UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Monitoramento de Ambientes Virtuais Distribuídos com suporte a clientes sob projeção única e multiprojeção

MAURÍCIO GIACOMINI PENTEADO

ORIENTADOR: PROF. DR. LUIS CARLOS TREVELIN

São Carlos - SP
Julho/2012

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Monitoramento de Ambientes Virtuais Distribuídos com suporte a clientes sob projeção única e multiprojeção

MAURÍCIO GIACOMINI PENTEADO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Redes e Sistemas Distribuídos.
Orientador: Prof. Dr. Luis Carlos Trevelin.

São Carlos - SP
Julho/2012

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

P419ma

Penteado, Maurício Giacomini.
Monitoramento de Ambientes Virtuais Distribuídos com suporte a clientes sob projeção única e multiprojeção / Maurício Giacomini Penteado. -- São Carlos : UFSCar, 2012.
99 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2012.

1. Ambientes virtuais colaborativos. 2. Ambientes virtuais distribuídos. 3. Sistemas distribuídos. 4. Realidade virtual. 5. Plataforma JAMP. 6. CORBA - Common Object Request Broker Architecture. I. Título.

CDD: 006 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Monitoramento de Ambientes Virtuais Distribuídos com suporte a clientes sob projeção única e multiprojeção"

Mauricio Giacomini Penteado

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Prof. Dr. Luis Carlos Trevelin
(Orientador - DC/UFSCar)

Prof. Dr. Hélio Crestana Guardia
(DC/UFSCar)

Prof. Dr. José Remo Ferreira Brega
(UNESP/Bauru)

São Carlos
Setembro/2012

Dedico este trabalho a minha mãe Vera Lucia, meu pai Juvenal e

aos meus irmãos Cesar e Agnes, pelo amor e incentivos

dispensados ao longo de toda minha vida

AGRADECIMENTO

Primeiramente e acima de tudo, a Deus, pois "tudo posso naquele que me fortalece" (Fl 4,13).

A meus pais, Juvenal de Aguiar P. Neto e Vera Lucia Giacomini Penteado, e irmãos, Agnes Giacomini Penteado Lodi e Cesar Giacomini Penteado, pelo apoio incondicional, amor, amizade e presença constante, mesmo que distante, durante estes anos de estudos.

Ao meu amigo irmão de consideração Luis Henrique Argenton e família que sempre me acolheram com sinceridade e palavras de apoio em diversos momentos de minha vida.

A minha amiga e namorada Renata Barbosa Sanches que me proporcionou momentos de inspiração para enfrentar os desafios encontrados neste trabalho. E a família Sanches por sempre me acolher com carinho e atenção.

Ao meu orientador Dr. Luis Carlos Trevelin, pela atenção que sempre teve comigo, e por ser um grande mestre e amigo.

Agradeço aos amigos de república Douglas Fabiano de Sousa Nunes, Felipe Bezerra Reis e Renato Fernando dos Santos, pelo crescimento pessoal proporcionado e pelos muitos momentos gratificantes que vivemos juntos.

A minha amiga "sonho" Eliane Mahl, parceira de muitas horas.

A Barbara Castanheira, com certeza seu apoio durante o mestrado será guardado com muito carinho.

A todos os amigos , professores e funcionários do PPGCC/DC/UFSCar por toda ajuda e dedicação profissional desses anos de mestrado.

Agradeço a Bruno Muller, Erick Kathib e Paulinho Martins, grandes amigos. Fabrício Sanches, Daniel Pereira, Fabio Dias e Thiago Dias parceiros de muitos anos e demais amigos de Marília, por momentos de descontração necessários para concluir qualquer projeto.

Agradecimentos especiais para Cesar Giacomini Penteado meu grande ídolo e Luis Henrique Argenton que sempre me ensina muito.

RESUMO

Aplicações Distribuídas Imersivas, Interativas e Colaborativas, dadas suas características de tempo real e multiplataformas, demandam uma estrutura de monitoramento de suas funcionalidades, não plenamente disponíveis nos sistemas atuais de monitoramento, em geral específicos das linguagens e plataformas em que são disponibilizadas. Monitorar sistemas que necessitam de sincronismo no processamento de objetos distribuídos é uma tarefa complexa. Principalmente em sistemas distribuídos e de multiprojeção, onde o sincronismo é fundamental para seu funcionamento. Muitas vezes, é difícil apurar o correto funcionamento dos mesmos. Pode ser útil a desenvolvedores e usuários de tais sistemas a capacidade de se identificar, por exemplo, se a lentidão do sistema está sendo ocasionada por: uso abusivo de um recurso; por um erro na programação de um objeto ou; a identificação de qual objeto travou, comprometendo o sistema como um todo. Sistemas de objetos distribuídos, também conhecidos como SOA (*Service Oriented Archtecture*), geralmente utilizam mediadores do tipo *broker* em sua infraestrutura e podem ter um número dinâmico de clientes ou servidores conectados quando estão em execução. Existem na literatura diversas ferramentas destinadas ao monitoramento de performance, recursos, depuração, auxílio, manutenção de tais sistemas, etc. Porém, as ferramentas encontradas na literatura são restritas a linguagens ou plataformas que possam processar os agentes de monitoramento vinculados às mesmas. Neste trabalho, é proposta uma estrutura para a concepção de ferramentas para o monitoramento de sistemas de objetos distribuídos baseados em comunicações descritas em documentos XML, o que permite independência de linguagem ou plataforma na concepção de tais sistemas. Além da estrutura que é teoricamente fundamentada, a concepção de uma ferramenta nos moldes estruturados é descrita e provada no monitoramento de um estudo de caso.

Palavras-chave: Ambientes Virtuais Distribuídos, Aglomerados Gráficos, Jamp, RMI, LibGlass, Ferramentas de Monitoramento.

ABSTRACT

Immersive, interactive, and collaborative distributed applications, due to their real time and multi-platform characteristics, require a monitoring structure of their functionalities during the development period, which is not fully available in the current monitoring systems, which are usually specific of the languages and platforms in which they are made available. Monitoring systems that require synchronism of distributed objects during the processing is a complex task. Especially in distributed systems from multi-projection where timing is critical to its operation. It is often difficult to check the correct execution of the system. It might be useful for developers and users of such systems to have the ability of identifying, for example, if the reason for the system delay is due to the abusive use of a resource or due to programming mistakes; the identification of which object has crashed, compromising the system as a whole. Distributed object systems, also known as SOA (Service Oriented Architecture), generally use middleware of the broker type in their infrastructure and can have a dynamic number of connected clients or servers when running. There are, in the literature, several tools for the monitoring of performance, resources, debugging, and support, among others, of such systems. Nevertheless, the tools available in the literature are limited to programming languages or platforms that are able to process the monitoring agents related to them. In this work, a tool for the monitoring of distributed object systems is proposed, based on communication described in XML documents, which allows language and platform independence in the such systems development. A structure for the design of tools for the monitoring of distributed object systems is proposed, based on communication described in XML documents, which allows language and platform independence in the such systems development. In addition to the structure that is theoretically founded, the conception of a tool is structured in the manner described and proven in monitoring data from a case study.

Keywords: Distributed Virtual Environment, VRCluster, Jamp, RMI, LibGlass, Monitoring Tools.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

.NET - *Development Microsoft Framework.*
AIX, - *Advanced Interactive eXecutive, uma versão da IBM para o Unix.*
API - *Application Programming Interface.*
AV - *Ambiente Virtual.*
AVD - *Ambiente Virtual Distribuído.*
AVC - *Ambiente Virtual Colaborativo.*
C++ - *Linguagem de Programação.*
CCM - *CORBA Component Model.*
CMIP - *Common Management Information Protocol.*
COM - *Component Model Object.*
COM+ - *Component Model Object Plus.*
CORBA - *Common Object Request Broker Architecture.*
CPU - *Central Processing Unit.*
DCOM - *Distributable Component Model Object.*
DNA - *Distributed interNetwork Architecture.*
DOS, - *Disk Operating System.*
EJB - *Enterprise Java Beans.*
ESIOP - - Environment-Specific Inter-ORB Protocols.
GIOP - *General Inter-ORB Protocol.*
GNU - *GNU Not Unix.*
HPC - *High-performance computing.*
IBM - *International Business Machines, empresa multinacional com sede nos Estados Unidos na área de informática.*
IDL - *Interface Definition Language.*
IIOP - *Internet Inter-ORB Protocol.*
INTEL - *Empresa multinacional com sede nos Estados Unidos na área de informática.*
JAMP - *Java Architecture for Media Processing.*
JAVA - *Linguagem de Programação.*
JDK - Java Development Kit.
JNI - *Java Native Interface.*
JVM - *Java Virtual Machine.*
LIBGLASS - *Arcabouço com rotinas escritas em C, para o desenvolvimento de aplicativos para aglomerados gráficos.*
LINUX - *Operating System.*
LRPC - *Lightweight Remote Procedure Call.*
MAC - *Media Access Control address.*
MIDDLEWARE - *Mediador.*
MIDL - *Microsoft Interface Definition Language.*
MIMD - *Multiple Instruction Multiple Data.*
MPI - *Message Passing Interface.*
MPICH - *Implementação livre das especificações MPI.*
OID - *Object Identifier.*
OLE - *Object Linking Embedding.*
OMG - *Object Management Group.*

ORACLE - *Empresa multinacional com sede nos Estados Unidos na área de informática.*
ORB - *Object Request Brokers.*
RMI - *Remote Method Invocation.*
RTEP, - *Real Time Event Protocol.*
SHA - *Secure Hash Algorithm.*
SNMP - *Simple Network Management Protocol.*
SPMD - *Single Program Multiple Data.*
SQL - *Structured Query Language.*
UDP - *Universal Datagram Protocol.*
UNIX - *Operating System.*
WINDOWS - *Operating System.*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

O avanço tecnológico de hardware e software tem possibilitado o desenvolvimento de aplicações voltadas ao entretenimento, saúde, negócios educação e treinamento de pessoas em diversas áreas. Neste escopo de aplicações, algumas pesquisas utilizam recursos de Realidade Virtual (RV) com a finalidade de incrementar tais práticas, aproximando-as da experiência real.

Pode-se dizer que a RV é um ambiente composto por recursos que promovem ao usuário interatividade, desencadeando assim a sensação de estar presente em um mundo virtual. Para que isso seja possível, o ambiente pode ser equipado com dispositivos sofisticados que melhoram a interação, navegação e imersão por parte dos usuários [BOTEGA; CRUVINEL, 2009].

O termo Realidade Virtual é bastante abrangente, dessa forma, acadêmicos, desenvolvedores de software e pesquisadores procuram defini-lo baseados em suas próprias experiências, gerando assim diversas definições [NETTO et. al, 2002].

Corrêa [CORRÊA, 2010] cita diferentes classificações de sistemas RV concebidos desde os primordiais impulsionados pelas indústrias de simuladores de vôo e cinematográficas em 1962 até os citados em 2010. Entre os tipos de sistemas relacionados estão: RV de Simulação, RV de Projeção, Realidade Aumentada, Tele-presença e RV de Mesa.

Em geral RV, refere-se a uma experiência imersiva (sensação de estar no ambiente) e interativa baseada em imagens gráficas 3D geradas em tempo real por computador, ou seja, é uma simulação gerada por computador, do mundo real ou apenas de um mundo imaginário [BRAGA, 2001].

Apesar da capacidade de imersão ser característica de sistemas RV existem também os sistemas não imersivos onde se têm a sensação parcial de presença no

ambiente. Nesse tipo de sistema geralmente estão disponíveis apenas dispositivos convencionais, tais como monitor, teclado e mouse. Assim o usuário pode interagir com o mundo virtual. Porém, se o olhar deste usuário desviar além dos limites da tela do monitor o mesmo perderá o contato com o mundo virtual.

Já nos sistemas imersivos a sensação de pertencer ao ambiente é potencializada pelo uso de dispositivos não convencionais, tais como: luvas, capacetes, vestimentas dotadas de sensores, detectores de gestos, projetores de retina, óculos especiais para a captura de imagens 3D e CAVE's.

Para este trabalho destaca-se o dispositivo não convencional CAVE. Tal como descrevem Netto [NETTO et. al, 2002], Corrêa [CORREA, 2010] e Dias [DIAS et al., 2010]; o conceito de CAVE (ou "Caverna Digital" em português) consiste de uma sala com grandes telas de projeção, através das quais o usuário tem a sensação de estar dentro do ambiente. Nesta sala, as telas podem cobrir paredes, teto e chão, possibilitando imersão do usuário como um todo no mundo virtual projetado.

Geralmente a CAVE possui um aglomerado gráfico que controla seu funcionamento. Um aglomerado gráfico é um conjunto de computadores dotados de placas gráficas e projetores, que são configurados de maneira customizada a somar seus recursos e prover a ilusão de um único computador com grande capacidade de processamento [GUIMARÃES, 2004].

O aglomerado gráfico paraleliza o processamento das imagens de maneira que cada computador torna-se responsável pelo processamento e projeção de uma parte da imagem. Essa técnica permite a divisão de processamento entre os nós do aglomerado e possibilita melhor qualidade na imagem final. Esta técnica conhecida como multiprojeção [DIAS, 2010].

Kirner e Tori [KIRNER; TORI, 2006], descrevem possibilidades de aumentar a capacidade de imersão na CAVE propiciando em sua infraestrutura as possibilidades de se ter som espacial, geração de visão estereoscópica, comandos sob voz e recursos multisensorias como reação de tato e força, sensação de calor, frio, vento, etc.

Aplicações de RV imersivas e não imersivas ainda podem ser divididas em aplicação de uma única pessoa ou fazer parte de um sistema distribuído conectando diversos usuários ao mesmo ambiente virtual.

Quando um conjunto de elementos é modelado sob a tecnologia de RV simulando ambientes reais ou fictícios dispondo navegações passivas ou interativas, associa-se ao termo "Ambientes Virtuais" (AVs). Apesar da disponibilidade dos recursos tridimensionais da RV os AVs podem também ser concebidos sob tecnologias gráficas 2D presentes na área da Computação Gráfica [OSÓRIO et. al., 2004].

Ambientes Virtuais podem ser concebidos como: *(i)* sistemas mono usuário, onde um único processo controla o ambiente e as interações efetuadas pelo usuário ou; *(ii)* multi usuário, onde se destacam características de sistemas distribuídos. Na modalidade multi usuário os AVs são chamados "Ambientes Virtuais Distribuídos" (AVDs).

Ambientes Virtuais Distribuídos (AVDs) são caracterizados como Ambientes Virtuais (AVs) interativos interconectados em que os usuários remotos têm como objetivos a cooperação e o compartilhamento dos recursos computacionais em tempo real usando um suporte de rede de computadores para melhorar o desempenho coletivo por meio da troca de informações [BENFORD, 1994; ZYDA, 1999 apud RODRIGUES et al., 2006]

Os AVDs são também subdivididos em AVDs cliente e servidor e AVDs com mediadores *broker*s. Nos AVDs cliente e servidor, clientes compartilham um mesmo AV disponibilizado por um único servidor; já nos AVDs com mediadores *brokers* é permitida a replicação de servidores como prevenção de falhas e também a interconexão de diferentes servidores de AVs.

Estendendo os AVDs existem também os Ambientes Virtuais Colaborativos (AVCs) os quais podem possuir a mesma infraestrutura dos AVDs cliente e servidor ou AVDs com mediadores *brokers*. Apesar de mesma infraestrutura os AVCs possuem foco em disponibilizar os AVDs para trabalho cooperativo.

AVCs compartilham informações disponíveis no ambiente virtual de forma que qualquer um possa modificar suas propriedades refletindo essas modificações a todos os demais usuários em tempo real sob um mesmo ponto de vista.

A codificação de sistemas AVDs, incluindo AVCs, que permitem conexão entre diferentes dispositivos de imersão e plataformas de *hardware*, geralmente agregam em suas infraestruturas codificações escritas em diferentes linguagens de programação.

Com independência de linguagem de programação e características distribuídas, imersivas, interativas e colaborativas, sistemas AVDs e AVCs demandam uma estrutura de monitoramento de suas funcionalidades, ainda não plenamente disponíveis nos sistemas atuais de monitoramento, em geral específicos das linguagens e plataformas em que são disponibilizados.

Monitorar sistemas que necessitam de sincronismo no processamento distribuído é uma tarefa complexa. Muitas vezes, é difícil apurar o correto funcionamento dos mesmos.

Pode ser útil a desenvolvedores e usuários de tais sistemas a capacidade de se identificar, por exemplo, se a lentidão do sistema está sendo ocasionada por: uso abusivo de um recurso; por um erro na programação de um dos processos ou; a identificação de qual processo teve sua execução interrompida, comprometendo parcialmente o sistema ou até mesmo o sistema como um todo.

Para que seja feita uma analogia do uso de AVCs no mundo real, foi idealizada a situação fictícia em que dois laboratórios químicos situados em diferentes cidades simulam virtualmente a interligação de cadeias moleculares objetivando a concepção de novos medicamentos. Essas simulações são visualizadas em cavernas digitais de maneira a prover imersão e interação para os engenheiros químicos, além de melhor qualidade das imagens finais.

Supondo que os diferentes laboratórios desenvolvem um mesmo projeto de medicamentos, a modelagem virtual deve ter a mesma visualização refletida em ambas as cavernas digitais. Dessa maneira, as modificações efetuadas pelos químicos de um laboratório podem ser visualizadas pelos químicos do outro laboratório e vice-versa.

Aumentando as possíveis localizações dos projetistas conectados à formulação do mesmo medicamento. Engenheiros químicos podem também, acompanhar e sugerir modificações através de seus computadores pessoais em suas casas, através de dispositivos móveis quando em trânsito ou por conveniência através da tecnologia de TV Digital.

Esse tipo de aplicativo só seria funcional se existisse visualização distribuída e colaborativa das modificações do modelo virtual entre os diferentes dispositivos que o acessam. Em outras palavras, todas as modificações efetuadas por qualquer um dos dispositivos devem ser refletidas e visualizadas por todos os demais.

Além da característica de colaboração e visualização distribuída, aplicativos com tais especificações teriam considerável segurança se o servidor que mantém a estrutura pudesse ser replicado em rede local ou remota. Assim, a perda de um servidor não colocaria em risco o projeto como um todo.

Também seria interessante se as cadeias químicas virtuais envolvidas pudessem ser divididas entre diferentes servidores no caso de cadeias representando diferentes medicamentos.

Um sistema como o de laboratórios químicos idealizado, pode ser imaginado como um sistema de AVC sob infraestrutura de AVD com mediador *broker*.

A complexidade de se conceber um sistema como o do cenário fictício de cadeias químicas indica a necessidade de uma ferramenta que permita: (*i*) acompanhar dados simples para fins de depuração; (*ii*) acompanhar dados como disponibilidade e utilização dos recursos computacionais envolvidos; (*iii*) informações sobre como acessar ou prover aplicativos com tais especificações e; (*iv*) permitir também identificar dinamicamente quais dispositivos participam de um ambiente em tempo real.

Existem na literatura diversas ferramentas conhecidas como ferramentas de monitoramento. Estas, permitem acompanhar dados sobre a utilização de recursos computacionais, detalhes sobre comunicação e depuração de processos de diferentes tipos de arquiteturas. Como por exemplo: PCs, multiprocessadores, multicomputadores (como no caso de aglomerados gráficos), dispositivos móveis, TV Digital, etc.

Porém, existe carência de uma ferramenta de monitoramento que permita acompanhar a comunicação dos dispositivos e a utilização dos recursos de forma integral, independente de plataforma ou linguagem de programação.

A ferramenta da qual foi identificada esta carência deveria representar informações de servidores e clientes remotamente distribuídos na internet, tendo foco na concepção de AVDs. Esses, executados sob qualquer arquitetura dentre os dispositivos do cenário proposto.

O projeto deste trabalho é o de fundamentar, implementar e demonstrar com um estudo de caso, uma ferramenta de monitoramento (JMonitor) capaz de suprir a carência indicada, tendo foco em aplicativos de AVDs tal como aplicativos AVCs.

A Figura 1.1 esboça como seria a comunicação entre os dispositivos e servidores do cenário fictício de cadeias químicas apresentado. Nessa figura, é

possível observar diferentes redes de computadores disponibilizando clientes, servidores de AVs ou replicas de servidores de AVs já existentes.

Continuando com a visualização da Figura 1.1, é possível observar: cavernas digitais, arquiteturas PC, dispositivos de computação móvel, e tecnologias de TV Digital se comunicando com os AVs disponíveis. Esse cenário de forma geral reflete como seria a comunicação de um AVC sob a infraestrutura AVD com mediador *broker*.

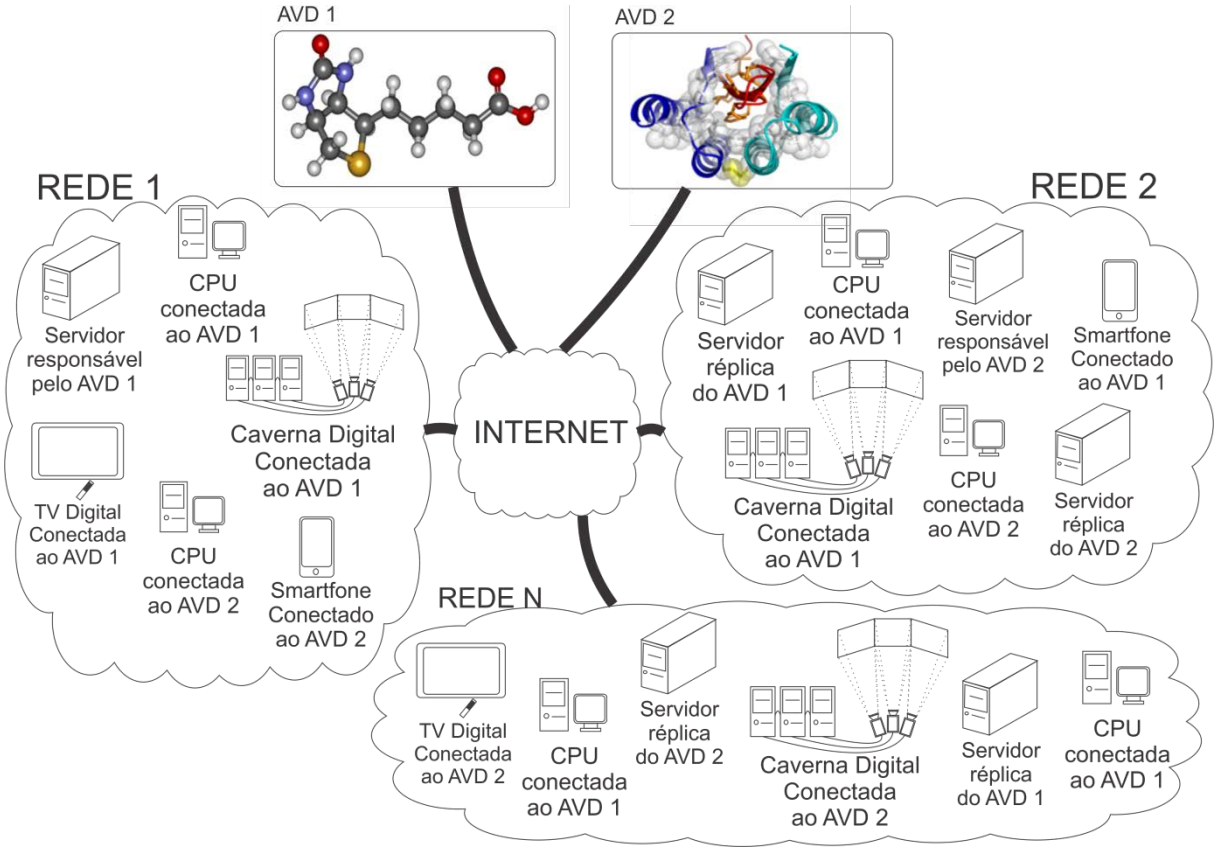

Figura 1.1 - Cenário de comunicação para AVCs com suporte a clientes sob diferentes tipos de plataformas. Cenário ideal.

Conceber AVCs que suportem diferentes tipos de plataformas, tendo em mente toda a sincronização e transporte de dados em tempo real, reflexão de interações entre participantes e AVDs, agregando tecnologias PC, aglomerados gráficos, TV digital, dispositivos móveis e toda a heterogeneidade de redes e dispositivos que essas tecnologias demandam, não é uma tarefa trivial.

Seria de grande benefício se existisse um arcabouço que abstraísse toda a tecnologia de baixo nível envolvida, permitindo que o programador se concentre apenas na criação do AVD ou do AVC.

Esse arcabouço proposto foi idealizado no projeto XPTA.lab [XPTA, 2011], incluindo documentação, aplicativos como casos de uso, documentação e suporte pelas empresas e laboratórios consorciados.

O arcabouço do projeto XPTA.lab foi idealizado a partir de uma junção de três tecnologias: (*i*) o arcabouço desenvolvido sob código aberto denominado LibGlass [LIBGLASS, 2011; GUIMARÃES, 2004]; (*ii*) a plataforma de desenvolvimento de *software* distribuído concebida por alunos da UFSCar (Universidade Federal de São Carlos), denominada JAMP [BAPTISTA, 2004; BARBOSA, 2006; CORRÊA, 2010] e; (*iii*) diretivas de programação para a computação móvel e TV digital concebidas pelo Lince (Laboratório Inovação em Computação e Engenharia), também situado na UFSCar.

O arcabouço LibGlass contém em sua estrutura bibliotecas de funções que permitem sincronizar variáveis de processos que executam em diferentes computadores ligados por uma rede de comunicação [GUIMARÃES, 2004].

Diante do arcabouço XPTA.Lab, LibGlass assume o papel de recurso para a sincronização dos processos que executam nos diferentes nós dos aglomerados gráficos. Estes processos geram as multiprojeções das cavernas digitais.

No projeto XPTA.lab foram desenvolvidos diversos AVDs como estudos de caso. Estes AVDs foram modelados através das bibliotecas gráficas OpenGL ou da XJ3D, ambas tecnologias destinadas à criação de modelos tridimensionais.

Com o correto alinhamento de câmeras entre os diferentes processos do aglomerado gráfico para uma multiprojeção panorâmica, pode-se utilizar as funções providas pelo arcabouço LibGlass para sincronizar valores que controlam a movimentação sobre o modelo tridimensional.

O processamento de um aglomerado gráfico diante de uma caverna digital necessita de recursos computacionais para a sincronia das multiprojeções. Estas, sem sincronia descaracterizam o objetivo da caverna digital, que é o de prover maior sensação de imersão por parte do espectador.

Para que fosse possível a visualização e interação sob o modelo através de TV digital ou computação móvel, os desenvolvedores dos estudos de caso do projeto XPTA.Lab utilizaram as diretivas de programação criadas pelo laboratório Lince.

A plataforma JAMP, por sua vez, uniu a esse ambiente uma série de arcabouços e servidores disponíveis em sua estrutura, destinados a facilitar a concepção e manutenção de sistemas de objetos distribuídos.

Sistemas de objetos distribuídos foram originados por meio da junção da orientação a objetos com os sistemas distribuídos. A base para esse modelo é o conceito fundamental de objetos, que podem ser clientes, servidores ou ambos. Objetos definidos como entidades com comportamentos específicos e atributos configuráveis, podem ser combinados de maneira simples provendo a capacidade de executar serviços customizados [CORRÊA, 2010].

Mediadores do tipo *broker* podem ser vistos como componentes essenciais aos sistemas de objetos distribuídos. Um *broker* tem a função de gerenciar a comunicação entre os diferentes objetos distribuídos que compõem um ou mais sistemas.

Através do *broker*, objetos distribuídos podem realizar o registro de suas interfaces de comunicação disponibilizando um modo de localiza-los. Objetos podem também buscar por interfaces já registradas no *broker* e, através dessas interfaces buscadas, o consumo dos serviços providos por objetos remotos pode ser efetuado.

Quando a arquitetura de sistemas distribuídos utiliza objetos provedores e objetos consumidores de serviços, é possível afirmar que estes sistemas estão estruturados sobre o modelo SOA (Service Oriented Archtecture).

Existem várias tecnologias para a concepção de sistemas SOA como: CORBA, RMI, JINI, Web Services, entre outras [AL BELUSHI, BAGHDADI, 2007][RAFE et al., 2009].

Na Figura 1.2 a plataforma JAMP é contextualizada entre outras plataformas que disponibilizam mediadores do tipo *broker*. Na mesma Figura é possível identificar que na JAMP o mediador do tipo *broker* é denominado JBroker.

O JBroker estende a tecnologia Java RMI. Esta extensão provê mecanismos de localização e distribuição de objetos. Isto permite o registro dos provedores de serviços e a invocação destes serviços pelos objetos consumidores.

