FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ELTER GIROTTO DA CRUZ

IMPLEMENTAÇÃO DE UMA BIBLIOTECA DE COMUNICAÇÃO PARA AMBIENTES VIRTUAIS DISTRIBUÍDOS

MARÍLIA
2006

ELTER GIROTTO DA CRUZ

IMPLEMENTAÇÃO DE UMA BIBLIOTECA DE COMUNICAÇÃO PARA AMBIENTES VIRTUAIS DISTRIBUÍDOS

Monografia apresentada ao Curso de Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", Mantenedora do Centro Universitário Eurípides de Marília - UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. Ildeberto Aparecido Rodello

MARÍLIA
2006

ELTER GIROTTO DA CRUZ

IMPLEMENTAÇÃO DE UMA BIBLIOTECA DE COMUNICAÇÃO PARA AMBIENTES VIRTUAIS DISTRIBUÍDOS

Banca examinadora do Trabalho de Conclusão de Curso apresentado ao Programa de Graduação da UNIVEM/F.E.E.S.R., para obtenção do título de Bacharel em Ciência da Computação. Área de Concentração: Realidade Virtual.

Resultado: 6,0

ORIENTADOR: Prof. Dr. Ildeberto Aparecido Rodello

1º EXAMINADOR: Prof. Dr. José Remo Ferreira Brega

2º EXAMINADOR: Prof. Dr. Antônio Carlos Sementille

Marília, 07 de dezembro de 2006.

A mente que se abre a uma nova idéia
jamais voltará ao seu tamanho original.
Albert Einstein

Cruz, Elter Girotto da. **Implementação de uma biblioteca de comunicação para Ambientes Virtuais Distribuídos.** 2006. 46 f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

RESUMO

Com a grande utilização da Realidade Virtual e conseqüentemente dos Ambientes Virtuais Distribuídos, tanto em redes locais como na mundial, no caso da Internet, surgiu a necessidade de um suporte de comunicação para Sistemas Distribuídos de Realidade Virtual. Essa comunicação permite a usuários que estejam geograficamente dispersos se comunicarem, acessando um mesmo mundo virtual, utilizando os mesmos objetos pertencentes a este ambiente. Este trabalho consiste em se criar uma biblioteca que possa fornecer um suporte à comunicação de usuários em Ambientes Virtuais Distribuídos utilizando Java RMI. Além disso, fornece aos programadores a possibilidade de criação de seus próprios métodos de conexão e comunicação para esses ambientes.

Palavras-chave: Realidade Virtual, Ambientes Virtuais Distribuídos, Sistemas Distribuídos, Java RMI, biblioteca.

Cruz, Elter Girotto da. **Implementação de uma biblioteca de comunicação para Ambientes Virtuais Distribuídos.** 2006. 46 f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

ABSTRACT

With the growing of Virtual Reality and consequently of Distributed Virtual Environments, such local ones as at the worldwide one, like Internet, the necessity of communication support to Distributed Virtual Reality Systems appeared. This communication allows geographically dispersed users to communicate among them, accessing the same virtual world, using the same belong objects to this environment. This work consists of creating a library for providing communication to users in Distributed Virtual Environments using Java RMI. Besides, it supplies the programmers the possibility to create their own methods of connection and communication to these environments.

Keywords: Virtual Reality, Distributed Virtual Environments, Distributed Systems, Java RMI, library.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATUARS E SIGLAS

API:	*Application Programming Interface*
AV:	Ambiente Virtual
AVD:	Ambiente Virtual Distribuído
CAVE:	*Cave Automatic Virtual Environment*
CORBA:	*Common Object Request Broker Architecture*
DARPA:	*Defense Advanced Research Projects Agency*
DIS:	*Distributed Interactive Simulation*
DoD:	*Department of Defense*
DWTP:	*Distributed Worlds Transfer and communication Protocol*
HMD:	*Head Mounted Display*
HTML:	*Hyper Text Markup Language*
HTTP:	*Hyper Text Transfer Protocol*
IEEE:	*Institute of Electrical and Electronic Engineers*
IP:	*Internet Protocol*
MU3D:	*Multi-user 3D Protocol*
PDU:	*Protocol Data Unit*
RMI:	*Remote Method Invocation*
RV:	Realidade Virtual
TCP:	*Trasmission Control Protocol*
UDP:	*User Datagram Protocol*
URL:	*Uniform Resource Locator*
VRML:	*Virtual Reality Modeling Language*
VRTP:	*Virtual Reality Transfer Protocol*

W2W: *World2World*

WTK: *WorldToolKit*

SUMÁRIO

INTRODUÇÃO

Com a tecnologia cada vez mais acessível, a Realidade Virtual (RV) se expandiu para diversas áreas do conhecimento, como jogos e entretenimento, comunicação à distância, médica e engenharia, entre outras.

A RV é o uso do computador e de interfaces humano-computador para criar o efeito de mundos tridimensionais com objetos interativos (Ambientes Virtuais – AV), apoiada em três fundamentos: interação, imersão e navegação (Cardoso, 2006; Kirner & Pinho, 1996).

Ainda segundo Cardoso(2006); Kirner & Pinho(1996) é necessário um forte senso de presença neste espaço tridimensional (imersão), permitindo experimentar (navegação) e interagir (interação) de forma a propiciar sensações de prazer e de conhecimento.

Além disso, com o desenvolvimento dos AVDs, há a possibilidade de se acessar um AV localizado em outra máquina. Assim pessoas em lugares diferentes podem trocar informações e acessarem um mesmo AV. É isso que se refere o Ambiente Virtual Distribuído (AVD), fazendo com que usuários dispersos possam acessar o mesmo AV simultaneamente, em tempo real.

Para acessar as informações contidas em um AVD, é necessário, além do meio de comunicação, sendo ele uma rede local ou a própria Internet, uma forma para haver essa comunicação, ou seja, um suporte para facilitar a troca de mensagens entre os participantes. Esse suporte visa facilitar a interação nos AVDs, provendo mecanismos necessários para a comunicação entre os participantes e o mundo virtual.

Dentro desse contexto, esse trabalho consiste na criação de uma biblioteca de comunicação para AVDs utilizando-se da API Java RMI (*Remote Method Invocation*) (Sun, 2006). Como principais características o Java RMI traz a portabilidade e a possibilidade de programação para Sistemas Distribuídos. Visto a dificuldade para se obter a comunicação em

AVDs, essa biblioteca tem como objetivo fornecer um suporte para essa comunicação, facilitando a utilização por implementadores.

Assim, a implementação desta biblioteca visa tornar mais fácil o trabalho do programador que pretende criar uma aplicação que utilize uma conexão entre cliente e servidor na implementação de um AVD.

