MAC-5701 - Tópicos em Ciência da Computação
Profa. Yoshiko Wakabayashi
Relatório de Estudo

Leonardo Marques Alves de Pinho
Orientador: Alfredo Goldman

25 de Junho de 2002

1 Introdução

Este documento consiste no relatório de estudo realizado no primeiro semestre de 2002, com base no plano de estudos montado para a disciplina *MAC 5701 - Tópicos em Ciência da Computação*.

O plano de estudos foi baseado na pesquisa e leitura de artigos científicos sobre os assuntos relativos a tese de mestrado a ser desenvolvida (ver seção 2). Podemos dividir os assuntos estudados em algumas áreas: localização de serviços, sistemas de localização física e mobilidade.

Na seção de localização de serviços foram estudados vários artigos sobre os protocolos existentes: *Jini*, *Service Location Protocol (SLP)*, *Microsoft Universal Plug and Play*, *Bluetooth Service Discovery Protocol* e *Salutation*. Em sistemas de localização estudamos soluções para localização física de unidades móveis (UMs), tais como GPS, Active Badge Systems, etc. Além disso, estumos alguns tópicos sobre computação móvel, como o suporte a mobilidade do IPv6 e o DHCP (*Dynamic Host Configuration Protocol*).

2 Área de Estudo - Resumo

O número de serviços disponíveis em uma rede (em particular a Internet) está crescendo drasticamente. Além de serviços básicos, como impressão, fax, envio de emails, etc, estão surgindo novos serviços a cada dia. Com isso, é cada vez mais importante que estes serviços estejam sempre disponíveis, e que possam ser acessados com o mínimo de esforço possível, ou seja, com a menor reconfiguração possível no *device* do cliente, que deseja utilizar o serviço.

Isso é importante quando consideramos que o número de usuários móveis está aumentando, através da popularização cada vez maior dos PDAs (*Personal Digital Assistent*). Assim, não queremos que em cada local (rede) que o usuário esteja, este precise reconfingurar manualmente sua unidade móvel para utilizar um serviço; o que pode implicar em uma simples troca de IP, ou até a instalação de *drivers*.

Um exemplo típico é um usuário que deseja imprimir um arquivo do seu PDA, que está conectado a um rede *wireless* de um prédio comercial, por exemplo. Não é desejável que ele procure o IP do *device* (no caso a impressora) mais próximo que provê o serviço, queremos que o usuário simplesmente informe o desejo de utilizar esse tipo de serviço (impressão), e que o processo de reconfiguração seja automático. Devido a isso, as tarefas do administrador da rede são reduzidas, não é mais necessário reconfigurar cada cliente novo da rede para acessar todos os serviços oferecidos. Essa característica é chamada "administração zero", é o ideal de uma rede LAN.

Para isso ser possível é necessário um protocolo de localização de serviços, que deve descobrir dinamicamente os serviços disponíveis em uma rede, e permitir o acesso transparente ao serviço mais próximo. Atualmente existem várias tecnologias que visam resolver esse problema.

O *Jini* é uma tecnologia desenvolvida pela Sun Microsystems, sobre linguagem Java e utiliza fortemente sua camada para objetos distribuídos: *RMI - Remote Method Invocation*; o *Service Location Protocol (SLP)* foi desenvolvido pelo *IETF SrvLoc working group* e foi projetado para rodar sobre TCP/IP e em grandes redes; o *Universal Plug and Play* é uma arquitetura proprietária desenvolvida pela Microsoft; o *Salutation*, por sua vez, está sendo desenvolvido por um consórcio aberto, chamado *Salutation Consortium*; assim como o *Bluetooth Service Discovery Protocol* que foi criado por um consórcio de empresas de telecomunicações.

Cada tecnologia tem características próprias, vantagens e desvantagens, entretando nenhuma delas consegue resolver o problema de localizar o servidor mais próximo ao cliente. Isso envolve o estudo de soluções para localização física de unidades móveis (UMs), tais como GPS, Active Badge Systems, etc.

Assim, o objetivo nessa disciplina é estudar mais profundamente essas tecnologias e analisar a que melhor resolve o problema descrito anteriormente como exemplo, de um usuário que deseja imprimir um arquivo do seu PDA, em uma rede *wireless*, e escolher a que pode ser melhor adaptada para otimizarmos o uso das impressoras, sempre utilizando a mais próxima do cliente, através de um sistema de localização.