Pode ser visualizado também na Figura 1.2, destaque para o serviço JMonitor. Esse destaque se deve ao fato deste serviço ser fundamentado, implementado e demonstrado através de um estudo de caso neste trabalho.

JMonitor foi projetado para prestar serviços como ferramenta de monitoramento possibilitando o acompanhamento de eventos de usuários dos AVDs

que desencadeiam alterações no sistema como um todo. Dessa forma pode-se identificar falhas de sincronismo entre os processos que podem ser escritos com independência de linguagem de programação.

Também estão disponíveis em JMonitor, meios de acompanhar a utilização dos recursos computacionais envolvidos no ambiente, além de ser possível dissipar pela rede informações gerais sobre os AVDs utilizando a ferramenta.

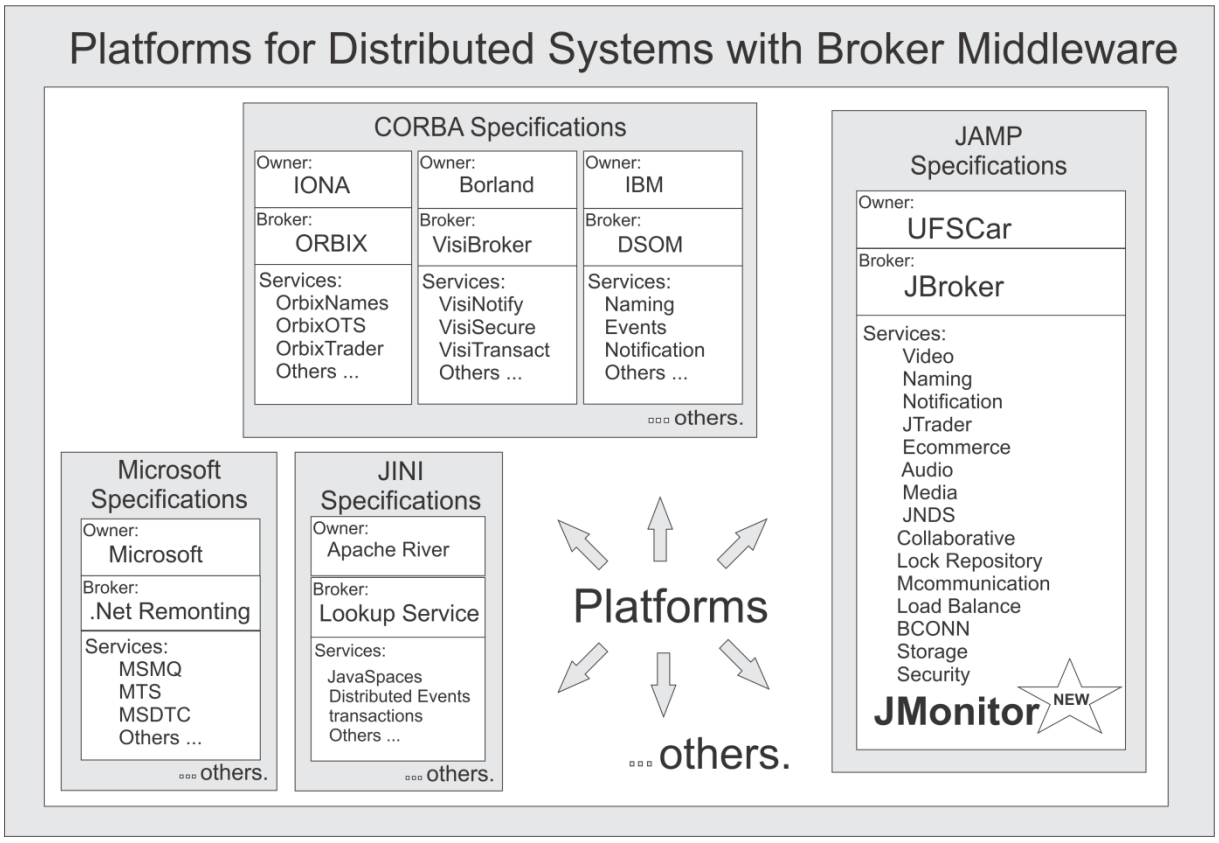

Figura 1.2. Comparação entre plataformas para sistemas distribuídos

1.1 Motivação

A possibilidade do desenvolvimento de ferramentas de apoio, que facilitem a concepção de ambientes virtuais distribuídos provendo meios de diminuir as dificuldades envolvidas neste processo, motivam este trabalho.

Disponibilizar uma ferramenta que monitore recursos computacionais e dados para depuração envolvidos na troca de informações entre servidores, arquiteturas PC e aglomerados gráficos, podendo estes dados estar geograficamente distantes

uns dos outros de forma dinâmica conforme o acesso aos servidores aumente ou diminua, são incentivos a proposta.

Por meio da ferramenta de monitoramento pretende-se dispor também recursos onde se possa verificar como estão as taxas de memória, processamento e utilização de rede pelos clientes e servidores que participam do AVD.

Com a ferramenta de monitoramento sendo disponibilizada junto à plataforma JAMP qualquer dispositivo que possa se conectar ao JBroker poderá também ter acesso às informações disponibilizadas pela mesma.

A ferramenta de monitoramento será destinada a facilitar a localização, conexão, concepção e disseminação de informações sobre aplicativos AVDs. Sendo estas, acessíveis com independência de plataforma e linguagem de programação.

1.2 Objetivos

(i) Idealizar um modelo genérico de ferramenta de monitoramento para SDs tais como sistemas AVDs sendo esses independente de plataforma e linguagem de programação.

(ii) Prototipar uma ferramenta seguindo os moldes propostos.

(iii) Testar a ferramenta em um Estudo de Caso.

1.3 Limitações

Neste trabalho não serão abordados:

(i) Detalhes sobre tecnologias para a concepção de modelos tridimensionais, tais como XJ3D, OpenGL ou similares.

(ii) Diretivas de programação para a computação móvel ou TV digital as quais existem no projeto XPTA.lab, porém, o objetivo deste trabalho não possui foco nesses dispositivos.

(iv) Monitoramento de processos que executam sob arquiteturas móveis ou TV digital.

1.4 Metodologia de Desenvolvimento do Trabalho

Para alcançar os objetivos, as principais tarefas realizadas são:

(i) Levantamento bibliográfico, para analisar modelos de ferramentas de monitoramento já existentes, direcionadas a arquiteturas distribuídas e/ou paralelas.

(ii) Estudar em detalhes como funcionam as tecnologias envolvidas no projeto XPTA.lab, com foco em aglomerados gráficos, multiprojeção e distribuição das informações pela rede, tais como: JAMP e LibGlass.

(iii) Estudar tecnologias para programação de agentes de rede, os quais permitem a coleta de dados em sistemas distribuídos, disponibilizando estatísticas sobre recursos monitorados.

(iv) Fundamentar como poderia ser estruturada uma ferramenta de monitoramento com as características desejadas.

(v) Implementar um protótipo da ferramenta de monitoramento fundamentada.

(vi) Implementar, testar e apresentar um estudo de caso de AVD que faça utilização da ferramenta protótipo implementada.

1.5 Organização do Trabalho

No Capítulo 2, serão introduzidas técnicas para a coleta, transformação e visualização de dados. Essas técnicas, são essenciais para a concepção de ferramentas de monitoramento de dados.

No Capítulo 3, é efetuado um embasamento teórico. Um introdutório sobre sistemas distribuídos, arcabouços e mediadores. Esses conceitos e tecnologias estão vinculados a infraestrutura dos AVDs.

O capítulo 4 estende o embasamento teórico efetuado no Capítulo 3, abordando tecnologias de sistemas paralelos e aglomerados gráficos. Essas tecnologias podem ser utilizadas na conexão à AVDs por meio de aglomerados gráficos e cavernas digitais.

No Capítulo 5, são descritas as diferentes estruturas de AVDs que existem e como uma ferramenta de monitoramento com independência de linguagem poderia ser auxiliar na sincronização dos dados.

O Capítulo 6, descreve detalhes sobre a estrutura de uma ferramenta de monitoramento para sistemas AVDs .

O Capitulo 7, descreve a implementação da ferramenta JMonitor. Uma implementação da ferramenta de monitoramento detalhada no Capítulo 6.

O Capítulo 8, provê detalhes de como um AVD de estudo de caso foi concebido e como o mesmo foi interligado com a ferramenta JMonitor. Provando a funcionalidade da ferramenta implementada.

No Capítulo 9, são apresentadas as conclusões do trabalho efetuado, as contribuição providas pelo mesmo e ideias para trabalhos futuros identificados a partir deste.

No final do trabalho são apresentas as principais referencias que fundamentaram o mesmo.

Capítulo 2

MONITORAMENTO DE SISTEMAS COMPUTACIONAIS

A concepção da ferramenta possui foco no monitoramento de dados com fins de depuração, monitoramento de recursos computacionais e informações sobre conexão e disponibilização de AVDs sobre as especificações do projeto XPTA.lab.

A técnica para acompanhar o monitoramento das informações, que é utilizada neste trabalho é uma adaptação da técnica proposta por Foster [FOSTER, 1995]. Alega-se ser uma adaptação pois a técnica de Foster é destinada a ferramentas de monitoramento que utilizam interceptação e instrumentações de código mas não descreve o monitoramento de recursos computacionais.

A instrumentação e interceptação de código da técnica de Foster está vinculada à inserção e coleta de símbolos de depuração diante dos processos a serem monitorados. Isso pode também ser utilizado com a ferramenta a ser fundamentada porém, os resultados da coleta desses dados precisam ser descritos nos documentos aos quais a ferramenta está preparada para processar.

Apesar das adaptações efetuadas ainda existe uma divisão de fases nas técnicas de Foster que descrevem como pode ser estruturada uma ferramenta de monitoramento. Esta divisão de fases não foi alterada e pode-se descrevê-la em três passos básicos:

- **Coleta de dados**: é o meio pelo qual informações são extraídas dos processos ou dos sistemas a serem monitorados. Esta pode ser implementada através de técnicas de instrumentação; pode ser codificada junto ao processo a ser monitorado ou; pode também ser efetuada por

meios periódicos de consultas através de comandos do sistema operacional.

Os dados obtidos são utilizados pelos desenvolvedores para realizar validações sobre os processos e sistemas monitorados, garantindo assim que os mesmos não estejam executando sob mau funcionamento ou de forma inesperada.

- **Transformação dos dados**: é aplicada com o objetivo de reduzir o volume total de dados. Pode ser usada para determinar valores médios ou outras estatísticas. Compiladores e ferramentas que permitem a instrumentação de código fonte tendem a obter uma quantidade excessiva de detalhes sobre a execução de um processo monitorado. Na fase de transformação os dados coletados podem ser transformados em métricas sob o interesse do desenvolvedor [PIOLA, 2007].

- **Visualização dos dados**: é utilizada para produzir imagens para auxiliar na compreensão dos resultados pelo programador ou usuário dos sistemas.

2.1 Coleta de Dados

Os dados computacionais coletados são geralmente armazenados em arquivos durante ou após a execução do processo monitorado sendo possível também apresentação em tempo real.

A coleta de dados pode ser efetuada de três formas:

- **Estática**: gera arquivos de registro contendo os dados computacionais coletados após finalizar a execução do processo monitorado.

  No caso de haver necessidade de modificações diante da coleta de dados esta deve ser recodificada, o que exige recompilação do código fonte do processo a ser monitorado.

  Um exemplo de sistema que faz coleta de dados estática é o gprof [FENLASON, 1997]. Mais exemplos podem ser encontrados no trabalho de Piola [PIOLA, 2007] ou em Boccardo [BOCCARDO et al., 2005].

- **Dinâmica**: processa informações derivadas da execução do processo em tempo real, permitindo customizações sobre a coleta de dados. Como por exemplo, permite modificações referentes a quais partes do código devem ser monitoradas sem que o processo tenha que ser recompilado para isso.
  Exemplos de sistemas que fazem coleta de dados dinâmica são: DTrace [CANTRILL; SHAPIRO, 2004] e Paradyn [MILLER; CALLAGHAN, 1995].
- **Por meio de agentes**: coleta de dados geralmente utilizada por sistemas de monitoramento destinados a gerencia de redes. Agentes são estruturados para realizarem coletas de dados customizadas em processos, sistemas ou dispositivos computacionais; geralmente armazenam os dados coletados em bases locais disponibilizando esses dados a processos gerenciadores conforme requisição dos mesmos.
  Agentes também podem enviar notificações a processos gerenciadores quando detectam que determinada situação pré-conFigurada ocorreu.
  Exemplos de sistemas que utilizam a coleta de dados por meio de agentes são: Nagios [KOCJAN, 2008], Zenoss, [BADGER, 2008], Zabbix [OLUPS, 2010] e Cacti [URBAN, 2011].

Os níveis mais comuns em que a coleta de dados pode ser efetuada são:

- **Nível de Hardware**: através dessa abordagem um dispositivo dedicado coleta dados de eventos registrando sinais presentes no barramento de dados do sistema alvo.
  O dispositivo pode ser incorporado à arquitetura sendo monitorada ou simplesmente conectado ao barramento de dados disponível. Este tipo de dispositivo pode monitorar dados de eventos, inclusive registrando temporizações entre diferentes coletas de dados efetuadas.
  Possui a vantagem de não ser intrusivo, por não compartilhar recursos de sistema junto com os processos a serem monitorados [WILD, 2000].
  Piola [PIOLA, 2007], enaltece algumas desvantagens as quais são: (*i*) necessidade hardware adicional o que o torna mais caro o monitoramento; (*ii*) fornece dados de nível muito baixo; (*iii*) não satisfaz exigências de monitoramento quanto a programação em ambientes paralelos e; (*iv*) forma a classe menos portável de mecanismos de monitoramento.

- **Nível de Software**: checagens de dados a serem coletados podem ser inseridas pelo programador ou pelo compilador diretamente no código do processo a ser monitorado [DUARTE, 2003].

  A implementação das checagens via software deve ser cautelosa pois estas geram sobrecarga nos processos a serem monitorados. O fato das checagens usarem os mesmos recursos que estão medindo pode afetar o monitoramento resultando dados errôneos e monitoramento inadequado.

  Existem várias abordagens para a coleta de dados por software, por exemplo:

  - **Checagens no Código Fonte**: instruções disponibilizadas no núcleo das linguagens de programação ou por meio de extensões das mesmas, permitem inserção de rotinas para aquisição de dados sob o código fonte do programa de forma manual (pelo programador) ou automática (pelo compilador). Piola [PIOLA, 2007], destaca como desvantagem deste método que modificações dos pontos a serem monitorados necessitam de recompilação dos processos.
  - **Checagens em Rotinas de Bibliotecas**: complementando o método anterior, instruções podem ser inseridas no código fonte e compiladas para bibliotecas nativas. Como vantagem desse método se tem o fato que bibliotecas nativas podem acessar diferentes recursos do sistema operacional. Porém, a desvantagem detectada no método anterior se mantém neste caso.
  - **Checagens via *Kernel***: permite interceptar e registrar as chamadas de sistema efetuadas pelo processo alvo bem como os sinais que são recebidos pelo mesmo. Podendo disponibilizar aos desenvolvedores de sistemas detalhes como: nome e argumentos das chamadas de sistema invocadas, tempo decorrido, além de registros no caso de erros sobre as invocações [KRETSCHEK, 2002].

    Método similar à técnica de checagens em rotinas de bibliotecas, tem a vantagem de fazer a detecção do evento transparente para o programa. Porém, somente eventos relacionados a chamadas do *kernel* podem ser detectados e não eventos da aplicação.

- **Checagens de Código Objeto:** usam alteração de *bytecode* para coletar dados obtidos em tempo de compilação, onde *bytecodes* são definidos como uma representação intermediária do programa.
  Piola [PIOLA, 2007] destaca que checagens introduzidas em *bytecodes* ao invés do código fonte tem a vantagem de serem independentes de plataforma e pode ser transparente ao programador. Isto resulta em menor sobrecarga pois o baixo nível das instruções de máquina pode ser usado ao invés do alto nível de comandos de código fonte que exigem compilação.
  A desvantagem deste método é que ele requer um compilador que permita inserir as checagens nos *bytecodes*.

- **Coleta de Dados Híbrida**: comporta-se como uma mistura entre os níveis de *hardware* e de *software*.
  Em sua porção a nível de *hardware* depende de dispositivo dedicado para detectar e processar eventos de um barramento. Por outro lado, partilha também da coleta de dados por *software* permitindo mudanças nos eventos que devem ser monitorados.
  Esse tipo de coleta de dados foi projetada para se beneficiar das vantagens de ambos os níveis. Causa menos sobrecarga se comparada a coleta a nível de *software* e possui menor custo comparado a coleta a nível de *hardware*, pois, compartilha recursos com o sistema a ser monitorado e necessita de menos código inserido sob o processo ou sistema alvo.
  A coleta de dados híbrida possui desvantagem na portabilidade comparada com a coleta efetuada somente a nível de *software* por causa do uso de *hardware* dedicado.

Uma ferramenta que monitora performance de *software* pode identificar, por exemplo: (*i*) os módulos e instruções mais executados; (*ii*) identificar quais módulos do *software* alocam mais memória ou; (*iii*) quais módulos geram mais requisições de entrada e saída. Ferramentas de performance de *hardware* podem ser construídas dentro do próprio sistema ou adicionadas mais tarde.

2.2 Transformação dos Dados

As ferramentas de monitoramento podem em alguns casos obter grande quantidade de dados que correspondem a uma observação detalhada da operação do sistema. Geralmente essa massa de dados especifica intervalos de tempo, contadores de eventos, porcentagens e, por fim, tem uma relação com funções lógicas do sistema. Para ser utilizável, essa massa de dados precisa ser analisada e transformada em informação significativa. Algumas métricas comumente usadas são:

- **largura de banda:** é a taxa de transmissão de dados na qual um enlace pode propagar a informação. A largura de banda nominal é dada em bits por segundo. Essa ainda recebe algumas sub-divisões:
  - **largura de banda de contenção:** é a taxa máxima que uma rede pode transmitir dados de um transmissor para um receptor ou seja, a taxa máxima de transmissão é equivalente ao enlace com a menor largura de banda no caminho entre o transmissor e o receptor, esta métrica não é afetada pelo aumento ou a diminuição do tráfego.
  - **largura de banda utilizada:** representa a largura de banda utilizada em um enlace ou seja, a quantidade de dados trafegando por um enlace em um determinado momento.
  - **largura de banda disponível:** é a taxa máxima na qual um host consegue transmitir dados ao longo de um caminho da rede em um certo momento, esta taxa considera o tráfego existente no momento da medição. Em um caminho de rede a largura de banda disponível é determinada pelo enlace com a menor largura de banda não utilizada.
- **atraso:** corresponde ao tempo de transmissão de um host de uma rede a um outro host da mesma rede ou fora dela, geralmente é medido em milissegundos. A métrica atraso também possui algumas subclassificações:
  - **atraso de ida e volta:** é o tempo necessário para transmitir um pacote a um host destino e retransmiti-lo à origem, chamado de *Round-Trip-Time* (RTT). Alguns tipos de comunicação exigem um pacote de retorna a origem que confirme o recebimento de dados enviados ao destino.

  - **atraso em um sentido:** é o tempo necessário para um pacote atravessar de uma origem para um destino ou do destino para origem. Intuitivamente, o atraso em um sentido, deve ser a metade do atraso de ida e volta.

- **perda de pacotes:** é a porcentagem de pacotes corretamente recebidos pelo destino em relação ao total de pacotes enviados pela origem. O fator que mais influencia na perda de pacotes é a falta de espaço em *buffer* nos roteadores intermediários e pacotes corrompidos causados por erros de transmissão.

  Uma alta taxa na perda de pacotes pode ser prejudicial aos serviços da rede, pois, pode implicar em retransmissão dos pacotes perdidos e degradar drasticamente o desempenho principalmente em aplicações multimídia.
- **latência ou tempo de resposta:** é medida em unidades de tempo decorrido. Essa métrica deve especificar um evento de início e um evento de término para que se possa contabilizar o tempo total. Por exemplo:
  - o tempo decorrido entre a solicitação de acesso de um cliente a um sistema e a real conexão do mesmo no dispositivo cliente;
  - o tempo entre uma informação de atualização ser enviada de um servidor a seus clientes e esses realmente a receberem.

  Em outros casos, a latência pode ser relatada ou especificada como uma distribuição estatística. Por exemplo, pode ser exigido que um novo servidor a fazer parte de um AVD deve ter a capacidade mínima de refletir no mínimo 99,5% das requisições dos clientes. Desse modo sacrificando um pouco o sincronismo pela velocidade.
- **taxa de serviço (ou *throughput*):** é medida em unidades de tarefas executadas em um intervalo de tempo, e é inversamente proporcional ao tempo de resposta por tarefa. Por exemplo: (*i*) o número de requisições de atualização que podem ser completadas por minuto ou; (*ii*) um cenário novo sendo acessado pelo cliente no AVD - quanto tempo levará a transferência de dados que representa este cenário entre cliente e servidor.
- **taxa custo/desempenho:** é geralmente usada como uma métrica para comparar dois ou mais sistemas. O custo inclui licença de

hardware/software, instalação e manutenção sobre um período de tempo, geralmente meses ou anos.

- **Confiabilidade:** é uma medida do número de interrupções críticas durante o tempo em que um programa ou sistema está em funcionamento.
- **Disponibilidade:** é a porcentagem de tempo que um determinado sistema está ativo e trabalhando.

No trabalho de Piola [PIOLA, 2007], são relatadas também algumas técnicas para se reduzir a quantidade de dados após a coleta e a definição de métricas. Essas técnicas de redução podem ser atrativas para sistemas que geram grande quantidade de dados.

As técnicas relatadas por Piola propõem de forma geral, maneiras de se obter métricas avaliando pequenas porções dos dados coletados. Essa propriedade de "não necessariamente exigir tratamento dos dados em sua totalidade" aumenta a velocidade da aquisição de dados úteis aos desenvolvedores.

2.3 Visualização dos Dados

Após a etapa de coleta e transformação de dados, vem a etapa de visualização. Nesta etapa, os dados que foram coletados, tratados e armazenados geralmente em arquivos de registros ou bases de dados são agora apresentados aos desenvolvedores de sistemas.

Existem diferentes formas de visualização em que os dados podem ser dispostos o que também está relacionado ao tipo do dado coletado. Por exemplo:

- **visualização por processador:** tipo mais básico de exibição, demonstra geralmente por meio de gráficos a porcentagem do tempo de execução gasto por um dado processador durante a execução do sistema a ser monitorado;
- **visualização por comunicação:** a comunicação entre os processadores pode ser descrita logicamente (sem considerar a rede de interconexão) ou fisicamente (em termos do trajeto percorrido em uma rede específica). A exibição lógica de comunicação é independente de qualquer topologia de rede particular e

portanto, pode ser portável entre arquiteturas [PIOLA, 2007];

- **visualização por *threads*:** representação gráfica dos estados das threads ativas e informações relativas a medidas que envolvem a soma de bytes enviados e recebidos entre uma aplicação servidora e seus clientes ou o conjunto das aplicações inter relacionadas que constituem uma aplicação distribuída [BRUGNARA et al., 2005];
- **visualização com interatividade:** este tipo de visualização apresenta todos os fluxos de execução em atividade permitindo remotamente a ativação ou desativação dos mesmos [DUARTE, 2003].
- **visualização por custo de função:** dados representando rotinas executadas juntamente ao custo definido pelo tempo gasto em suas execuções podem ser visualizados graficamente [PARADYN, 2011];
- **visualização por árvore de chamadas:** neste modo de representação todo o histórico hierárquico entre invocações de diferentes funções pode ser visualizado textualmente ou graficamente. Este modo de visualização vem a completar o modo por comunicação onde somente os eventos que realizam comunicações entre processos podem ser visualizados [VTUNE, 2011; TAU, 20011];
- **visualização por uso de memória:** este modo de visualização, ajuda a detectar erros relacionados a alocação, acesso ou desempenho, dados relacionados a memória [VALGRIND, 2011].

Diversos tipos de visualizações textuais ou gráficas podem ser obtidas por ferramentas existentes na atualidade. A Figura 2.1 demonstra algumas ferramentas que geram gráficos em execução. Porém, nenhuma dessas ferramentas objetiva a criação de AVDs com comunicação multiplataforma e independente de linguagem de programação.

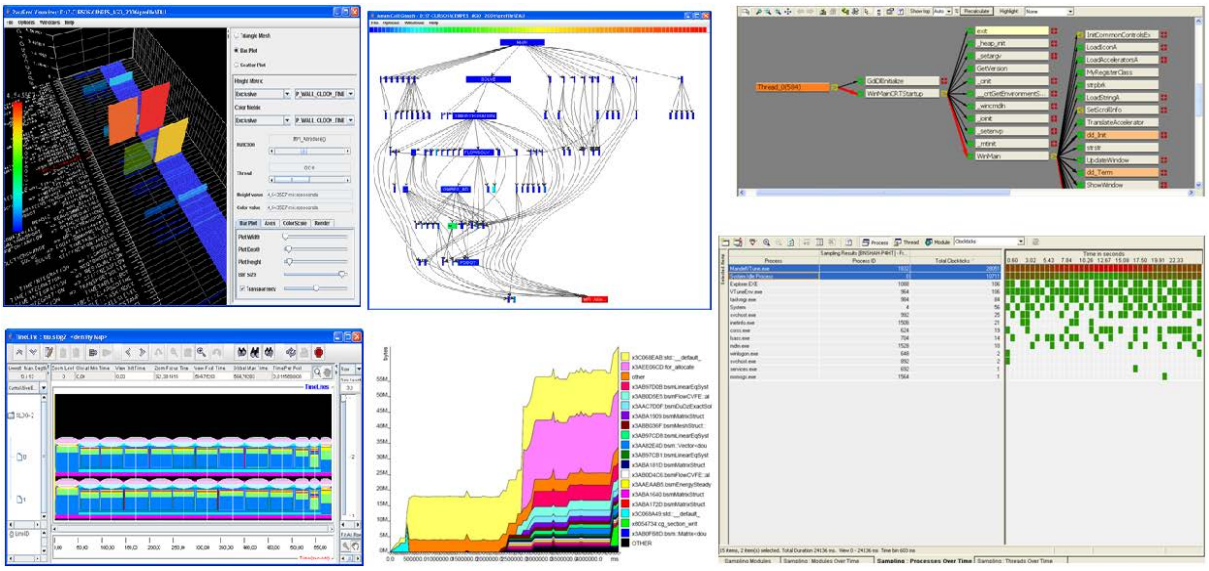

Figura 2.1- Exemplos de visualizações gráficas [VGIMG].

Como exemplos de problemas que podem ser reconhecidos através dos gráficos gerados na etapa de visualização podem ser citados:

- **Vazamento de memória:** fenômeno que ocorre em sistemas computacionais quando uma porção de memória alocada para uma determinada operação não é liberada quando não é mais necessária.

  A ocorrência de vazamentos de memória é quase sempre relacionada a erros de programação e pode levar a falhas no sistema se a memória for completamente consumida;
- **Condição de corrida:** se duas ou mais tarefas (*threads* ou processos) tentam alocar o mesmo recurso ou dado compartilhado ao mesmo tempo, pode ocorrer inconsistências [TANENBAUM; WOODHULL, 2008].
- **Gargalos de processamento:** trechos na execução de um programa que causam picos de processamento podem até mesmo parar o sistema por inteiro. A descoberta de gargalos de processamento podem indicar que o trecho processado deve receber otimizações em suas rotinas.
- ***heap dump*:** o *heap* diante de uma maquina virtual java, representa uma área reservada de memória onde os objetos java são criados. É no *heap* que o coletor de memória já liberada (Garbage Collector) realiza seu processamento.

Quando um objeto ou uma ilha de objetos existe no *heap* porém não são acessados por nenhum processo em execução, o coletor de memória os destrói liberando a memória alocada para outros processos.

Um *heap dump* representa uma limpeza forçada, a qual é utilizada quando o coletor de memória por algum motivo desconhecido não consegue limpar o *heap* saturando a memória da JVM [IBM, 2011].

Ferramentas de monitoramento que demonstram estatísticas sobre os *heaps* das JVMs envolvidas em um sistema podem evidenciar a necessidade de realização de *heap dump* ajudando a identificar erros de programação na criação e destruição de objetos em um sistema monitorado.