Organização do Trabalho

Além desse Capítulo de Introdução, esta monografia também aborda os seguintes tópicos: no Capítulo 1 serão apresentados os AVDs, seus componentes e suas principais características. No Capítulo 2 serão abordados o RMI, seus aspectos e criação de programas. No Capítulo 3, serão abordados os aspectos de implementação da biblioteca e suas primitivas. Por fim, serão apresentadas a conclusão e sugestões para trabalhos futuros.

1. AMBIENTES VIRTUAIS DISTRIBUÍDOS

1.1 Introdução

Ambientes Virtuais Distribuídos (AVDs) são caracterizados como um Ambiente Virtual (AV) interativo, onde os usuários remotamente localizados podem compartilhar recursos computacionais em tempo real usando um suporte de rede para troca de informações, aumentando assim o desempenho coletivo (Benford, 1994; Zyda & Singhal, 1999).

A seguir serão abordados alguns temas relacionados a AVDs, tais como seus componentes, seus meios de comunicação, aspectos importantes para sua implementação, a possibilidade de particionamento e protocolos de comunicação.

1.2 Componentes de um AVD

Um AVD consiste de quatro componentes básicos que trabalham em conjunto para fornecer a sensação de imersão em diferentes localidades. Os componentes são: displays gráficos, dispositivos de comunicação e controle, sistema de processamento e rede de comunicação (Sementille, 1999).

Os Displays gráficos oferecem ao usuário a visão tridimensional do ambiente. Exemplos: monitores, capacetes (HMD) e CAVEs.

A visão que o usuário tem do AV é, geralmente, obtida por visualizadores. O visualizador é um programa que controla a movimentação de um usuário e exibe a visão do Ambiente Virtual, permitindo assim o controle sobre o que quer ver e quando ver. Quando há outras máquinas controlando visões diferentes de um mesmo ambiente, estas devem ser sincronizadas para dar a ilusão de um conjunto.

Os dispositivos de comunicação e controle servem para o usuário manipular objetos e se comunicarem em um AV. Assim dispositivos de entrada como mouses, teclados, ou mais sofisticados como luvas (*datagloves*) são usados para executar essas tarefas. Para uma

completa imersão em um AVD, os usuários podem se comunicar verbalmente por meio do uso de microfones.

A unidade de processamento em um AVD é responsável por receber os eventos dos dispositivos de entrada do usuário e computar o efeito dessas entradas dentro do ambiente. Além disso, o processador deve enviar mensagens aos outros participantes, informando as mudanças feitas por um usuário e manter uma atualização do ponto de vista do ambiente.

A rede de comunicação é responsável pela troca de informações em um AVD, como por exemplo, o posicionamento de um avatar, movimentação de objeto e sincronização do estado compartilhado no AVD, tais como tempo e visibilidade.

1.3 A Taxonomia dos AVDs

Os AVDs podem ser classificados em centralizados ou distribuídos (Kirner; Pinho, 1996).

No modelo centralizado, todos os usuários compartilham o AV, onde existe uma única base de dados. Obtém-se consistência por meio de mecanismos de trava (*lock)*, porém pode apresentar problemas com escalabilidade e confiabilidade.

Ainda segundo Kirner; Pinho (1996), o modelo distribuído pode ser dividido em replicado ou particionado.

Os ambientes replicados são geralmente de pequeno porte, onde cada usuário tem a réplica do AV. Quando se altera algo no ambiente, a alteração deve ser transmitida para os outros usuários.

Os ambientes particionados são de grande escala, assim apenas uma parte do ambiente fica em cada máquina. O usuário pode penetrar em qualquer região do ambiente, que receberá uma cópia daquela região e caso haja uma alteração nele, esta é repassada para todos os usuários que estão nessa região.

1.4 Modelos de Comunicação

Os modelos de comunicação são basicamente divididos em Modelo Cliente/Servidor, *Peer-to-Peer*, computação distribuída orientada a objetos e híbrido (Macedonia, 1997; Zyda & Singhal, 1999).

O modelo Cliente/Servidor é simples e implica somente na comunicação entre o cliente e o servidor, não havendo comunicação entre os clientes.

Uma desvantagem seria a falha do servidor, deixando o sistema inoperante, porém isso poderia ser resolvido com a técnica de replicação de servidores. Outro problema seria o atraso na comunicação no caso de muitos usuários acessarem o servidor ao mesmo tempo.

O modelo *Peer-to-Peer* implica na comunicação direta entre *hosts*, sem usar servidores. O problema é que além do gerenciamento local, cada *host* deve também tratar de acessos remotos.

O paradigma da computação distribuída orientada a objetos consiste na comunicação do cliente com o serviço de objetos do solicitador e do receptor de serviços que são intermediados por um agente (*broker*).

O modelo Híbrido combina o modelo Cliente/Servidor com o modelo *Peer-to-Peer*. Nesse modelo o servidor atua como administrador, gerenciando a conexão, autenticação e desconexão de usuários.

1.5 Estruturas Básicas de Comunicação

As estruturas básicas de comunicação são *unicast*, *broadcast* e *multicast*.

Na estrutura *unicast* a comunicação de dados é realizada apenas entre dois *hosts* (um-para-um).

No *broadcast*, a comunicação é feita da forma “um-para-vários”, em que um *host* envia um dado para todos os outros da rede. Ele é usado em ambientes virtuais replicados.

O *multicast* é conhecido como “um-para-vários” ou “vários-para-um” em uma única operação, onde os dados são enviados para um grupo de máquinas simultaneamente. Ele limita o número de participantes em uma mesma recepção de mensagem e economiza na largura de banda, sendo o fluxo de mensagem reduzido. Usado em ambientes virtuais particionados.

Formas de Distribuição e Armazenamento de Dados

A escolha da forma de distribuição e armazenamento de dados é restrita ao objetivo da aplicação.

Os modelos são apresentados a seguir:

- Banco de Dados Centralizado e Compartilhado*:* os dados estão centralizados em uma máquina que atua como servidor. Os clientes mantêm somente os dados pertinentes à cena em memória e apenas um usuário pode modificar o banco de dados em um determinado tempo.
- Banco de Dados Replicado de Mundos Homogêneos*:* cada *host* conectado na aplicação possui uma réplica completa dos dados da aplicação. A consistência é obtida através do *heartbeat* (o *host* envia periodicamente mensagens informando o seu estado). Tem como vantagem o tamanho pequeno das mensagens e como desvantagem o aumento de usuários, podendo gerar inconsistência de dados.
- Banco de Dados Distribuído e Compartilhado com Alterações *Peer-to-Peer*: cada ambiente possui apenas uma parte do ambiente em sua base de dados. Devido ao grande número de mensagens, usa-se o *multicast* para diminuir o

tráfego. Usado para Ambientes Virtuais de Grande Porte e tem como desvantagem a fragilidade na segurança.

