The Jini driver factory is downloaded using GET method over HTTP. -The Jini-UPnP bridge performs attributes transformation from UPnP format to Jini format to prepare for service registration. -Upon successfully translating the entire service attributes and obtaining the Jini driver factory, the Jini-UPnP bridge registers the discovered service with Jini Lookup Service. Using the Jini driver factory, the bridge creates a service object that is used for registration. Registration is done by sending a join request with all necessary attributes to Jini Lookup Service that adds the new service to its cache. -Whenever a Jini client needs our bridging service, it contacts Jini Lookup Service and downloads the instantiated object that is used to drive the service. Like any typical Jini service, the Jini-UPnP bridge should be equipped with JVM to be able to participate in the Jini SDP.

4 MODELING DEPLOYMENT OF HYBRID NETWORK-WEB SERVICES

WML/XML encoding of messages allows for interoperability and standardization. Each and every node/device in the heterogeneous environment should have the ability to use standard socket communication and parse WML/XML strings. The preferred method of communication is UDP sockets for the flexibility and the ability to use message broadcasting. In addition, UDP sockets would fit more with the active nature of the wireless multimedia environment and the need for rapid and quick switching of sources and targets of communication. For example a node/device in the environment may listen for incoming requests from

different clients while responding to every client using the same open socket without opening another one. Nodes/devices that are Java-based can also use RMI for intercommunication.

Server services which are ready to expose their services are run once with the command line parameter "/r" to inform the broker residing on the same node/device of their existence and exposed interfaces. This is done through sending XML messages to the broker on the port it is listening to, or a Java RMI registry. Once the system is up, there are three possible scenarios that can occur: The first is called transparent addressing, where server components register themselves with their local brokers. When a client component needs to invoke a method on a previously registered server component but doesn't know its location, a remote method invocation for a *GetObject method* is sent as a broadcast XML message containing the server component name to be instantiated. Each broker component receives the message and searches its local registration files for the component name. The broker which finds the component then starts the required component as a process; sending it the address and port of the client in the form "xxx.xxx.xxx.xxx:xxxx" as a command line parameter. Then the broker becomes free to handle more requests. If the client component times out waiting for a response, the component is considered not registered. When a server component starts up, it sends an XML encoded result message to the client component containing its address and port, which will be considered the component reference by the client afterwards. Subsequently the server component becomes ready to process further requests. The client component now begins to send XML-RPCs to the server component and receives results also as XML encoded result messages. The client can reuse the instantiated server as many times as it wants. When the client component does not need the services of the server anymore, it sends a shutdown message to the server, which then closes its port (Figure 1).

The second is called targeted addressing. Due to the overhead that broadcasts cause to the networking resources, Targeted component addressing would provide a less transparent solution with less networking overheads. In this scenario, the client component happens to know the node address where the server component resides. It sends a direct message to the broker on the target node requesting a reference of the server component. This could be incorporated into the previous scenario by adding a functionality to the client component to save the location of the node of a certain server component into local registration files for future reference. The broker then performs as before and passes the address and port of the client as a command line parameter to the server component as it starts. The rest of the scenario continues as in Transparent Addressing (Figure 2).

The third is called parallel processing. The system here could be viewed as a message passing protocol for implementing parallel processing. The scenario is based on redundant components that have the same functionality present on different nodes/devices in the environment and makes use of the targeted Component addressing method above.A master component divides the problem into several smaller tasks (a divide and conquer step). It encodes the function calls to process these tasks into XML messages. It then starts sending different brokers the *GetObject* method invocation as required and waits for component references, which it stores, for future use. Each broker, receiving an invocation from the master component, will spawn a server process, passing it the master component address as a command line parameter. The master component then starts communicating with the required identical components sending each its share of the problem (single program multiple data stream). It will then start receiving the results from the invoked components and then reassembles the parts of the main problem (embarrassingly parallel master-slaves). Each of the invoked server components will wait for a *SHUTDOWN* call and then close its port (Figure 3). In the experimental part (section 5), a library called XMLRPCLib.dll is built which exposes a class called XMLRPC used in formatting XMLRPC calls and results as mentioned above. This library is ported to the three languages used for coding network-Web services: Visual Basic,Visual C++ and Java. As for Visual Basic, it is built as a COM library which