2.4 Trabalhos relacionados.

Existem vários projetos que investigam diferentes tipos de monitoramentos para os sistemas de objetos distribuídos. As técnicas e apresentações do projeto de Karunamoorthy [KARUNAMOORTHY; DEVINUWARA, 2005], o projeto PerfMoon [XU; XU, 2009], CENNI [PENG; CAO, 2010] e outros existentes são exemplos dessa categoria

No projeto de Karunamoorthy [KARUNAMOORTHY; DEVINUWARA, 2005] são relatadas técnicas para a aquisição de dados descritivos sobre os recursos disponibilizados aos sistemas de objetos distribuídos. Estas técnicas utilizam a plataforma JINI como infraestrutura. Reconhecem e disponibilizam os recursos computacionais utilizados onde os objetos distribuídos estão em execução, de forma centralizada. Porém, a customização dos monitoramentos não é permitida.

PerfMoon [XU; XU, 2009] por sua vez, utiliza em sua arquitetura de monitoramento módulos a nível de *kernel*. Estes devem ser habilitados nos sistemas a serem monitorados disponibilizando interfaces de comunicação RMI preparadas para receberem chamadas originadas em um *daemon* de monitoramento central. Entretanto, conceber novos módulos para o monitoramento de características customizadas são técnicas de alta complexidade.

Em CEMMI [PENG, 2010] a plataforma JMX e componentes MBeans são utilizados para prover um sistema de monitoramento distribuído baseado em regras, que podem ser customizadas de acordo com a necessidade do alvo a ser monitorado. No caso de CEMMI a utilização da plataforma JMX não permite o monitoramento de variáveis simples para aplicativos "não Java".

Na arquitetura da ferramenta de monitoramento abordada neste trabalho, quaisquer dados disponibilizados pelos sistemas de objetos distribuídos podem ser monitorados através de arquivos XML. Isto permite fácil customização, independência de linguagem e independência de arquitetura.

Capítulo 3

SISTEMAS DISTRIBUÍDOS MODELO CLIENTE E SERVIDOR, MEDIADORES E ARCABOUÇOS

Um sistema distribuído segundo Tanenbaum [TANENBAUM(a), 2007], pode ser definido como “uma coleção de dois ou mais computadores independentes que se apresentam aos seus usuários como um sistema único e consistente”.

Coulouris [COULOURIS et al., 2007], define sistema distribuído como ambiente onde “os componentes de hardware ou software, localizados em computadores, interligados em redes, se comunicam e coordenam suas ações apenas enviando mensagens entre si”.

Como a Figura 3.1 exemplifica, um servidor para uma solicitação de serviço pode se tornar um cliente para uma outra solicitação de serviço. Mensagens são trocadas de forma tipicamente interativa.

O modelo C/S não suporta processamento *off-line* dos pedidos. Mas existem exceções como, por exemplo, os sistemas com servidores de mensagem. Estes, permititem que os clientes armazenem mensagens em uma fila para serem processadas de forma asíncrona pelos servidores em um estágio posterior.

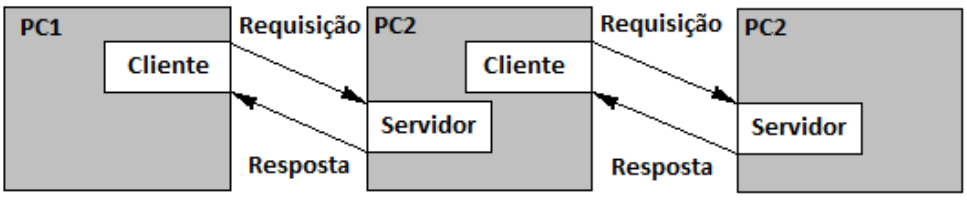

Figura 3.1 - Comunicação Cliente Servidor.

Os clientes e servidores normalmente residem em máquinas distintas, conectadas através de uma rede de trabalho. Conceitualmente, clientes e servidores também podem executar na mesma máquina.

3.1 Modelo lógico de camadas e arquiteturas cliente/servidor estrutural de duas camadas

Para facilitar a concepção de sistemas criados sobre a arquitetura C/S, com o passar dos anos foi se fortalecendo o conceito lógico de camadas, onde, atribuições funcionais que poderiam existir entre o cliente e o servidor diante de quaisquer tipos de sistemas de informação foram propostos.

Segundo Stock relatou em seu trabalho [STOCK, 2001], três modelos lógicos de camadas se estabeleceram: (*i*) camada de apresentação responsável pela interface do usuário da aplicação; (*ii*) camada de aplicação responsável pela funcionalidade da aplicação e; (*iii*) camada de dados responsável pelo gerenciamento dos dados da aplicação.

Cada uma das camadas é então atribuída a uma máquina específica ou um conjunto de máquinas. Além do modelo lógico de camadas, surgiram também os modelos estruturais. O modelo lógico de camadas é distribuído através dos modelos estruturais e essa divisão define se: (*i*) o sistema terá exigências quanto a servidores com grande capacidade de processamento; (*ii*) se o processamento das informações se concentrarão nos clientes ou; (*iii*) se existirão divisões das cargas a serem processadas entre servidores e clientes.

No trabalho de Stock [STOCK, 2001], foram definidas cinco arquiteturas base de divisões de camadas lógicas entre a arquitetura cliente/servidor sob a estrutura de duas camadas, Figura 3.2.

Apesar de um determinado aplicativo C/S poder ser projetado em qualquer uma das referidas conFigurações, dados remotos e apresentações distribuídas/remotas são as arquiteturas mais comuns.

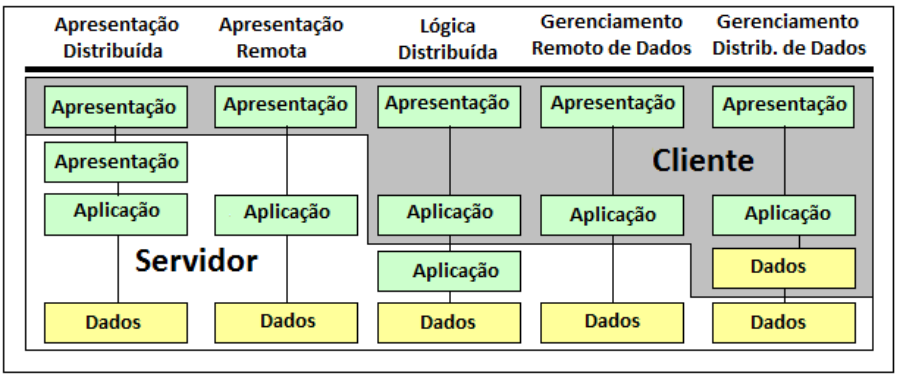

Figura 3.2 - Arquiteturas Cliente/Servidor de duas camadas.
Adaptado de Technologies for Thin Client Architectures [STOCK, 2001]

3.2 Modelo cliente/servidor estrutural de três ou multi camadas e mediadores

A arquitetura estrutural de três camadas surgiu para superar as limitações da arquitetura de duas camadas. Na arquitetura de três camadas uma camada intermediária foi adicionada entre o sistema cliente e o sistema servidor.

Essa camada adicionada é conhecida na literatura como mediador (*middleware*). Este disponibiliza uma API (*Application Programming Interface*) de fácil utilização localizada entre o cliente e os recursos que ele necessita de um ou mais servidores.

A principal função de um mediador é prover acesso transparente para sistemas e serviços sem exigir do programador ou desenvolvedor o conhecimento de como os sistemas são e onde eles estão.

Os mediadores objetivam habilitar a comunicação entre componentes distribuídos. Para tal, fornecem aos programadores abstrações de alto nível construídas usando primitivas de rede do sistemas operacional, que escondem a complexidade introduzida pela distribuição.

Tecnologias de mediadores existentes para sistemas distribuídos foram construídas com a metáfora de caixa preta, onde a gerência da distribuição tornou-se transparente ao usuário e ao projetista de sistemas [BLAIR, 98; EMMERICK 2000].

Quando a quantidade de mediadores e servidores disponibilizando recursos e serviços aumenta na rede a estrutura da aplicação passou então a ser conhecida como arquitetura multi camadas [ORFALI, 1998].

A Figura 3.3 ilustra como se comportam os componentes das arquiteturas multicamadas. Em uma arquitetura deste tipo quando um componente precisa acessar um recurso de um servidor o mediador necessário para o acesso a esse recurso é invocado.

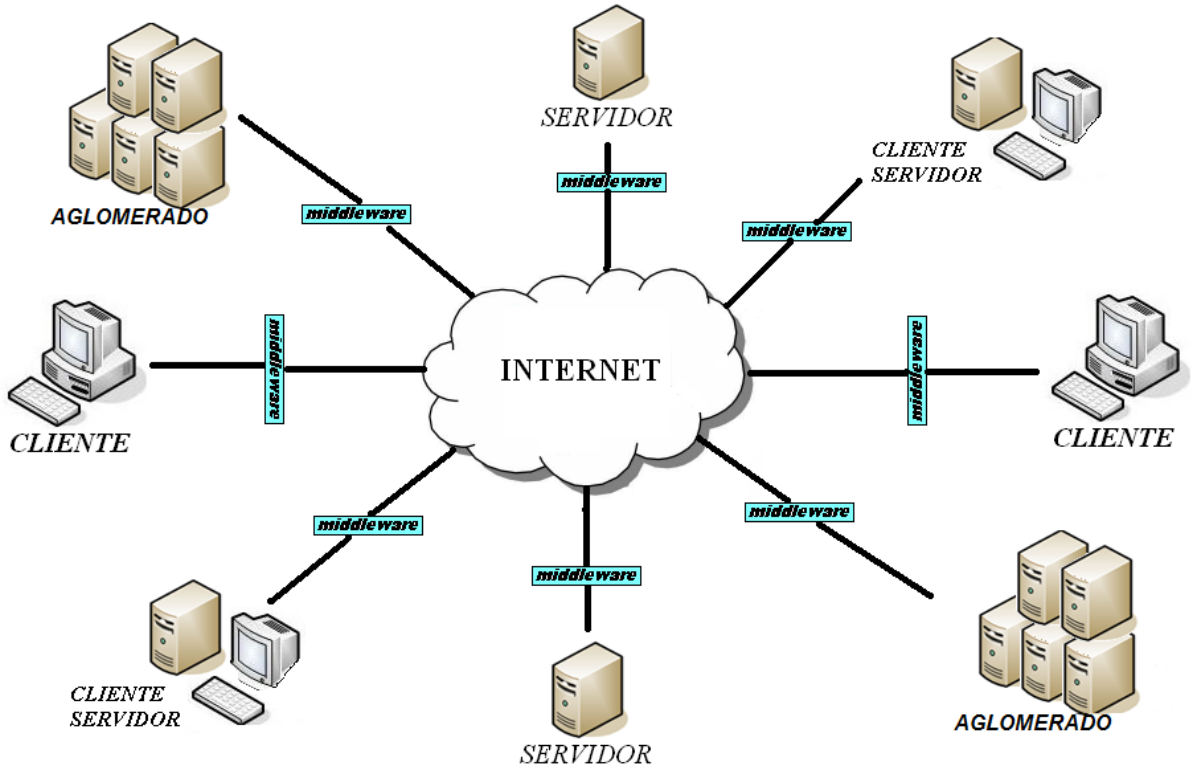

Figura 3.3 - Arquitetura cliente e servidor multicamadas.

Dentre os vários modelos de mediadores existentes, cinco foram estabelecidos como tradicionais [GEYER, 2004; SOUZA, 2001]: (*i*) mediadores RPCs (*Remote Procedure Calls*); (*ii*) mediadores conectores de bases de dados; (*iii*) mediadores orientados a mensagens; (*iv*) mediadores orientados a transação e; (*v*) mediadores de comunicação (*brokers*).

Além dos mediadores tradicionais existem também na literatura e no mercado tecnológico propostas e produtos com diferentes objetivos: computação móvel, grades computacionais, computação em nuvem, computação pervasiva, etc.

Porém, com um número cada vez maior de tipos de mediadores se estabelecendo no mercado e na literatura, a complexidade para concepção de arquiteturas cliente e servidor que utilizam estes mediadores também aumentou consideravelmente.

Para minimizar essa complexidade foram criadas ferramentas de apoio chamadas de arcabouços (*frameworks*).

3.3 Arcabouços

À medida que as aplicações de *software* se tornaram mais complexas, tecnologias e ferramentas de apoio que proporcionam a reutilização destas aplicações se tornaram cada vez mais necessárias.

Com a capacidade da orientação a objetos de definir comportamentos através de interfaces, classes abstratas e classes concretas previamente definidas, tornou-se possível empregar um comportamento genérico aos sistemas. Assim, esses sistemas por meio de extensões e implementações passaram a se comportar sempre de uma mesma forma, conforme esperado.

Segundo Buschmann [BUSCHMANN, 1996] e Pree [PREE, 1995] um arcabouço foi definido como um software parcialmente completo projetado para ser instanciado. O arcabouço define uma arquitetura para uma família de subsistemas e oferece os construtores básicos para criá-los.

Várias outras definições para arcabouços são encontradas na literatura, tal como nas publicações de Mattsson [MATTSON, 2000] ou Fayad [FAYAD; SCHMIDT, 1997]. Porém, apesar de serem diferentes não são contraditórias.

Em uma definição mais genérica, um arcabouço funciona como um molde para a construção de aplicações ou subsistemas dentro de um domínio de aplicações.

Existem vários tipos de arcabouços para a concepção de aplicativos sobre a arquitetura de objetos distribuídos. Em geral, arcabouços para este tipo de arquitetura utilizam ou disponibilizam mediadores de comunicação (*brokers*) em suas estruturas.

A seção seguinte é dedicada à arquitetura de objetos distribuídos. Neste trabalho, os conceitos da arquitetura de objetos distribuídos, junto aos conceitos de arcabouço, servirão de base para o entendimento de como funciona a plataforma JAMP e o arcabouço LibGlass.

3.4 Sistemas Distribuídos Modelo Objetos Distribuídos

A união da orientação a objetos com sistemas distribuídos originou a área de objetos distribuídos. A base para esse modelo é o conceito fundamental de objetos, que são definidos como entidades que possuem comportamentos específicos e atributos, fornecendo uma maneira simples de combiná-los. Os atributos permitem que se recuperem ou definam dados para um objeto, que podem conter operações e invocações. Uma operação identifica uma ação que pode ser executada em um objeto, dado um conjunto especificado de argumentos. Os objetos podem ser clientes, servidores ou ambos [CORRÊA, 2010].

Para que fosse possível a localização de objetos servidores ou clientes distribuídos sobre uma rede de interconexão e compor uma aplicação distribuída sob o modelo de objetos distribuídos, foi concedida pelo consórcio OMG (*Object Management Group*), a especificação CORBA (*Common Object Request Broker Architecture*), que propõe um padrão, sobre como os mediadores do tipo *broker* devem ser programados.

3.4.1 CORBA, especificação padrão para mediadores do tipo *broker*

Um *broker* é responsável por todos os mecanismos necessários para encontrar a implementação de um objeto distribuído em uma rede de acordo com uma requisição, preparar a implementação de um objeto para receber requisições e transferir os dados que compõem a requisição ao objeto destino [CORBA, 2011].

Uma especificação padrão permite que empresas desenvolvam *brokers* que suportem portabilidade e interoperabilidade para aplicações escritas em diferentes linguagens podendo ser compiladas para diversas plataformas e sistemas

operacionais e que se comuniquem com outras implementações de *brokers* [SOUZA 2001].

O cliente por sua vez tem acesso a uma referência ao objeto distribuído disponibilizada pelo *broker* e invoca operações sobre o objeto remoto através dessa referência. Um cliente só conhece a estrutura lógica do objeto de acordo com sua interface.

As interfaces podem ser estáticas ou dinâmicas e são a única forma pela qual um cliente pode se comunicar com o objeto distribuído servidor. As interfaces estáticas são disponibilizadas aos clientes antes do código ser executado. As interfaces dinâmicas são carregadas pelos clientes conforme a necessidade. Após serem compiladas, essas interfaces são denominadas "*stub*" e "*skeleton*".

A comunicação entre os componentes cliente e servidor (objeto implementação) e as interfaces *stub* e *skeleton* pode ser visualizada na Figura 3.4.

Figura 3.4 - Comunicação interna no ORB via interfaces.

A partir de 1997, CORBA iniciou o ciclo de desenvolvimento de suas versões 2.x, sendo mantidas em evolução até junho de 2002 [CORBAHISTORY, 2011]. Apesar de ter se tornado um padrão bastante difundido diante das soluções de tecnologias de mediadores do tipo *broker*, com o passar dos anos novas requisitos passaram a ser exigidos e o padrão CORBA 2.x teve que ser atualizado.

Em junho de 2002, a OMG disponibilizou a primeira versão das especificações CORBA 3.x, sendo que sua última atualização foi disponibilizada em janeiro de 2008 [CORBAHISTORY, 2011]. Com CORBA 3.x é possível que seja codificada grande parte do código dos servidores, permitindo que o desenvolvedor concentre sua atenção na lógica de negócios.

Maiores detalhes sobre as especificações CORBA 3.x, podem ser encontrados nos trabalhos de Nardi [NARDI,2003], Marques [MARQUES, 2006] ou diretamente nas especificações [CORBA, 2011].

Apesar de existirem as especificações da OMG para padronizar a concepção de mediadores do tipo *broker* prezando portabilidade e interoperabilidade entre linguagens e plataformas, as empresas de tecnologias computacionais Microsoft e Oracle, criaram especificações e implementações proprietárias para mediadores de comunicação, que são denominadas DCOM (*Distributed Component Object Model*) e Java/RMI (*Java Remote Method Invocation*) respectivamente.

3.4.2 Mediadores de comunicação da *Microsoft*

A evolução de tecnologias que gerou DCOM, começou a partir de uma tecnologia de estruturação de documentos chamada *Object Linking Embedding* (OLE), presente na maioria das aplicações Windows. Seguindo a evolução tecnológica, OLE foi transformada em uma tecnologia mais genérica e orientada a objetos chamada de *Component Object Model*, ou COM.

A tecnologia COM está longe de ser um mediador do tipo *broker*, pois não define um protocolo que permite interoperabilidade entre componentes remotos. Dessa forma, COM pode ser usada para definir componentes com interfaces comuns de interoperabilidade, porém com restrição de que somente pode-se vincular componentes disponíveis na mesma máquina.

Com o objetivo de criar um produto com as funcionalidades de mediadores do tipo ORB, a partir do Windows NT 4.0, a Microsoft estendeu e aperfeiçoou COM resultando na tecnologia DCOM.

A partir de então, a ligação entre um componente e outro passou a ser um pouco mais extensa. No NT4 foi acrescentado o protocolo TCP/IP a arquitetura de redes COM e essencialmente, tornou o DCOM capaz de ser compatível com a internet. Além de conFigurar o cliente e o servidor na tecnologia DCOM as chamadas entre objetos locais ou remotos são transparentes ao cliente e até mesmo ao programador [BARWEL, 2004].

O barramento DCOM gera um objeto *stub* que permanece armazenado no servidor e recebe chamadas do objeto *stub proxy* armazenado no cliente. Quando uma chamada remota a um objeto servidor é invocada a mesma é convertida em uma chamada local à interface solicitada. Assim, tudo se passa transparentemente no barramento DCOM.

Os objetos *stub proxy* e *stub*, são muito similares ao modo como CORBA implementa transparência, utilizando interfaces *stub* para comunicação do cliente e *skeleton* para o servidor. Numa visão bastante simples, pode-se assumir DCOM como a especificação de um *broker* específico para plataformas Windows.

Um objeto DCOM, além de poder ser iniciado por meio de um aplicativo executável, pode também ser executado por meio de um navegador. Nesse último modo de execução, passa a ser conhecido como objeto ActiveX.

Além das especificações e implementações de serviços que originaram a plataforma DCOM, no final da década de 1990, a Microsoft anunciou um novo modelo de componentes conhecido como COM+.

O modelo de componentes COM+ estende o modelo DCOM adicionando diversas melhorias em sua infraestrutura. COM+ se manteve como principal tecnologia de componentes para sistemas distribuídos Microsoft até meados de 2003, passando por diversas versões.

Mantendo os avanços sobre a evolução de sistemas de objetos distribuídos a Microsoft anunciou junto a tecnologia COM+, uma nova proposta denominada *Distributed interNetwork Architecture* (DNA) que sugeria uma forma de se organizar a estrutura de aplicativos para a internet baseados em componentes COM+ (também conhecidos como ActiveX).

No início da década de 2000 a Microsoft migrou a arquitetura DNA para o arcabouço .NET, uma especificação para o desenvolvimento de aplicações distribuídas para a internet que possui fortes fundamentos na disponibilidade de serviços web (*Web-Services*).

O arcabouço .Net 1.0 lançado em 2002, trouxe uma nova tecnologia para aplicações distribuídas denominada *.Net Remoting* que se manteve como principal tecnologia Microsoft até o lançamento do WCF (*Windows Communication Foundation*) em 2008 [REMOTING, 2011].

A tecnologia WCF disponibilizada junto com a versão 3.0 do arcabouço .Net, é a principal tecnologia para sistemas distribuídos da Microsoft na atualidade e está disposta também na versão 4.0 do arcabouço .Net, o qual teve sua última versão disponibilizada em fevereiro de 2011 [NET, 2011].

3.4.3 Java Remote Method Invocation (Java RMI), mediador de comunicação da Oracle

Desde a versão 1.1 do kit de desenvolvimento Java (JDK 1.1) da Oracle, foi incluído na linguagem Java, *Remote Method Invocation* (RMI) como parte das bibliotecas padrões da linguagem. RMI assume que todos os objetos que constituem o sistema distribuído são escritos em Java e executam em suas respectivas máquinas virtuais. Assim, supõe-se que a rede é uma coleção homogênea de máquinas virtuais.

RMI permite que objetos clientes Java invoquem métodos em objetos servidores Java, não importando se eles estão se comunicando na mesma JVM ou até mesmo, no mesmo computador.

Pode se contextualizar RMI como uma maneira natural de implementação distribuída quando todos os participantes são escritos em Java. Assim, métodos remotos são chamados como se fossem locais, não existe carga extra adicionada de protocolos ou mecanismos de manipulação de comunicação subjacente (como exigências quanto a criação e manipulação de sockets) [BATRA et al., 2008].

Em termos de complexidade de programação e ambiente, é muito simples construir aplicações RMI comparando-a com outras tecnologias. Em termos de ambiente, exige somente suporte TCP/IP e um serviço de nomes de objetos (rmiregistry), disponibilizado gratuitamente com o JDK [BORSOI; SCHULTS, 2004].

Tal como nas especificações CORBA ou na plataforma DCOM, que só acessam objetos remotos através de interfaces, clientes Java RMI interagem com os objetos remotos somente através de objetos *stub* e objetos *skeleton* de intermédio conforme pode ser visualizado na Figura 3.5.

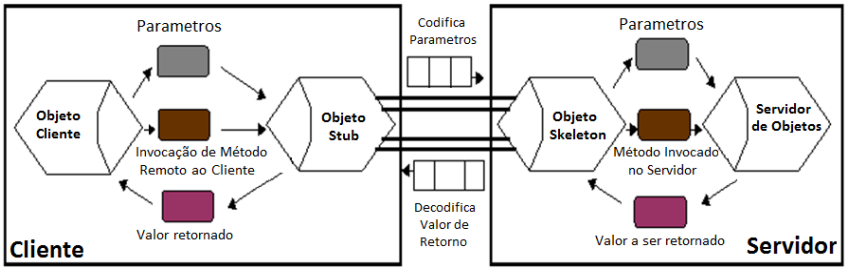

Figura 3.5 - Objetos que compõem uma comunicação distribuída em Java RMI

Quando um objeto servidor RMI é requisitado por um objeto cliente e ambos os objetos são codificados sob a linguagem Java, a comunicação geralmente é realizada através do protocolo JRMP (*Java Remote Method Protocol*). Porém, existem outras formas de comunicação, conforme demonstrado na Figura 3.6.

Na comunicação Java/IDL (disponível a partir do jdk 2.0) as interfaces são mapeadas por meio da linguagem IDL neutra de CORBA e, ao serem compiladas, são geradas automaticamente as classes da aplicação. Esta possibilidade de compilação de interfaces permite que objetos RMI se comportem como objetos CORBA.

Com a comunicação RMI-IIOP as interfaces são desenvolvidas em Java porém, ao serem compiladas a comunicação IIOP passa a ser permitida (disponível a partir do jdk 5.0). A tecnologia Enterprise JavaBeans versão 3.0, é um exemplo de componente Java que pode se comunicar através de RMI-IIOP [BURKE; MONSON, 2006].

Figura 3.6 - Tipos de comunicação presentes em RMI.

O modelo RMI pode ser bem vantajoso para programadores Java. Ele torna o *broker* transparente, estendendo naturalmente o alcance das funcionalidades da linguagem Java. Além disso, possui variantes que permitem comunicação com objetos que reconhecem o protocolo IIOP, como o caso de componentes criados sob as especificações CORBA.

3.4.4 A Plataforma JAMP

Inicialmente, JAMP (*Java Architecture for Media Processing*) foi projetada como um ambiente para o desenvolvimento de aplicações multimídia cooperativas em ambiente distribuído.

Estendendo Java RMI, JAMP provê recursos de localização transparente dos objetos distribuídos. Além disso oferece diversos servidores e arcabouços que auxiliam o desenvolvimento de aplicações distribuídas com suporte a vários formatos de mídia.

JAMP foi criada como parte de um projeto maior chamado MultiEng, patrocinado pela Finep/Recope, Computação de Alto Desempenho [SOUZA, 2001]. Seu desenvolvimento teve início em 1997 e sua continuidade passou por inovações em vários trabalhos científicos [BAPTISTA, 2004] [BARBOSA, 2006] [GUIMARÃES, 2000] [GONÇALVEZ, 2002] [SOUZA,2001] [MENDONÇA, 2000] [CORRÊA, 2010]. A plataforma JAMP se apresenta como tecnologia padrão para a distribuição dos ambientes virtuais do projeto XPTA.lab [XPTA, 2011].

A plataforma JAMP visa o desenvolvimento de aplicações distribuídas em ambientes abertos (internet). Consiste de um conjunto escalável de servidores e arcabouços que podem ser utilizados no desenvolvimento de aplicações distribuídas. Assim, as aplicações podem utilizar os arcabouços disponíveis na plataforma habilitando a execução dos serviços oferecidos.

O principal componente dentre os disponíveis na plataforma JAMP é o *broker*, denominado nesta plataforma como "JBroker". O componente JBroker tem a função de localizar todos os demais servidores e arcabouços da plataforma. Isto permite que aplicativos com objetos clientes a JAMP invoquem métodos em objetos remotos na rede sem conhecer detalhes de localização, implementação e protocolos desses objetos.

A Figura 13.7 ilustra como funciona o processo de *Trading* (Processo de Negociação) que é o meio pelo qual um objeto cliente JAMP se comunica com um objeto servidor. Na JAMP o processo se inicia com o objeto servidor sendo registrado no JBroker para uso direto das aplicações.

Após o registro, o objeto servidor passa a disponibilizar seus métodos e implementações para que objetos clientes possam invocá-los. Dessa forma qualquer cliente JAMP pode fazer acessos ao JBroker e procurar pelos serviços disponíveis.

O cliente quando requisita um serviço ao JBroker recebe uma referência ao objeto que contém o serviço requisitado assim, pode fazer uso dos serviços disponibilizados.

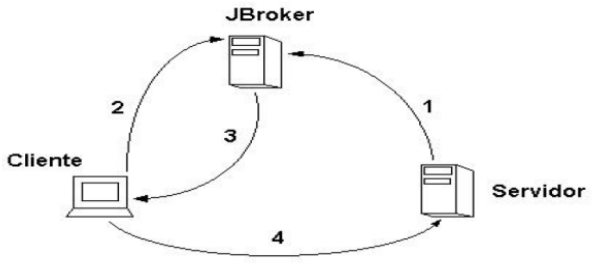

Figura 3.7 - Processo de Trading. (1) Publicação do serviço. (2) Consulta ao JBroker. (3) Referência do objeto de serviço. (4) Acesso direto ao serviço [BARBOSA, 2006].

Como o JBroker utiliza RMI em sua comunicação, uma das vantagens providas pelo JBroker diante de aplicações AVDs está na capacidade que RMI possui de processar invocações remotas de métodos diretamente na porta 80. Desta forma as classes que representam o JBroker podem ser hospedadas e receber requisição diretamente em servidores http [RMI, 2011].

Essa característica de comunicação traz a vantagem de não ser necessário modificações em regras de *firewall* entre os computadores que acessam os objetos servidores referenciados pelo JBroker, pois, é comum nessas regras que a comunicação via porta 80 seja por padrão liberada.

Por meio da JAMP pode-se conceber aplicativos sobre a estrutura de múltiplos servidores. Nesta estrutura, servidores disponibilizados em rede local são acessíveis por clientes locais e, em redes remotas, servidores são acessíveis por clientes remotos. Assim, os múltiplos servidores podem utilizar o JBroker para se manterem sincronizados, o que é um dos requisitos para sistemas de AVDs.