- Banco de Dados Distribuído e Compartilhado Cliente/Servidor: baseado na técnica cliente/servidor, tendo o banco de dados particionado em seus clientes. O *broker* é responsável em manter o mundo atualizado.

1.6 Desenvolvimento de AVDs

Aspectos Principais

Os fatores a serem apresentados podem afetar de alguma maneira o desempenho e/ou complexidade de um AVD, podendo prejudicar o realismo com atrasos e lentidão.

- Número de usuários

Os usuários podem ser classificados em:

Monousuário: permite a existência de outros usuários, porém só um interage com o ambiente.

Multiusuário: sistemas complexos que exigem um poder maior de computação, como o grande número de tarefas.

- Número e tamanho dos mundos virtuais

Em sistemas multiusuários pode-se ter vários mundos ativos, aumentando a complexidade do gerenciamento. Além disso, mundos muito grandes demoram muito para serem carregados.

- Interconexão de Mundos

Corresponde a definir limites entre mundos e o portal de entrada e também localizar e carregar o mundo antecipadamente no modelo particionado.

- Objetos presentes no mundo

Vários fatores implicam em um melhor ou pior desempenho do sistema: nível de detalhes, quantidade de objetos e comportamento.

Objetos mais próximos do usuário necessitam um nível de detalhes maior, diferente dos objetos mais afastados.

Em um AVC, o usuário pode explorar o objeto segundo dois aspectos: objetivo ou subjetivo. No aspecto objetivo todos os usuários acessam as mesmas informações do objeto. No aspecto subjetivo, o usuário tem uma maior interação, acessando informações exclusivas do objeto.

- Interação entre os usuários no mundo

Aborda a representação dos usuários (avatares) e troca de mensagens entre eles.

Pontos de vista, riqueza de movimentos, expressões faciais, comunicação por texto ou voz influem no desempenho de um Ambiente Virtual.

1.7 Particionamento

O AV é particionado em áreas menores para o melhor processamento do ambiente.

O AVD pode ser particionado em algumas formas, como *grids* (retângulos) e hexágonos. Os *grids* têm a desvantagem de muitos pontos onde os retângulos se encontram, porém é mais fácil a implementação. Quando o avatar chega em um ponto, os quatro

retângulos em volta são renderizados, Usando hexágono a renderização é feita em somente três regiões, facilitando o processamento.

A relação entre o processamento do AVD e a comunicação em rede representa o fundamento para melhorar a escalabilidade e o desempenho dos AVDs.

O gerenciamento de fluxo de dados pode ser dividido em três principais categorias: filtragem por área de interesse, *multicast* e agregação *multicast* híbrida. (Rodrigues et al., 2006).

a) Subscrição de filtragem por área de interesse

Os *hosts* transmitem informações para um grupo de gerenciadores de subscrição, chamados de "servidores de filtragem". Informações originárias de regiões distantes ou ocultas são geralmente de menor importância em relação às vizinhas. Filtros de alta prioridade garantem que a informação mais importante seja entregue a cada *host*.

A filtragem intrínseca inspeciona o conteúdo da aplicação dentro de cada pacote para determinar se deve ser entregue para um *host* em particular.

A filtragem extrínseca filtra os pacotes com base em prioridades de rede.

b) Multicast

No *Multicast* é possível a um *host* optar se deseja participar de um grupo. O pacote só será entregue para os *hosts* que se inscreverem no grupo *multicast*.

Pode-se determinar um endereço *multicast* diferente para cada entidade no AVD.

Cada grupo *Multicast* tem um único remetente, cuja entidade não muda. Desta forma, os roteadores mantêm uma única distribuição para cada grupo *multicast*.

c) Agregação Multicast Híbrida

Permite que cada cliente sintonize seu próprio conjunto de subscrições baseadas em interesses locais, permitindo uma forma de filtragem intrínseca. Garante também que o particionamento não seja tão refinado a ponto da transmissão de dados se degenerar num simples *unicast*.

1.8 Protocolos de Comunicação para AVDs

A seguir serão apresentados os protocolos de comunicação mais utilizados em AVDs, mostrando suas características e em quais aplicações são mais utilizados.

TCP/IP

A arquitetura TCP (*Transmission Control Protocol*) / IP (*Internet Protocol*) representa um grupo de protocolos para transmitir informações pela Internet. O TCP e o IP são apenas dois dos protocolos abrangidos pela arquitetura TCP/IP (Rodrigues et al., 2006). Ela fornece como serviços a entrega e recepção de serviços confiáveis.

O IP define a unidade de transferência e o formato de todos os dados enviados pela Internet. Ele escolhe o caminho pelo qual o pacote vai passar, do *host* origem até o *host* destino. Esses *hosts* são identificados por endereços fixos chamados endereços IPs. Porém o IP não garante a entrega, assim é usado o protocolo da camada mais alta, o TCP.

O TCP divide a mensagem em pacotes, enumera-os e envia as mensagens, recupera pacotes perdidos e o destino monta a mensagem recebida de acordo com o original.

Para que essa comunicação seja perfeita, o *host* destino envia uma mensagem de confirmação (ACK) para o *host* destino. Para que não haja duplicação de pacotes, estes são enumerados, assim o *host* destino sabe qual pacote recebeu.

O UDP não é um protocolo confiável, pois envia as mensagens em um único pacote e não há confirmação da chegada da mesma. Apesar disso o UDP pode ser usado em aplicações em que os pacotes perdidos não tenham tanta importância, pois é muito mais rápido.

DIS (Distributed Interactive Simulation)

Em 1989, o *Departament of Defense* dos Estados Unidos (DoD) iniciou um projeto chamado *Distributed Interactive Simulation* (DIS), resultando em um protocolo do mesmo nome, reconhecido pelo *Institute of Electrical and Electronics Engineers* (IEEE) em 1993. O protocolo é baseado em IP *multicasting* e usado para simulação distribuída de cenários militares (Rodrigues et al., 2006).

O DIS define um número de unidades (PDU's) que são transferidas para todos os outros participantes da simulação para transferir o estado de cada objeto. A especificação do DIS contém 27 formatos de mensagem PDU e cada serviço há um tipo de PDU específica, por exemplo, disparos de projéteis ou marcação de campos minados. A PDU mais importante é a de estado da entidade, utilizada para transmitir o estado completo do objeto, posição, orientação e velocidade.

O DIS exige que cada entidade envie constantemente mensagens de estado dos objetos ativos, para todos os *hosts*. Mesmo sendo um protocolo *peer-to-peer* ele não suporta um grande número de usuários, assim, não pode melhorar sua escalabilidade.