3 Localização de serviços

Nos artigos estudados obtemos informações mais detalhadas sobre cada um dos protocolos existentes, comparações entre eles, e uma prosposta de um modelo para os sistemas de localização de serviços.

Depois de estudar os artigos pesquisados, elaboramos um texto sobre o modelo proposto para localização de serviços, e sobre os sistemas existentes no mercado. Para isso, utilizamos principalmente os seguintes artigos[1]:

[1]Lista completa dos artigos lidos: http://www.ime.usp.br/~lmap/mestrado.html

- **Service Location Protocol: Automatic Discovery of IP Network Services**
  Erik Guttman
  IEEE Internet Computing - aug/1999

- **Service Advertisement and Discovery: Enabling Universal Device Cooperation**
  Golden G. Richard III
  University of New Orleans, USA - sep/2000

- **Development of a Service Discovery Architecture for the Bluetooh Radio System**
  Christian Schwingenschlögl and Anton Heigl
  University of Munich, Germany - 2000

- **Alive and Well: Jini Technology Today**
  Jim Waldo
  IEEE Computer, jun/2000

- **Service Discovery protocols for Ad Hoc Networking**
  Michel Barbeau
  Carleton University, Canada - nov/2000

- **Mobile, Distributed, and Pervasive Computing**
  Michel Barbeau
  Carleton University, Canada - may/2001

- **A Comparison of Service Discovery protocols and implementation of the SLP**
  Christian Bettstetter and Christoph Renner
  University of Munich, Germany - 2000

- **Service Location Protocol for Mobile Users**
  Charles E. Perkins
  Sun Microsystems, Menlo Park, CA - 2000

- **Service Recommendation using SLP**
  Evan Hughes, David McCormack, M. Barbeau and Francis Bordeleau
  Carleton University, Canada - may/2001

O texto elaborado segue abaixo:

3.1 Service Discovery Model

Descreveremos um modelo genérico para protocolos de localização de serviços, que pretende abranger as características comuns dos *frameworks* existentes.

Em todos os cenários existem pelo menos dois elementos: o usuário do serviço e o provedor do serviço. Há um outro elemento(opcional) intermediário entre as requisições dos clientes e as respostas dos provedores, geralmente chamado de localizador de servíços, ou gerenciador de serviços. Neste modelo genérico utilizaremos a convenção do SLP(*Service Location Protocol*) e chamaremos os três elementos de: UA - *user agent*, SA - *service agent* e DA *directory agent*, respectivamente.

Conforme o número de particapantes, com ou sem DA, temos que considerar duas arquiteturas ou modelos de interação, que chamaremos de *DAwith* e *DAless*. Para cada caso, teremos diferentes fases do protocolo, que descrevemos a seguir.

3.1.1 DAwith

Como dissemos anteriormente, a principal função do DA é atender as requisições do cliente, e encontrar os serviços disponíveis de acordo com o tipo requisitado, ou através das caracteríticas do serviço, descritas pelo cliente(modo opcional).

Fase de Inicialização:

- **Localização do DA**

  Inicialmente é necessário que o cliente, de alguma forma, tenha a informação(o IP) de como acessar o DA da rede em que ele está conectado. Para isso existem quatro possibilidades:

  - **static:** o ip do DA é atribuído manualmente.
  - **passive:** o DA realiza *multi/broadcast* periódicos com a sua localização.
  - **active:** os UAs e SAs realizam *multi/broadcast*, requisitando a localização do DA.
  - **dhcp:** pode-se adaptar o DHCP, para que este forneça o DA responsável por aquela rede.

  Uma observação importante, o modo **passive** pode aumentar bastante o tráfego na rede, dependendo do período escolhido, assim como o modo **active**, de acordo com o número de elementos no sistema. Assim os três últimos modos são recomendados para redes locais(LAN), e o primeiro para redes globais(WAN), como a *Internet*.

- **Registro dos SAs**

  Cada agente de serviço, depois de localizar o DA, deve se registrar ao mesmo, com as informações necessárias para seu acesso, e em alguns casos com a lista de atributos que descrevem o serviço. Geralmente existe um mecanismo para atualização das propriedades do serviço do SA, depois deste ter sido registrado.

Os registros podem ser persistentes, ou seja o DA armazena fisicamente as informações dos SAs disponíveis, assim caso ocorra um *reboot* no mesmo, os servidores não precisarão registrar-se novamente, entretanto pode-se adicionar um *overhead* maior ao processo de registro.

