The μJini Proxy Architecture for Impromptu Mobile Services

Choonhwa Lee[1], Sumi Helal[2] and David Nordstedt[3]

[1] College of Info. & Comm.
Hanyang University, Korea
lee@hanyang.ac.kr

[2] CISE Department
University of Florida, USA
helal@cise.ufl.edu

[3] Phoneomena, Inc.,
Gainesville, Florida, USA
david@phoneomena.com

***Abstract* – Wireless hotspots and broadband coverage are permeating the globe giving rise to interesting spontaneous networking scenarios. The discovery and delivery of the most relevant and suitable services in a spontaneous network is a problem of paramount importance. In this paper, we propose an architecture that can support context-aware service discovery and delivery for resource-constrained mobile devices. At the heart of the architecture is the μJini proxy which mediates the discovery and delivery processes. Its delivery subsystem provides thin-client based adaptation for fat service delivery that does not fit into thin client devices. To demonstrate its effectiveness, we developed a proof-of-concept prototype, focusing first on J2ME smart-phones.**

I. INTRODUCTION

Thanks to recent technology advances, mobile devices are shrinking in size and increasing in capability and widespread use. Additionally, the proliferation of spontaneous networks is creating new opportunities for mobile users who come close in proximity to location-based and other context-relevant services. In fact, we should be very close from realizing the vision of fully impromptu, service discovery and delivery for mobile users. There are a number of challenging problems that need to be solved however before this vision is satisfactorily realized. One challenge is maintaining an acceptable quality of service in discovering and receiving (delivering) a service on the mobile device. This is indeed challenging when the variability of the mobile device capability and the variability of the service resource needs are considered. Another challenge is the focusing the discovery to the most relevant and suitable services given the contexts of the mobile user. Without such focusing the mobile user could get bombarded with a morass of unrelated or unusable services.

To meet these challenges, we have developed a new architecture for dynamic service discovery and delivery for a full spectrum of client devices, ranging from tiny, resource-poor devices to powerful workstations. Our μJini proxy architecture has been designed to fill the gap (i.e., context-aware service discovery and device-independent service delivery) that current technology is lacking to fulfill the promise of impromptu service discovery. It is capable of capturing implicit context information relevant to the specific type of service being discovered. On delivering a desired service, its delivery subsystem performs thin-client based adaptations that use target device capability as first-class context information to complete the dynamic service acquisition scenario.

In this paper, we describe the **μJini** architecture and prototype implementation, and present experimental results of an evaluation study of the service discovery and delivery system.

The rest of the paper is organized as follows. Section 2 motivates our research by reviewing Java technologies for mobile services. After introducing the architecture of the **μJini** proxy system, Section 3 elaborates on the thin-client based service adaptation followed by performance measurements. Finally, previous research efforts related to mobile service access are discussed in Section 4.

II. JAVA TECHNOLOGIES FOR MOBILE SERVICES

User mobility inherent in mobile computing environments dictates support for service portability as well as dynamic service discovery. For instance, a user should be able to access the same (dynamically discovered) service on his desktop computer at work, on his auto PC on the road, or on his smart-phone in the street. To achieve such mobility and device independence, we take advantage of Java's platform independence by basing our framework on Java technologies such as Java runtime environments for mobile devices and Java service discovery protocols.

J2ME technology is a Java platform for consumer and embedded devices such as mobile phones, PDAs, and set-top boxes [1] [2]. It defines Java runtime environments optimized for a particular range of devices and target markets in terms of configurations (CLDC and CDC), profiles (MIDP, FP, PP, etc), and optional packages. For instance, a Java environment for J2ME smart-phones is typically made up of CLDC, MIDP, and other optional packages, which can not execute full-blown J2SE services or even J2ME CDC services. However, the ubiquitous vision of mobile computing requires that a Java application be usable by any mobile device, regardless of which target platform was initially intended.

To access services on the move, a mobile device should be provided with a means to first discover it. Jini [3] fills up the need for this dynamic service discovery in Java environments. Built on top of Java RMI system, it provides a framework for dynamic service advertisement,

discovery, and invocation. Jini's RMI requirement sets its minimal runtime environments to J2ME CDC/RMI Profile, which typically can not be met by small mobile devices. Therefore, the Jini Surrogate Architecture [4] has been introduced to allow the resource-poor devices to participate in a Jini network. Our µJini protocol, which is part of the µJini proxy architecture, addresses this problem. Our µJini proxy system addresses the two main issues of context-aware service discovery and subsequent device-independent service delivery.