VRTP (Virtual Reality Transfer Protocol) (Brutzman, 1997).

O *Virtual Reality Transfer Protocol* (VRTP) foi desenvolvido pela DARPA, e espera-se que tenha o mesmo papel em AVD para o VRML (*Virtual Reality Modeling Language*), quanto o HTTP (*Hyper Text Transfer Protocol*) tem para as páginas HTML (*Hypertext Markup Language*). Além do HTTP, são necessários outros recursos na realização dessa

tarefa, que combinem a comunicação peer-to-peer, cliente/servidor e monitoramento de rede (Brutzman, 1997).

Apesar de não haver implementações nem especificações completas. Alguns objetivos podem ser analisados. Um computador consegue executar várias tarefas ao mesmo tempo no VRTP. Ele pode ser um cliente, um *host* ou um *peer*. A parte crucial de sua arquitetura é a comunicação *peer-to-peer* e o *multicast*.

A recuperação de erros também é importante ser destacada, pois é fundamental para um protocolo escalável para AVDs.

Na parte de transmissão, para detectar e resolver os problemas, o VRTP deve escolher automaticamente o protocolo de transporte para cada aplicação.

DWTP (Distributed Worlds Transfer and communication Protocol)

O *Distributed Worlds Transfer and communication Protocol* (DWTP) é um protocolo de rede para camada de aplicação de ambientes virtuais compartilhados. Diferente do DIS, ele é heterogêneo e independente da aplicação, permitindo que dados de diferentes tipos possam ser transmitidos, como por exemplo, eventos, mensagens, arquivos.

Esse tipo de protocolo utiliza mensagens de reconhecimento negativo (NACK – *Negative Acknowledgement*), precisando ser confirmado quando não há o recebimento correto da mensagem. Para evitar erros na transmissão causados por roteadores *multicast* sobrecarregados com NACKs adicionais dos clientes, alguns protocolos instituem um tempo de espera antes do envio de um NACK.

Multi-user 3D Protocol

O protocolo *Multi-user 3D Protocol* (MU3D) foi desenvolvido por Galli e Luo na *University of Balearic Islands*, para um projeto de arquitetura colaborativa. O protocolo *MU3D* é *peer-to-peer* e envia apenas atualizações e não os dados completos.

No envio das mensagens, o cliente bloqueia o nó enquanto estiver fazendo alterações e libera somente depois de enviar a mensagem de alteração para todos os clientes. O tamanho das mensagens é muito pequeno (200 a 300 bytes), incluindo todos os *overloads* das camadas inferiores do protocolo. Isso se deve ao fato do protocolo fazer somente atualizações.

1.9 Modelos de Suporte para Comunicação

Os modelos de suporte para comunicação podem ser divididos em quatro categorias: baseados em *sockets, toolkits, middlewares* e *frameworks*.

Os *sockets* são primitivas que estabelecem um canal de comunicação inter-processos por meio de chamadas de sistema. As mensagens são enviadas e recebidas através de uma porta de comunicação estabelecida.

Os *toolkits* são modelos desenvolvidos especificamente para um conjunto de aplicações. Como exemplos, podem ser citados o *World2World* (W2W), que é uma solução em rede baseada no modelo cliente/servidor para simulações 3D interativas em tempo real, e *WorldToolKit* (WTK) (Sense8, 1998), DIVE (Hagsand, 1996), MASSIVE (Greenhalgh; Benford, 1995), entre outros.

Os *middlewares* são soluções independentes de plataformas e domínio de aplicação. Como exemplo poder ser citado o modelo CORBA (*Common Object Request Broker Architecture*) (Corba, 2001), Java RMI (*Remote Method Invocation*) (JavaRMI, 2002) além de outros.

O modelo CORBA é um padrão para desenvolvimento de aplicações distribuídas segundo o modelo cliente/servidor, usando orientação a objetos. Ele baseia-se em um barramento de interconexão de objetos (*broker*), habilitando a comunicação transparente entre clientes e objetos remotos.

Os *frameworks* são soluções específicas para um domínio de aplicação. Procura promover colaboração em redes heterogêneas, suportando computação móvel.

Segundo Deriggi et. al, os sistemas desenvolvidos com base em *toolkits* tendem a ter um desempenho melhor em comparação com outras plataformas, entretanto apresentam problemas relacionados a portabilidade e dependência de plataforma específica. Esse problema pode ser solucionado utilizando-se a plataforma CORBA, por exemplo. A desvantagem relacionada ao CORBA é a ausência de um padrão para estabelecimento de comunicação em ambientes virtuais.

2. Remote Method Invocation

O sistema RMI (Remote Method Invocation) permite que um objeto em execução em uma máquina virtual Java invoque métodos de um objeto em execução em outra máquina virtual Java. O RMI possibilita a comunicação remota entre programas escritos na linguagem de programação Java (Sun, 2006).

O RMI tem como vantagens: simples implementação, objetos podem ser passados por valor facilmente, objetos podem ser transferidos de servidores remotos para clientes de forma segura e o custo zero, pois já vem na instalação do JDK. Como desvantagens, aponta-se a falta de mecanismo para descrição de objetos e a não possibilidade de chamadas geradas dinamicamente.

2.1 Visão Geral

As aplicações RMI são geralmente compostas de dois programas separados: um cliente e um servidor. Uma aplicação servidora típica cria alguns objetos remotos, faz referências a eles e aguarda que clientes invoquem métodos desses objetos. Uma aplicação cliente típica busca uma referência remota para um ou mais objetos remotos no servidor e invoca seus métodos neles. O RMI provém um mecanismo pelo qual o servidor e o cliente se comunicam e passam informações. A aplicação é às vezes referida como uma aplicação objeto distribuída.

As aplicações objeto distribuídas precisam (Sun, 2006):

- Localizar objetos remotos: Aplicações podem usar vários mecanismos para obter referência a objetos remotos. Por exemplo, uma aplicação pode registrar seus objetos remotos com o *rmiregistry,* ou a aplicação pode passar e retornar referências a objetos remotos como parte de sua operação normal.

- Comunicar com objetos remotos: detalhes de comunicação entre objetos remotos são controlados pelo RMI; para o programador, comunicação remota parece um padrão Java de invocação de métodos.
- Carregar classes *bytecodes* para objetos que são passados: pelo RMI permitir que objetos sejam passados de um lado para o outro, o RMI provém mecanismos necessários para carregar um código de objeto, tanto quanto para transmitir seus dados.

Uma aplicação distribuída RMI usa o *registry* para obter a referência para um objeto remoto. O servidor chama o *registry* para associar o nome com o objeto remoto. O cliente procura o objeto remoto pelo seu nome no *registry* do servidor e então pode invocar seus métodos.