Fase de Consulta:
O objetivo dessa fase é informar ao DA o interesse do UA por um tipo de serviço, enviando uma mensagem de consulta com o tipo. Alguns protocolos também suportam a requisição por um serviço com certas características, ou seja, com alguns atributos especificados pelo UA, por exemplo: um serviço de impressão colorida. Neste caso é necessário uma fase anterior de requisição da lista de atributos do serviço, geralmente chamada de *service template*. Esta fase continua, até que é terminada pelo próprio UA, atráves de *timeout*, ou número de requisições enviadas sem resposta, etc.

Fase de Seleção:
Nesta fase desejamos selecionar o SA correspondente ao pedido do cliente(UA). Vamos dividi-la em duas subfaces:

- **Subfase de comparação**

  Nós desejamos encontrar o subconjunto de SAs da lista de SAs disponíveis(registrados no DA), que atendem a requisição do cliente. Antes de adicionar o SA ao subconjunto, seria conveniente testar se este ainda está disponível.

- **Subfase de decisão**

  Depois de encontrado o subconjunto, temos que decidir como será a resposta dada ao UA, se iremos simplesmente retorna a lista de todos os SAs que atendem o seu pedido, se iremos escolher aleatóriamente um, ou o mais apropriado, se temos algum críterio para a decisão(como o serviço mais próximo) e elegemos o "melhor" SA. Este último modo é chamado de *Service Selection Facilitation*, poucos protocolos suportam essa funcionalidade.

Fase de Configuração:
Nesta última fase é modificada a configuração do UA, para que este possa acessar o serviço selecionado. A configuração pode ser simples, como abrir um *socket* para o serviço, ou mais complexa, como a instalação de *drivers*. Alguns protocolos só suportam este primeiro caso.

3.1.2 DAless

Nessa arquitetura temos apenas dois participantes: UAs e SAs e o processo é mais simples. Como não existe DA, não temos a fase de inicialização.

Fase de Consulta:
Ao invés do UA se comunicar com o DA via *unicast*, ele se comunica com os SAs usando UDP *multicast*, enviando a sua requisição. Os agentes de serviços que estiverem escutando,

irão processar a requisição.

Fase de Seleção:
Agora temos apenas a subfase de comparação, que será executada localmente por cada agente de serviço. Caso ele atenda o pedido do cliente, o SA se comunica com o UA via *unicast*, dando-lhe as informações de como acessa-lo.

Como não temos a subfase de seleção, esta arquitetura não suporta a funcionalidade de *Service Selection Facilitation*.

Fase de Configuração:
Idêntica à fase do *DAwith*.

3.2 Service Discovery Protocols

3.2.1 Service Location Protocol (SLP)

O *Service Location Protocol* está sendo desenvolvido pelo *IETF SrvLoc working group*. Ele foi designado para rodar sobre TCP/IP e em grandes redes.

Ele suporta os dois modelos de interação descritos na seção anterior, e a comunicação entre os componentes do sistema é toda feita através de pares de mensagens: Service Request(SrvRqst) - Service Reply(SrvRply), Service Registratiton(SrvReg) - Service Acklonegement(SrvAck), Attribute Request(AtrReq) - AttributeReply(AtrRply).

Seus elementos são: *Directory Agent* (DA), equivalente ao LS do Jini; *Service Agente* (SA), o provedor do serviço e *User Agent* (UA), o usuário do serviço.

Existe um modelo para descrição dos serviços chamado *Service Template*, que contém todas as informações do mesmo. Um exemplo de uma impressora é descrito abaixo:

service:printer://192.168.0.183:1020/queue1
scopes = tum, bmw, administrator
printer-name = papel
printer-model = HP LJ4050
printer-location = Room 0409
color-support = false
pages-per-minute = 9

O *Service Template* pode ser conhecido pelo UA, utilizando a mensagem *AtrReq*, e assim este pode requisitar um tipo de serviço com determinadas características(valores dos atributos do *template*), no diagrama identificado como *Predicated*.

O *service URL* também chamado de SAP(*Service Acess Point*) contém as informações necessárias para o UA acessar o serviço: o IP do serviço, sua porta e o caminho(*path*), bastando agora que o cliente conecte via *socket* no servidor escolhido. Não há menção sobre o caso do serviço ser mais complexo de ser acessado, por exemplo: necessitar de um *driver*.