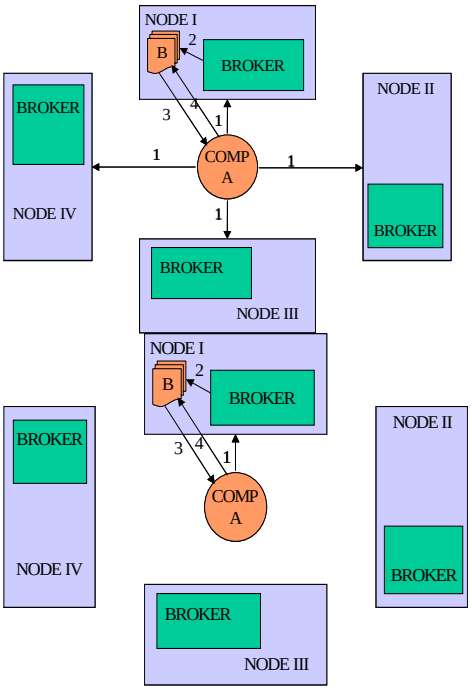


Figure 1: 1) Broadcast message from component A, 2) Broker on node I found required component B and starts it, 3) Component B sends its address to component A, 4) Component A sends method invocation message on component B.

Figure 2: 1) Direct message from component A to Broker, 2) Broker on node I found required component B and Starts it, 3) Component B sends its address to component A, 4) Component A sends invocation message to component B.

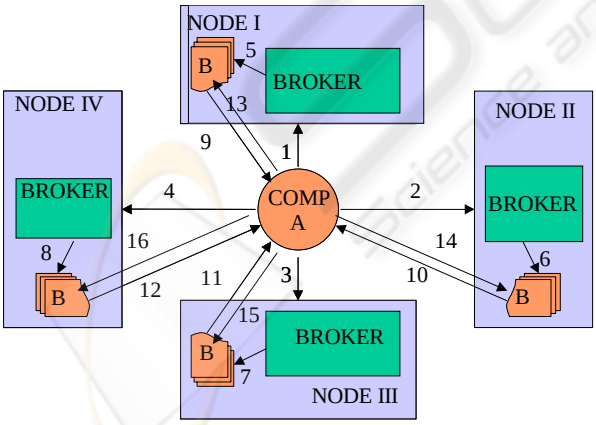


Figure 3: 1,2,3,4) Component A sent direct message to broker on nodes I, II, III, IV, 5, 6, 7, 8) Brokers on appropriate nodes starts component B, 9, 10, 11, 12) Component B on each node send their addresses to the master component A, 13,14,15,16) The master component starts sending each B component its part of the problem to solve.

demonstrates the interoperability of the proposed system with COM. For Visual C++, this library is built as a native C++ class and for Java, it is built in 2 forms; as a native Java class and as an RMI object to demonstrate interoperability with Java RMI. As such Java-based services can intercommunicate using either XML messaging or RMI invocation. The XMLRPC library is built to expose a set of utility functions to simplify the process of encoding and decoding of XML messages, an essential task of each component in the heterogenous environment. Each function in the library simplifies the access of XML Documents and maps system related functions to XML related tasks. Functions in the library are described in (El-Ashmawi, 2003). The library is implemented as a COM dynamic link library in Visual Basic to demonstrate interoperability of the proposed system and COM. As mentioned before the library is also ported to Visual C++ as a native C++ class and to Java as a native Java class. The Rapide ADL has a number of preprocessors to instrument the above service codes written in Java, VB, and VC++ so that they produce time-stamped events during their execution in the modeled environment. The generated events are then analyzed by the Rapide tool- see section 5.