Para a comunicação entre servidores e o JBroker, a plataforma JAMP possui também comunicação via protocolo RTEP que pode ser visto em detalhes no trabalho de Correa [CORRÊA, 2010]. O protocolo RTEP possui maior velocidade de comunicação comparado ao RMI e foi projetado para o serviço de sincronismo entre diferentes servidores registrados junto ao JBroker.

A Figura 3.8 apresenta uma visão geral da plataforma JAMP. Nesta pode ser observada a divisão das três camadas existente na plataforma: (*i*) aplicativos de casos de uso são disponibilizados na camada de aplicações; (*ii*) diversos objetos servidores de serviços e arcabouços são disponibilizados na camada de serviços e; (*iii*) na camada de infraestrutura são demonstrados protocolos e tecnologias que sustentam a plataforma.

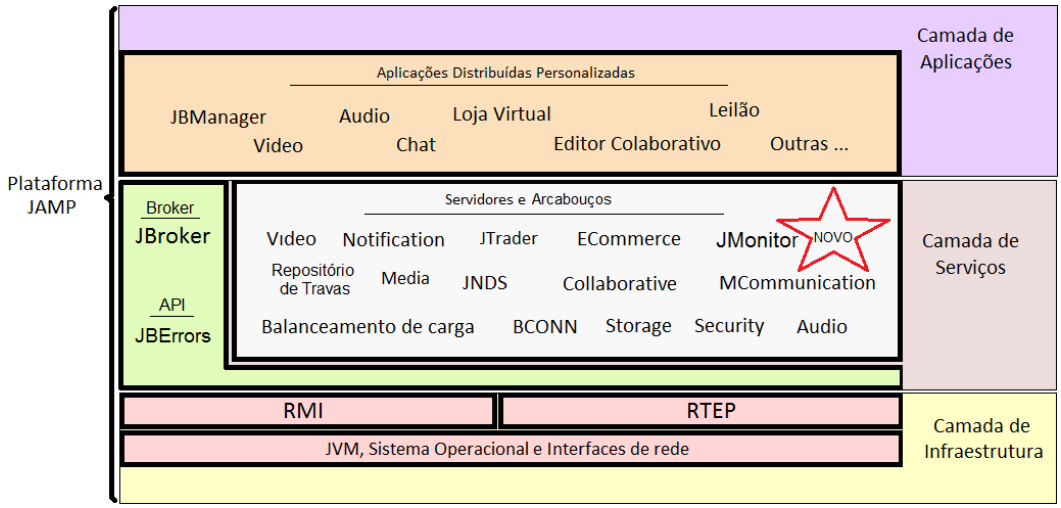

Figura 3.8 - Arquitetura da plataforma JAMP

O destaque na Figura 3.8 presente na camada de serviços junto a JMonitor se deve ao fato deste serviço representar a ferramenta de monitoramento que foi fundamentada e implementada junto a plataforma JAMP neste trabalho.

Após detalhes sobre a fundamentação e a implementação de JMonitor, este serviço será demonstrado diante de um sistema AVD de estudo de caso.

No estudo de caso, o arcabouço JNDS (*JAMP Networked Directory System*) e o servidor Repositório de Travas, ambos da camada de serviços da plataforma JAMP, também serão utilizados.

O arcabouço JNDS provê um serviço de diretórios distribuídos para os clientes que o acessam, permitindo que sejam armazenados objetos nesses diretórios [BAPTISTA, 2004].

No estudo de caso, um servidor de AVD será tratado como um objeto. Assim, poderá ser acessível para qualquer cliente que possa também se conectar ao JBroker e sucessivamente aos diretórios distribuídos.

Com o Repositório de Travas é possível criar acesso ordenado aos objetos que serão compartilhados entre diferentes clientes [BAPTISTA, 2004]. Dessa forma, sempre que um cliente tentar alterar um objeto, este, será travado antes da alteração e a liberação da trava só será efetuada quando todos os demais clientes receberem a alteração efetuada, mantendo o ambiente síncrono.

A plataforma JAMP fornece também uma API com várias definições de erros remotos, a JBErrors. Ao incorporá-la, a aplicação pode ter uma descrição mais detalhada sobre a falha ocorrida facilitando a tarefa da resolução.

O entendimento da plataforma JAMP é crucial para o trabalho a ser proposto neste trabalho. Porém, antes da formalização do estudo de caso a ser monitorado é necessário o entendimento de outras tecnologias vinculadas a computação paralela e aglomerados gráficos. Esses são os assuntos do próximo capítulo.

Capítulo 4

SISTEMAS PARALELOS, COMUNICAÇÃO E SINCRONIZAÇÃO EM AGLOMERADOS GRÁFICOS

As arquiteturas paralelas podem ser organizadas de diversas formas de acordo com a clássica taxonomia de Flynn [FLYNN; RUDD, 1996]. Em uma dessas formas, denominada MIMD (*Multiple Instruction Multiple Data*), Flynn destaca arquiteturas que possuem dois ou mais processadores que podem operar de forma cooperativa ou concorrente na execução de um ou mais aplicativos.

Tanenbaum [TANEMBAUM (b), 2007], ainda subdivide a classificação MIMD proposta por Flynn em multiprocessadores e multicomputadores.

Os multiprocessadores usam memória global e única compartilhada entre os processadores para realizar a comunicação entre os processos. Os multicomputadores possuem memória distribuída e comunicam-se por meio de passagem de mensagens.

Dash [DASH; DESMKY, 2009] e Yu [YU et al., 2004] conceituam uma categoria híbrida, denominada por memória compartilhada distribuída (DSM), cuja comunicação é realizada por meio do mapeamento das memórias compartilhadas de cada máquina fornecendo a visão de uma única memória para todo o ambiente de execução.

Dentro do contexto exposto por Flynn e complementado por Tanenbaum, Dash e YU, surgiu uma classe arquitetural que atraiu a atenção de muitos pesquisadores: o aglomerado (ou *cluster*).

Um aglomerado é uma arquitetura de multicomputadores interligados por um meio de comunicação dedicado comportando-se como um sistema único do ponto de vista do usuário.

Os aglomerados são atrativos devido ao custo/benefício proporcionado pela facilidade de aproveitamento de máquinas que já não atendem mais as expectativas de uso enquanto isoladas.

Todos os aspectos relativos à distribuição de dados, tarefas e comunicação entre os computadores podem ser abstraídos do usuário por meio de uma plataforma de programação paralela baseada nos modelos de passagem de mensagem sob memória compartilhada ou memória compartilhada distribuída.

4.1 Passagem de Mensagens

A visão lógica do paradigma de passagem de mensagens consiste em um número arbitrário de processos, cada um com seu endereço exclusivo. Para enviar um dado de um processo a outro é utilizada uma função que necessita basicamente de parâmetros como o endereço de destino, a mensagem e o tamanho da mensagem. O processo receptor normalmente especifica o buffer, o seu tamanho máximo e aguarda a mensagem do endereço remetente. Este modelo de programação pode ter comunicação síncrona e assíncrona, grupos de comunicação e outras funções agregadas.

A maioria dos programas escritos sob o paradigma de passagem de mensagens usam a abordagem SPMD (*Single Program Multiple Data*). Em programas SPMD o código executado por diferentes processos é idêntico, exceto por um processo (ex.: processo raiz ou mestre) que realiza a distribuição das tarefas a serem executadas nos nós de processamento.

O desempenho do modelo de passagem de mensagem pode ser medido pelo tempo ou tráfego de dados e são afetados pelo o número de vezes que uma mensagem precisa ser enviada ou recebida, e também pelo tamanho da mensagem [PIOLA, 2007].

Além disso, devem ser considerados a banda de rede agregada, o volume de concorrência, a segurança, a escalabilidade e o gerenciamento do tráfego de mensagens, fatores que influenciam no tempo de execução da aplicação.

4.2 MPI - Message Passing Interface

MPI é uma especificação padrão, para a concepção de bibliotecas para passagem de mensagens, concebido pelo MPI Fórum (amplo consórcio entre fabricantes de computadores paralelos, desenvolvedores de bibliotecas e especialistas em aplicações paralelas).

Segundo Pitanga [PITANGA, 2008], MPI é uma biblioteca de programação com funções para troca de mensagens entre os nós de um aglomerado e é responsável também pela comunicação e sincronização dos processos do mesmo.

A especificação de um padrão se fez necessária pelo fato de que, no passado, cada fabricante tinha sua própria biblioteca, otimizada para arquitetura de sua máquina em particular, não havendo portabilidade, dificultando assim a difusão do uso de troca de mensagens.

MPI descreve detalhes de algumas rotinas para operações coletivas, envolvendo conjuntos de nós do aglomerado e, aproximadamente 125 rotinas para a passagem e o gerenciamento de mensagens [MPI, 2011]. E é implementada em vários produtos disponíveis no mercado, inclusive em versões gratuitas sob diversas plataformas. Exemplos: MPICH (Argonne National Laboratory, Mississippi State University), FT-MPI (University of Tennessee), Open MPI Intel, etc.

MPI padroniza uma biblioteca de rotinas para passagem de mensagens entre processos *clusterizados*. Essa, foi descrita nesse trabalho com objetivo comparativo às características disponíveis no arcabouço LibGlass descritas no próximo tópico.

LibGlass possui diversas similaridades com a especificação MPI porém, seu projeto foi direcionado a um tipo especializado de aglomerado, denominado aglomerado gráfico ou VRCluster.

Nos aglomerados gráficos, os nós possuem dispositivos especiais, tais como, placas gráficas pela qual suportam a multiprojeção de vídeo. Através da multiprojeção de vídeo podem ser montados dispositivos que provêm maior

sensação de imersão sobre as projetações, tal como: cavernas digitais, *power walls* ou panoramas [GUIMARÃES, 2004].

Para a proposta de ambientes virtuais distribuídos com suporte a clientes sob projeção única e multiprojeção. Os clientes sob multiprojeção estarão sendo executados em aglomerados gráficos controlando cavernas digitais . A Figura 4.1, exemplifica essa arquitetura.

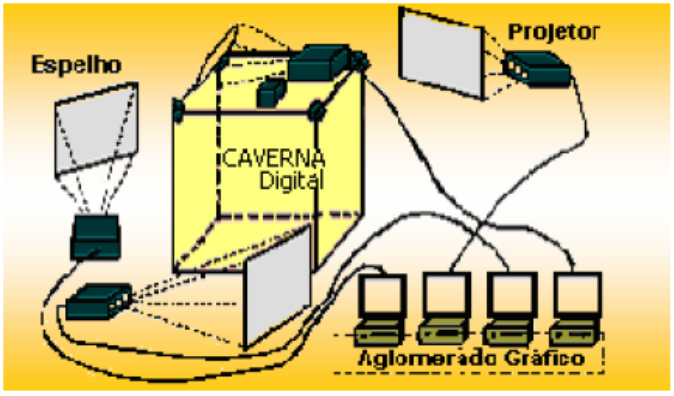

Figura 4.1 - Caverna Digital sendo gerenciado por um aglomerado gráficos [GUIMARÃES, 2004].

4.3 O Arcabouço LibGlass

LibGlass é um arcabouço para sincronização de aglomerados gráficos em que funcionalidades são fornecidas através de plug-ins [GUIMARÃES; GNECCO, 2011]. Por meio do sincronismo realizado pela LibGlass, jogos, aplicações científicas e educacionais podem ser executadas em aglomerados gráficos usando cavernas digitais.

A LibGlass se apresenta como tecnologia padrão para sincronia dos aglomerados gráficos do projeto XPTA.lab [XPTA, 2011] e também será utilizada para sincronizar o aglomerado gráfico que se conectará ao sistema AVD do estudo de caso deste trabalho.

Aplicações tridimensionais, executadas em aglomerados gráficos possuem requisitos específicos que incluem: baixo atraso na troca de informações,

sincronização rápida, banda larga, entre outros [CARDOSO et al., 2003][GNECCO et al., 2003].

A LibGlass é totalmente escrita em C++ porém, possui em sua estrutura *bindings* de suas funções compiladas para código nativo (.dll ou .so). Em outras palavras, as funções *LibGlass* são acessíveis a qualquer tecnologia que possa acessar código nativo, como a tecnologia JNI (*Java Native Interface*) que permite invocar funções escritas em código nativo através de um aplicativo Java.

Atualmente a LibGlass é composta por 4 plugins [LIBGLASS, 2011]: variáveis compartilhadas síncronas, sincronização de barreiras, eventos e *alias*.

LibGlass possui também algumas extensões que permitem a geração de registros (*logs*) sobre funções invocadas e facilidades para controle de processos em rede por meio de dispositivos móveis ou até mesmo por eventos sob reconhecimento de voz.

Nos projeto de AVDs de acordo com as especificações XPTA.lab a utilização da LibGlass tem a função de sincronizar os aglomerados gráficos que fazem acesso aos AVDs.

Para que esta função seja realizada com sucesso, o AVD deve ser concebido de forma a permitir o desenho e redesenho da imagem além dos limites da projeção central. De modo a explorar os recursos de tela das cavernas digitais.

Levando em conta as projeções laterais, estas podem ser representadas como: projeção lateral direita e projeção lateral esquerda (dependendo de como a caverna digital é montada, as projeções podem variar como projeção solo "chão", projeção acima "céu", projeção traseira "costas", etc.).

No estudo de caso fundamentado neste trabalho, cada nó do aglomerado gráfico apesar de processar diferentes partes da imagem é tratado como um cliente único para os servidores que mantêm os AVDs. Portanto, qualquer interação, vinda de um nó do aglomerado gráfico ou de qualquer outro cliente conectado ao AVD, é refletida para todos os demais clientes do ambiente.

Apesar de todo o sincronismo oferecido pela plataforma JAMP, existem casos onde valores devem ser sincronizados somente entre os nós do aglomerado gráfico. Em outras palavras, existem "interações locais" entre os clientes AVDs que não devem ser refletidas pelo ambiente como um todo, mas devem ser compartilhadas entre os nós de um mesmo aglomerado.

As interações locais podem ser adversas e dependem de quais recursos de interatividade os AVDs oferecem. Interações de movimento são típicas interações que devem ser replicadas por todo o ambiente por meio da JAMP. Interações locais a serem compartilhadas e sincronizadas pela LibGlass têm foco na escolha de como é possível visualizar pela caverna digital o AVD que está sendo compartilhado, por exemplo, trocando valores de aproximação (*zoom*), destaque de cores, luminosidade, entre outras.

Tem-se em mente que um valor de aproximação por exemplo, deve ser o mesmo para todos os nós de um aglomerado gráfico. Porém, como os aglomerados gráficos permitem visualizações com limites de tela maiores do que um computador comum, esse valor de zoom não deve ser compartilhado por todos os clientes. Corre-se o risco de um valor de zoom alto impedir a visualização de clientes que não disponibilizam recursos com grandes limites de tela.

Para a sincronia de valores locais entre os nós do aglomerado gráfico, a LibGlass oferece os *plugins* de variáveis compartilhadas síncronas e sincronização de barreiras.

O *plugin* de variáveis compartilhadas síncronas permite declarar variáveis que são compartilhadas entre os nós do aglomerado, abstraindo do programador detalhes quanto a tratamentos de conexão da rede entre os nós.

A sincronização de barreiras, por sua vez, garante que todos os nós do aglomerado processem o desenho e redesenho de tela na mesma velocidade, eliminando atrasos nas diferentes projeções.

Capítulo 5

AMBIENTES VIRTUAIS DISTRIBUÍDOS E O MONITORAMENTO

Sob uma perspectiva geral, os denominados Ambientes Virtuais (AVs) podem ser concebidos sob tecnologias gráficas 2D e 3D, derivadas das áreas da Computação Gráfica e da Realidade Virtual.

Na concepção de AVs a tecnologia gráfica 3D se destaca quando comparada a tecnologia 2D por oferecer a possibilidade de representar as informações de um modo realístico, organizando-a de uma maneira espacial. Deste modo, obtém-se maior intuição na visualização da informação por ser mais natural ao ser humano.

Um sistema de Ambiente Virtual denota-se por ser a representação sintética de um modelo real ou fictício, pela qual o usuário pode manipular, navegar e interagir modificando o estado dessa representação. Por meio de técnicas na concepção desses ambientes os estados entre as interações dos usuários são armazenados. Assim, a representação mesmo não sendo real pode ser considerada real enquanto o sistema estiver em execução.

A navegação e interação são propriedades importantes em um AV. A navegação corresponde à habilidade de se movimentar através do ambiente, explorando as características deste, sendo que este deslocamento deve respeitar um comportamento adequado em relação ao tipo de elemento representado: evitar a colisão contra paredes e obstáculos, simular o movimento de forma adequada dos objetos (incluindo a sua cinemática e dinâmica em modelos simulados) e animar os entes de forma adequada (movimentação das pessoas, animais e outros seres) [OSÓRIO et al., 2004].

Ambientes Virtuais podem possuir as características: *(i)* passiva, sendo todos os elementos autônomos e sinteticamente modelados, como por exemplo um grupo de pessoas, um grupo de peixes ou um grupo de pássaros; *(ii)* interativa, sendo todos os elementos também sinteticamente modelados, porém neste caso as representações dos elementos são controladas por usuários ou; *(iii)* mista, onde existem representações autônomas junto às representações sob o controle de usuários.

5.1 Ambientes Virtuais (AVs): Distribuídos (AVDs) e Colaborativos (AVCs)

Os AVs possuem subdivisões quanto ao tipo de conexão disponível aos usuários e também sob a estrutura interna dos mesmos. Essas subdivisões podem ser observadas na Figura 5.1.

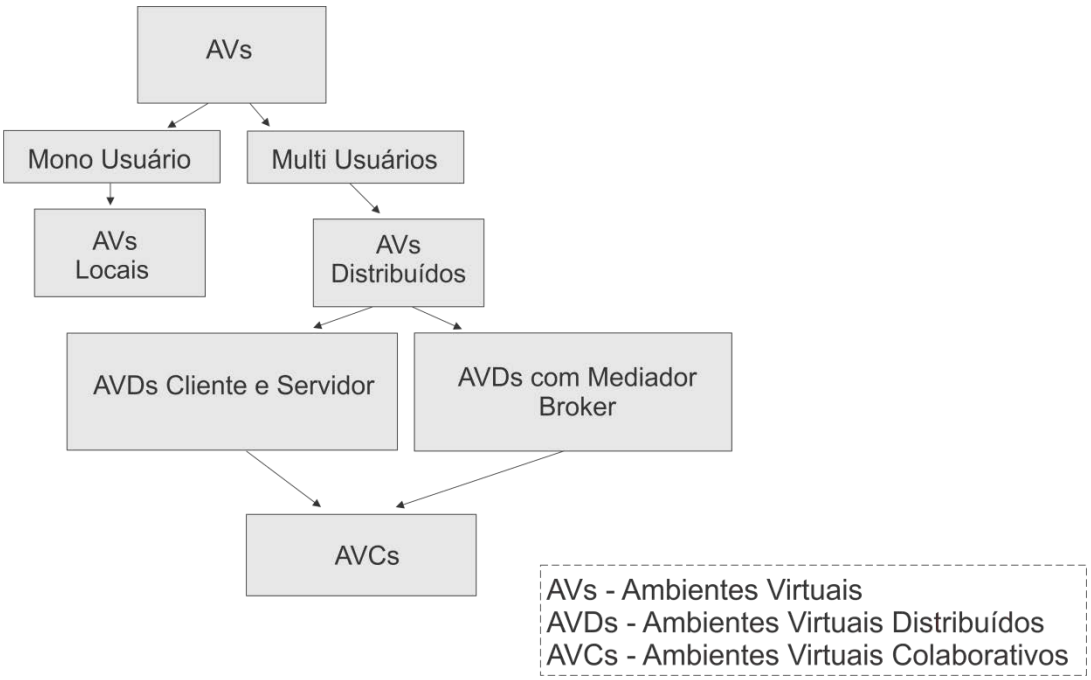

Figura 5.1 - Classificação de diferentes tipos de Ambientes Virtuais.

Também pode ser observado na Figura 5.1, que os AVCs são tipos específicos de AVDs. O estudo de caso apresentado no Capítulo 8 deste trabalho é um exemplo de AVC.

Rodrigues [RODRIGUES et. al., 2006] e Chen [CHEN et. al., 2011] além das estruturas apresentadas na figura, descrevem também a estrutura *peer-to-peer*, a qual também podem basear sistemas AVDs.

Sistemas AVDs ganharam popularidade em jogos online, tais como: *Counter Strike*, *World of Warcraft* ou *Second Life* [CHEN et. al., 2011]. Sistemas AVCs, como os ambientes de biologia e química molecular de Lee [LEE et. al., 2009] e de Corrêa [CORRÊA, 2010], são geralmente direcionados ao trabalho em equipe.

Sistemas de ambientes virtuais concebidos para execução apenas em cavernas digitais (sem clientes externos) também podem ser considerados distribuídos (AVDs), como por exemplo o sistema de estruturas dentárias de Dias [DIAS et. al., 2010]. Neste caso, os computadores que compõem o aglomerado gráfico para gerência da caverna digital executam um mesmo processo replicado com diferentes direcionamentos de câmeras. Assim, nota-se a distribuição do ambiente pelos nós do aglomerado gráfico que o acessa.

As múltiplas imagens geradas pelos computadores que compõem o aglomerado gráfico se recombinam após serem projetadas formando uma única imagem final por toda a caverna digital. Para que essa estrutura funcione corretamente deve existir algum recurso que permita a sincronização dos processos, tal como o arcabouço LibGlass, apresentado no tópico 4.3.

O uso do arcabouço LibGlass como recurso de sincronização para AVDs de caverna digital faz com que os processos replicados no aglomerado gráfico processem através de diretivas, acesso ao servidor de sincronização também presente no arcabouço dessa forma é obtido a multiprojeção síncrona de imagens que é característica a esses tipos de AVDs.

AVDs com mediadores *brokers* que utilizam recursos tais como JBroker da plataforma JAMP (tópico 3.4.4) para localização e serviços como JNDS e Repositório de Travas para gerenciamento de sincronização possuem a característica de projeções síncronas entre os diferentes clientes que o acessam.

No estudo de caso deste trabalho que será apresentado no Capítulo 8, foi concebido um AVC sobre a união das estruturas de AVD com mediador *broker* e esse mesmo AVC para acesso por meio de caverna digital. Após essa concepção foi

efetuado monitoramento através da ferramenta JMonitor (que é apresentada no Capítulo 7) para auxiliar a sincronização do ambiente como um todo.

5.2 Monitoramento de Sistemas AVDs

Pode-se observar na Figura 5.2 o relacionamento entre tecnologias utilizadas no desenvolvimento de AVs. Esse relacionamento é efetuado para melhor entendimento da necessidade de independência de linguagem na ferramenta desenvolvida.

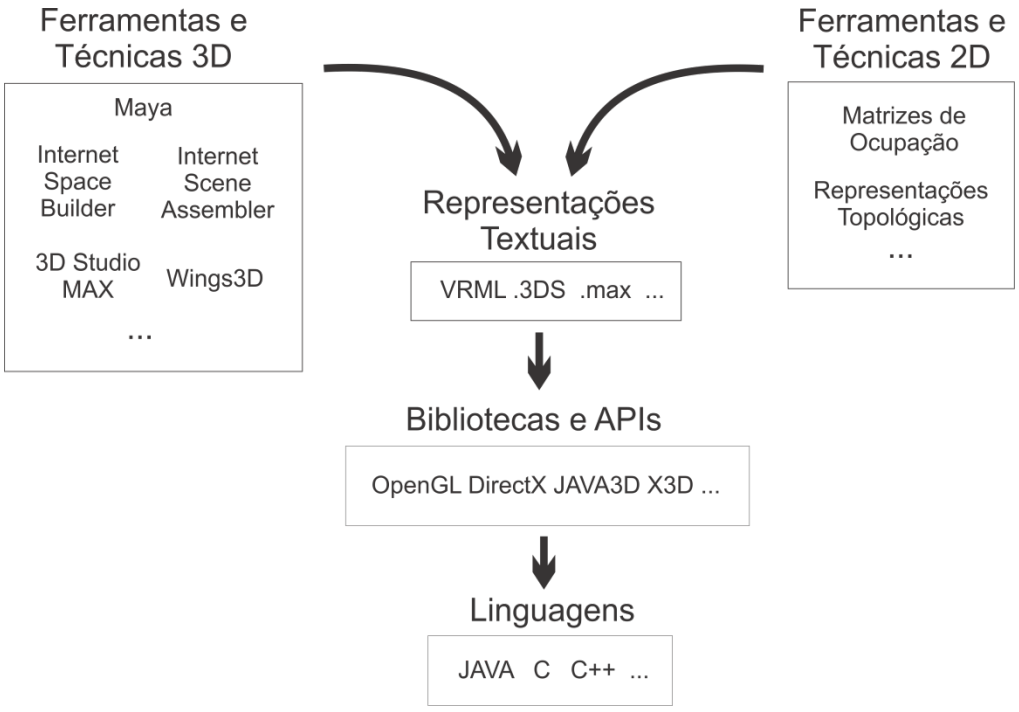

Figura 5.2 - Relacionamento entre diferentes tecnologias para a concepção de AVs.

Osório [OSÓRIO et. al., 2004] provém melhores detalhes sobre esse relacionamento. O desenvolvimento de AVs é iniciado com a modelagem do mesmo por meio de ferramentas e técnicas 2D e ou 3D.

Após as modelagens serem efetuadas as mesmas são descritas em representações textuais. Bibliotecas e APIs, são então utilizadas como recursos auxiliares provendo a capacidade de incorporar as modelagens sob representações

textuais em aplicativos desenvolvidos sob linguagens diversas tais como: JAVA, C, C++, etc.

A capacidade de efetuar movimentos sob as modelagens que são concebidas por meio das linguagens de programação é crítica quanto ao sincronismo exigido em AVDs.

A Figura 5.3 (A) ilustra como seria a multiprojeção efetuada por um AVD de caverna digital com 3 telas de projeção. Nesta figura, pode ser observado falta de sincronia entre as projeções das telas 1 e 2 quanto relacionadas a projeção da tela 3. Na Figura 5.3 (B) é ilustrado como seria a multiprojeção com correta sincronia entre as projeções.

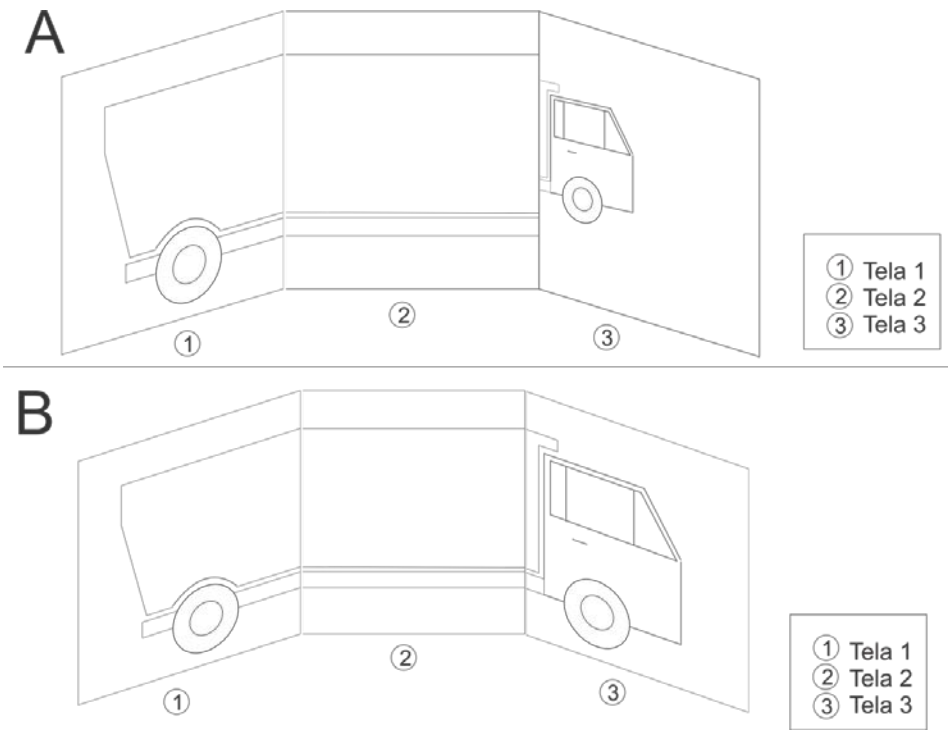

Figura 5.3 -Sincronismo em AVDs de caverna digital.

Em AVDs sob a estrutura de mediadores *brokers*, a sincronização deve ser efetuada entre os diferentes clientes que o acessam. No caso de AVCs que estendem essa estrutura a interação por qualquer cliente deve ser refletida a partir de um mesmo ponto de vista. Assim, a reflexão envolvida é crítica quanto a sincronização e real utilização desses ambientes para os trabalhos colaborativos.