O protocolo SLP não suporta *Service Selection Facitation*. Ele apenas ser adaptado utilizando uma função(limitada) de *Ranking* em alguns dos atributos do tipo de serviço.

Apenas na arquitetura *DAwith*), é claro.

3.2.2 Jini

A tecnologia *Jini*, desenvolvida pela Sun Microsystems, é uma "extensão" da linguagem Java, que tem como objetivo criar uma rede, em que seus elementos possam prestar serviços a outros elementos da rede, chamada de *Jini community*. Cada participante da comunidade deve possuir uma JVM(*Java Virtual Machine*).

A arquitetura do sistema *Jini* consiste de *Clients*, *Lookup Services(LS)* e *Service Providers(SP)*, que são equivalentes aos elementos UA, DA e SA do SLP. Ele também suporta os dois modos de interação. A primeira sem o LS é chamada de *peer lookup*, e a segunda contém o LS, atuando como central de informações ou repositório de objetos.

O protocolo de funcionamento do *Jini* é dividido em micro-protocolos. O primeiro deles, chamado *Protocol Join*, é utilizado quando um SP deseja se registrar a um LS. O *Service Provider* chama o método *register* do LS, passando como parâmetro o objeto de serviço, que será gravado em uma tabela denominada *Lookup Table*. O objeto de serviço consiste de uma interface Java, com a assinatura de métodos e atributos descritivos. Se o registro tem sucesso, o *Lookup Service* retorna um objeto(remoto) de registro do serviço, que pode ser usado para a manutenção do mesmo.

Outro micro-protocolo, chamado de *Protocol Lookup*, é utilizado pelo cliente para requisitar um serviço ao LS, de acordo com o tipo(na verdade um tipo de interface Java) ou através de valores de atributos do tipo de serviço. Quando o SP é escolhido, o objeto de serviço correspondente é enviado para o Cliente, e o objeto passa a atuar como um *proxy* entre o *Client* e o *Service Provider*, para acesso ao serviço. O protocolo usado na comunicação entre SP e o objeto de serviço é o Java RMI, solucionando o problema de instalação de drivers, e do acesso de servíços mais complexos pelo cliente.

Um problemas do *Jini* além de requirimento da JVM nos elementos da rede, é a falta de padronização das interfaces, que seria necessário para localizar provedores de servíços do mesmo tipo.

3.2.3 Salutation

Essa arquitetura está sendo desenvolvido por um consórcio aberto, chamado *Salutation Consortium*. A principal diferença entre seu sistemas e os outros, é que ele contém um elemento principal, o Salutation Manager(SLM), também chamado de *service broker*, que é responsavél por todo o processo de operação do sistema.

Os servidores utilizam o SLM para registrar os seus serviços, e os clientes para requisitar um determinado serviço. Quando o serviço desejado é localizado, o acesso ao mesmo é feito através do SLM.

Além do SLM, o sistema contém um outro elemento, o *Transport Manager*, em uma camada abaixo do SLM. Ele é responsavél por informar os clientes e servidores a localização do SLM, e pela abstração da camada de tranporte. Dessa forma, o *Salutation* se torna independente de protoloco de transporte, mas no momento só existe implementação sob IrDA e TCP/IP.

3.2.4 Universal Plug and Play (UPnP)

Nesta arquitetura, desenvolvida pela Microsoft, não existe o elemento DA, portanto ele suporta apenas um modo de iteração, baseado em requisições via UDP *multi/broadcast* ou http. Seu protocolo de localização de serviços é chamado *Simple Service Discovery Protocol*(SSDP).

O cliente envia uma mensagem requisitando um tipo de servíço(a busca por atributos não é suportada), e os provedores de serviços que estiverem escutando e suportarem o serviço requisitado, respondem via *unicast* ao cliente.

3.2.5 Bluetooth Service Discovery Protocol (SDP)

O SDP suporta consultas por um determinado tipo de serviço e também de acordo com os atributos do serviço. Mas depois que o serviço é encontrado, ele não contém a funcionalidade de seleção e acesso ao serviço, que deve ser feito através de um outro mecanismo.

Por esse motivo, ele pode ser usado em conjunto com outros protocolos de localização de serviço, como o SLP ou Salutation, sendo o SDP apenas responsável por localizar os servíços disponíveis, de acordo com a requisição.