As a simple proof of concept; an ADL model for a video stream broadcasting service is built using the Rapide ADL. The services development environment chosen was Microsoft Visual Basic 6.0 IDE using the Microsoft winsock ActiveX control for socket communications and the Microsoft XML parser COM library MSXML.DLL for XML parsing and building. The DOM model for XML parsing was used rather than the SAX oriented one, which suits the nature of the proposed environment more. Microsoft Visual C++ 6.0 was used to build a C++ server service components using the same COM XML parser library and native MFC Socket classes. Borland's JBuilder IDE with JDK 1.3.1 was used to develop Java client components using the built-in Datagram Socket class and the freely available Apache Xerces XML parser library (Grag, 1996). For the video stream broadcasting service three components using the above were built: 1. A Broker component, which listens to port 5000 waiting for requests using the *GETOBJECT* method. It then fetches the path for the executable for this object from an XML-Formatted file called components.xml, which resides on the node/device and contains all the registered components within this node/device. When it finds the path of the object, it executes it and passes it the address and port of the caller process. The Broker component had to be designed as a multithreaded component, which queues the incoming requests in a queue of messages and handles the queue in another thread.

This way the time for processing each incoming message would not affect the response time to another incoming message, which may be dropped if intense processing is required. 2. A server component is a media player that broadcasts/displays the video stream. The server component is spawned by the broker and immediately responds to the caller by sending it its address and port. It gets the reference for the caller from the command line parameter set by the broker. It then gets the actual RPC call with the method RECEIVE-STREAMS with parameter 1 as the screen size, and parameter 2 as the name of the video file. The component initializes its local stream player and starts to buffer the received video streams then sends back to the calling process an RPC Acknowledge message. It then waits until it receives a SHUTDOWN message from the caller and then ends execution. In case of failure to communicate, the component waits for one minute and shuts itself down if it does not receive any messages during this time. 3. A master component that contains the streaming video to be broadcasted gets an array of the required object references from the broker by sending it a *GETOBJECT* method call and then begins distributing the video stream through its local media player to the server processes and waits for them to pass back acknowledgments. Finally, it sends a shutdown call to all spawned processes using the array of object references it has. In case of transparent addressing, it broadcasts the message call *GETOBJECT* to all brokers on the network until it gets all the object references it needs. Afterwards, unnecessary spawned server processes (running media players) will die if they don't receive any messages within a minute.

In the model, the Video Stream Broadcasting service is ADL modeled using 2 different addressing: -Master and n-node/device components intercommunicating using direct addressing method. -Master and n-node/device broadcast components intercommunicating using transparent addressing method. Setting up the services was done by running the broker on each node/device in the environment and then registering the server component by running it (media players) with the switch /r. The master component is then run (media player) from any node/device, which may or may not have a running broker. In our case, this node/device had a running broker. For running the Java RMI services, the RMI registry was run on the node/device harboring the XMLRPC class component and the component was run to register itself with the registry, ready for use. In the model, the Master is a wireless access Web node that represent a kind of Wireless Application Service Provider-video on-demand broadcasting station. The other 4 devices act as clients who perform kind of video on demand from the master node. Clients and servers can spontaneously and unpredictably join and leave the environment. The Master services are re-entrant, that is, multiple instances of the same service might be running on the same node/device serving multiple clients. It is capable of preparing contents for different devices. This is useful in particular with the video on-demand facility of this video stream broadcasting service. In this figure we envision four wireless terminal devices and one master node bridge (called base station or access point) between the air and a physical wired network. Then a number of servers hosting the video broadcasting service application in the form of network-Web services to be used by the terminal devices are located on either the wireless devices themselves or on the nodes of the wired network as shown.

5 EXPERIMENTAL WORK: TESTING FUNCTIONALITY, CONSISTENCY AND PERFORMANCE MEASURES

After modeling both the discovery and deployment separately, we brought them together in one large Rapide model. The Rapide toolset provides a set of compilation and runtime execution tools whose output is a simulation of the Rapide architectural model. The output of the simulation could be analyzed in various ways, including constraint checking, analysis for surprises and depiction of behavior. We chose to analyze the output of our simulation using the **P**artial **O**rder **S**et (Poset) browser. We have conducted three experiments to test functionality/consistency and measure the performance of the proposed hybrid discovery. The usage of a bridge in a hybrid system implies the presence of an overhead in time and resources. In the first performance experiment we are interested in measuring the overhead of discovering a UPnP service from a Jini-based device compared to having that same service as a native Jini service. The overhead is measured in terms of time and the number of messages exchange. First, we ran the Jini Rapide model with a topology of one Jini Service Cache Manager (SCM), two Jini Service Users (Jini SUs) and one Jini Service manager (Jini SM), where one of the Jini SUs requests a service (view a video stream) of the same type as that offered by the Jini SM. We measure the time taken and the number of messages exchanged since the Jini SM starts up and until the Jini SU receives the service description. Next, we run our Jini-UPnP Bridged model with a