A Figura 5.4 ilustra como seria um AVC com falta de sincronismo. Nessa Figura, é possível observar que os PCs 1 e 2 da rede 1 e o PC 4 da rede 2 estão em

sincronismo. Porém, o PC 3 da rede 2 está fora de sincronismo o que não pode ocorrer nesses ambientes.

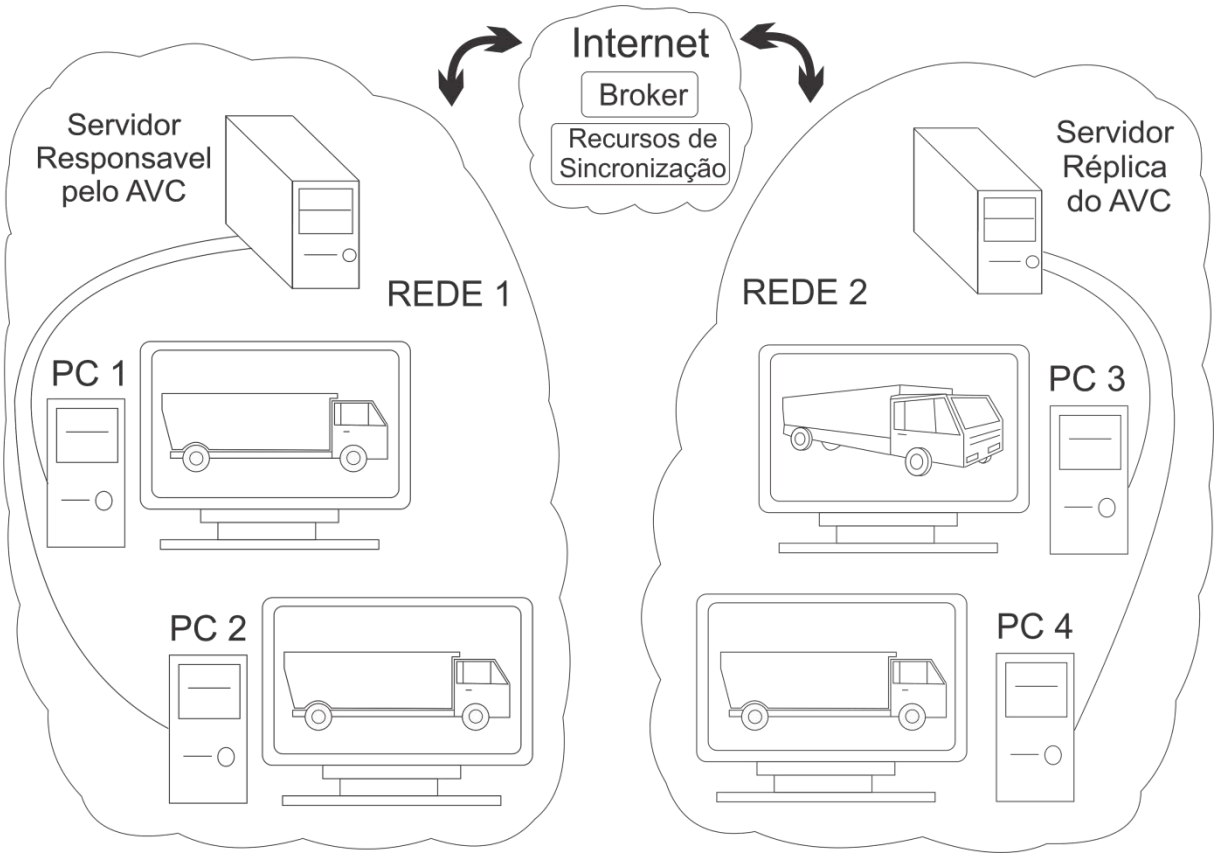

Figura 5.4 -Falta de sincronismo em AVC com mediador *broker*.

Como sistemas AVDs podem ser codificados em diferentes linguagens partiu-se do pré-suposto que os desenvolvedores que quiserem utilizar a ferramenta em seus projetos devem codificar um módulo gerador de registros (log) em seus sistemas.

O gerador de registros deve armazenar todas as informações que os desenvolvedores julgarem necessárias de serem acompanhadas, como por exemplo, informações que afetam movimentações sob AVD e podem acarretar em problemas de falta de sincronia.

Os registros devem ser periodicamente gravados e atualizados em arquivos, sobre as regras da ferramenta de monitoramento. A ferramenta possui lógica programada que monitora esses arquivos, checando sintaxes e modificações antes de processa-los, dessa forma é obtida a independência de linguagem por parte da ferramenta.

Capítulo 6

FERRAMENTA PARA MONITORAMENTO: FUNDAMENTAÇÃO

Este capítulo iniciará a fundamentação da ferramenta de monitoramento partindo de uma estrutura envolvendo um AV (Ambiente Virtual) genérico, a princípio funcionando com servidor e cliente em rede local.

Partindo deste ambiente genérico idealizado, são adicionados recursos da plataforma JAMP que permitem as características de interações colaborativas e visualização distribuída, em outras palavras o sistema passa a ser um AVC.

Contando ainda com os recursos providos pela plataforma JAMP o AVC é modificado à uma estrutura de AVD com mediador *broker*. Essa modificação, permite que servidores espelhem o AVC existente em redes remotas ou local e também proporciona a adição e navegação por servidores de outros ambientes virtuais.

O acesso através de uma caverna digital ao ambiente genérico também é descrito, o que envolve a utilização do arcabouço LibGlass.

Com a estrutura imaginária fundamentada, é abordada a proposta na qual o serviço de monitoramento irá atuar. Este executará três etapas funcionais: (*i*) coleta de dados dos clientes e servidores que compõem o AVC; (*ii*) transformação dos dados coletados em estruturas com mais usabilidade para o sistema de monitoramento e facilidade na geração de métricas e; (*iii*) disponibilizar dados transformados pela rede através visualização intuitiva.

6.1 Evoluindo um AV simples à um AVC sob estrutura AVD com mediador *broker*

A estruturação do AVC se inicia sob uma arquitetura de AV cliente/servidor de duas camadas. Conforme ilustrado na Figura 6.1-A, o servidor tem a função de prover o ambiente virtual tridimensional e gerenciar a conexão e interação entre cliente e ambiente.

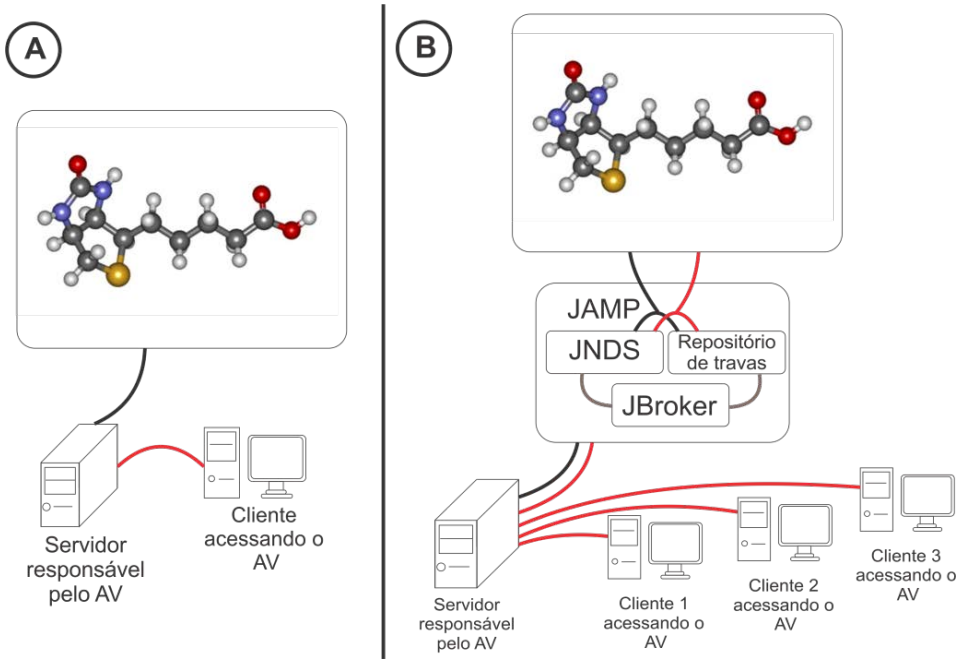

Figura - 6.1 - A) Ambiente virtual arquitetura de 2 camadas. B) AVC, arquitetura de 3 camadas.

Na Figura 6.1-B, pode ser observada a plataforma JAMP agindo como intermediária na conexão entre clientes e servidor. Nesta Figura pode ser observado também que o ambiente virtual passa a ter a capacidade de interação colaborativa e visualização distribuída entre os clientes, características típicas de AVCs.

A interação colaborativa junto com a visualização distribuída permite que mais de um cliente realize interações com o ambiente, sendo que estas são refletidas em tempo real a todos os demais clientes conectados ao mesmo ambiente.

Os serviços JNDS (*JAMP Networked Directory System*) e Repositório de Travas providos pela plataforma JAMP podem ser utilizados para a concepção de ambientes virtuais com características de AVCs.

JNDS funciona de maneira a permitir que o servidor do ambiente virtual possa exportar referências de diretórios remotos aos clientes que o acessam. Estes clientes, realizam então a importação dessas referências espelhando o conteúdo destes diretórios remotos.

No desenvolvimento de AVDs ou AVCs, pode ser imaginado que no diretório remoto é armazenado o grafo de cena junto com o histórico de interações que foram executadas. Assim, um cliente com a capacidade de renderizar o ambiente, importa o grafo de cena, carrega a disposição das formas a serem renderizadas, aplica o histórico de interações e obtém o ambiente na atual disposição das suas formas, mesmo se este ambiente tenha sido alterado em relação as disposições iniciais.

O serviço de Repositório de Travas, por sua vez, age quando uma interação é disparada entre um cliente qualquer e o ambiente virtual. Para que a interação possa ser processada, o cliente requisita ao Repositório de Travas pela permissão de alteração do ambiente virtual. Caso seu estado temporário indique situação bloqueada significa que alguma alteração de outro cliente está sendo processada. Dessa forma, a interação entra em um fila de processamento aguardando a liberação da trava para ser processada.

Quando uma interação é efetuada com sucesso entre cliente e o ambiente virtual, essa interação passa por três etapas: (*i*) requisição da trava temporária, bloqueando o ambiente a demais interações; (*ii*) envio da interação a todos os clientes conectados ao ambiente, para que todos a processem e se mantenham síncronos e; (*iii*) liberação da trava para demais interações.

Na Figura 6.2 é apresentada a evolução da estrutura idealizada. Nessa Figura, podem ser observadas duas redes de computadores se comunicando pela internet através dos recursos providos pela plataforma JAMP. Na rede 1, pode ser identificado o servidor responsável pela execução do AVD 1 e clientes conectados ao mesmo. Na rede 2, pode ser identificado o servidor responsável pelo AVD 2 com seus respectivos clientes e também um servidor réplica, que sincroniza os dados do AVD 1 e permite também a conexão de clientes.

Como a plataforma JAMP estende a tecnologia RMI em sua infraestrutura, os arquivos binários que representam seus respectivos serviços podem ser hospedados

em servidores (tais como http ou ftp) acessíveis através da internet [RMI, 2011]. Assim, quando um servidor de sistema AVC se torna disponível, este servidor pode se conecta ao JBroker através dos servidores http ou ftp e utilizar os serviços tais como o JNDS provido pela plataforma JAMP

O serviço JNDS disponível na plataforma JAMP, provê recursos adicionais, tais como: (*i*) gerenciamento; (*ii*) localização; (*iii*) adição de novos AVDs ao ambiente e; (*iv*) a replicação de ambientes já existentes em redes locais ou remotas.

Para maiores detalhes de como os serviços JNDS e Repositório de Travas foram fundamentados e implementados pode-se consultar o trabalho de Baptista [BAPTISTA, 2004].

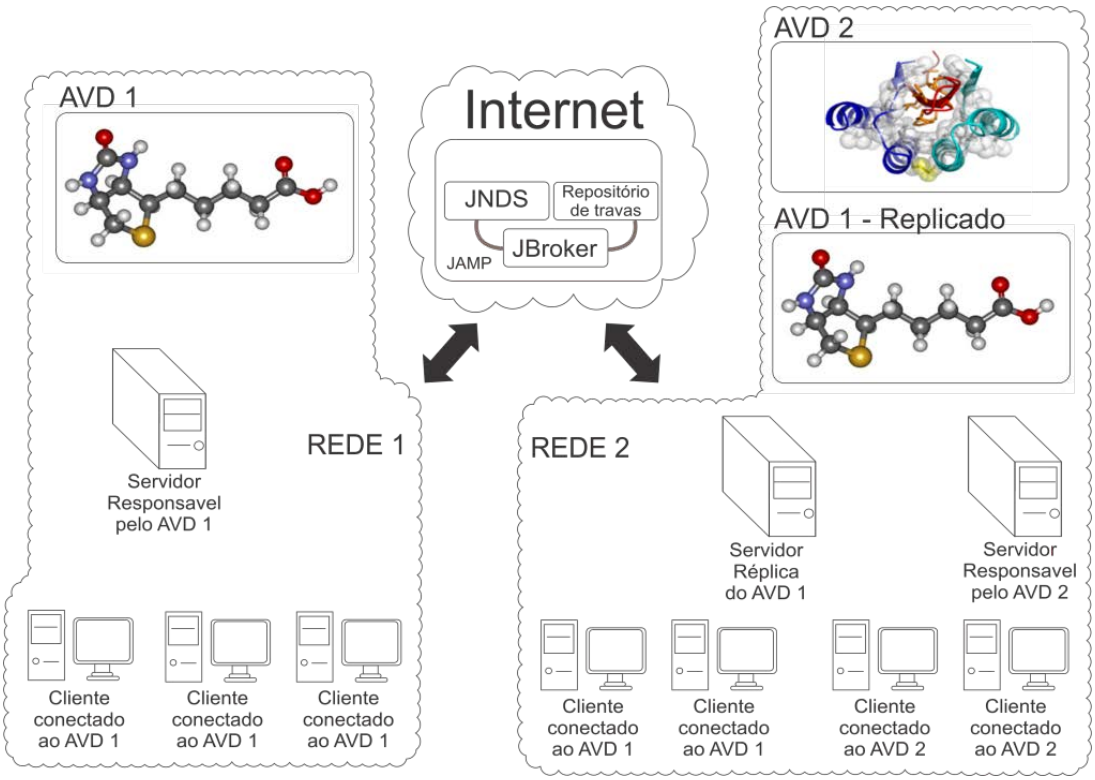

Figura 6.2 - Ambiente Virtual Distribuído com Interação colaborativa e Visualização Distribuída sob a Internet.

A estrutura idealizada, possui características de sistemas AVCs. Para atender os pré-requisitos inicialmente idealizados neste trabalho, falta a conexão dos dispositivos capazes de prover multiprojeções de vídeo tais como as cavernas digitais.

6.2 Conectando Cavernas Digitais aos AVCs

A conexão de cavernas digitais ao AVC não difere muito das conexões de clientes comuns via PC. As cavernas digitais apenas devem ser organizadas de modo que cada nó dos aglomerados gráficos que as gerenciam irá executar um processo cliente conectado ao servidor de ambientes virtuais, como se fossem clientes PCs conectados ao mesmo.

Com cada nó executando um processo cliente é proporcionado à caverna digital uma visão refletida em todas as projeções que a mesma processa, como ilustra a Figura 6.3.

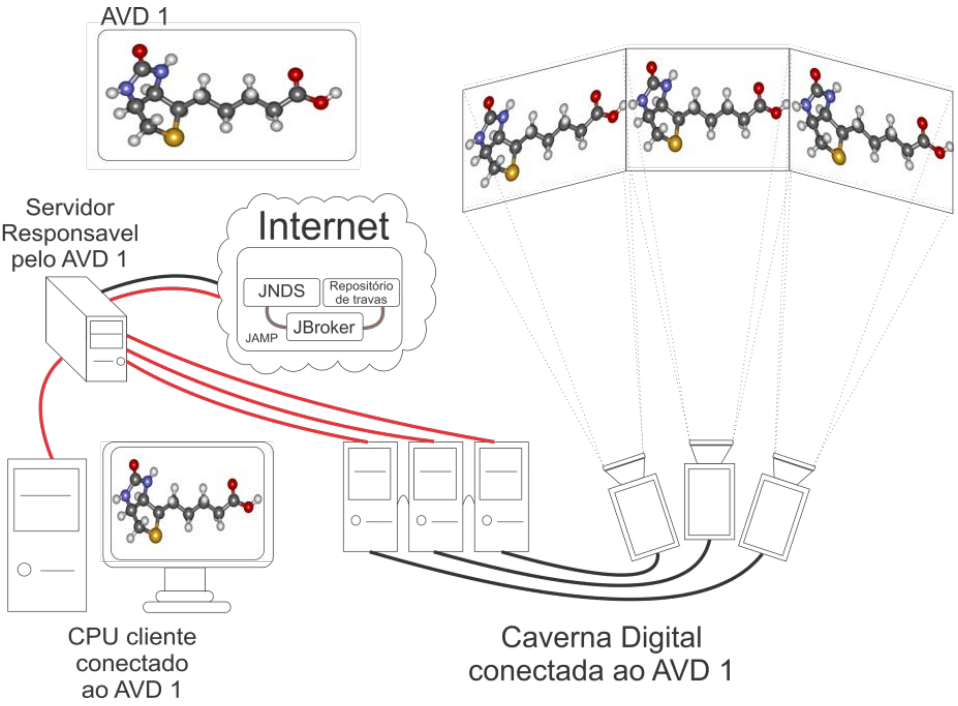

Figura 6.3 - Caverna Digital executando em cada nó do aglomerado gráfico um processo cliente do servidor de AVC.

Porém, na Figura 6.3 pode ser observado que a caverna digital está sendo subutilizada. O propósito das cavernas digitais está em utilizar as múltiplas projeções de maneira que cada nó do aglomerado gráfico processa uma parte da imagem e não a mesma imagem replicada, como apresentado.

Mesmo com a subutilização, a conexão da caverna digital e AVC da maneira como foi demonstrado provê os requisitos de interação cooperativa e visualização

distribuída do ambiente como um todo, o que mantém a caverna sincronizada com os demais clientes do AVC.

Com os requisitos de interação cooperativa e visualização distribuída já presentes no ambiente, a próxima etapa para correta utilização da caverna digital é a especialização dos processos clientes que executam nos nós do aglomerado gráfico.

Os processos clientes que executam nos nós do aglomerado gráfico devem ser modificados de forma a permitir deslocamentos de câmera de acordo com a disposição das múltiplas projeções da caverna digital em questão. Esse deslocamento deve ser efetuado de forma que as várias imagens se completem, expandindo a imagem virtual final projetada.

O resultado do correto deslocamento de câmeras na conexão de AVDs e cavernas digitais pode ser visualizado na Figura 6.4. Porém, na mesma Figura pode ser observado que, por se tratar de um AVC, quando valores de aproximação (*zoom*) são ajustados de modo a explorar os limites de tela da caverna digital, outros clientes conectados ao ambiente podem ter a visualização da imagem comprometida. Por exemplo no caso de um cliente visualizando a imagem através de um computador pessoal (PC).

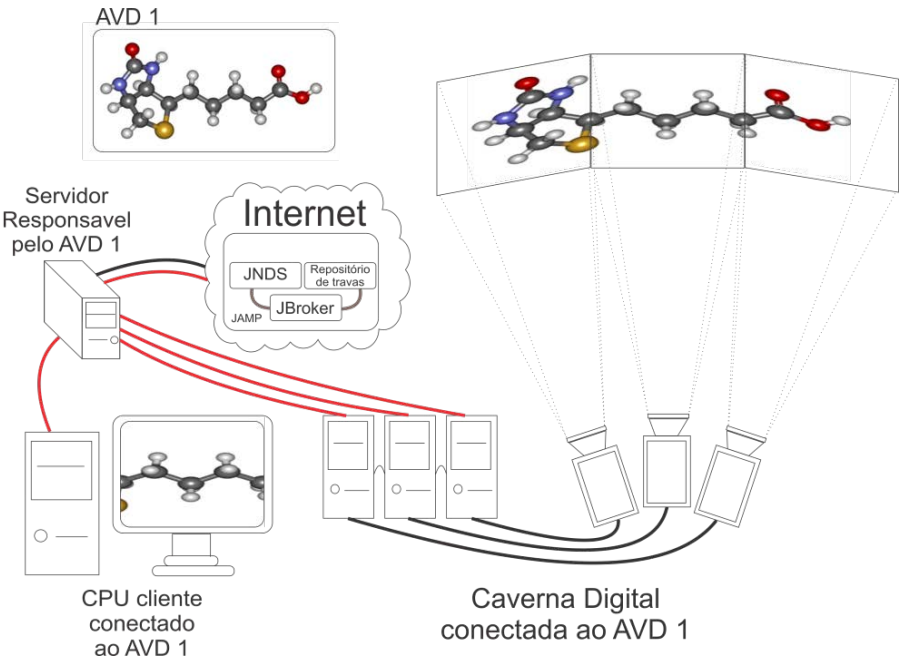

Figura 6.4 - Caverna Digital com especialização dos processos dos clientes que projetam as imagens das telas laterais da mesma.

Com a atual estrutura do AVC fundamentado, foi identificado que existem formas diferentes de interações. As interações de direcionamento que controlam rotações e deslocamentos do modelo virtual visualizado devem ser sincronizadas com todos os clientes conectados ao mesmo AVC. Porém, existem interações que devem ser consideradas como locais aos diferentes clientes: (*i*) valores de aproximação (*zoom*) da imagem; (*ii*) filtros de destaque de coloração e; (*iii*) outras propriedades adversas dos modelos virtuais.

As interações que trocam valores de propriedades locais a processos clientes dos AVCs trazem um novo problema na conexão da caverna digital. Se a troca de um valor de aproximação da imagem é efetuada em um processo cliente que projeta a imagem de uma das telas da caverna, esta fica com a aproximação da imagem gerada diferente das demais, perdendo sincronia na multiprojeção (como foi apresentado no tópico 5.2).

Para solucionar o problema das interações locais nas cavernas digitais, uma nova especialização dos clientes destinados a aglomerados gráficos deve ser efetuada. Nessa nova especialização, esses clientes devem utilizar as diretivas de sincronização disponibilizadas pelo arcabouço LibGlass para manter a sincronia das interações locais.

A sincronização desses valores será efetuada somente entre os nós de um mesmo aglomerado gráfico, não sendo replicada aos demais clientes. Para que essa alteração seja efetuada com sucesso, todos os nós de um mesmo aglomerado gráfico devem se tornar também clientes do servidor de sincronização do arcabouço LibGlass. Este servidor, unido às diretivas do arcabouço trata sincronizações entre nós de um aglomerado gráfico.

Pode ser observado na Figura 6.5 um exemplo de estrutura onde o servidor da LibGlass possui a responsabilidade de sincronizar interações locais como a aproximação da imagem a todos os nós do aglomerado gráfico. As interações de movimentação sobre o AVC continuam sob a responsabilidade da plataforma JAMP.

Com sincronismo entre as interações distribuídas e locais no AVC e com a capacidade de conexão através de caverna digital, os pré-requisitos para a fundamentação da ferramenta de monitoramento são alcançados.

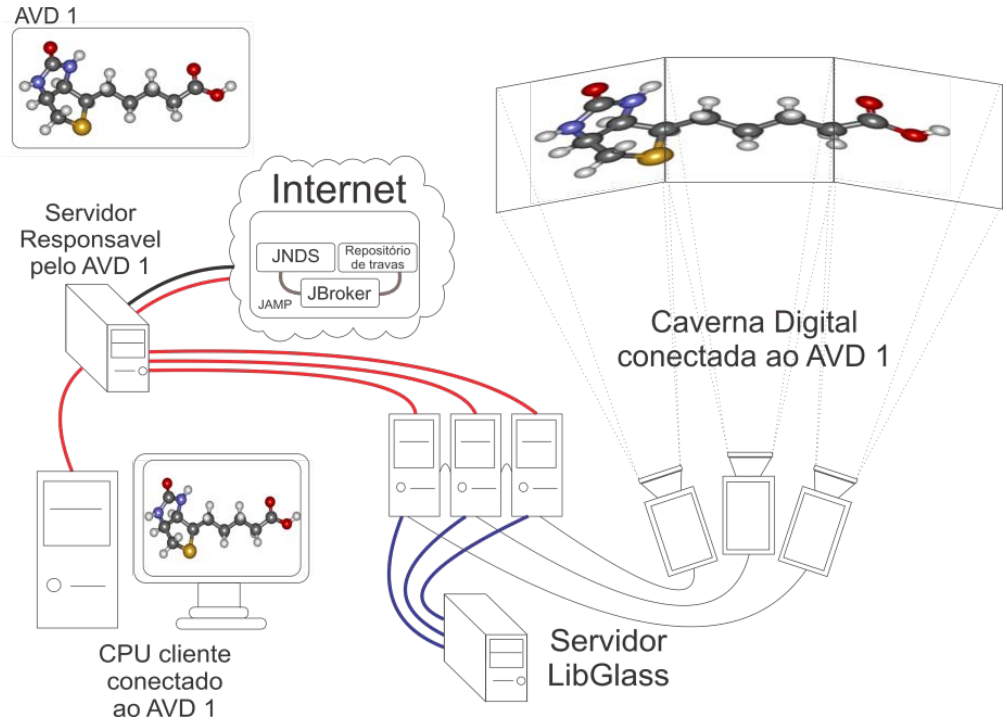

Figura 6.5 - Caverna Digital com especialização dos processos clientes sincronizando valores de interações locais através do arcabouço LibGlass.

6.3 Ferramenta de Monitoramento, Funções e Disposições dos módulos

A ferramenta de monitoramento idealizada se baseia nas etapas de coleta de dados, transformação e apresentação destes dados a usuários e desenvolvedores de sistemas AVDs. O objetivo é auxiliar a concepção e utilização de tais sistemas.

Como ilustra a Figura 6.6, as etapas foram transformadas em três módulos a serem programados, declarados como: (*i*) agente de monitoramento, responsável pela etapa de coleta de dados; (*ii*) gerenciador de monitoramento, responsável pela etapa de transformação dos dados e; (*iii*) *applet* de monitoramento, responsável pela etapa de visualização dos dados.

Foi idealizado que a etapa de coleta de dados é parcialmente de responsabilidade dos desenvolvedores, que terão que codificar módulos geradores de registros em seus AVDs. Estes registros deverão ser descritos em documentos para pós-processamento pela ferramenta de monitoramento.

Consultas como carga dos processadores, taxa de uso da memória, uso da banda de rede, valores de variáveis internas ao código dos AVDs e dados informativos dos mesmos, são exemplos de coletas de dados que podem ser cruzados gerando métricas. Essas métricas, podem interessar aos desenvolvedores e usuários dos sistemas de AVDs.

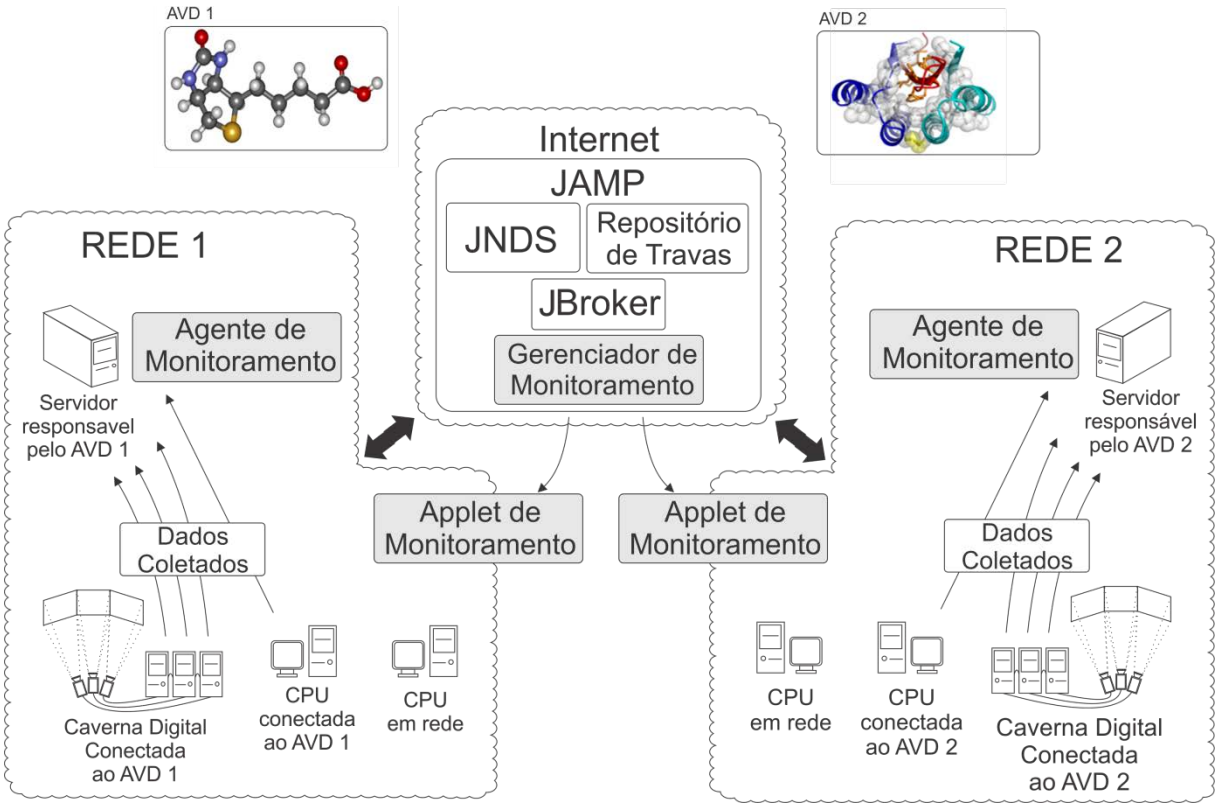

Figura 6.6 - Disposição dos módulos que compõem a ferramenta de monitoramento executando sobre o ambiente idealizado.