3.3 Tabela de Comparações

Feature	**SLP**	**Jini**	**Salutation**	**UPnP**
Developer	IETF	Sun Microsystems	Salutation Consortium	Microsoft
License	open source	open license	open source	open(only for members)
Version	2	1.0	2.1	0.91
Network Transport	TCP/IP	independent	independent	TCP/IP
Programming Language	independent	Java	independent	idependent
OS and plataform	dependent	independent	dependent	dependent
Code mobility	no	yes(Java RMI)	no	no
Srv attributes searchable	yes	yes	yes	no
Support for Directory Agent	yes	yes	yes	no
Central cache repository	yes(optional)	optional	yes(optional)	no
Operation w/o directory	yes	Lookup Table required	yes	-
Leasing concept	yes	yes	no	yes
Security	IP dependent	Java based	authentication	IP dependent

3.4 Conclusões e trabalhos futuros

Devido a existência desses vários protocolos, surgi um problema: acabam-se criando "ilhas de comunicação" entre as redes, em que cada protocolo está sendo utilizado. Para resolver este problema, estão começando a surgir *bridges*, que poderiam facilitar a integração entre alguns dos framework.

Técnicas de aceleração da localização do serviço também estão sendo estudadas. Geralmente são baseados em *cache*, através da capacidade de reuso e experiência, isto é, se um serviço é muito utilizado em uma mesma subrede(sempre visitada), é interessante guardar a localização deste, para que numa próxima interação ele não precise ser localizado novamente.

4 Sistemas de Localização

Foram estudados textos sobre sistemas de localização física de unidades móveis, como eles surgiram, suas características, como podem ser utilizados, etc. Dentre os sistemas estudados destam-se o Active Bagde e o GPS.

Os principais artigos analisados foram:

- **Location Systems for Ubiquitous Computing**
  Jeffrey Hightower and Gaetano Borriello
  Computer Magazine, aug/2001

- **Implementing a Sentient Computing System**
  Mike Addlesee, Rupert Curwen, Steve Hodges, Joe Newman, Pete Steggles, Andy Ward and Andy Hopper
  Computer Magazine, aug/2001

- **Using and Determining Location in a Context-Sensitive Tour Guide**
  Nigel Davies, Keith Cheverst, Keith Mitchell and Alon Efrat
  Computer Magazine, aug/2001

- **Expanding the Horizons of Location-Aware Computing**
  Roy Want and Bill Schilit
  Computer Magazine, aug/2001

5 Mobilidade

Neste tópico estudamos alguns artigos sobre protocolos de transporte com suporte a mobilidade, como o IPv6 e também protocolos de configuração dinâmica, como o DHCP. Dentre os artigos estudados, destacam-se:

- **Is IPv6 Finally Gaining Ground?**
  George Lawton
  Computer Magazine, aug/2001

- **Automated Configuration of TCP/IP with DHCP**
  Ralph Droms
  Computer Magazine, jul/1999

- **An Experimental Implementation of Mobile IPv6 on a Personal Digital Assistant**
  Zhao-shu Zeng and M. Barbeau
  Carleton University, Canada - jun/2001

- **Mobility Support in IPv6**
  Charles E. Perkins and Daniel B. Johnson
  Carnegie Mellon University, USA - nov/1996

6 Análise da Pesquisa

Com base nos artigos estudados e no texto escrito, podemos realizar uma análise crítica e comparativa entre os sistemas existentes, implementando pequenos programas (cliente/servidor) utilizando cada um dos *frameworks*.

O sistema escolhido para a implementação inicial do protótipo foi o Jini, da Sun Microsystems. Assim, foi realizado um seminário[2] registrado na CPG no dia 14 de junho, para expor essa tecnologia e o problema a ser estudado. O texto da apresentação segue abaixo:

6.1 A tecnologia Jini

6.1.1 O que é?

Visando alcançar esses objetivos, a Sun Microsystems desenvolveu a tecnologia Jini. Uma arquitetura para localização de serviços utilizando a linguagem Java. Com isso, os usuários de um rede conseguem descobrir facilmente os serviços existentes nesta.

Para isso ser possível, o Jini armazena uma lista dos serviços da rede, e por isso, inicialmente ele pode ser comparado com o DNS (*Domain Name Server*), mas veremos na próxima seção que ele tem muitas outras vantagens, não sendo apenas um servidor de nomes de serviços.