2.2 Carregamento Dinâmico de Código

Uma das principais características do RMI é a habilidade de baixar os *bytecodes* da classe objeto, se a classe não é definida no recebedor da máquina virtual. Os tipos e o comportamento de um objeto, previamente disponível somente em uma *single virtual machine,* pode ser transmitida para outra, possivelmente remota, máquina virtual. O comportamento dos objetos não muda quando passados para outra máquina virtual, pois o RMI passa objetos pelo seu tipo verdadeiro. Isto permite novos tipos serem introduzidos em uma máquina remota virtual, entendendo o comportamento de uma aplicação dinamicamente.

2.3 Interfaces Remotas, Objetos e Métodos

Como qualquer aplicação Java, uma aplicação distribuída construída usando Java RMI é feita de interfaces e classes. As interfaces definem métodos e as classes implementam os métodos definidos nas interfaces. Em uma aplicação distribuída, algumas implementações

podem residir em diferentes máquinas virtuais. Objetos que têm métodos que podem ser chamados por máquinas virtuais são chamados objetos remotos.

RMI trata um objeto remoto diferentemente de um objeto não remoto quando o objeto é passado de uma máquina virtual para outra. Além de fazer uma cópia da implementação do objeto na máquina virtual receptora, o RMI passa um *stub* remoto para um objeto remoto. O *stub* atua como um representante local, ou *proxy*, para o objeto remoto e basicamente é, para o requisitante, a referência remota. O requisitante invoca um método no *stub* local, que é responsável por carregar a chamada do método no objeto remoto.

Um *stub* para um objeto remoto implementa o mesmo conjunto de interfaces remotas que um objeto implementa. Isso permite que um *stub* seja *cast* para qualquer interface que o objeto remoto implemente. Isso também significa que somente aqueles métodos definidos em uma interface remota são disponíveis para serem chamados na máquina virtual receptora.

2.4 Criando Aplicações Distribuídas Usando RMI

Para facilitar a criação de uma aplicação distribuída em RMI, sugere-se o cumprimento das etapas que são discutidas nessa seção.

Projetar e implementar os componentes de sua aplicação distribuída

Primeiramente, deve-se definir a arquitetura de aplicação e determinar quais componentes são objetos locais e quais devem ser remotamente acessados. Este passo inclui (Sun,2006) :

- Definir as interfaces remotas: uma interface remota especifica os métodos que podem ser invocados remotamente por um cliente. Programas clientes para interfaces remotas, não para a implementação de classes dessas interfaces.

- Implementar os objetos remotos: objetos remotos devem implementar uma ou mais interfaces remotas. A classe do objeto remoto pode incluir implementações de outras interfaces (local ou remota) e outros métodos (que são disponíveis somente localmente). Se qualquer classe local está para ser usada como parâmetro ou retorno de valores para quaisquer desses métodos, elas precisam ser implementadas.
- Implementar os clientes: Clientes que usam objetos remotos podem ser implementados a qualquer hora depois das interfaces remotas definidas.

Compilar fontes e gerar stubs

Este é um processo de dois passos (Sun, 2006). No primeiro passo, usa-se o compilador *javac* para compilar os arquivos fontes, contendo a implementação das interfaces remotas e implementações das classes servidor e cliente.No segundo passo, usa-se o compilador *rmic* para criar *stubs* para os objetos remotos. O RMI usa uma classe *stub* do objeto remoto como um *proxy* nos clientes, então os clientes podem comunicar-se com um objeto remoto particular.

Construir classes acessíveis à rede

Neste passo os arquivos classes são associados com as interfaces remotas, *stubs* e outras classes que precisam ser enviadas para clientes, acessíveis via um servidor *Web*.

Iniciar a Aplicação

Iniciar a aplicação inclui rodar o objeto remoto *registry* RMI, o servidor e o cliente.

2.5 Construindo um Servidor *Compute Engine*

Esta sessão foca em uma aplicação distribuída simples chamada *compute engine*. O *compute engine* é um objeto remoto no servidor, que busca as tarefas dos clientes, executa e retorna o resultado. As tarefas são executadas na máquina onde o servidor está executando. Este tipo de aplicação distribuída permite que máquinas clientes façam uso de uma máquina poderosa ou de um hardware especializado.

As tarefas não precisam ser definidas quando o *compute engine* é escrito, elas podem ser criadas a qualquer momento e enviadas para o *compute engine* executá-las. Mesmo que uma classe de uma tarefa tenha sido escrita muito tempo depois do *compute engine* ter sido iniciado, ela pode ser enviada.

O RMI dinamicamente carrega o código tarefa no *compute engine* da máquina virtual Java e executa a tarefa sem conhecimento prévio da classe que implementa a tarefa. Uma aplicação assim, que tem a habilidade de carregar o código dinamicamente, é geralmente chamada de aplicação baseada em comportamento (Sun, 2006).

A Figura 1 mostra a implementação do *Compute Engine*:

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception:");
            e.printStackTrace();
        }
    }
}
```

Figura 1- Implementação do Compute Engine

As seções a seguir discutem cada componente da implementação do servidor *Compute Engine*.

2.6 Projetando uma Interface Remota

No *compute engine*, um protocolo é responsável pela permissão do envio de tarefas para o *compute engine*, pela execução das tarefas e dos resultados retornados ao cliente. Cada interface contém um método único. A interface do *compute engine*, compute, permite que as

tarefas sejam enviadas para o *engine*, a interface cliente, *Task*, define como o *compute engine* executa a tarefa enviada.

A interface *compute.Compute()* define a parte acessível do *compute engine*. A Figura 2 mostra a interface remota com seu único método:

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote {
      Object executeTask (Task t) throws Remote Exception;
}
```

Figura 2- Interface Compute

A interface necessária para o *compute engine* define o tipo *Task*. Esse tipo é usado como argumento do método *executeTas*k da interface Compute. A interface *compute.Task()* define a interface entre o *compute engine* e o trabalho a ser realizado. A Figura 3 mostra a interface *Task*:

```
package compute;
import Java.io.Serializable;
public interface Task extends Serializable {
      Object execute();
}
```

Figura 3- Interface Task

A interface *Task* define um método único, *execute*(), que retorna um *Object*, sem parâmetros e sem exceções. Desde que a interface não estenda *Remote*, o método nessa interface não precisa listar *java.rmi.RemoteException* na sua cláusula *throws*.

Na interface *Task* é utilizada a interface *java.io.Serializable*. O RMI usa esse mecanismo de serialização para transportar objetos por valor entre máquinas virtuais Java. A serialização faz a classe ser capaz de converter em um *byte stream* auto-descritivo, podendo ser usado para uma cópia exata do objeto serializado.