Os dados a serem coletados podem ser adquiridos com consultas programadas ao próprio sistema operacional, utilização de sistemas SNMP, codificadas através das próprias linguagens de programação dos AVDs ou de qualquer outra forma preferida dos desenvolvedores.

Existem quatro regras na etapa da coleta de dados: (*i*) dados coletados devem ser descritos em pares "nome e valor" indicando qual propriedade foi monitorada; (*ii*) os pares "nome e valor" devem ser descritos em documentos, que também descrevem detalhes do computador que executa o processo monitorado; (*iii*) o arquivo gerado pelo computador em questão deve ser periodicamente recriado, provendo informações atualizadas para a ferramenta de monitoramento e; (*iv*) deve

existir no documento um campo de validade, informando por quanto tempo a ferramenta deve tomar os dados relatados como válidos, caso estes não sejam atualizados dentro de um limite de tempo.

Pode existir algum conjunto de pré-requisições em documentos que descrevem os dados, como determinada estrutura pré-especificada. Esta estrutura pré-especificada, poderia por exemplo ser aguardada pelas etapas de transformação e visualização dos dados, de forma que quando presente poderia liberar recursos como a geração de gráficos específicos.

A parcela da coleta de dados de responsabilidade da ferramenta de monitoramento está relacionada à coleta dos documentos que descrevem os dados a serem relatados. Porém, ainda deve existir a centralização desses dados junto ao servidor do AVD que está sendo monitorado. A ferramenta de monitoramento somente coletará o arquivo gerado no computador que executa o processo servidor do AVD em questão.

É de responsabilidade dos desenvolvedores dos AVDs, que tenham intenção em utilizar a ferramenta de monitoramento, periodicamente coletar os dados de todos os clientes conectados aos servidores de seus sistemas e gerar o documento descritor. Este documento deve também, descrever os dados do próprio servidor de AVD em questão.

Um módulo representando um agente da ferramenta de monitoramento quando em execução, verifica constantemente o documento descritor de dados a serem monitorados. Caso o documento descritor seja atualizado pelo processo servidor de AVD, o agente através de uma invocação remota de método envia o documento atualizado ao módulo gerenciador disponibilizado junto a camada de serviços da plataforma JAMP.

O módulo gerenciador é responsável pela etapa de transformação dos dados. Diversos tipos de transformações podem ser efetuadas nessa etapa. Entre elas, os arquivos descritores de dados são validados e transformados em objetos contendo as propriedades do servidor AVD e membros coleções contendo as propriedades relacionados aos clientes deste AVD.

Com os arquivos descritores transformados em objetos, a facilidade em se localizar e relacionar propriedades, transformando dados em métricas, é mais intuitiva. Isso facilita a concepção da ferramenta de monitoramento.

Junto das transformações efetuadas pelo módulo gerenciador, deve ser programada uma *thread* com a função de periodicamente checar a validade das informações. Essa validade deve ser efetuada relacionando-se os tempos de criação do objeto de dados monitorados com o tempo limite que permite relatar as informações como válidas, caso estas não recebam atualizações.

Se na validação dos dados monitorados for identificado que os tempos expiraram, devido a falta de atualizações, o objeto é descartado.

A etapa de visualização das propriedades monitoradas deve ser programada em forma de *applet* ou alguma outra tecnologia capaz de executar chamadas remotas de métodos (RMI) ao módulo gerenciador. Dessa forma, qualquer usuário ou desenvolvedor dos sistemas AVDs que possa se conectar ao JBroker poderá carregar o *applet* de monitoramento e acompanhar quaisquer AVDs que estiverem em execução e relatando dados para a ferramenta de monitoramento.

Deve ser programada uma *thread* junto ao *applet* de monitoramento que periodicamente invoca o método remoto do módulo gerenciador e recebe como retorno da invocação os objetos de propriedades válidos e atualizados

É de responsabilidade do *applet* periodicamente recarregar os dados dos objetos propriedades buscados pela *thread* e apresentar essas informações aos usuários da ferramenta de monitoramento. As informações, podem ser apresentadas, por exemplo, em forma de gráficos ou mesmo como nome e valor.

No próximo capítulo, é mostrado um estudo de caso de implementação da ferramenta de monitoramento proposta.

Capítulo 7

JMONITOR: UMA IMPLEMENTAÇÃO DA FERRAMENTA DE MONITORAMENTO

A ferramenta JMonitor foi codificada sobre a infraestrutura oferecida pela plataforma JAMP e foi disponibilizada como um serviço adicional a mesma.

Apesar dessa codificação, é importante ressaltar que programadores que tenham a intenção de utilizar a ferramenta JMonitor não necessariamente terão de estruturar seus aplicativos pela mesma plataforma. A única regra, é que esses aplicativos devem disponibilizar as informações a serem processadas por meio de documentos XML, gerados de acordo com especificações que os validam.

A implementação da ferramenta JMonitor pode ser dividida em três módulos e dois arquivos que controlam a comunicação, como pode ser visualizado na Figura 7.1.

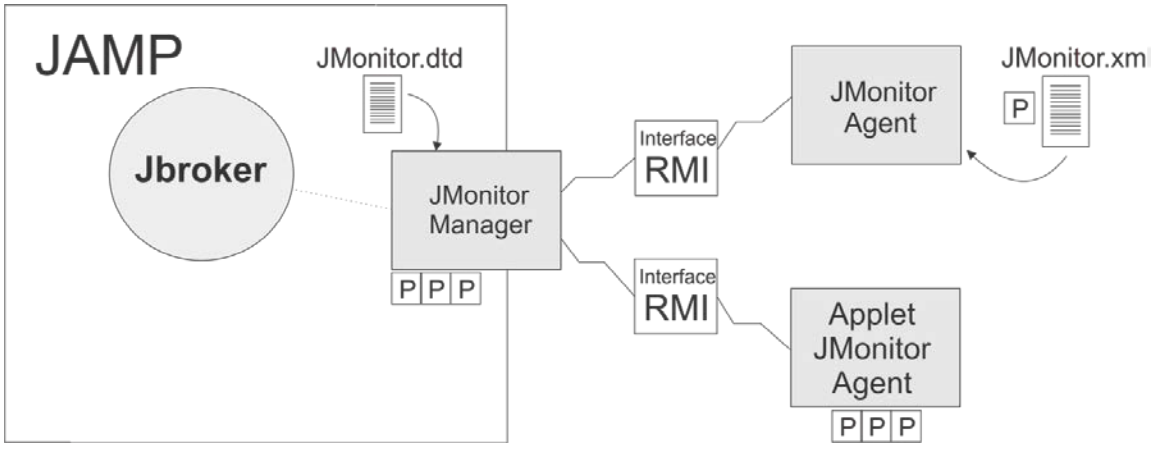

Figura 7.1 - JMonitor, comunicação entre módulos

Os módulos são denominados: JMonitorManager, JMonitorAgent e JMonitorAppletAgent. Os arquivos são JMonitor.dtd e JMonitor.xml.

Módulo JMonitorAgent

Este módulo foi preparado para ser executado junto ao processo que exerce a função de servidor do sistema AVD. Os programadores que quiserem utilizar a ferramenta JMonitor deverão preparar seus sistemas para isso através de geradores de registro (*logs*) descritos em arquivos.

Neste sistema a ser monitorado, o processo servidor do AVD deve recolher as informações de todos os processos clientes, por meios periódicos e customizados. De posse dessas informações, o servidor deve gerar ou atualizar um arquivo de registros no formato XML, relatando as informações a serem monitoradas.

Junto ao código do módulo JMonitorAgent foi concebida uma *thread* que checa constantemente se o arquivo de registros foi gerado ou atualizado pelo processo servidor do sistema AVD. Este arquivo é chamado JMonitor.xml.

Se detectada alguma alteração neste arquivo, o módulo JMonitorAgent transforma o arquivo em um conjunto de bits preparados a serem enviados ao módulo gerenciador. Isto é realizado por meio de técnicas de serialização.

Após a serialização do arquivo, o módulo JMonitorAgent invoca uma chamada RMI ao método *receiveProperties*, do modulo JMonitorManager. Assim, o módulo JMonitorAgent envia o conjunto de bits que representa o arquivo XML para o módulo gerenciador.

7.1 Módulo JMonitorManager

O módulo gerenciador JMonitorManager foi preparado para receber chamadas RMI originadas em objetos remotos. Com a chamada *receiveProperties*, o módulo recebe o conjunto de bits que foi enviado pelo modulo JMonitorAgent. De

posse desse conjunto de bits, o JMonitorManager realiza a deserialização do arquivo JMonitor.xml.

Após deserializar o arquivo, o módulo JMonitorManager realiza checagens, verificando se o mesmo está de acordo com as regras estabelecidas no arquivo JMonitor.dtd.

No arquivo JMonitor.dtd que é auxiliar ao módulo gerenciador estão definidas as regras que validam o arquivo JMonitor.xml. Através de JMonitor.dtd é possível verificar a compatibilidade do arquivo recebido com o tipo de arquivo esperado pela ferramenta JMonitor. Com a validação aceita, o arquivo recebido é então transformado em um objeto da classe Properties.

Um objeto da classe Properties representa todas as propriedades do sistema distribuído que foram marcadas no arquivo JMonitor.xml. Este objeto possui a identificação e informações relativas ao processo servidor e também, membros coleções com identificação e informações de cada cliente conectado ao mesmo.

Além de todas as informações do arquivo de registros, o objeto da classe Properties recebe também um tempo de vida, identificando por quanto tempo estas informações são tomadas como válidas. Se estas não forem atualizadas dentro do tempo de vida do objeto, este é destruído.

O módulo JMonitorManager possui a capacidade de armazenar objetos Properties representando todos os sistemas AVDs ou AVCs que estão sendo monitorados em dado momento. Através de uma *thread* vinculada ao módulo, é realizada a checagem do tempo de vida dessas propriedades armazenadas.

Caso o tempo vida tenha espirado, o objeto Properties é destruído. Caso contrário, o objeto Properties permanece disponível para ser enviado ao módulo de visualização. Isto é realizado por meio de invocações periódicas originadas no modulo de visualização.

7.2 Módulo JMonitorAppletAgent

O módulo de visualização JMonitorAppletAgent foi concebido com um aplicativo do tipo *applet* acessível a qualquer computador da rede que possa conectar o JBroker (ou conectar o servidor http ou ftp que hospeda JBroker). Este

módulo tem a função de disponibilizar as informações monitoradas a desenvolvedores e usuários dos sistemas monitorados, de forma intuitiva.

JMonitorAppletAgent também possui uma *thread* que periodicamente invoca através de RMI o método *informProperties* do módulo gerenciador. Desta forma são repassados ao *applet* todos os objetos Properties válidos armazenados no gerenciador.

No *applet* de monitoramento existem geradores de gráficos que mostram: (*i*) cargas de memória RAM; (*ii*) memória Swap; (*iii*) carga média de uso dos processadores e; (*iv*) utilização da interface de rede. Essas informações são obtidas dos objetos Properties providos pelo módulo gerenciador.

Por meio dos objetos Properties, são adquiridos e disponibilizados em forma de lista os processos servidores e respectivos clientes dos sistemas AVDs ou AVCs que estão sendo monitorados.

Esses processos listados são selecionáveis e a organização é disposta de forma alfabética e hierárquica. No primeiro nível são disponibilizadas as informações de um primeiro processo servidor de sistemas AVDs ou AVCs e todos os seus clientes. No segundo nível um segundo processo servidor de AVDs e AVCs também com seus clientes, e assim sucessivamente, até que todos os sistemas monitorados estejam listados.

Abaixo da lista de processos listados selecionáveis é disposta a lista de propriedades monitoradas. Quando um processo servidor ou cliente de AVD ou AVC é selecionado na primeira lista, as propriedades relativas ao mesmo são disponibilizadas nessa segunda lista, mantendo constante atualização conforme estas são registradas.

Propriedades notificadas como valores de variáveis que representam interações sobre os AVDs ou AVCs e são críticas ao sincronismo são disponibilizadas nessa segunda lista.

Para visualizar a sincronização em tempo real, diferentes instâncias dos *applets* de monitoramento devem ser carregadas cada qual com a seleção efetuada sobre os diferentes processos que devem ser sincronizados. Conforme um evento de interação é disparado por qualquer um dos clientes do AVD que está sendo monitorado pode ser visualizado a mudança desse valor de interação em todos os *applets* carregados, mostrando se o sincronismo existe.

A seleção da primeira lista também atualiza os geradores de gráficos para que demonstrem corretamente os dados do sistema selecionado.

Através do *applet* é possível visualizar informações como: (*i*) uso e disponibilidade dos processadores envolvidos no sistema; (*ii*) valores de variáveis e; (*iii*) detalhes informativos, como o email do responsável por um sistema distribuído. Estes exemplos de informações podem ser visualizadas por qualquer computador da rede, através do *applet* executando em conjunto com o JBroker e os demais módulos da ferramenta JMonitor.

7.3 Arquivos JMonitor.dtd e JMonitor.xml

Estes dois arquivos são necessários para a ferramenta JMonitor. Com a utilização deles é possível garantir que documentos enviados por usuários estão de acordo com os documentos que a ferramenta está preparada para processar.

Um trecho do arquivo JMonitor.dtd pode ser visualizado na Figura 7.2. Este arquivo é utilizado pelo módulo gerenciador e possui regras que validam o arquivo JMonitor.xml. A Figura 7.3 ilustra um arquivo de exemplo JMonitor.xml relatando informações de um sistema fictício

O que pode ser observado na Figura 7.2 é que os sistemas devem obrigatoriamente informar o nome do processo servidor a ser monitorado e sua validade. Estas informações servem para que a ferramenta JMonitor possa determinar por quanto tempo as informações contidas nesse documento devem ser tratadas como válidas.

Caso o sistema a ser monitorado queira utilizar os geradores de gráficos contidos no *applet*, os elementos memory-ram-total, memory-ram-free, memory-swap-total, memory-swap-free, cpu-used, recv_packages e trans_packages devem ser informados. Porém, o uso desses geradores de gráficos é opcional ao uso da ferramenta.

Pelas regras de JMonitor.dtd, o elemento client também é opcional. Caso ele exista, deve obrigatoriamente possuir um nome único entre os processos clientes conectados a um mesmo processo servidor. O uso dos geradores de gráficos referentes ao sistema que processa o processo cliente também é opcional.

```
<!ELEMENT monitoring ( serverName, vality, memory-
ram-total?, memory-ram-free?, memory-swap-total?,
memory-swap-free?, cpu-used?, recv_packages?,
trans_packages?, static-properties, dynamic-
properties, client+ ) >
        . . . .
<!ELEMENT client ( clientName, memory-ram-total?,
memory-ram-free?, memory-swap-total?, memory-
swap-free?, cpu-used?, recv_packages?,
trans_packages?, static-properties?, dynamic-
properties ) >
        . . . .
```

Figura 7.2 - JMonitor.dtd.