Além de localizar o serviço, o Jini também fornece uma forma de acesso ao recurso requerido, sendo um *middleware* entre o cliente, que deseja utilizar um serviço, e o servidor, que disponibiliza o mesmo.

6.1.2 Vantagens

Para busca do serviço, a arquitutera fornece a consulta não somente pelo nome (ou tipo) do serviço como no caso do DNS, mas também através dos atributos do serviços. Por exemplo, um usuário que deseja encontrar uma impressora na rede, mas necessita de impressora colorida, ou de uma impressora laser. O Jini suporta este tipo de busca.

Se o serviço não é encontrado, o Jini oferece a possibilidade do cliente, interessado no recurso, ser notificado automaticamente quando este estiver disponível.

Outra importante vantagem dessa tecnologia é a disponibilização de uma forma transparente de acesso ao serviço, sem a necessidade de instalação de *drivers*, de conhecimento do IP do servidor, e da reconfiguração do cliente.

Devido a isso, as tarefas do administrador da rede são reduzidas, não é mais necessário reconfigurar cada cliente novo da rede para acessar todos os serviços oferecidos. Essa característica é chamada “administração zero”.

O sistema ainda tolera falhas dos servidor, se reajustando automaticamente a esse novo cenário, e desabilitanto o serviço oferecido por ele, evitando que o cliente tente acessar um servidor indisponível no momento.

[2] Veja http://www.ime.usp.br/~rcastro/seminars/

6.2 Arquitetura

O Jini foi desenvolvido sobre a linguagem Java, utilizando fortemente sua tecnologia para objetos distribuídos: *RMI - Remote Method Invocation*. Além disso ele depende da camada de rede para se comunicar com os elementos de seu sistema, e obviamente ele depende do sistema operacional.

Portanto podemos traduzir sua arquitetura nas seguintes camadas:

- **Aplicações Cliente e Servidor**
- **Jini Network Tecnology**
- **Java - RMI**
- **Sistema Operacional**
- **Camada de transporte de rede**

6.3 Elementos do sistema

No seu cenário existem três elementos:

- **Lookup Service (LS)**
- **Service Provider (SP)**
- **Client (C)**

Os dois últimos elementos são os usuários finais do sistema. O *Service Provider* é o elemento que disponibiliza o serviço, o *Cliente* é o elemento que deseja utilizar os serviços.

O *Lookup Service* é um elemento intermediário entre as requisições dos clientes e as respostas dos provedores, geralmente chamado de localizador de serviços, ou gerenciador de serviços. Ele é o elemento principal do sistema, responsável pelas funcionalidades principais do Jini: mantém a lista dos serviços registrados, gerencia acesso a um serviço, notifica clientes sobre novos serviços, etc.

6.4 Funcionamento

O seu funcionamente é descrito através de micro-protocolos, que serão explicados abaixo:

6.4.1 Localização do LS

Para um cliente acessar um serviço, ou um servidor publicar um serviço, ele precisa inicialmente ter acesso de alguma forma ao LS da rede, em que ele está conectado. Para isso existem três possibilidades:

- **Multicast Request:** os SPs e clientes realizam *multi/broadcast*, requisitando a localização do LS.

- **Multicast Announcement:** o LS realiza *multi/broadcast* periódicos com a sua localização.

- **Unicast Discovery:** a localização do LS é atribuída manualmente.

Depois da execução de qualquer um dos modos acima, o elemento que deseja localizar o LS, recebe um objeto remoto que referencia o LS, podendo ser acessado através do RMI.

Uma observação importante: o modo **Multicast Announcement** pode aumentar bastante o tráfego na rede, dependendo do período escolhido. Assim ele é recomendado apenas para redes locais (LAN), e os outros dois modos para redes globais (WAN), como a *Internet*.

6.4.2 Publicação do Serviço

Cada agente de serviço, depois de localizar o LS, deve se registrar ao mesmo, com as informações necessárias para o seu acesso, e com a lista de atributos que descrevem o serviço.

Para isso, é utilizado um objeto da biblibloteca do Jini: *ServiceItem*, que é instanciado recebendo três argumentos: o *id* do serviço (opcional, pode ser *null*), utilizado somente quando o serviço já foi registrado uma vez; *service proxy*, que é a implementação do serviço, pode ser um objeto serializavél, para ser possível ser enviado para o cliente, ou um objeto remoto, para o cliente acessa-lo via RMI; e o último argumento é um vetor de atributos que descrevem o serviço, que está sendo registrado.