topology of one Jini SCM, two Jini SU, one Jini SM, one Jini-UPnP Bridge, one UPnP SU and two UPnP SM. The time taken by a Jini SU to discover a requested UPnP service is measured. This time value is the sum of the time taken for Jini-UPnP Bridge to discover the services; the time the bridge registers this service with the Jini SCM and the time the Jini SCM forwards the service description to the interested Jini SU.

Measurements for Jini are done on two stages; first we measure the time taken for Jini SM to register with SCM and the number of messages needed. We assume that SCM discovery has already taken place. The time taken for this operation, as shown in the results is **0.064s** , and the number of messages exchanged is four messages (NUM MSGs 1 :4). The second stage is where the SCM starts matching the newly added service description to the available SU requests. Two messages are exchanged for this operation to complete and the total time needed is **0.00081s**. Thus the total time for the whole operation starting with SM registration to SU discovery takes **TOTAL TIME = 0.06481s** on average.

Bridging a UPnP SM service to be reachable for Jini SUs is done in three stages. First the Service SM is discovered by the Jini-UPnP bridge, then the bridge registers the service with Jini SCM. The time taken for a Jini-UPnP bridge to discovery and obtain the complete description of Jini Factory service is **1.00132s** where five messages are exchanged in this operation. Secondly, the bridge registers the newly discovered service with the SCM by exchanging two messages in **.00022**. The last stage is where the SCM matches the added service to the notification for services that SUs have registered with the SCM earlier. This operation exhausts about **0.00061s**. The total time consumed in the process of bridging **TOTAL TIME = 1.00215s.** Comparing the results for a native Jini service to that of bridging the service through Jini-UPnP Bridge, it is clear that the bridging process has an overhead of about **0.93734s** or a 93.5% overhead.

Network Bandwidth is a main factor in the behavior of any distributed system. The performance of different entities in a SDP is very much affected by network delays as a main parameter. In our model for Jini-UpnP-Jini discovery, we simulate network bandwidth by having network delay as one of the main Rapide ADL model input parameters. Parameters are defined for unicast and multicast delays between any pair of nodes and also for the network as a whole. The following tests record the effect of varying network delays on the performance of UPnP-Jini-UPnP discovery. In the pervious experiment we were interested in measuring the overhead of discovering a service in terms of time and number of messages. We fixed the TCP/IP network delay to a typical network delay value of 10-100 µs uniform. To measure the performance of the Jini-UPnP-Jini discovery in a light loaded network, we repeat the experiment done in the previous section with the same input parameters, yet changing the TCP/IP network delay to **10-30 µs** uniform. The results would be compared to those obtain in the pervious experiment. We repeated the experiment ten times to compute the average overall time taken by the bridge. Compared to the results obtained in the previous experiment, the discovery performance increases about 0.071 % with a less loaded network (i.e. higher bandwidth) of 10-30 µs uniform delay. The results show an improved value for the time of registration with the bridge from 1.00132 s in normal network to 1.000617 in a less loaded network. We are more interested in the last time value (**Overall Time)** since the time taken to download the Jini driver factory is a factor of it. The results are up to our expectations since an overall improvement in time delay is noticed.