III. µJINI PROXY ARCHITECTURE

Built on top of Jini and AT&T VNC (Virtual Network Computing) [5], the µJini proxy system in Figure 1 performs service discovery and delivery. More specifically, the µJini proxy functions as a proxy to a Jini network with regard to service discovery, and serves client devices as VTC (Virtual Thin Client) servers for service delivery. The figure also shows the context-aware service discovery part, which provides context-aware discovery service to µJini proxy.

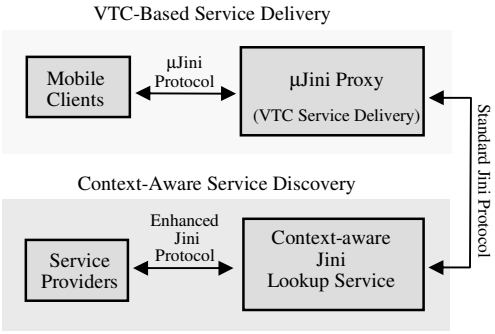

Fig. 1 - µJini Proxy System Components

To bring discovered services to resource-constrained mobile devices, our µJini proxy employs three different approaches. The first and most straightforward mode is to have a service code executed entirely on the client side (*Client executable mode*). In case of fat services or if client resources needed by the services are not available, we have to resort to two adaptation approaches. These approaches rely on the proxy to execute any kind of fat services, either J2ME MIDlets [1] or J2SE services, through the network-side thin-client system, which sends only screen updates to the client. The first adaptation enables MIDlets to be executed by a MIDlet emulator on the proxy side (*VTC MIDlet emulation mode*), while the second is for J2SE services (*VTC J2SE emulation mode*). The decision on which mode to use is made transparently to the users. Our µJini proxy system is capable of selecting the best adaptation, depending on the service context, while users sees largely the same behaviors whichever delivery mode is used to service their requests.

A. Context-Aware Service Discovery

Context awareness for service discovery means ability to provide the most appropriate service(s) to mobile users by exploiting any meaningful contextual information. The concept of *context attribute* underlies our context-aware service discovery subsystem. It is a special kind of attributes that are parts of service announcements. The context attribute is also called *dynamic attribute* in that its actual value is dynamically determined at the moment of lookups, as opposed to the current *static attribute* that has a fixed value set at the time of announcements. For instance, a *queue-length* context attribute may be used to figure out which printer is least loaded. Relevant context information is captured via an evaluation process. In this case, the attribute connects back to its printer service for the current queue length, which returns up-to-date load information. In contrast, a static queue-length attribute associated with a printer service must be updated, whenever the printer load changes, causing the waste of precious network resource and processing power.

Jini does not provide support for context-awareness. Based on Jini framework, our discovery subsystem extends its service matching mechanism to enable context-aware service discovery. We added context-awareness processing to Sun's Jini reference implementation, named *reggie*, version 1.2. The details of our context-aware discovery system are reported in [6].

B. Thin-Client Approach to Service Delivery

B.1 µJini Proxy System Architecture

As shown in Figure 2, the µJini proxy system architecture is made up of four main components: µJini Protocol, Resident Client, VTC, and µJini Proxy. The Resident Client resides on a client device and takes care of interfacing between the client and the proxy server as well as handling the VTC client/server communication through a VNC channel. The µJini Proxy performs context-aware service discovery on the network side for the client as well as allocating a thin-client server for the client to access the discovered service. The VTC is split into two pieces, VTC Client on the client side and VTC Server on the µJini Proxy side. Most of the processing is pushed to the proxy server side so the low-resource client device can operate more efficiently. The µJini protocol allows the client and proxy to talk, and a separate VNC channel is used for the VTC server connection.

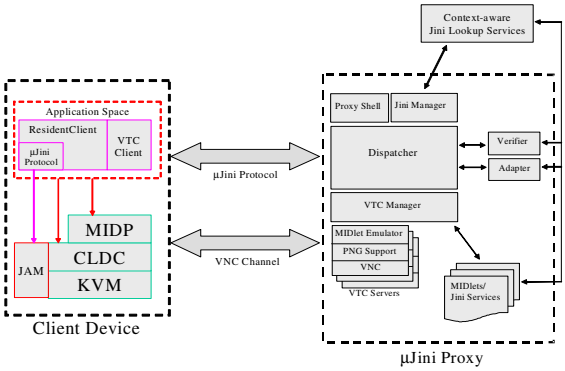

Fig. 2 - µJini Proxy System Architecture

B.1.1 μJini Protocol

The μJini proxy has been introduced to offload heavy Jini requirements from resource-constrained client devices like J2ME CLDC/MIDP smart-phones. The protocol allows the client device and proxy server to communicate using a common set of library APIs. It is lightweight enough to fit into the small devices, but supports all necessary messages for μJini proxy discovery, Jini/MIDlet service discovery, service invocation, lease management, and client device description passing. Any μJini protocol session begins with passing the client device profile to the μJini proxy, similar to W3C CC/PP device profile passing, and the protocol is translated into Jini protocol by the μJini Proxy.