Além disso, é preciso definir a política de *leasing*, pois ao registrar o serviço, este será valido somente por um período de tempo especificado durante o registro, depois disso o *leasing* precisa ser renovado para o servidor continuar disponível. Portanto, para servidor permanentes é necessário inicializar uma *Thread*, que mantenha sempre o leasing válido, renovando-o periodicamente. Através disso, que é possível ao Jini identificar os servidores inacessíveis, retirando-os da lista dos servidores disponíveis.

Depois de realizado o registro, o servidor recebe um id identificando o seu registro, e podendo ser mais tarde utilizado em um novo registro, caso o servidor tenha ficado não disponível momentaneamente, por exemplo.

6.4.3 Consulta

O objetivo dessa fase é informar ao LS o interesse do cliente por um tipo de serviço, enviando uma requisição de busca por tipo, ou por um serviço com certas características, ou seja, com alguns atributos especificados pelo cliente, por exemplo: um serviço de impressão colorida.

Assim é necessário instanciar um objeto que representará a consulta, no caso do Jini nomeado *ServiceTemplate*, que conterá a lista de atributos do serviço procurado.

Este objeto é instanciado recebendo três argumentos: *id*, o identificador do serviço, é opcional, pode ser *null*; o vetor de atributos requeridos, e o último argumento, que representa o tipo do serviço, e que é na verdade o nome de uma *interface* Java, ou seja, o LS irá buscar o serviço através do tipo, utilizando o "comando" *instanceof* do Java.

Quando o serviço é encontrado, é enviado ao cliente o *Service Proxy* do servidor, e este será executado pelo o cliente, para ter acesso ao recurso desejado.

6.5 Exemplos de código

Abaixo, temos um exemplo de código de um serviço simples, que devolve apenas ao cliente um string: “Hello World”.

6.5.1 Servidor

```
// O programa publica um proxy que retorna
// uma string quando executado pelo cliente

package corejini.chapter5;

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import java.util.Hashtable;
import java.io.IOException;
import java.io.Serializable;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

// Este é o objeto proxy que será feito download
// pelos clientes, por isso é serializável. Além disso
// ele implementa a interface deste serviço: HelloWorldServiceInterface
class HelloWorldServiceProxy implements Serializable,
    HelloWorldServiceInterface {
    public HelloWorldServiceProxy() {
    }
    public String getMessage() {
        return "Hello, world!";
    }
}

// HelloWorldService é a classe que irá publicar o serviço (service item) no LS.
public class HelloWorldService implements Runnable {
    // define o tempo do leasing, tempo que o serviço será válido
    protected final int LEASE_TIME = 10 * 60 * 1000;
    protected Hashtable registrations = new Hashtable();
    protected ServiceItem item;
    protected LookupDiscovery disco;

    // Inner class que esperará evento de localização do LS
```

```
class Listener implements DiscoveryListener {
    // Método chamado quando um novo LS é encontrado
    public void discovered(DiscoveryEvent ev) {
        System.out.println("Localizado um novo LS!");
        ServiceRegistrar[] newregs = ev.getRegistrars();
        for (int i=0 ; i<newregs.length ; i++) {
            if (!registrations.containsKey(newregs[i])) {
                registerWithLookup(newregs[i]);
            }
        }
    }

    // Chamado quando um LS é terminado.
    public void discarded(DiscoveryEvent ev) {
        ServiceRegistrar[] deadregs = ev.getRegistrars();
        for (int i=0 ; i<deadregs.length ; i++) {
            registrations.remove(deadregs[i]);
        }
    }
}

public HelloWorldService() throws IOException {
    item = new ServiceItem(null, createProxy(), null);

    // Seta um security manager
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }

    // Argumento recebe o grupo que pertence o LS procurdo,
    // string vazia é o grupo default
    disco = new LookupDiscovery(new String[] { "" });

    // Instala um listener
    disco.addDiscoveryListener(new Listener());
}

protected HelloWorldServiceInterface createProxy() {
    return new HelloWorldServiceProxy();
}

// Depois de localizado o LS, esse método é chamado para
// realizar o registro do serviço
protected synchronized void registerWithLookup(ServiceRegistrar registrar) {
    ServiceRegistration registration = null;
```

```
        try {
            registration = registrar.register(item, LEASE_TIME);
        } catch (RemoteException ex) {
            System.out.println("Couldn't register: " + ex.getMessage());
            return;
        }

        // Se é o primeiro registro, guardamos o serviceID para
// utilizarmos no futuro, no caso de reinicialização do
        // servidor
        if (item.serviceID == null) {
            item.serviceID = registration.getServiceID();
            System.out.println("Set serviceID to " + item.serviceID);
        }

        registrations.put(registrar, registration);
    }

    public void run() {
        while (true) {
            try {
                Thread.sleep(1000000);
            } catch (InterruptedException ex) {
            }
        }
    }

    // Cria um novo HelloWorldService and inicia sua thread
    public static void main(String args[]) {
        try {
            HelloWorldService hws = new HelloWorldService();
            new Thread(hws).start();
        } catch (IOException ex) {
            System.out.println("Couldn't create service: " +
                               ex.getMessage());
        }
    }
}
```