To measure the performance of Jini-UPnP-Jini discovery in a congested network, we apply the same experiment with a higher network load with the same input parameters, yet changing the TCP/IP network delay to **80-100 µs** uniform. The results would be compared to those obtain in case of typical network delays. We repeated the experiment ten times to compute the average overall time taken by the bridge. Compared to the results in normal network condition that are obtained in the previous experiment, the discovery performance degraded about 0.034 % with a congested network (i.e. low bandwidth) of 80-100 µs uniform delay. The result is as expected since the effect of having a low bandwidth is of direct effect on the time taken to transfer messages and to download Jini driver factory. The overhead in time is more obvious in the time taken for registration with the bridge, as downloading the Jini driver factory file is a factor in it. We conduct a topology of five UPnP SMs to be bridged, one UPnP Jini Bridge, one UPnP Service User, one Jini SCM, two Jini SUs and one Jini SM. We assume the same input delays and parameters presented above. We record the time taken for a Jini SU to discover a requested UPnP service. This time value is the sum of the time taken for Jini UPnP Bridge to discover the services; the time the bridge registers this service with Jini SCM and the time the Jini SCM forwards the service description to the interested Jini SU.

Experiments were run with different addressing techniques for each type of component (VB, VC++ and Java). The parallel video stream broadcasting was run on 8 to 14 nodes/devices using both the targeted component addressing and the transparent

(Broadcasting) component addressing scenarios. Also the Java component was tested using both the native XMLRPC library and the RMI dependent library. For each run in the experiments the timing in seconds was recorded and a mean of 10 different runs was obtained. It is important to note that Rapide (Lucham, 2003) has a number of preprocessors to instrument service codes written in Java, VB, and C++ so that they produce time-stamped events during the execution of the modeled environment. The generated events are then analyzed by Rapide tools. Although Visual Basic does not support multithreading programmatically, using the DoEvents statement in the queue-handler function and the event driven socket implementation provided a workaround for this. The process of registering and deregistering a component is merely adding a <COMPONENT> tag to the Components.xml file with the appropriate attributes (NAME and PATH) or updating the attributes for an already existing component and removing the whole tag for a component to deregister it as mentioned in the function implementations in the XML library.

Surprisingly enough, the indirect (broadcasting) addressing scheme on n nodes/devices outperformed the direct targeted addressing method using all types of components but was more outstanding in the case of VB components. The broadcast method, although exerts heavy loads on the network, prefers the most responding nodes/devices and hence the better performance. As regards to the VC++ and JAVA components, the same applies, but due to the delayed response time, the difference between the two topologies is masked (Figures 4, 5, 6, 7 and 8) (El-Ashmawi, 2003). Finally, as for the comparison made between the Java native components and the Java RMI components, the RMI components showed a delay in both addressing used which was more or less of the same proportion (Figure 9). However, as can be noticed, as the nodes/devices size gets larger this delay is masked and the performance gets even better with the Java RMI components.

6 CONCLUSIONS

The problem we addressed in this research is enabling thin servers and lightweight devices to offer their services to hybrid clients through passive and indirect registration using heretrogenous discovery/deployment strategies. We used architectural models of Jini, UPnP, Jini-UPnP bridge, UPnP-Jini bridge, VB services, VC++ services, and Java services as a basis to create hybrid discovery/deployment environment. For testing and simulating the environment, we created a hypothetical topology of Jini, UPnP clients and VB, VC++, and Java services. Using Rapide ADL, we have simulated the topology to verify its correctness and measure its performance. The proposed federation confines to Internet standards, maintains platform and language neutrality, integrates with current distributed architectures and conforms to object orientation standards.

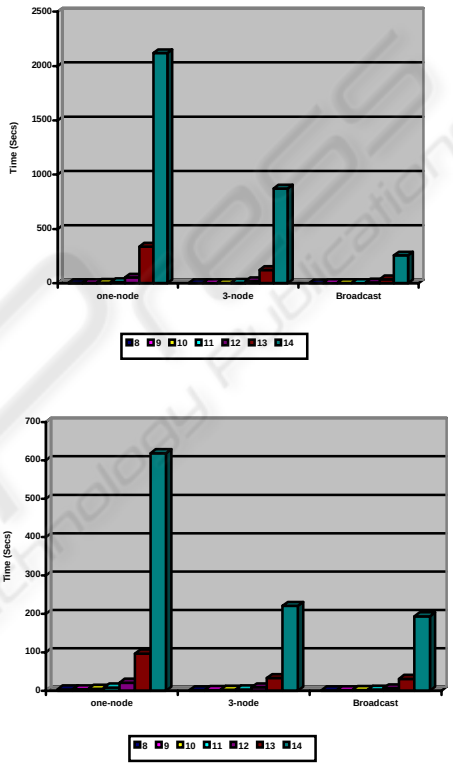
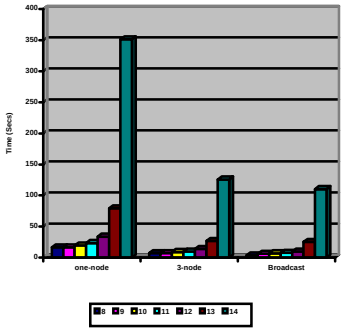