Diferentes tipos de tarefas podem ser executadas por um objeto *compute* desde que sejam implementações do tipo *Task*. As classes que implementam esta interface podem conter quaisquer dados ou métodos usados para a computação da tarefa.

2.7 Implementando uma Interface Remota

Em geral, a classe implementação de uma interface remota deve, no mínimo: declarar as interfaces remotas a serem implementadas, definir o método construtor para o objeto remoto, fornecer uma implementação para cada método remoto nas interfaces remotas.

O servidor precisa criar e instalar os objetos remotos. Este procedimento pode ser encapsulado em um método *main* na classe implementação do objeto remoto, ou incluso completamente em outra classe. O procedimento deve criar e instalar um *security manager*, criar uma ou mais instâncias de um objeto remoto e registrar pelo menos um dos objetos remotos com o objeto remoto RMI *registry* (Sun, 2006).

2.7.1 Declarar as Interfaces remotas a serem implementadas

O *UnicastRemoteObject* é uma classe de conveniência definida no API no RMI, que pode ser usado como uma superclasse para implementações de objetos remotos.

Uma implementação de um objeto remoto não tem que estender do *UnicastRemoteObject* e qualquer implementação não precisa prover implementações apropriadas dos métodos *java.lang.Object*. Além disso, a implementação de um objeto remoto deve fazer uma chamada explícita para um dos métodos do *UnicastRemoteObject*, assim o objeto pode aceitar chamadas entrantes.

2.7.2 Definir o construtor

O construtor simplesmente chama a superclasse construtor que não tem argumentos e faz parte da classe *UnicastRemoteObject*. Durante a construção, um *UnicastRemote object* é exportado, significando que está disponível para novas requisições ouvindo chamadas vindas de clientes em uma porta anônima.

2.7.3 Fornecer Implementações para cada método remoto

Este método implementa o protocolo entre o *ComputeEngine e* seus clientes. Os clientes fornecem o *ComputeEngine* com um objeto *Task*, que tem uma implementação do método execute da tarefa. O *ComputeEngine* executa o *Task* e retorna o resultado do método execute da tarefa diretamente para o requisitante.

2.7.4 Passando Objetos em RMI

Objetos remotos são essencialmente passados por referência (Sun, 2006). Uma referência para um objeto remoto é um *stub*, que é um *Proxy* do lado cliente que implementa o conjunto de interfaces remotas.

Os objetos locais são passados por cópias, usando serialização. Todos os campos são passados, exceto aqueles que são marcados como *static* ou *transient*.

Ao passar um objeto por referência, toda mudança feita neste objeto chamado por um método remoto será refletido no objeto remoto original.

Os parâmetros, retorno de valores exceções são passados por valor, assim uma cópia do objeto é criada na máquina virtual receptora e todas as mudanças feitas neste objeto são refletidas apenas na cópia, mantendo a original intacta.

2.7.5 Criar e Instalar um Security Manager

A primeira coisa que o método *main* faz é criar e instalar um *security manager* (Sun, 2006). Ele determina se o código carregado tem acesso ao sistema de arquivo local ou outras operações privilegiadas.

Caso um programa RMI não tenha um *security manager* instalado, ele não carrega classes para objetos recebidos como parâmetros, retorno de valores ou exceções em chamadas de métodos remotos. Isso mantém a segurança no código carregado.

2.7.6 Tornar o Objeto Remoto disponível para os Clientes

Antes de o requisitante poder invocar um método em um objeto remoto,ele deve obter primeiro a referência para o objeto remoto (Sun, 2006). O sistema fornece um objeto remoto particular, o RMI *registry*, para encontrar referências para objetos remotos.

O RMI *registry* é um serviço de nome de objeto remoto que permite clientes remotos adquirirem uma referência para o objeto remoto através de um nome. O registro é usado somente para localizar o primeiro objeto remoto que um cliente RMI necessita usar. Esse primeiro objeto remoto ajuda a encontrar outros objetos.

2.8 Criando um programa Cliente

O *compute engine* é um programa bem simples: roda as tarefas que são dadas a ele (Sun, 2006). Já os clientes para o *compute engine* são mais complexos. Um cliente precisa chamar o *compute engine*, mas também tem que definir a tarefa a ser executada por ele.

Como no servidor, o cliente começa pela instalação do *security manager*. Isso é necessário, pois o RMI pode estar carregando o código para o cliente. Toda vez que um código é carregado pelo RMI, o *security manager* deve estar presente.

Depois da instalação, o cliente constrói um nome para observar um objeto remoto *Compute*. O cliente utiliza o método *Naming.lookup* para localizar o objeto remoto pelo nome no registro do *host* remoto. Ao localizá-lo, o código cria uma URL que especifica o *host* onde o *compute server* está rodando.

Finalmente, o cliente invoca o método *execute.Task* para o objeto remoto *Compute* e o programa apresenta o resultado de acordo com a entrada fornecida. A principal classe cliente pode ser vista na Figura 4.

```
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

Figura 4- ComputePi - Principal Classe Cliente

3. DESCRIÇÃO DA BIBLIOTECA

Como mencionado, a implementação desta biblioteca visa tornar mais fácil o trabalho do programador que pretende criar uma aplicação que utilize uma conexão entre cliente e servidor na implementação de um AVD. O programador poderá utilizar as classes e métodos da biblioteca para fazer a conexão, troca de mensagens e manipulação de objetos de um AVD.

De maneira geral, a biblioteca é composta de seis arquivos Java, sendo dois arquivos para a interface e implementação do cliente, dois para a interface e implementação do servidor e dois arquivos para armazenar informações dos clientes do ambiente e para armazenar os arquivos.

Do lado do cliente há o módulo *Client*, que é a interface que fornece os métodos que podem ser utilizados para o uso do cliente e o módulo *ClientImpl,* que implementa esses métodos contidos na interface. Do lado do servidor outros dois módulos aparecem: o *Server*, que fornece a interface do servidor e o *ServerImpl*, que implementa métodos contidos na interface desse servidor. Para armazenar a lista de clientes do ambiente é usada a classe *UserInfo* (Figura 5) e para armazenar os arquivos do ambiente é usada a classe *FileInfo* (Figura 6).