B.1.2 Resident Client

The Resident Client is a MIDlet program (in the case of J2ME devices) running on the client devices. It provides a graphical user interface to the user, speaking the μJini protocol to find and request services from the μJini Proxy server. It also takes care of instantiating a VTC viewer and cleaning up resources after a service session is complete. In the case of downloaded services (i.e., the client executable mode), the Resident Client interacts with the proxy, downloads the MIDlet(s) from a specified HTTP server, and loads the code onto the client device using native operating system functions.

B.1.3 VTC (Virtual Thin Client)

As a scaled-down version of Java VNC client available from [5], the VTC Client on the client side implements the main encoding schemes the VNC Remote Frame Buffer (RFB) protocol defines, including Raw, Hextile, RRE, and CoRRE encoding schemes. The Java client has been modified to execute as an application under the J2ME libraries.

B.1.4 μJini Proxy

The μJini proxy functions as not only a proxy to a Jini network to translate the μJini protocol messages to Jini protocol, but also a thin-client adaptor that executes heavyweight services on behalf of the client device. As depicted in Figure 2, the Dispatcher is the central piece of the μJini Proxy, which mediates all the interactions among components. First of all, it communicates with a client device, speaking the μJini protocol. Upon requests from the client, it performs service discovery on behalf of the Resident Clients and may provide MIDlet or J2SE emulation service though a VNC channel. More specifically, the Dispatcher makes service discovery requests through the Jini Manager, and it arranges VTC service invocation by contacting the VTC manager, if necessary.

VTC J2SE emulation is straightforward. A VNC desktop is allocated to a Jini service. Then its display is sent to the client through a VNC channel and, in response, the user input is forwarded to the service via VNC event messages. For MIDlet emulation, we have developed a MIDlet emulator prototype built on Sun CLDC/MIDP reference implementations [1] [2]. The Resident Client on the phone intercepts keypad input events to send them to the μJini Proxy which, in turn, passes them to a corresponding MIDlet emulator instance. These events are injected to the point where the original emulator translated native operating system events to corresponding KVM events. Then the LCD screen part of the resultant emulator display is mirrored to the real phone screen through the VNC channel. This input event handling mechanism threads two separate pieces together (i.e. the user interface part on the client device and the MIDlet running on the network proxy side) so as to create an illusion that the MIDlet is running on the client device.

B.2 Performance Measurements

We present basic performance measurements for the two delivery options of our prototype implementation. As client devices, we used the Motorola i85s smart-phone that has built-in support for MIDP 1.0. Besides the phone's processing power, limitations are also seen in the speed of the network, of which current maximum network speed is about 40kbps and the actual throughput is often much less. The μJini Proxy runs on a Sun Ultra-60 workstation with 512M RAM. During testing of our prototype implementation, it was noticed that the relative slow speed and high latency of current cellular network is the determining factor in the whole system performance.

After the over-the-air download of corresponding JAR and JAD files is complete, the application still has to go through a verification and installation process on the device itself. After being run through a simple verifier, the classes in the JAR file are unpacked and expanded into a memory-image file, so the application starts quickly when accessed by the user. Table 1 shows the results of tests for starting a VTC session versus downloading the JAD and JAR files, installing the application, and local execution startup time. The experiments have used four J2ME MIDlets which are *iCoffee (monitoring coffee machines status in a workplace), Shopping (a shopping list/assistant), PhoneClient (control program allowing the phone to control a powerpoint presentation on a networked computer),* and *MapIt (a mapquest-based turn-by-turn directions program).* They are among the MIDlets developed by the Mobile Computing "Killer App" Competition at the University of Florida [7]. Also, we note that the download size (D/L size) column indicates the combined size of the JAR and JAD files.

MIDlet	D/L size (bytes)	D/L (s)	Install (s)	Local Startup (s)	Total Startup (s)	VTC Startup (s)
iCoffee	36073	58	168	6	232	12
Shopping	23413	46	91	5	142	11
Phone Client	24355	40	104	1	145	11
MapIt	35106	43	190	7	240	12

Table 1. Performance for Delivery Options

From Table 1, it could be noted that there is a huge difference in the time between a downloaded service and

VTC emulation service usage. A service application that is run locally must first be downloaded to the client device and then installed before it can be executed. The low bandwidth that are common with today's cellular networks, along with the long process of installing an application on a resource-constrained device, add up to a long time for the user to wait to run the service. A service application that is run as a thin-client, however, only needs to wait for the VTC server to start the service. The application can be started in seconds rather than minutes for a downloaded service. As repeated in Figure 3, the startup time takes an order of magnitude less time for VTC startup.