Os dados do elemento static-properties são aguardados em pares nome e valor. Estes dados estão preparados para receber informações como por exemplo: (*i*) email de um responsável pelo sistema; (*ii*) onde pode ser adquirido uma cópia de instalação do mesmo e; (*iii*) relação de dependências para ser instalado. Estas informações variam de acordo com a necessidade dos desenvolvedores ou usuários dos sistemas.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE monitoring SYSTEM "JMonitor.dtd">
<JMonitor>
  <serverName value="Demo Server 1" />
  <vality value="20" />
  <memory-ram-total value="3062808" />
  <memory-ram-free value="923938" />
  <memory-swap-total value="6256636" />
  <memory-swap-free value="4654613" />
  <static-properties>
    <property name="Email" value="mauricio@dc.ufscar.br" />
    <property name="Source" value="http://www.laviic.dc.br" />
  </static-properties>
  <dynamic-properties>
    <property name="Dynamic-test" value="80" />
  </dynamic-properties>
  <client>
    <clientName value="Client 1" />
    <cpu-used value="43" />
    <static-properties>
      <property name="Email" value="mauricio@dc.ufscar.br" />
      <property name="Source" value="http://www.laviic.dc.br" />
    <dynamic-properties>
      <property name="Dynamic-test" value="4" />
    </dynamic-properties>
  </client>
</Jmonitor>
```

Figura 7.3 - JMonitor.xml.

A mudança dos valores demarcados com o elemento dinamic-properties é checada e recarregada a cada atualização registrada pelos eventos periódicos internos da ferramenta JMonitor. Estes elementos dinamic-properties são apropriados para notificar a mudança de valores de variáveis para fins de depuração, como por exemplo, para verificar se um valor de interação foi modificado por um cliente do AVD ou AVC em questão.

Os elementos geradores de gráficos também utilizam esta forma de checar e recarregar as informações a cada atualização.

Na Figura 7.3 pode ser observado que o arquivo JMonitor.xml foi gerado com um único processo cliente conectado. Esta informação está demarcada pelas tags <client> e </client>. No caso de não existir nenhum processo cliente, a ferramenta está preparada para manter o monitoramento apenas do objeto servidor.

A utilização da ferramenta JMonitor no desenvolvimento e monitoramento de um estudo de caso é descrita no próximo capítulo.

Capítulo 8

ESTUDO DE CASO

Para o desenvolvimento de sistemas com características de AVCs possíveis de conexão via caverna digital, está disponibilizado na Universidade Federal de São Carlos o laboratório LAVIIC (Laboratorio de Visualização Avançada, Interação, Imersão e Colaboração) o qual dispõe entre outros dispositivos e tecnologias, uma caverna digital e recursos para o processamento de ambientes virtuais distribuídos.

Como as especificações da estrutura idealizada no decorrer deste trabalho preveem a utilização de caverna digital no acesso à AVCs e tem-se como um dos objetivos disponibilizar a ferramenta de monitoramento JMonitor como auxiliar na concepção destes tipos de sistemas, a caverna digital do laboratório LAVIIC foi utilizada nesse estudo de caso.

Neste capítulo é exposto como um AVC concebido sob estrutura de AVD com mediador *broker* acessível por meio de arquiteturas PC foi estendido a cavernas digitais e também como o mesmo foi preparado para utilizar a ferramenta JMonitor.

8.1 AVC acessível a dispositivos sob projeção única

Com o objetivo de apresentar um estudo de caso que utilize as ferramentas propostas no projeto XPTA.lab [XPTA, 2011], tais como: plataforma JAMP para distribuição dos eventos pela rede e LibGlass para sincronia de aglomerados gráficos. Será tomado como base o estudo de caso do trabalho de Baptista [BAPTISTA, 2004].

O trabalho realizado por Baptista [BAPTISTA, 2004] concebeu como estudo de caso um AVC de testes. Este AVC utiliza: (*i*) recursos da linguagem C/C++ como estrutura base de desenvolvimento; (*ii*) modelagem 3D incorporada ao sistema AVC por meio das bibliotecas gráficas OpenGL; (*iii*) recursos de um arcabouço denominado OpenReality [BELLEZI, 2003] [OPENREALITY, 2012] para gerenciamento de *sockets* e *threads* e; (*iv*) servidor JNDS e Repositório de Travas que fazem parte da plataforma JAMP.

A Figura 8.1 mostra um esboço geral da arquitetura. Nessa Figura pode ser observado que a comunicação entre modelagem 3D e os servidores é intermediada por tecnologias de *proxy*.

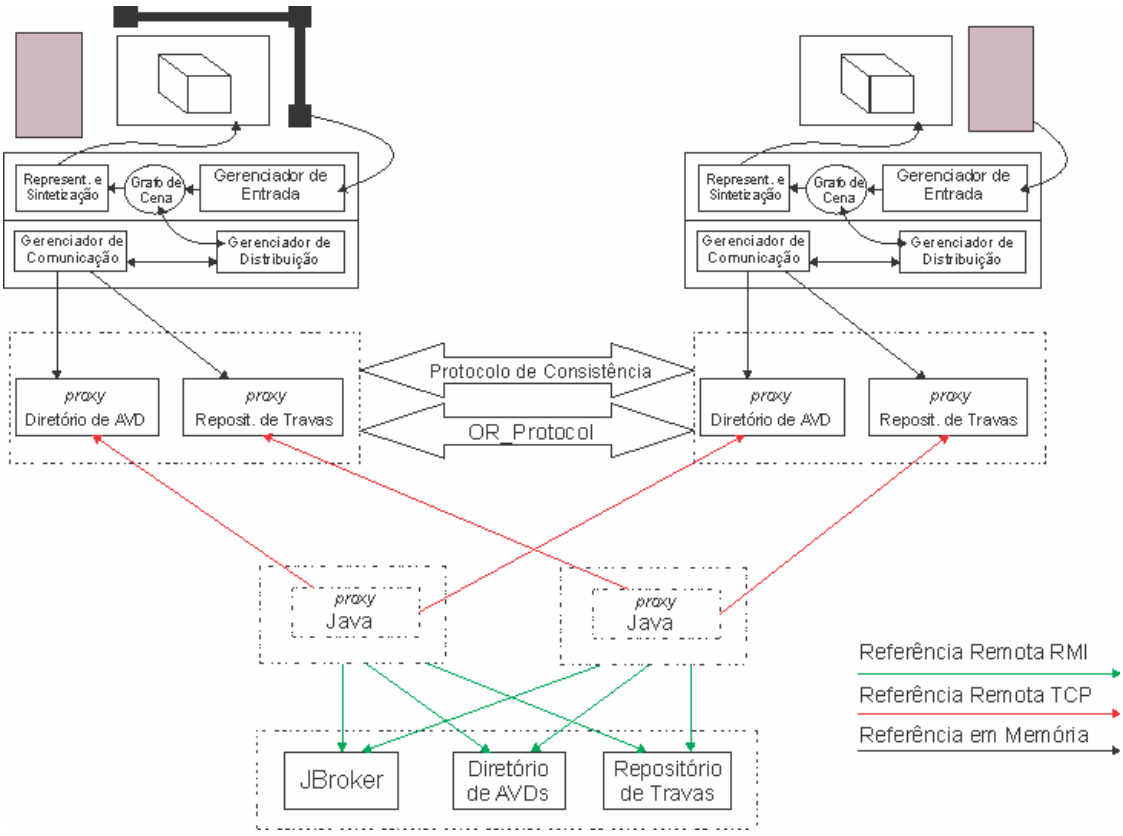

Figura 8.1 - Visão geral da arquitetura proposta por Baptista [BAPTISTA, 2004].

Baptista mostrou a necessidade deste intermédio de tecnologias de *proxys* nas comunicações, para que fosse possível que o AVC codificado por meio da OpenGL e C/C++ utilizasse, sem problemas, os serviços de diretórios distribuídos e Repositório de Travas oferecidos pela plataforma JAMP, que são codificados sob a linguagem JAVA.

O AVC concebido sob a arquitetura de Baptista utiliza a plataforma JAMP para distribuição dos eventos de interação entre clientes que o acessam. Desse modo, a utilização da plataforma JAMP para a distribuição já está codificada, o que é desejável de acordo com a estrutura idealizada no capítulo 6 deste trabalho.

Apesar do gerenciamento de protocolos de distribuição oferecidos pela plataforma JAMP já estarem codificados no AVC de testes, este não é acessível através de dispositivos com suporte a multiprojeção, como as cavernas digitais e os aglomerados gráficos que as gerenciam.

8.2 AVC acessível através de Cavernas Digitais

Para que o estudo de Baptista se enquadre nos moldes desejados, algumas extensões tiveram de ser efetuadas. Entre elas, codificações clientes ao AVC de testes preparados para execução em aglomerados gráficos tiveram de ser concebidas. Estes novos clientes codificados receberam ajustes de câmera limitados à sua porção da imagem.

Diante do aglomerado gráfico, cada nó do mesmo foi preparado para executar uma instância do processo cliente modificado. Dessa forma, a multiprojeção provida pela caverna digital unifica novamente as imagens, provendo maior sensação de imersão ao ambiente.

Com o sincronismo de interações já disponibilizado pela plataforma JAMP e os processos clientes modificados para as corretas porções da imagem, o acesso ao AVD de testes através de dispositivos com suporte a multiprojeção, foram concebidos.

Na Figura 8.2 pode ser visualizado o aglomerado gráfico do laboratório LAVIIC acessando o ambiente de testes.

Figura 8.2 - Aglomerado gráfico utilizado para testes

Apesar da concepção do acesso ao AVC através de cavernas digitais, foi verificado que ao aumentar a aproximação (*zoom*) dentro da caverna digital, inviabilizava-se a visualização por dispositivos que acessam o ambiente sobre projeção única.

Com altos valores de aproximação na caverna digital sendo replicados no ambiente, a projeção em dispositivos simples, tais como arquiteturas PC, excedem limites de tela não permitindo a identificação da imagem projetada.

Assim, a integração do arcabouço LibGlass para gerenciamento da caverna digital junto à codificação do AVC fechou com as especificações idealizadas e solucionou a característica de exploração dos limites de aproximação da imagem dentro das cavernas digitais.

Através dos recursos providos pela LibGlass adicionados ao AVD de testes a capacidade de distribuir interações somente entre os nós do aglomerado gráfico foi obtida, dessa forma a aproximação efetuada na caverna é refletida por todos os nós que participam da mesma multiprojeção, não afetando clientes conectados sobre dispositivos de projeção única.

A Figura 8.3 demonstra o estudo de caso estendido com recursos da LibGlass em execução. Nesta Figura, pode ser visualizada a multiprojeção provida pela caverna digital (ao fundo) e também dois computadores portáteis acessando o ambiente por meio de projeções únicas, todos com diferentes valores de aproximação da imagem (*zoom*).

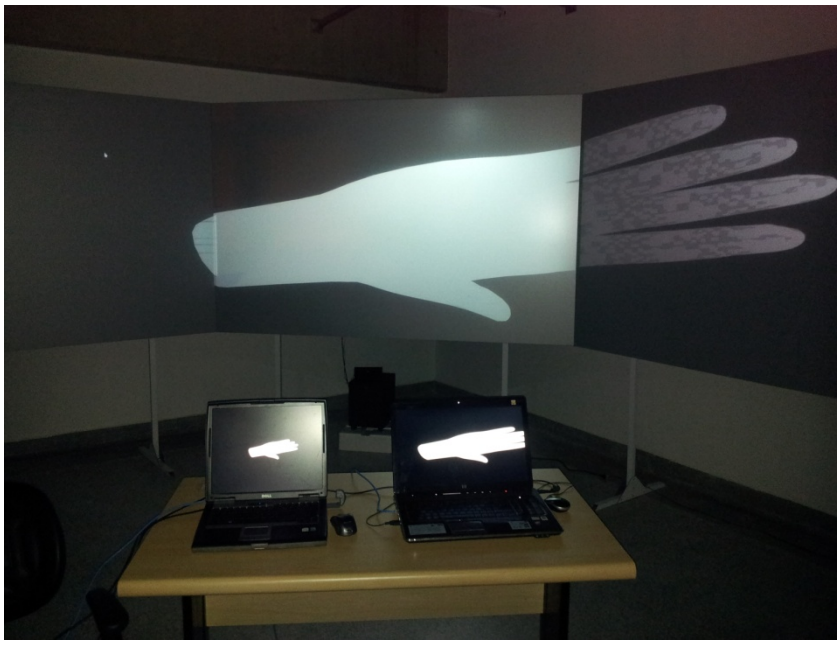

Figura 8.3 - Clientes compartilhando um mesmo AVC com diferentes valores de aproximação (*zoom*).

Com o estudo de caso executando de acordo com as especificações idealizadas, utilizando plataforma JAMP na distribuição e LibGlass no gerenciamento do aglomerado gráfico, passa-se então a um novo tópico, onde a integração do estudo de caso com a ferramenta JMonitor é descrita.

8.3 Adaptação do estudo de caso para uso de JMonitor

Como relatado no capítulo 6, os desafios de adquirir e disponibilizar as informações a serem monitoradas pela ferramenta JMonitor estão vinculados a concepção dos sistemas distribuídos a serem monitorados.

Durante a concepção dos sistemas AVDs ou AVCs à serem monitorados, processos clientes periodicamente terão de enviar informações requisitadas ao processo servidor. O processo servidor por sua vez terá de criar um documento para o serviço de monitoramento. Este documento disponibilizará suas informações, bem como as informações provenientes de todos os seus clientes. Este documento deverá ser também disponibilizado periodicamente com valores atualizados.

Para que fosse possível o monitoramento do AVC de testes, o mesmo teve que receber extensões, conforme demonstrado na Figura 8.4. Na mesma Figura,

pode ser observado que o código que representa os clientes do AVC de testes recebeu três extensões: (*i*) coleta de Dados; (*ii*) geração do arquivo XML e; (*iii*) envio do arquivo XML ao processo servidor.

O processo servidor, por sua vez, recebeu também três extensões: (*i*) código preparado para receber arquivos XML enviados pelos clientes; (*ii*) coleta de dados do processo servidor e; (*iii*) junção dos dados de todos os clientes com os dados do processo servidor em um mesmo arquivo XML final gerado. Este XML já está preparado para ser processado pela ferramenta JMonitor.

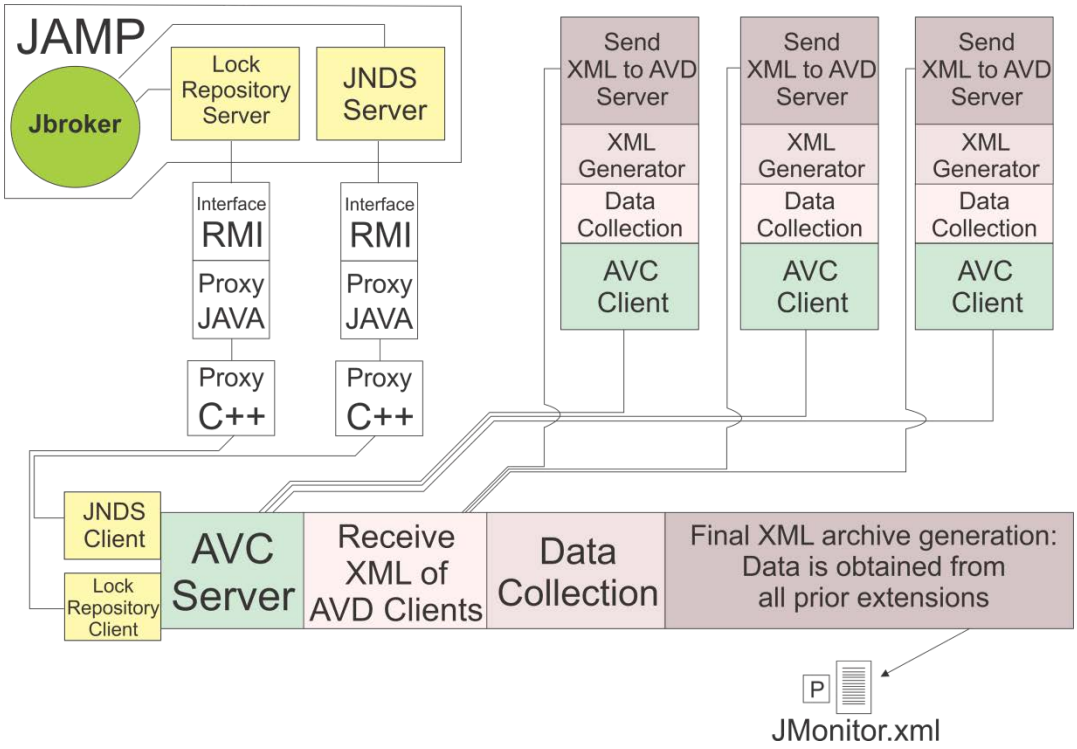

Figura 8.4 - Arquitetura do AVC de testes pronta para enviar dados à ferramenta JMonitor.

Com foco no processo cliente, codificou-se as extensões por meio de uma *thread* que periodicamente executa as três etapas mantendo sempre a mesma sequência: coleta de dados, geração do arquivo XML e envio do arquivo XML ao processo servidor

A extensão referente à coleta de dados foi estruturada de forma a armazená-los temporariamente como pares nome e valor. Os dados são mantidos armazenados até que todos sejam coletados e enviados para a próxima extensão.

Na próxima extensão o arquivo XML que descreve os dados do processo em execução será gerado.

Existem diversas formas de se coletar dados e armazená-los em pares nome e valor. Essa variedade se deve às diferentes linguagens de programação e sistemas operacionais existentes na atualidade.

No estudo de caso desse trabalho optou-se por utilizar a tecnologia SNMP [SCHIMID, 2007] e comandos do próprio sistema operacional Ubuntu [UBUNTU, 2012] para a coleta de dados referentes à utilização e disponibilidade dos recursos computacionais envolvidos no ambiente.

As coletas de dados referentes ao acompanhamento de valores com fins de depuração, foram codificadas nas próprias linguagens que as declaram e foram também registradas sobre pares "nome e valor". Esses pares "nome e valor" representam por exemplo, interações de usuários e são valores críticos a um ambiente sincronizado.

Como ilustrado na Figura 8.4 após ser processada à extensão de coleta de dados, esses dados são também descritos no arquivo XML gerado na próxima extensão. Seguindo a sequência, a extensão de geração do arquivo XML é executada. Nesta, os dados que foram coletados e temporariamente armazenados em pares nome e valor, são descritos em documentos XML.

As tecnologias utilizadas para a geração dos documentos também pode variar de acordo com a linguagem de programação e gosto dos desenvolvedores dos sistemas AVDs ou AVCs. Para o estudo de caso desse trabalho foi utilizado a tecnologia libxml [LIBXML, 2012].

Com o arquivo XML que descreve os dados coletados gerado, a *thread* executa a última extensão dos processos clientes. Essa última extensão processa o envio do arquivo XML ao servidor do AVC.

O envio do arquivo XML ao servidor do AVC foi codificado através de um cliente *socket* que conecta um respectivo servidor *socket* presente na primeira extensão do processo servidor do AVC de testes.

Com foco no processo servidor do AVC de testes, codificou-se as extensões também por meio de *thread*, da mesma forma que nos processos clientes. Uma *thread* periodicamente executa três etapas mantendo sempre a mesma sequência: código preparado para receber arquivos XML enviados pelos clientes, coleta de

dados do processo servidor e junção dos dados de todos os clientes em um mesmo arquivo XML final gerado.

A extensão referente ao recebimento de dados dos clientes foi codificada por meio de um servidor *socket* e uma *thread* interna da extensão, que é mantida aguardando os arquivos XML de seus respectivos clientes.

Quando a *thread* que processa sequencialmente as extensões executa esta etapa, esta *thread* coleta todos os arquivos XML já disponíveis no código do servidor do AVC de testes. Em outras palavras, coleta os arquivos XML já recebidos e disponibilizados pela *thread* interna da extensão.

Seguindo a próxima extensão, a coleta de dados do processo servidor do AVC de testes é executada. A coleta de dados não difere à dos processos clientes. Da mesma forma, dados são consultados por meio da tecnologia SNMP ou comandos do próprio sistema operacional e temporariamente armazenados em pares "nome e valor", até que seja executada a próxima extensão do processo servidor do AVC.

Na próxima e última extensão do servidor do AVC de testes a ser executada, é realizada a junção dos dados coletados do processo servidor com os dados de todos os respectivos clientes conectados ao mesmo. Essa junção é efetuada por meio da geração de um arquivo XML final, preparado para ser processado pela ferramenta JMonitor.

Com todas as extensões descritas e codificadas, o AVC de testes está pronto para utilizar a ferramenta JMonitor. Pretende-se utilizar a ferramenta para facilitar a visualização de informações sobre todos os processos em execução, que mesmo estando distribuídos por diferentes redes, compõem um mesmo ambiente virtual distribuído e ou colaborativo.

O próximo tópico provê detalhes de como foram integrados o AVC de testes com os diferentes módulos da ferramenta JMonitor.

8.4 Estudo de caso e JMonitor: integração

Para integrar o estudo de caso com a ferramenta JMonitor, a estrutura do AVC de testes é apresentada de maneira que os dados do AVC são coletados pela

ferramenta. Uma visão geral da estrutura com a integração pode ser visualizada na Figura 8.5.

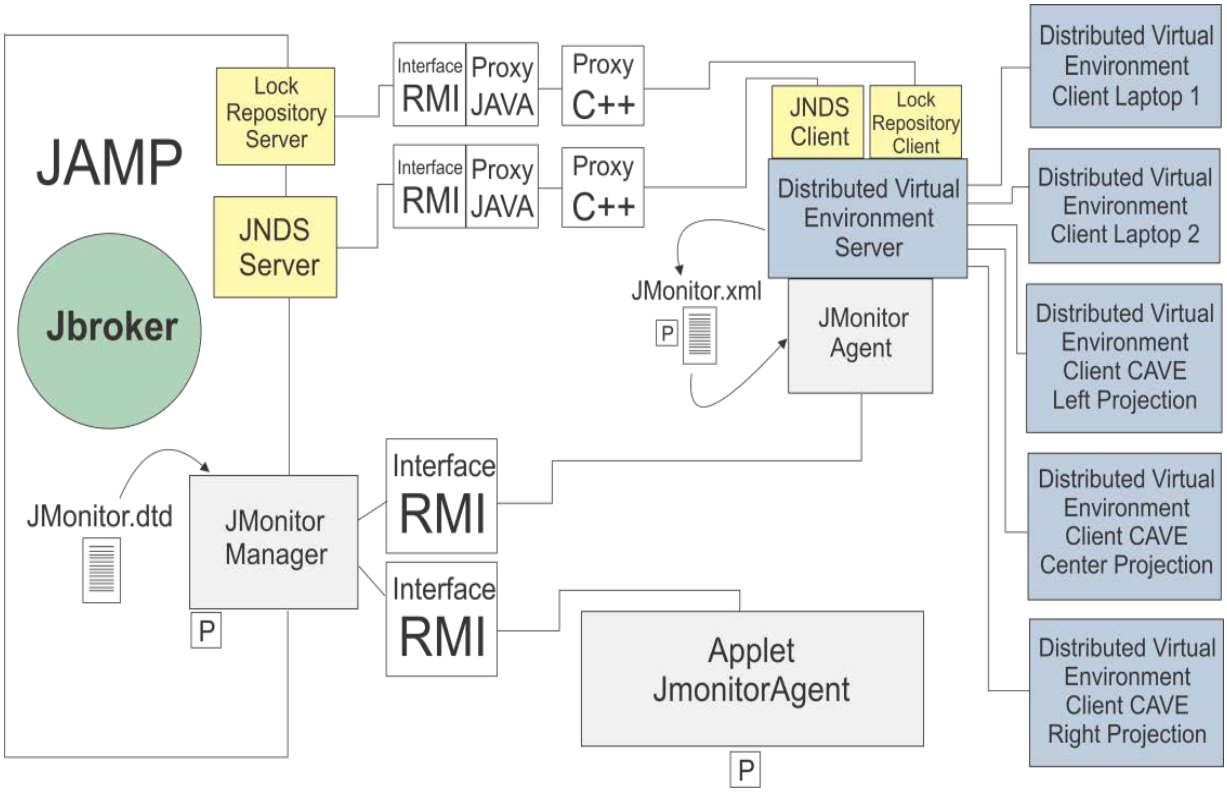

Figura 8.5 - Estrutura integrando estudo de caso com a ferramenta JMonitor.

Para que se torne funcional a estrutura apresentada, o gerenciador de monitoramento disponibilizado na camada de serviços da plataforma JAMP é colocado em execução.

O processo gerenciador de monitoramento se mantém preparado para executar quatro funções essenciais: (*i*) receber arquivos descritores de dados através de chamadas RMI; (*ii*) validar os arquivos recebidos comparando-os com as especificações descritas em JMonitor.dtd; (*iii*) realizar a etapa de transformação dos dados e; (*iv*) disponibilizar esses dados ao módulo de visualização.

A checagem de validade dos dados, comparando o tempo em que eles foram recebidos com o tempo corrente, é de responsabilidade de uma *thread* interna do módulo gerenciador.

Complementando a estrutura, foi executado o aplicativo que representa o módulo agente de monitoramento junto com o processo servidor do AVC de testes. O processo servidor do AVC de testes, gera e atualiza periodicamente o arquivo

JMonitor.xml contendo dados do mesmo e de todos os respectivos clientes conectados a ele.

O agente da ferramenta JMonitor foi programado de forma a identificar qualquer atualização do arquivo JMonitor.xml. Mesmo se o arquivo for recriado com dados idênticos o agente detecta e o envia ao módulo gerenciador, que por sua vez atualiza pelo menos o tempo de vida das informações. Neste caso, a atualização é realizada, apesar de valores iguais ao arquivo já anteriormente enviado.

Para finalizar a estrutura apresentada, qualquer computador da rede que possa se comunicar com o JBroker pode também obter acesso ao *applet* de monitoramento. A Figura 8.6 mostra o *applet* de monitoramento em execução disponibilizando informações do AVC de testes.

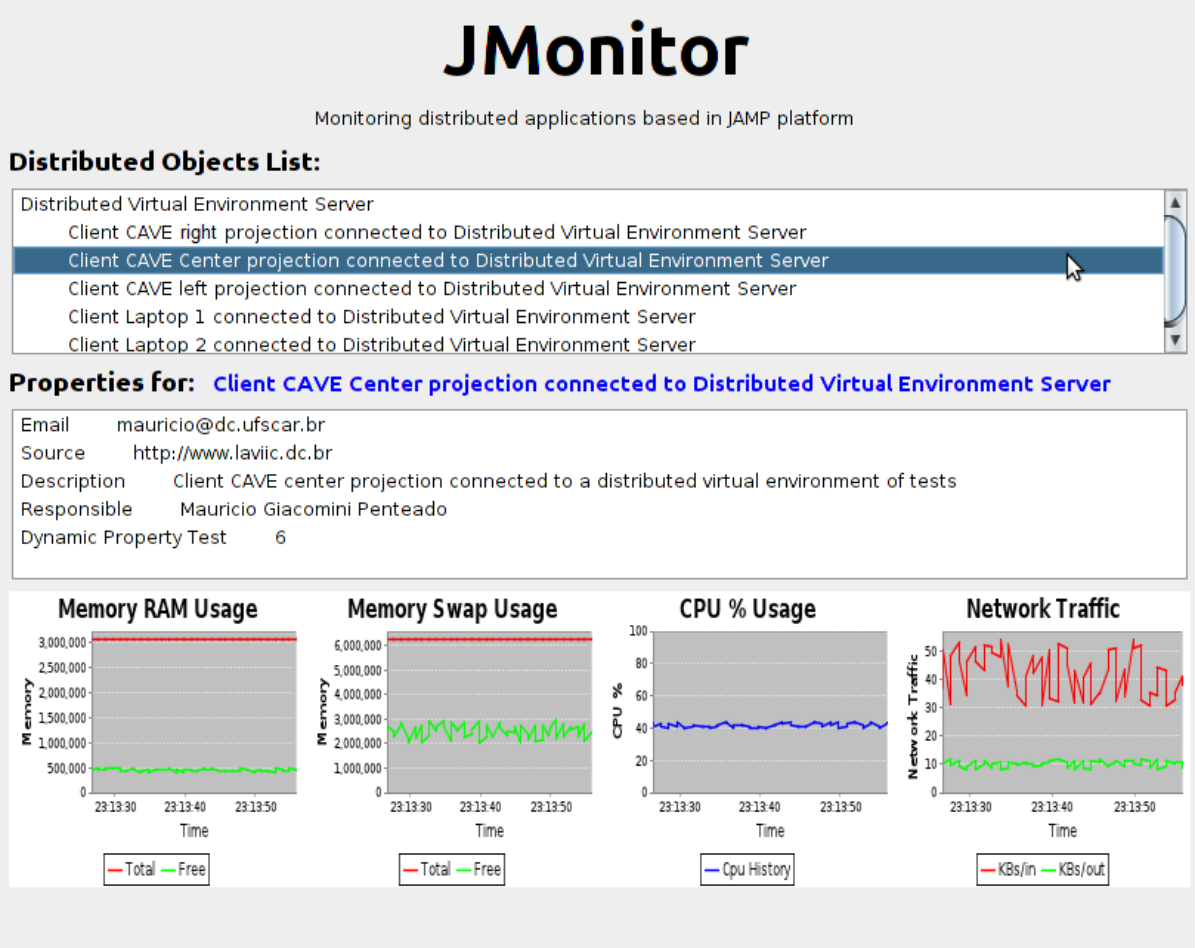

Figura 8.6 -Applet de monitoramento disponibilizando dados sobre o AVC de testes.

Também pode ser observado na Figura 8.6, que entre a lista de objetos distribuídos que fazem parte do AVC de testes, o cliente responsável pela projeção

central do aglomerado gráfico está selecionado. Esta seleção faz com que os dados da lista de propriedades e os geradores de gráficos executem dados providos por este determinado cliente.

Pode-se alternar entre visualização de dados de clientes ou mesmo do servidor do AVC de testes conforme requisitado pela diferente seleção entre os processos que compõem o ambiente.

O *applet* de monitoramento, através de uma *thread* interna ao mesmo, periodicamente executa chamadas RMI ao módulo gerenciador, carregando somente informações de sistemas AVDs ou AVCs válidas para terem suas propriedades reportadas.

Dessa forma, se um dos processos de um sistema AVD ou AVC monitorado tem execução interrompida ou se desconecta do ambiente parando de enviar suas atualizações ao sistema de monitoramento, pode-se facilmente identificar, pois o *applet* para de reportar esse processo em sua lista de processos selecionáveis.

Este capítulo apresentou um estudo de caso de AVC de testes concebido por meio da estrutura de AVD com mediador *broker*, acessível através de dispositivos de projeção única e de multiprojeções. E apresentou também a integração entre o AVC de testes com a ferramenta de monitoramento implementada (JMonitor), fechando com a proposta do trabalho.

A ferramenta JMonitor está disponibilizada como extensão à plataforma JAMP. Situada na camada de serviços da plataforma, JMonitor permite realizar o monitoramento de sistemas similares aos AVDs propostos nesse trabalho ou quaisquer sistemas distribuídos que descrevam dados a serem monitorados sob as especificações da ferramenta.

O próximo capítulo é destinado a conclusões e considerações finais dessa dissertação.

Capítulo 9

CONCLUSÕES DO TRABALHO

Neste capítulo são apresentados os principais resultados da proposta de solução do problema estabelecido inicialmente e quais foram as contribuições do trabalho para a área acadêmica.

São apresentados também trabalhos que podem ser realizados com base na pesquisa efetuada nessa dissertação, os quais podem corresponder á continuidade do tema abordado ou a um novo item de pesquisa identificado a partir deste.

9.1 Conclusões

A partir da proposta de um cenário introdutório, foram apresentadas as tecnologias JAMP, LibGlass, Diretivas para Computação Móvel e Diretivas para TV Digital; as quais juntas compõem o arcabouço concebido pelo projeto XPTA.lab [XPTA, 2011].

O arcabouço concebido no projeto XPTA.lab foi projetado para ser auxiliar na concepção de sistemas AVDs tais como sistemas AVCs. Esses AVDs, acessíveis por diferentes tipos de dispositivos com suporte a projeção única e multiprojeção de imagens.

De modo a facilitar o desenvolvimento dos AVDs, o arcabouço do projeto XPTA.lab provê recursos auxiliares ao desenvolvimento desses aplicativos. Esses recursos permitem sincronizar e explorar características intrínsecas à geração de imagens de cada um dos dispositivos que podem acessá-los.

Diante do cenário proposto foram expostas as limitações desse trabalho, as quais relataram que o mesmo não abordaria o acesso aos AVDs por meio das tecnologias de computação móvel ou TV Digital.

Partindo do cenário com AVDs acessíveis por dispositivos de projeção única e multiprojeção, foram apresentados os objetivos dessa dissertação, os quais possuem foco na fundamentação de uma ferramenta de monitoramento, capaz de disponibilizar dados sobre o acesso, quantidade de participantes, dados para depuração e utilização de recursos computacionais por cada um dos múltiplos processos que compõem esses AVDs.

A concepção de um protótipo da ferramenta fundamentada e a prova desta em estudo de caso, também fazem parte dos objetivos deste trabalho.

Foi efetuada uma revisão bibliográfica sobre diferentes trabalhos que, direta ou indiretamente, utilizam as técnicas de coleta, transformação e visualização de dados, para realizar o monitoramento de diferentes propriedades agregadas a processos, recursos computacionais ou, mais genericamente, dispositivos em rede.

Uma revisão teórica de como poderia ser concebido um sistema similar aos AVDs introdutórios, adicionando-se o uso de JAMP e LibGlass nessa concepção, foi efetuada e a partir desta revisão, a ferramenta de monitoramento foi fundamentada.

Após a fundamentação teórica, foi apresentada a implementação de uma ferramenta de monitoramento que segue os moldes propostos, denominada JMonitor. Concebida através da infraestrutura oferecida pela plataforma JAMP, JMonitor pode ser utilizada por qualquer sistema distribuído que gere informações demarcadas através de documentos XML. Isto permite que os sistemas a serem monitorados tenham independência de plataforma ou linguagem de programação.

A ferramenta JMonitor foi concebida com intenção de facilitar acesso a diferentes propriedades de sistemas AVDs e pode ser útil na identificação de comportamentos anormais no uso dos recursos computacionais envolvidos, no acompanhamento de valores de variáveis em fases de desenvolvimento e também na disseminação de informações dos AVDs pela rede.

Apesar da ferramenta JMonitor implementada, esse trabalho possui ênfase na ideia de como estruturar a concepção de outras ferramentas de monitoramento similares. Partindo dessa ideia, o protótipo poderia ter tido diferentes focos, como: explorar características de desempenho, identificar vazamentos de memória, regiões

críticas de comunicação, gargalos de comunicação, entre outros. Todos, também presentes nos diversos processos comunicantes que compõem os AVDs.

Esses diferentes focos no monitoramento poderiam ter sido atingidos com pequenas modificações nos dados coletados, com diferentes cálculos de métricas na fase de transformação e apresentação. Apesar dos diferentes focos que poderiam ter sido aplicados, essas modificações não afetariam a estrutura da ferramenta de monitoramento fundamentada, mantendo a estrutura idealizada para a mesma.

Para confirmar a funcionalidade da ferramenta JMonitor, a mesma foi preparada para monitorar um AVD de testes. Nesta etapa, um protótipo resultante do trabalho de Baptista [BAPTISTA, 2004] foi utilizado. O mesmo teve que receber extensões que o adequaram para o acesso por meio de cavernas digitais.

Também foram estendidas ao protótipo as etapas computacionais de coleta de dados e a geração de documentos descritores desses dados, adequados de acordo com as especificações da ferramenta JMonitor.

Como estudo de caso deste trabalho, foi apresentado a ferramenta JMonitor integrada com o protótipo com as devidas extensões efetuadas.

O *applet* de visualização identificou valores de variáveis que mudam dinamicamente entre os processos comunicantes, o que pode facilitar a identificação de erros de programação em fase de desenvolvimento.

A visualização e o acompanhamento de valores de variáveis codificadas em diferentes linguagens também foi efetuada sem problemas, fechando com os requisitos de independência de plataforma e linguagem de programação.

Na visualização quanto ao número de participantes de um mesmo AVD, a perda de comunicação de algum cliente é facilmente observada utilizando o JMonitor. Nesse caso, o tempo de vida das informações que ficam sem atualizações não demora a expirar. Assim, o processo com funcionamento interrompido se torna indisponível no *applet* de monitoramento, o que é bastante intuitivo.

Os geradores de gráficos da ferramenta mostraram gráficos similares ao real uso de recursos computacionais utilizados nos computadores remotos. Essa avaliação foi efetuada por meio de comparações com resultados de monitores de recursos computacionais locais aos computadores. Os resultados dos monitores locais, quanto comparados aos dados providos por JMonitor tiveram similaridade.

9.2 Contribuições

Foram propiciados nesse trabalho:

- Revisão bibliográfica de ferramentas de monitoramento de dados computacionais que direta ou indiretamente utilizam as técnicas de Foster [FOSTER, 1995], estruturadas com fases de coleta, transformação e visualização de dados.
- Estudo de como poderiam ser criados sistemas de AVDs tais como sistemas AVCs acessíveis à cavernas digitais utilizando a plataforma JAMP e o arcabouço LibGlass.
- Estudo de como poderia ser concebida uma ferramenta de monitoramento para o acompanhamento de dados computacionais de sistemas de AVDs ou AVCs.
- Extensão à plataforma JAMP com a ferramenta JMonitor
- Implementação de um protótipo da ferramenta de monitoramento fundamentada.
- Estudo de caso utilizando a ferramenta de monitoramento protótipo implementada.
- Artigo publicado no SMC 2012. The 2012 IEEE International Conference on Systems, Man, and Cybernetics.

9.3 Trabalhos Futuros

Os itens identificados como continuidades a este trabalho ou como nova pesquisa identificada a partir deste, são:

- Extensão da ferramenta de monitoramento fundamentada para que a mesma seja capaz de também monitorar plataformas da computação móvel e plataformas da TV Digital.
- Fundamentação e concepção de agentes para a ferramenta de monitoramento de AVDs capazes de também gerenciar [os] computadores remotos.

REFERÊNCIAS

AL BELUSHI, W.; BAGHDADI, Y.; , "An Approach to Wrap Legacy Applications into Web Services," Service Systems and Service Management, 2007 International Conference on , vol., no., pp.1-6, 9-11 June 2007

BADGER, M.. Zenos Core Network and System Monitoring, ed. PACKT, 261 pags. 2008.

BAPTISTA, B. A. D. Projeto do Subsistema de Comunicação e Distribuição e da Camada de Serviços da Arquitetura OpenReality para Suporte à Criação de Aplicações de Visualização Distribuída. Dissertação de Mestrado. Universidade Federal de São Carlos, São Carlos, SP, 2004.

BARBOSA, L. A. F. Uma arquitetura de Conectividade para Dispositivos Móveis na Plataforma JAMP. Dissertação de Mestrado. Universidade Federal de São Carlos, São Carlos, SP, 2006.

BATRA, U; DAHIYA, D; BHARDWAJ, S. Analytical Comparison of Distributed Object Components. In: 7th WSEAS International Conference on Applied Computer and Applied Computational. Department of CSE&IT, Institute of Technology and Management, Gurgaon, India, 2008.

BELLEZI, M. A.. Estudos de Mecanismos de Representação e Sintetização em Ambientes Virtuais Distribuídos: Projeto Inicial do Framework OpenReality. Dissertação de Mestrado. Universidade Federal de São Carlos, São Carlos, SP, 2003.

BLAIR, G. S. Na Architecture for Next Generation Middleware. In: ACM MIDDLEWARE COMFERENCE. Lake District, England. 1998.

BOCCARDO, D. R.; NAKASHIMA, H. J.; JÚNIOR, J. N. F.; CASAGRANDE, L. dos S.. Medição de Desempenho de Sistemas Paralelos - Estendendo o Paradyn -. Projeto Final de Curso. Departamento de Ciências da Computação e Estatística. Universidade Estadual Paulista - UNESP. São José do Rio Preto, SP. 2007.

BORSOI, B. T.; SCHULTS, R. E. de O. Estudo Comparativo entre CORBA e Java RMI. Congresso Anual de Tecnologia de Informação. FGV-EAESP Escola de Administração de Empresas de São Paulo da Fundação Getulio Vargas, Centro de Informática Aplicada. São Paulo, Brasil, 2004.

BOTEGA, L. C.. CRUVINEL, P. E.. Realidade Virtual: Histórico, Conceitos e Dispositivos. In: COSTA, Rosa Maria E. M. RIBEIRO, Marcos Wagner S. – Organizadores – Porto Alegre - RS. Aplicações de Realidade Virtual e Aumentada. Editora SBC - Sociedade Brasileira de Computação, Porto Alegre, 2009.

BRAGA, Mariluci. Realidade Virtual e Educação. Revista de Biologia e Ciências da Terra. Volume 1 – Número 1, 2001.

BRUGNARA, T.; CECHIN, S. L.; LISBÔA, M. L. B.. Uma ferramenta para análise de desempenho de aplicações distribuídas. XXV Congresso da Sociedade Brasileira de Computação. Instituto de Informática. Universidade Federal do Rio Grande do Sul, UFRGS, Porto Alegre, RS. 2005.

BURKE, R.; MONSON H.. Enterprise JavaBeans 3.0. ed. O'Reilly. 768 pag. 2006.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H., SOMMERLAND, P; STAL, M. Pattern-Oriented Software Architecture. A System of Patterns, John Wiley & Sons, 1996.

CANTRILL, B. M.; SHAPIRO, M. W.. Dynamic instrumentation of production systems. Proceedings of the annual conference on USENIX Annual Technical Conference. Boston, MA, USENIX Association: 2-2. 2004.

CARDOSO, A.; TEIXEIRA, C. A. C.; LAMOUNIER JR., E.. VRML - a Web em 3D. ed. SBC - Porto Alegre, RS. 2003.

DIAS, D.R.C.; LA MARCA, A.F.; MOIA VIEIRA, A.; NETO, M.P.; BREGA, J.R.F.; de PAIVA GUIMARÃES, M.; LAURIS, J.R.P.; , "Dental arches multi-projection system with semantic descriptions," Virtual Systems and Multimedia (VSMM), 2010 16th.

CHEN, J.; GROTTKE, S.; SABLATNIG, J.; SEILER, R.; WOLISZ, A.. "Scalability of a distributed virtual environment based on a structured peer-to-peer architecture," Communication Systems and Networks (COMSNETS), 2011 Third International Conference on, vol., no., pp.1-8, 4-8 Jan. 2011.

CORBA - Common Object Request Broker Architecture - Especificações. Disponível em: http://www.omg.org/spec/CORBA.
Acessado em: fevereiro de 2011.

CORBAHISTORY - History Of CORBA. Disponível em:
http://www.omg.org/gettingstarted/history_of_corba.htm
Acessado em: maio de 2011.

CORRÊA, M. F. Arquitetura de Alto Desempenho Para Integração de Ambientes Interativos e Imersivos Remotos para Visualização Molecular. Tese de Doutorado. Universidade Federal de São Carlos, São Carlos, SP. 2010.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. Sistemas distribuídos. Conceitos e Projetos. Porto Alegre: Bookman, 2007.

DASH, A; DESMKY, B. Software Transactional Distributed Shared Memory. In Proceedings of the Symposium on Principles and Practice of Parallel Programming, paginas 297-288, New York/USA. 2009.

DUARTE, V. A.. Uma Arquitetura para a Monitorização de Computações paralelas e Distribuídas. Tese de Doutorado. Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia. Lisboa, Portugal. 2003.

EMMERICK, W. Software Engineering and Middleware: a Roadmap.The Future of Software Engineering. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 22., Limerick, Ireland, 2000.

FAYAD, M. E.; SCHMIDT, D. C. Object-oriented Application frameworks. Communications of the ACM, Vol. 40, 10 p., 1997.

FENLASON, J.; Stallman, R.. GNU gprof: The GNU Profiler, September 1997.

FLYNN, M. J.; RUDD, K. W. Parallel Architectures. ACM Computing Survey, 1996.

FOSTER, I.. Designing and Building Parallel Programs. Addison-Wesley,. 381 pag. 1995

GEYER, C. F. R. Arquitetura para um Ambiente de Grade Computacional Direcionado às Aplicações Distribuídas Móveis e Consistentes do Contexto da Computação Pervasiva. Universidade Federal do Rio Grande do Sul, Porto Alegre, RS. 2004.

GNECCO, B.B.; GUIMARÃES, M.P.; ZUFFO, M.K.. Um framework flexível e transparente para computação distribuída de alto desempenho. Simpósio Brasileiro de Realidade Virtual. Ribeirão Preto. 2003.

GUIMARÃES, M. P. Um Ambiente para o Desenvolvimento de Aplicações de Realidade Virtual Baseadas em Aglomerados Gráficos. Tese de Doutorado. Escola Politécnica da Universidade de São Paulo. USP. São Paulo, SP, 2004.

GUIMARÃES, M. P. Projeto e Implementação de Suporte para Trabalho Cooperativo na Plataforma JAMP. Dissertação de Mestrado. Universidade Federal de São Carlos, São Carlos, SP, 2000.

GUIMARÃES, M. P.; GNECCO, B. B. http://www.lsi.usp.br/~paiva/glass/glass.htm. Acessado em: janeiro/2011

GONÇALVEZ, L. C. Projeto e Implementação de Suporte para Comércio Eletrônico na Plataforma JAMP. Dissertação de Mestrado. Universidade Federal de São Carlos, São Carlos, SP, 2000.

IBM - IBM Forums. Disponível em:
http://publib.boulder.ibm.com/infocenter/realtime/v2r0/index.jsp?topic=/com.ibm.softrt.aix64.doc/diag/tools/heapdump.html
Acesso: junho de 2011.

KARUNAMOORTHY, D.; DEVINUWARA, N.; , "Monitoring & manging dynamic distributed systems," Electronics, Circuits and Systems, 2005. ICECS 2005. 12th IEEE International Conference on , vol., no., pp.1-4, 11-14 Dec. 2005

KERGOMMEAUX, J. C.. Monitoring parallel programs for performance tuning in cluster environments. Parallel program development for cluster computing, Nova Science Publishers, Inc.: 131-150. 2001.

KIRNER, C., TORI, R. Fundamentos de Realidade Aumentada. In: Romero Tori; Claudio Kirner. (Org.). Fundamentos e Tecnologia de Realidade Virtual e Aumentada. Porto Alegre: SBC, 2006, v.1 , p. 2-21.

KOCJAN, W.. Learning NAGIOS 3.0, ed. PACKT, 300 pags. 2008.

KRETSCHEK, M. A.. Panalyser, uma ferramenta de baixo impacto para medição de utilização de recursos do sistema operacional linux. Dissertação de Mestrado. Universidade Federal do Paraná. Curitiba, PR. 2002.

LEE, J.; QUY, P.; KIM, J.; KANG, L.; SEO, A.; Kim, H.; , "A Collaborative Virtual Reality Environment for Molecular Biology," Ubiquitous Virtual Reality, 2009. ISUVR '09. International Symposium on , vol., no., pp.68-71, 8-11 July 2009

LIBGLASS - Homepage - Disponível em:
http://LibGlass.sourceforge.net/
Acessado em: janeiro/2011

LIBXML - The XML C parser and toolkit of Gnome - Disponível em:
http://xmlsoft.org/
Acessado em: maio/2012

MARQUES, C. F. dos S.. Avaliação do Modelo de Componentes CORBA. Dissertação de Mestrado. Universidade Católica de Santos. Santos, SP. 2006.

MATTSON, M. Evolution and Composition Object-Oriented Frameworks, PhD Thesis, University of Karlskrona, Department of Software Engineering and Computer Science, Ronneby, Sweden 2000.

MENDONÇA, S. P. Implementação de Serviço Multicast na Plataforma JAMP para Suporte a Aplicações Colaborativas. Dissertação de Mestrado. Universidade Federal de São Carlos, São Carlos, SP, 2000.

MILLER, B. P.; CALLAGHAN, M. D.. The Paradyn Parallel Performance Measurement Tool. Computer 28(11): 37-46. 1995.

MPI - Message Passing Interface Specifications Disponível em:
http://www.mcs.anl.gov/research/projects/mpi/
Acessado em: março/2011

NARDI, A. R.. Componentes CORBA. Dissertação de Mestrado. Instituto de Matemática e Estatística da Universidade de São Paulo. São Paulo, SP. 2003.

NET - .Net framework 4.0. Disponível em:
http://www.microsoft.com/downloads/en/details.aspx?displaylang=en&FamilyID=0a391abd-25c1-4fc0-919f-b21f31ab88b7
Acessado em: maio de 2011.

NETTO, A.V; MACHADO, L.S.; OLIVEIRA, M.C.F. Realidade Virtual: Definições, Dispositivos e Aplicações. Tutorial. Revista Eletrônica de Iniciação Científica da SBC. Ano II, v.II, n.2, ISSN 1519-8219. Março de 2002.

OLUPS, R.. Zabbix 1.8 Network Monitoring, ed. PACKT, 410 pags. 2010.

OPENREALITY, 2012. Disponível em:
http://sourceforge.net/projects/openreality/
Acessado em: março de 2012.

OSÓRIO, F.S.; MUSSE, S. R.; SANTOS, C. T.; HEINEN, F.; BRAUN, A.; SILVA, A. T. Ambientes Virtuais Interativos e Inteligentes: Fundamentos, Implementação e Aplicações Práticas. Tutorial in JAI – Jornada de Atualização em Informática/SBC. Salvador, Bahia, 2004.

PARADYN - Homepage. Disponível em:
http://www.paradyn.org
Acessado em maio/2011

PENG, J.; CAO, J.; , "ECA rule-based conFigurable frame of distributed system monitoring," Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on , vol.1, no., pp.674-677, 10-12 Dec. 2010.

PIOLA, T. de F.. *Tracing* de Aplicações Paralelas com Informações de Alto Nível de Abstração. Tese de Doutorado. Instituto de Física de São Carlos da Universidade de São Paulo. USP. São Carlos, SP, 2007.

PITANGA, M. Construindo Supercomputadores com Linux. 3a ed. Brasport Livros e Multimídia Ltda. pagina: 171. Rio de Janeiro, RJ. 2008.

PREE, W. Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995.

RAFE, V.; RAFEH, R.; FAKHRI, P.; ZANGARAKI, S.; , "Using MDA for Developing SOA-Based Applications," Computer Technology and Development, 2009. ICCTD '09. International Conference on , vol.1, no., pp.196-200, 13-15 Nov. 2009

REMOTING Microsoft .NET Remoting: A Technical Overview. Disponível em:
http://msdn.microsoft.com/en-us/library/ms973857.aspx
Acessado em maio/2011

RMI - RMI Tutorial. Disponível em:
http://docs.oracle.com/javase/tutorial/rmi/running.html
Acessado em maio/2011

RODRIGUES, L. C. R. ; KUBO, M. M. ; RODELLO, I. A. ; SEMENTILLE, A. C ; TORI, R. ; BREGA, J. R. F . Ambientes Virtuais Distribuidos e Compartilhados. In: Romero Tori;Claudio Kirner. (Org.). Fundamentos e Tecnologia de Realidade Virtual e Aumentada. Porto Alegre: SBC, v.1 , p. 60-78, 2006.

SCHIMID, C. R.. Desenvolvimento de gerência de SNMP para dispositivos de redes totalmente ópticas. Dissertação de Mestrado. Pontifícia Universidade Católica de Campinas. Campinas, SP. 2007.

SOUZA, L. F. H. Estudo de Modelos de Serviços para middleware e proposta de extensão à plataforma JAMP. Universidade Federal de São Carlos, São Carlos, SP. 2001.

STOCK, M. Technologies for Thin Client Architectures. Master Thesis in Computer Science. Department of Information Technology University of Zurich, Zurich, Switzerland. 2001.

TANENBAUM, A. S.; WOODHULL, A. S. Sistemas Operacionais: projeto e implementação. 3.ed. Porto Alegre: Ed. Bookman, 2008.

TANENBAUM(a), A. S.; STEEN, M. V.. Sistemas distribuídos. Princípios e paradigmas. 2a ed. Pearson Prentice Hall. São Paulo, 2007.

TANENBAUM(b), A. S.. Organização Estruturada de Computadores. 5a ed. LTC, Rio de Janeiro, RJ. 2007.

TAU - Tuning and Analysis Utilities. Disponível em:
http://www.cs.uoregon.edu/research/tau
Acessado em maio/2011

UBUNTU - Official Page. Distponível em:
http://www.ubuntu.com/
Acessado em maio/2012

URBAN, T.. Cacti 0.8 Beginner's Guide, ed. PACKT, 327 pags. 2011.

VALGRIND - Valgrind Homepage. Disponível em:
http://valgrind.org
Acessado em: janeiro/2011

VTUNE - Intel Tune V8. Disponível em:
http://www.intel.com
Acessado em maio/2011

VGIMG - [VALGRIND,2011][VTUNE, 2011][PARADYN, 2011][TAU,2011]

WHALEY, J. A portable sampling-based profiler for Java virtual machines. Proceedings of the ACM 2000 conference on Java Grande. San Francisco, California, United States, ACM: 78-87. 2000.

XPTA - XPTA.lab Homepage. Disponível em:
http://sourceforge.net/apps/mediawiki/xpta/index.php
Acessado em: janeiro/2011

XU, J.; XU, M.; , "A Performance Monitoring Tool for Predicting Degradation in Distributed Systems," Web Information Systems and Mining, 2009. WISM 2009. International Conference on , vol., no., pp.669-673, 7-8 Nov. 2009.

YU, B. H.; HUANG, Z.; CRANEFIELD, S; Purvis, M. Homeless and Homebased Lazy Release Consistecy Protocols on Distributed Shared Memory. In Proceedings of the Conference on Computer Science, paginas 117-123, Australia. 2004.

ANEXO

Este anexo descreve as extensões efetuadas no estudo de caso deste trabalho para utilização da ferramenta JMonitor.

No *header* OR_CommunicationManager.h pode ser observada a declaração dos métodos *consultaSNMP* e *consultaIfStat*, métodos auxiliares a coleta de dados. Nesse *header* pode ser observada também a herança da classe *Thread*, que vincula à *thread* de coleta de dados de forma periódica presente no estudo de caso. Esta thread é implementada no código OR_CommunicationManager.cpp.

OR_CommunicationManager.h:

```
class Monitoring : public Thread {
public:
    OR_CommunicationManager *orcm;
    Monitoring(OR_CommunicationManager *orcmPTR);
    string intToString(int number);
    char* substring(char *origem, int inicio, int quant);
    char* consultaSNMP(char *snmpCommand);
    char* consultaIfstat(char* ifstatCommand);
    void* run();
};
```

O código OR_CommunicationManager.cpp implementa a *thread* de coleta de dados vinculada a ferramenta JMonitor. Neste código é pode-se observar o principal método da *thread*, o método *run*.

OR_CommunicationManager.cpp

```
void* Monitoring::run() {
    // Início do laço infinito que processa a coleta de dados
    while (true) {

   // Taxas de entrada e saída de comunicação das interfaces de rede
char* chInterfacesTaxasOutTemp;
char* chInterfacesTaxasInTemp;

// ############################################################
// ## início do bloco de dados que utiliza o comando ifstat ##
// ## do Ubuntu para registrar as taxas de utilização da    ##
// ## interface de rede.                                    ##
// ############################################################

chInterfacesTaxasOutTemp = consultaIfstat("ifstat -i wlan0 1 1");
chInterfacesTaxasInTemp = (char*) malloc(sizeof (char)
     *strlen(chInterfacesTaxasOutTemp));
strcpy(chInterfacesTaxasInTemp, chInterfacesTaxasOutTemp);
int fimChIfacesTxsTemp = strlen(chInterfacesTaxasOutTemp);
int comecoChIfacesTxsTemp = fimChIfacesTxsTemp;

   while (chInterfacesTaxasOutTemp[--comecoChIfacesTxsTemp] != ' ')
   {  }
```

```
   orcm->props-
>setRateInterfaceOut(substring(chInterfacesTaxasOutTemp,
comecoChIfacesTxsTemp, (fimChIfacesTxsTemp -
comecoChIfacesTxsTemp)));

   while (chInterfacesTaxasOutTemp[--comecoChIfacesTxsTemp] == ' ')
   {  }
    orcm->props-
>setRateInterfaceIn(substring(chInterfacesTaxasInTemp, 0,
comecoChIfacesTxsTemp));

// ###############################################################
// ## fim do bloco de dados ifstat                              ##
// ###############################################################

// ###############################################################
// ## início do bloco de dados que utiliza consultas SNMP       ##
// ## para obter diferentes tipos de dados disponibilizados     ##
// ## por essa tecnologia                                       ##
// ###############################################################

        orcm->props-
>setMemoryRamTotal(consultaSNMP("MIBDIRS=/usr/local/share/snmp/mibs;
export MIBDIRS;MIBS=ALL;export MIBS;snmpget -v 1 -c public 127.0.0.1
.1.3.6.1.4.1.2021.4.5.0"));
        orcm->props-
>setMemoryRamFree(consultaSNMP("MIBDIRS=/usr/local/share/snmp/mibs;e
xport MIBDIRS;MIBS=ALL;export MIBS;snmpget -v 1 -c public 127.0.0.1
.1.3.6.1.4.1.2021.4.6.0"));
        orcm->props-
>setMemorySwapTotal(consultaSNMP("MIBDIRS=/usr/local/share/snmp/mibs
;export MIBDIRS;MIBS=ALL;export MIBS;snmpget -v 1 -c public
127.0.0.1 .1.3.6.1.4.1.2021.4.3.0"));
        orcm->props-
>setMemorySwapFree(consultaSNMP("MIBDIRS=/usr/local/share/snmp/mibs;
export MIBDIRS;MIBS=ALL;export MIBS;snmpget -v 1 -c public 127.0.0.1
.1.3.6.1.4.1.2021.4.4.0"));
        orcm->props-
>setCpuUsed(consultaSNMP("MIBDIRS=/usr/local/share/snmp/mibs;export
MIBDIRS;MIBS=ALL;export MIBS;snmpget -v 1 -c public 127.0.0.1
.1.3.6.1.4.1.2021.11.9.0"));

// ###############################################################
// ## fim do bloco de dados SNMP                                ##
// ###############################################################

// Variáveis como de clientes do AVC, como: zoom, e valores de
// rotação X, Y e Z usam coleta de dados dinâmicos.
// Exemplo de coleta de dados dinâmica com fins de depuração do
// servidor:
        orcm->props->insertDynamicProperties("Connected Clients",
intToString(orcm->totalMembers));
```

```
// Geração do arquivo JMonitor.xml
// Pode ser observado que após a criação do arquivo o processo
// entra em estado de dormência por 1 segundo e coleta novamente
// os dados (isto é possível por que a coleta de dados está em um
// laço infinito.
        XMLFile *xmlFile = new XMLFile(orcm->props);
        delete xmlFile;
        sleep(1);
    }
    return (void*) 0;
}
```

O *header* Properties.h declara três classes auxiliares para a geração do arquivo JMonitor.xml: (*i*) a classe Property representa os pares de dados nome e valor a serem descritos em JMonitor.xml; (*ii*) a classe Properties representa o objeto utilizado para armazenar os dados temporariamente durante a coleta dos mesmos, esses dados permanecem no objeto Properties até que são escritos no arquivo JMonitor.xml e; (*iii*) a classe XMLFile, um objeto dessa classe é utilizado no código para realmente criar o arquivo JMonitor.xml e descrever nele os dados armazenados nos objetos Properties.

Properties.h

```
class Property{
public:
    string name;
    string value;
    Property( string name, string value);
    void setName (string name);
    string getName();

    void setValue (string name);
    string getValue();
};

class Properties {
public:
    string serverName;
    string vality;
    string memoryRamFree;
    string memoryRamTotal;
    string memorySwapFree;
    string memorySwapTotal;
    string cpuUsed;
    string rateInterfaceIn;
    string rateInterfaceOut;

    void setServerName (string serverName);
    string getServerName();
    void setVality (string vality);
    string getVality();

    void setMemoryRamFree (string memoryRamFree);
    string getMemoryRamFree();
    void setMemoryRamTotal (string memoryRamTotal);
```

```
    string getMemoryRamTotal();

    void setRateInterfaceIn (string rateInterfaceIn);
    string getRateInterfaceIn();
    void setRateInterfaceOut (string rateInterfaceOut);
    string getRateInterfaceOut();

    void setMemorySwapFree (string memorySwapFree);
    string getMemorySwapFree();
    void setMemorySwapTotal (string memorySwapTotal);
    string getMemorySwapTotal();

    void setCpuUsed (string cpuUsed);
    string getCpuUsed();

    Properties(string serverName, string vality);
    Properties(const Properties& orig);
    virtual ~Properties();

    void insertStaticProperties(string name, string value);
    list<Property> staticPropertiesList;
    list<Property>::iterator staticIt;

    void insertDynamicProperties(string name, string value);
    list<Property> dynamicPropertiesList;
    list<Property>::iterator dynamicIt;
};

class XMLFile {
public:
    xmlDocPtr doc;       /* document pointer */
    xmlNodePtr root_node, node, node1;/* node pointers */
    xmlDtdPtr dtd;       /* DTD pointer */
    char buff[256];
    int i, j;

    XMLFile(Properties *props);
    virtual ~XMLFile();
private:

};
```

O código Properties.cpp implementa as classes declaradas no *header* Properties.h. Neste anexo destaca-se a implementação da classe XMLFile utilizada pela *thread* de OR_communicationManager para realizar a escrita dos dados coletados.

Para descrever os dados coletados em um arquivo com estrutura XML de acordo com os requisitos de JMonitor foi utilizada a biblioteca libxml [LIBXML, 2012].

Properties.cpp

```
XMLFile::XMLFile(Properties *props) {
// ponteiro para o documento
doc = NULL; /* document pointer
// ponteiros para os nós xml
```

```
root_node = NULL, node = NULL, node1 = NULL;
// ponteiro para o dtd
dtd = NULL;
LIBXML_TEST_VERSION;

/*
 * Cria um novo documento e um nó configurado como root
 */
doc = xmlNewDoc(BAD_CAST "1.0");
root_node = xmlNewNode(NULL, BAD_CAST "monitoring");
xmlDocSetRootElement(doc, root_node);

/*
 * Cria uma declaração DTD
 */
dtd = xmlCreateIntSubset(doc, BAD_CAST "Monitoring", NULL, BAD_CAST
"monitoring.dtd");

/*
 * xmlNewChild() cria um novo nó, que é configurado como nó
 *“filho”
 */
BAD_CAST((string) props->getServerName()).c_str());
node = xmlNewChild(root_node, NULL, BAD_CAST "serverName", NULL);
xmlNewProp(node, BAD_CAST "value", BAD_CAST((string) props-
>getServerName()).c_str());
xmlAddChild(root_node, node1);

BAD_CAST((string) props->getVality()).c_str());
node = xmlNewChild(root_node, NULL, BAD_CAST "vality", NULL);
xmlNewProp(node, BAD_CAST "value", BAD_CAST((string) props-
>getVality()).c_str());
xmlAddChild(root_node, node1);

node = xmlNewChild(root_node, NULL, BAD_CAST "memory-ram-total",
NULL);
xmlNewProp(node, BAD_CAST "value", BAD_CAST((string) props-
>getMemoryRamTotal()).c_str());
xmlAddChild(root_node, node1);

node = xmlNewChild(root_node, NULL, BAD_CAST "memory-ram-free",
NULL);
xmlNewProp(node, BAD_CAST "value", BAD_CAST((string) props-
>getMemoryRamFree()).c_str());
xmlAddChild(root_node, node1);

node = xmlNewChild(root_node, NULL, BAD_CAST "memory-swap-total",
NULL);
xmlNewProp(node, BAD_CAST "value", BAD_CAST((string) props-
>getMemorySwapTotal()).c_str());
    xmlAddChild(root_node, node1);

node = xmlNewChild(root_node, NULL, BAD_CAST "memory-swap-free",
NULL);
xmlNewProp(node, BAD_CAST "value", BAD_CAST((string) props-
>getMemorySwapFree()).c_str());
```

```
xmlAddChild(root_node, node1);

node = xmlNewChild(root_node, NULL, BAD_CAST "cpu-used", NULL);
xmlNewProp(node, BAD_CAST "value", BAD_CAST((string) props-
>getCpuUsed()).c_str());
xmlAddChild(root_node, node1);

node = xmlNewChild(root_node, NULL, BAD_CAST "rate-interface-in",
NULL);
xmlNewProp(node, BAD_CAST "value", BAD_CAST((string) props-
>getRateInterfaceIn()).c_str());
xmlAddChild(root_node, node1);

node = xmlNewChild(root_node, NULL, BAD_CAST "rate-interface-out",
NULL);
xmlNewProp(node, BAD_CAST "value", BAD_CAST((string) props-
>getRateInterfaceOut()).c_str());
xmlAddChild(root_node, node1);

node = xmlNewNode(NULL, BAD_CAST "static-properties");

for (list<Property>::iterator it = props-
>staticPropertiesList.begin(); it != props-
>staticPropertiesList.end(); it++) {
node1 = xmlNewChild(node, NULL, BAD_CAST "property", NULL);
xmlNewProp(node1, BAD_CAST "name", BAD_CAST((string) ((Property) *
it).getName()).c_str());
xmlNewProp(node1, BAD_CAST "value", BAD_CAST((string) ((Property) *
it).getValue()).c_str());
xmlAddChild(node, node1);
}

xmlAddChild(root_node, node);

node = xmlNewNode(NULL, BAD_CAST "dynamic-properties");

for (list<Property>::iterator it = props-
>dynamicPropertiesList.begin(); it != props-
>dynamicPropertiesList.end(); it++) {
node1 = xmlNewChild(node, NULL, BAD_CAST "property", NULL);
xmlNewProp(node1, BAD_CAST "name", BAD_CAST((string) ((Property) *
it).getName()).c_str());
xmlNewProp(node1, BAD_CAST "value", BAD_CAST((string) ((Property) *
it).getValue()).c_str());
}

xmlAddChild(root_node, node);
/*
 * Descreve o documento criado em memória no arquivo JMonitor.xml
 */
xmlSaveFormatFileEnc("JMonitor.xml", doc, "iso-8859-1", 1);

/* Elimina o documento em memória */
xmlFreeDoc(doc);

/*
```

```
 *Elima variável global
 */
xmlCleanupParser(); }
```

Para sincronização de eventos locais a aglomerados gráficos foram codificados na arquivo app_client.cpp as variáveis que controlam a aproximação (*zoom*) ao AVC e propriedades da malha poligonal através dos recursos da LibGlass.

App_client.cpp

```
// Declaração
GlassClient *g = NULL;
Barrier *datalock;
Barrier *swaplock;
Shared<int, rawPack, rawUnpack> *zoom;
Shared<int, rawPack, rawUnpack> *filling;
Alias<void(*)(void) > *alias;

//método principal
main()
//inicialização de variáveis LibGlass
using namespace libglass;
    try {
        g = new GlassClient(new TCP(), server_libglass_IP);
    } catch (libglass::Exception e) {
        std::cerr << e.getMessage() << std::endl;
        exit(1);
    }
    /* Barreiras de sincronismo */
    datalock = new Barrier(1, false);
    swaplock = new Barrier(2, false);

    /* variáveis compartilhadas */
    zoom = new Shared<int, rawPack, rawUnpack > ("zoom");
    filling = new Shared<int, rawPack, rawUnpack > ("filling");

    if (g->isMaster()) {
        *zoom = -60.0f;
        *filling = 0.0f;
        zoom->sendUpdate();
        filling->sendUpdate();
    }
} // fim do método principal

// thread que recebe e processa cada evento recebido
void* App_Client::run() {
    try {
        cout << "PEDIDO DE CONEXAO =======> "<< this-
>intConnectionAVDEventCommunicationPort << endl;
        InetAddress addr(this->server_app_communication_IP);
        Socket sock(addr, this-
>intConnectionAVDEventCommunicationPort);
        cout << "CONECTOU < ========= "<< this-
>intConnectionAVDEventCommunicationPort << endl;
        //ostream & out = sock.getOStream();
```

```
        istream & in = sock.getIStream();

        while (true) {
            char txt[5] = "";
            while (true) {
                in >> txt;
                if (in.eof())
                    break;
                if (txt != "") {
                 cout << "RECEBIDO POR MULTICAST = " << txt << endl;
                    switch (atoi(txt)) {
                        case 31007:
                            // evento de zoom recebido
                            // processando sincronização
                            if (g->isMaster()) {
                                *zoom = **zoom + 5.0;
                                zoom->sendUpdate();
                            }
                            glutPostRedisplay();
                            break;
                        case 31008:
                            if (g->isMaster()) {
                                *zoom = **zoom - 5.0;
                                zoom->sendUpdate();
                            }
                            glutPostRedisplay();
                            break;
                        case 31009:
                            // evento de malha poligonal recebido
                            // processando sincronização
                            if (g->isMaster()) {
                                *filling = 1.0f;
                                filling->sendUpdate();
                            }
                            glutPostRedisplay();
                            break;
                        case 310010:
                            if (g->isMaster()) {
                                *filling = 0.0f;
                                filling->sendUpdate();
                            }
                            glutPostRedisplay();
                            break;
                    }
                }
            }
        }

    } catch (OR::Exception & e) {
        cerr << e.what() << endl;
    }
    return (void*) 0;
}
```