```
public class UserInfo
{
  public String name;
  public Client client;

      public UserInfo(String name, Client client)
      {
            this.name = name;
            this.client = client;
      }

}
```

Figura 5 - Classe UserInfo

```
public class FileInfo {
	public String filename;

	public FileInfo(String file) {
		this.filename = file;
	}
}
```

Figura 6- Classe FileInfo

Para a utilização da biblioteca basta o programador criar dois arquivos: um para ser o seu cliente e o outro para ser seu servidor e assim utilizar os métodos contidos nas interfaces *Client* e *Server.*

3.1 Descrição das Primitivas

As primitivas da biblioteca são divididas em três grupos: métodos para conexão e desconexão, métodos para troca de mensagens textuais e métodos para manipulação e controle de objetos, como apresentado na Figura 7.

Classe Server	Classe Client
connectToServer()	receiveConnection()
login()	receiveEnter()
logout()	receiveExit()
chat()	receiveChat()
StoreOpenedFile()	requestFile()
requestFile()	receiveFile()
broadTranslation()	receiveTranslation()

Figura 7 – Métodos da biblioteca

3.1.1 Métodos para conexão e desconexão

O método para conexão do lado do cliente é chamado de *receiveConnection()* e possui dois parâmetros do tipo *String*: *server* e *service*. O parâmetro *server*, refere-se à localização do servidor e o parâmetro *service*, que é o nome do servidor utilizado. Esses parâmetros são passados pelo usuário quando o programa estiver executando ou pelo programador quando fizer a implementação do programa. Os parâmetros são recebidos pelo método

receiveConnection() e um objeto é criado para que aconteça a conexão com o servidor de nomes. Caso não seja possível criar esse objeto, a exceção correspondente é escrita na tela.

O método de conexão do lado do servidor é chamado de *connectToServer()* e também usa como parâmetros *server* e *service*. O método cria um objeto e o associa com o *service* passado como parâmetro.

A Figura 8, mostra o trecho de código correspondente ao método *connectToServer()*.

```
public void connectToServer(String server, String service) throws
java.rmi.RemoteException
      {
        try {
            ServerImpl server = new ServerImpl();
            Naming.rebind(service, server);
            System.out.println("Servidor pronto.");
        }
        catch (Exception e) {
            System.out.println("ServerImpl erro" + e.getMessage());
        }
      }
```

Figura 8- Método connectToServer

O método para entrada no ambiente do lado do servidor é chamado de *login()* e possui como parâmetros o *name,* do tipo *String*, que é o nome do usuário que entrará no ambiente e estará contido no servidor; e o *client*, que é um objeto da classe *Client* e é utilizado para armazenar as informações passados pelo programa do usuário.

A Figura 9 mostra o trecho de código correspondente a implementação do *login* do usuário.

```
public void login(String name, Client c) throws java.rmi.RemoteException
      {
            System.out.println(name+" entrou");
            if (c != null && name != null) {
                  Enumeration enume = clients.elements();
                  UserInfo u = new UserInfo(name, c);
                  clients.addElement(u);

                  while (enume.hasMoreElements()) {
                  UserInfo u2 = (UserInfo) enume.nextElement();
                  u2.client.receiveEnter(name, c);
                  }

                  Enumeration enumef = files.elements();

                  while (enumef.hasMoreElements()) {
                  FileInfo u2 = (FileInfo) enumef.nextElement();
                        c.requestFile(u2.filename);
                  }
            }
      }
```

Figura 9- Método login

Do lado do cliente há o método *receiveEnter()* e seus parâmetros são os mesmos do servidor. O método é invocado pelo servidor e então é feito o armazenamento das informações do cliente com o auxílio da classe *UserInfo*.

O método para desconexão do módulo cliente é chamando de receive*Exit()* e possui como parâmetro o *name*. Do lado do servidor tem-se o método *logout()* com o parâmetro *name* também. Neste método, o parâmetro passado é procurado na lista de informações de clientes, gerenciada pela classe *UserInfo* e é então excluído, desconectando o usuário do ambiente.

O trecho da implementação do método *logout()* pode ser visto na Figura 10.

```
public void logout(String name) throws java.rmi.RemoteException
      {
        if (name == null) {
          System.out.println("Name = null: Não é possível remover name");
          return;
        }

    UserInfo u_gone = null;
    Enumeration enum = null;

    synchronized (clients) {
              for (int i = 0; i < clients.size(); i++) {
                    UserInfo u = (UserInfo) clients.elementAt(i);
                    if (u.name.equals(name)) {
                      System.out.println(name+" saiu");
                      u_gone = u;
                          clients.removeElementAt(i);
                          enum = clients.elements();
                          break;
                    }
              }
            }

    if (u_gone == null || enum == null) {
      System.out.println("Nenhum usuario com nome "+name+" encontrado ");
      return;
    }

    while (enum.hasMoreElements()) {
      UserInfo u = (UserInfo) enum.nextElement();
      u.client.receiveExit(name);
    }
      }
```

Figura 10- Método logout

3.1.2 Métodos para troca de mensagens textuais

Para a troca de mensagens textuais são usados os métodos *chat()* e *receiveChat()*. No cliente o método usado é o receive*chat()*, que tem como parâmetros o nome do usuário destino, *name*, e a mensagem transcrita pelo usuário a ser entregue ao destino, *message*. Essa mensagem é enviada pelo usuário ao servidor por meio do método *chat()*, de mesmos parâmetros (Figura 11), sendo o responsável pelo envio da mensagem ao participante destinatário do ambiente. O participante pode então visualizar a mensagem escrita recebida pelo método *receiveChat()*.

```
public void chat(String name, String message) throws
java.rmi.RemoteException
      {
        Enumeration enume = clients.elements();

        while (enume.hasMoreElements()) {
          UserInfo u = (UserInfo) enume.nextElement();
          u.client.receiveChat(name, message);
        }
      }
```

Figura 11- Método chat

3.1.3 Métodos para manipulação e controle de objetos

Para manipulação e controle dos objetos os métodos são os seguintes: *receiveFile(), requestFile() e receiveTranslation()*; do lado cliente e *StoreOpenedFile(), requestFile(), e, broadTranslation()*, do lado servidor.

Um arquivo, ao ser carregado por algum participante, tem sua referência imediatamente armazenada no servidor. Este, por fim, comunica a todos os participantes do AVD sobre o novo objeto que acaba de ser inserido. Além disso, todo participante recebe uma cópia deste arquivo.

O método *StoreOpenedFile()* usa os parâmetros *file*, que é arquivo a ser armazenado e *textString*, que é o nome do arquivo a ser armazenado. Do lado cliente o método usado é o *requestFile()*, de um único parâmetro, *file*. O usuário solicita ao servidor que o objeto seja armazenado no AV. Assim o servidor, usando a classe *FileInfo,* faz o armazenamento do arquivo no ambiente virtual.

O *requestFile(),* por meio dos parâmetros *name* e *filename*, avisa ao participante do AV sobre o carregamento do novo objeto. O método lê todo o arquivo e o envia ao participante, utilizando o método *lookup()* para achar a localização do participante no ambiente. Assim o cliente recebe o arquivo através do método *receiveFile().*

Os métodos de *broadcast* utilizados para a rotação e translação dos objetos são: *broadTranslation(),* que faz as modificações, como rotação e translação, utilizando os

parâmetros *x, y* e *z.* Do outro lado, do lado cliente, o método *receiveTranslation()* recebe as modificações ocorridas no método *broadTranslation()*, representado pela Figura 12.