6.5.2 Cliente

```
// Um simples Cliente que deseja acessar o HelloWorldService

package corejini.chapter5;
```

```
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import java.util.Vector;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

public class HelloWorldClient implements Runnable {
    protected ServiceTemplate template;
    protected LookupDiscovery disco;

    // Inner class que esperará evento de localização do LS
    class Listener implements DiscoveryListener {
        public void discovered(DiscoveryEvent ev) {
            ServiceRegistrar[] newregs = ev.getRegistrars();
            for (int i=0 ; i<newregs.length ; i++) {
                lookForService(newregs[i]);
            }
        }
        public void discarded(DiscoveryEvent ev) {
        }
    }

    public HelloWorldClient() throws IOException {
        Class[] types = { HelloWorldServiceInterface.class };

        template = new ServiceTemplate(null, types, null);

        // Seta um security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        // Procura pelo LS do grupo default
        disco = new LookupDiscovery(new String[] { "" });

        // Instala um listener
        disco.addDiscoveryListener(new Listener());
    }

    // Depois de localizar LS, esse método procura pelos
```

```
    // proxies que implementam a HelloWorldServiceInterface
    protected Object lookForService(ServiceRegistrar lusvc) {
        Object o = null;

        try {
            o = lusvc.lookup(template);
        } catch (RemoteException ex) {
            System.err.println("Erro durante lookup: " + ex.getMessage());
            return null;
        }

        if (o == null) {
            System.err.println("Serviço não foi encontrado.");
            return null;
        }

        System.out.println("Achei um servidor.");
        System.out.println("A mensagem é: " +
                           ((HelloWorldServiceInterface) o).getMessage());
        return o;
    }

    public void run() {
        while (true) {
            try {
                Thread.sleep(1000000);
            } catch (InterruptedException ex) {
            }
        }
    }

    // Cria um HelloWorldClient e inicia sua thread
    public static void main(String args[]) {
        try {
            HelloWorldClient hwc = new HelloWorldClient();
            new Thread(hwc).start();
        } catch (IOException ex) {
            System.out.println("Couldn't create client: " +
                               ex.getMessage());
        }
    }
}
```

6.6 Problemas

Um dos principais problemas do Jini, é a obrigatoriedade do cliente conhecer a *interface* do serviço em que ele está interessado, os seus métodos, e o seu nome, para ser possível o acesso e a busca por tipo, respectivamente.

Para amenizar esse problema, a Sun está propondo uma padronização de *interfaces* para os serviços mais conhecidos, como de impressão, envio de emails, etc. E a partir dessa *interface*, cada fabricante implementará o seu serviço.

Outro problema é a inexistência de um método para seleção serviços, pois depois de encontrado o conjunto de servidores, temos que decidir como será a resposta dada ao cliente, se iremos simplesmente retornar a lista de todos os SPs, que atendem o seu pedido, ou se iremos escolher aleatóriamente um, ou o mais apropriado (o ideal), se temos algum críterio para a decisão (como o serviço mais próximo, ou o com menos demanda) e assim elegemos o "melhor" SP.

7 Conclusões

Durante o primeiro semestre do ano de 2002 foi realizado um estudo intensivo sobre os sistemas de localização, a fim de resolver o problema da impressora que nos estudamos. Com isso, obtivemos uma base consistente para desenvolver o nosso projeto de tese.

Ademais, com a base já formada, foi realizado um seminário em que o aluno pode expor seu projeto e uma pequena amostra da especificação do mesmo.

Por isso julgamos que a experiência foi muito produtiva, ajudando a continuação do projeto de mestrado e o seu término com sucesso.