Figure 4: Chart showing comparison of the multi-requests one-node, Targeted n-nodes/devices and n-nodes/devices Broadcast algorithms with the VB components. Figure 5: Chart showing comparison of the multi-requests one-node, Targeted n-nodes/devices and n-nodes/devices Broadcast algorithms with the VC components



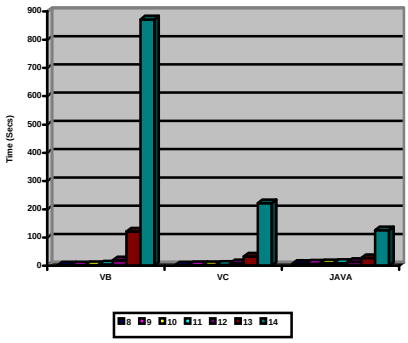
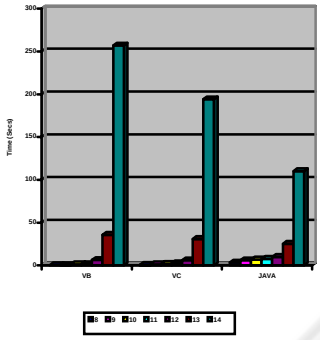


Figure 6: Chart showing comparison of the multi-requests one-node, Targeted n-nodes/devices and n-nodes/devices Broadcast algorithms with the JAVA native components

Figure 7: Chart showing times taken for the Targeted n-nodes/devices algorithm with the VB, VC and JAVA native components



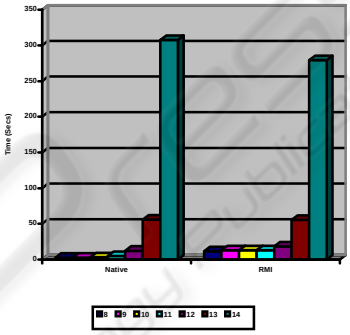


Figure 8: Chart showing times taken for the n-nodes/devices broadcast algorithm with the VB, VC and JAVA native components.

Figure 9: Chart showing times taken for the multi-requests one-node algorithm with the JAVA native components and Java RMI Components

REFERENCES

Richard, G., 2000. Service Advertisement and Discovery: Enabling Universal Device Cooperation, *IEEE Internet Computing*.

El-Kharboutly, R., 2002. Modeling Jini-UpnP Bridge Using Rapide ADL, M.Sc. thesis in Computer Science, The American University in Cairo.

El-Ashmawi, H., 2003. A New Message-Based Protocol for Building a Platform and Language Independent Distributed Object Model, M.Sc. Thesis in Computer Science, The American University in Cairo.

Guttman, E. and Kempf, J., 1999. Automatic Discovery of Thin Servers: SLP, Jini and the SLP-Jini Bridge, *Proc. 25th Ann. Conf. IEEE Industrial Electronics Soc. (IECON 99), IEEE*, Press, Piscataway, N.J.

Luckham, D.: Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events, http://anna.stanford.edu/rapide

Garg, V., and Wilkes, J., 1996. Wireless and Personal Communications Systems, Prentice Hall Wireless.

Dabrowski, C. and Mills, K., 2001. Analyzing Properties and Behavior of Service Discovery Protocols using an Architecture-based Approach, *Proceedings of Working Conference on Complex and Dynamic Systems Architecture*.