```
public void broadTranslation(double x, double y, double z) throws
java.rmi.RemoteException
      {
        Enumeration enume = clients.elements();

        while (enume.hasMoreElements()) {
          UserInfo u = (UserInfo) enume.nextElement();
          u.client.receiveTranslation(x, y, z);
        }
      }
```

Figura 12- Método broadTranslation

Na classe *FileInfo*, o único método existente é o método *FileInfo()*, com parâmetro *file*, do tipo *String*. Este método faz o armazenamento das informações de um determinado arquivo.

A classe *UserInfo* também contém um único método também chamado de *UserInfo()*, com parâmetros *name*, do tipo *String* e *client*, do tipo *Client.* Este método faz o armazenamento dos Clientes que entram no ambiente, juntamente com suas informações.

Essas são as classes e métodos que compõem a biblioteca e que oferecem o suporte ao programador para criar sua própria aplicação. Com eles é possível fazer conexão, desconexão, manipulação e controle de objetos e de mensagens textuais.

CONCLUSÃO

Esta monografia teve como objetivo mostrar o projeto de criação de uma biblioteca de comunicação para AVDs, que visa facilitar o trabalho do programador, utilizando-a como ferramenta para que ele possa usufruir em suas aplicações, obtendo assim um bom resultado em relação à comunicação. Com os métodos criados, o programador passa a focar apenas no seu aplicativo, deixando de lado problemas como conexão e troca de mensagens.

A utilização do RMI ajudou na implementação da biblioteca, pois trabalha com interfaces, separadamente da implementação dos métodos. Isso proporciona a utilização desses métodos pelo programador para construir sua aplicação. Através do seu programa ele acessa os métodos da biblioteca e faz operações necessárias para que esses métodos funcionem.

Porém, algumas dificuldades foram encontradas na hora de projetar a biblioteca, como a falta de materiais sobre o assunto, seja na Internet ou em livros, tornando um pouco prejudicial o andamento do projeto, além de alguns problemas na hora da implementação da biblioteca.

Mesmo com alguns problemas foi possível realizar o trabalho e aprender mais sobre a linguagem Java, o RMI e sobre os AVDs. O trabalho criou a oportunidade de se aprofundar no assunto e agregar conhecimento sobre o tema, sendo importante para a formação acadêmica e profissional.

Trabalhos Futuros

Alguns assuntos não puderam ser pesquisados devido à falta de tempo, pois foram percebidos no decorrer da implementação da biblioteca, como o problema da instanciação da classe criada pelo programador na classe *ClientImpl*, isso faria com que a classe criada pelo

programador ficasse totalmente independente da biblioteca, não necessitando ser citado na implementação da mesma. Outros métodos podem ser adicionados, como por exemplo, o envio de uma mensagem avisando a todos os usuários do AV que outro usuário acabou de entrar.

REFERÊNCIAS

Benford, J. B., Fahlén, L. E., Mariani, J., Rodden, T. (1994) **Supporting Cooperative Work in Virtual Environments**, The Computer Journal, V. 37, N. 8, pp. 653-668.

Brutzman, D.; Zyda, M.; Watsen K.; Macedonia, M. (1997) **Virtual reality transfer protocol (VRTP) design rationale**. In: Proceedings of the IEEE Sixth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprise. (WETICE'97). IEEE Computer Society. Cambridge, MA. p. 179-186. Disponível em: <http://citeseer.nj.nec.com/brutzman97virtual.html>. Acessado em: 22 fev. 2003.

Cardoso, Alexandre (2006). **Conceito Básico de Realidade Virtual**. Disponível em: http://www.compgraf.ufu.br/alexandre/rv.htm, acesso em 15 novembro 2006.

Corba. (2001). **Common Object Request Broker Architecture**. Disponível em <http://www.corba.org>.

Deriggi Jr., F. V. (1998) **Suporte de Comunicação para Sistemas Distribuídos de Realidade Virtual**, Dissertação (Mestrado), UFSCar, Agosto.

Deriggi Jr., F., Kubo, M.M., Sementille, A.C., Santos, S.G., Brega, J.R.F., Kirner, C. (1999) **CORBA Platform as Support for Distributed Virtual Environment**. In: Proceedings of the IEEE Virtual Reality, Texas, March.

Greenhalgh, C., Benford, S. (1995) MASSIVE: **A Collaborate Virtual Environment for Teleconferencing**. ACM Transactions on Computer-Human Interaction, V. 2, N. 3, pp. 239-261.

Hagsand, O. (1996) **Interactive Multiuser Ves in the DIVE System**, IEEE Multimedia, Spring, V. 3, N. 1, pp. 30-39.

JavaRMI. (2002) **Getting Started using RMI** . Disponível em : <http://java.sun.com/j2se/1.4/docs/guide/rmi/getstart.doc.html>.

Kirner, C., Pinho, M. (1996) **Introdução à Realidade Virtual. Minicurso JAI/SBC**. Recife. PE.

Macedonia, M. R., Zyda, M. J. (1997) **A Taxonomy for Networked Virtual Environments**. IEEE Multimedia. V. 4, N. 1. pp. 48-56.

Rodrigues, L.C.R., Kubo, M.M., Rodello, I.A., Sementille, A.C., Tori, R., Brega, J.R.F. (2006). **Ambientes Virtuais Distribuídos e Compartilhados. In: VIII Symposium on Virtual Reality**, Belém, Maio.

Sementille, A. C. (1999) **A Utilização da Arquitetura CORBA na Construção de Ambientes Virtuais Distribuídos**, Tese (Doutorado) - Instituto de Física de São Carlos - USP, São Carlos, 186p.

Sense8 Corporation. (1998). **WorldToolKit Reference Manual – Release 7**. Mill Valley. CA.

Sun (2006) **Trail: RMI**. Disponível na Internet em <http://java.sun.com/docs/books/tutorial/rmi/index.html>, acesso em 18 maio 2006.

Zyda, M.; Singhal, S. (1999) **Networked Virtual Environments: Design an Implementation**. Addison Wesley Pub.

CRUZ, Elter Girotto da
Implementação de uma biblioteca de comunicação para Ambientes Virtuais Distribuídos / Elter Girotto da Cruz; orientador: Ildeberto Aparecido Rodello. Marília, SP: [s.n.], 2006.
46 f.

Monografia (Bacharelado em Ciência da Computação) – Centro Universitário – Fundação de Ensino Eurípides Soares da Rocha.

1. Biblioteca 2. Comunicação 3. Ambientes Virtuais Distribuídos.

CDD: 006