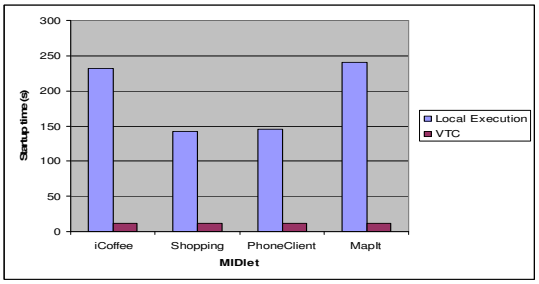

Fig. 3 - Service Startup Time Comparison

IV. Related Work

It is often the case where small, resource-poor devices can not afford Java RMI that is a prerequisite for Jini. Jini Surrogate Architecture addresses this problem by introducing a network side proxy that performs Jini operations on behalf of low-resource devices [4]. Its main components are surrogate host, surrogate, and interconnect protocol. The surrogate host providing an environment for executing the surrogate, i.e., a Java object that represents the small device, is comparable to our μJini proxy server, whereas the interconnect protocol is analogous to our μJini protocol. Our μJini proxy system does more than what the surrogate architecture does; it determines and executes the best service delivery method considering service implementations and client capability.

Equivalently to the VNC teleporting, Java user interfaces can be displayed on a remote host via Java Remote AWT [8]. Remote AWT objects allow a Java application to display its graphical outputs on a remote machine and to receive input events from the machine. In other words, a Java application itself and its graphical user interface are stitched together by remote AWT objects. However, there exist a certain set of mobile devices that even can not afford the graphical display part of the application, i.e., Java AWT or Swing classes. By requiring minimal graphical rendering classes on client devices, our VTC emulation approach can bring services to a broader set of device classes. The finest granularity of Java application partitioning is found in [9] in which expensive computation to execute a Java program is offloaded to a gateway machine (e.g., Java byte-code translation and code verification by a remote JIT compiler.)

Finally, it should be noted that there exist several scaled-down versions of VNC [10] [11]. They enable mobile devices to be thin clients of a traditional desktop, but not dealing with any of service discovery and delivery issues which the μJini proxy addresses.

V. Conclusion

The μJini proxy system architecture has been designed to overcome the current limits of mobile device, network, and service technologies. The proxy system provides a complete solution from a mobile service request to its delivery to client devices. To clients, it acts as a context-aware service discovery proxy, and it functions as virtual thin-client servers for service delivery to any class of client devices as well. We believe that the architecture will enhance user experiences with mobile services, which will accelerate the deployment of mobile services as well as wireless mobile devices.

Acknowledgement

This work was supported by a grant from Motorola, Inc., and Choonhwa Lee was also supported, in part, by grant No. R01-2005-000-10267-0 from Korea Science and Engineering Foundation in Ministry of Science & Technology.

References

[1] Sun Microsystems, "Mobile Information Device Profile," http://java.sun.com/products/midp

[2] Sun Microsystems, "Connected Limited Device Configuration," http://java.sun.com/products/cldc

[3] Sun Microsystems, "Jini Specifications v1.2," December 2001, www.sun.com/jini/specs

[4] Sun Microsystems, "Jini Surrogate Architecture," http://surrogate.jini.org

[5] AT&T Laboratories Cambridge, "Virtual Network Computing," http://www.uk.research.att.com/vnc

[6] C. Lee and A. Helal, "Context Attributes: An Approach to Enable Context Awareness for Service Discovery," In Proceedings of the Third IEEE/IPSJ Symposium on Applications and the Internet, pp. 22-30, January 2003

[7] S. F. Midkiff, "Mobile Computing 'Killer App' Competition," IEEE Pervasive Computing, vol. 1, no. 3, pp. 101-104, July-September 2002

[8] D. Malkhi and M. K. Reiter, "Secure Execution of Java Applets using a Remote Playground," IEEE Transactions on Software Engineering, vol. 26, no. 12, pp. 1197-1209, December 2000

[9] R. Teodorescu and R. Pandey, "Using JIT Compilation and Configurable Runtime Systems for Efficient Deployment of Java Programs on Ubiquitous Devices," In Proceedings of the 3rd International Conference on Ubiquitous Computing, pp. 76-95, September 2001

[10] B. Shizuke, M. Nakasu, and J. Tanaka, "VNC-Based Access to Remote Computers from Cellular Phones," In Proceedings of the IASTED International Conference on Communication Systems and Networks, pp. 74-79, September 2002

[11] J2ME VNC, http://j2mevnc.sourceforge.net