Comportamento de Aplicações
de Realidade Virtual Distribuídas
por meio da Análise da Troca de Mensagens

Luiz Francisco Contri

Outubro/2014

Dissertação de Mestrado em
Ciência da Computação

Comportamento de Aplicações de Realidade Virtual Distribuídas por meio da Análise da Troca de Mensagens

Esse documento corresponde à dissertação de mestrado apresentado à Banca Examinadora da Dissertação no curso de Mestrado em Ciência da Computação da Faculdade Campo Limpo Paulista.

Campo Limpo Paulista, 31 de Outubro de 2014.

Luiz Francisco Contri
Prof. Dr. Marcelo de Paiva Guimarães
Orientador

Faculdade Campo Limpo Paulista

Programa de Mestrado em Ciência da Computação

"Comportamento de Aplicações de Realidade Virtual Distribuídas por meio da Análise da Troca de Mensagens"

LUIZ FRANCISCO CONTRI

Dissertação de Mestrado apresentado ao Programa de Mestrado em Ciência da Computação da Faculdade Campo Limpo Paulista, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Prof. Dr. Marcelo de Paiva Guimarães

(Orientador – FACCAMP)

Prof. Dr. Osvaldo Luiz de Oliveira

(FACCAMP)

Prof. Dr. José Remo Ferreira Brega

(UNESP)

Campo Limpo Paulista, 31 de outubro de 2014.

FICHA CATALOGRÁFICA

Dados Internacionais de Catalogação na Publicação (CIP)

Câmara Brasileira do Livro, São Paulo, Brasil

Contri, Luiz Francisco
Comportamento de aplicações de realidade virtual distribuídas por meio da análise da troca de mensagens / Luiz Francisco Contri. Campo Limpo Paulista, SP: FACCAMP, 2014.

Orientador: Profº. Dr. Marcelo de Paiva Guimarães
Dissertação (mestrado) – Faculdade Campo Limpo Paulista – FACCAMP.

1. Realidade virtual. 2. Sistemas distribuídos. 3. Aplicações distribuídas. 4. Análise do comportamento. 5. Depurador de mensagens. 6. Troca de mensagens. I. Guimarães, Marcelo de Paiva. II. Faculdade Campo Limpo Paulista. III. Título.

CDD-006.68

AGRADECIMENTOS

A Deus pela vida, por sua maravilhosa graça e presença em todos os meus momentos.

Ao meu professor orientador, Dr. Marcelo de Paiva Guimarães, por sua paciência e orientação eficaz, por sua dedicação, pela transferência de conhecimentos, por suas idéias inteligentes que enriqueceram o conteúdo deste trabalho.

Aos professores do Mestrado em Ciência da Computação da Faccamp, em especial ao prof. Dr. Osvaldo Luiz de Oliveira, por estar sempre de portas abertas em atender, ouvir e apoiar as nossas solicitações.

Aos professores membros das bancas examinadoras: Dra. Ana Maria Monteiro, Dr. José Remo Ferreira Brega, Dr. Marcelo de Paiva Guimarães e Dr. Osvaldo Luiz de Oliveira pelas contribuições ao presente trabalho.

Ao Diego Colombo Dias, doutorando em Ciência da Computação na UFSCar que nos atendeu nos laboratórios da universidade, demonstrando o funcionamento da CAVERNA Digital e também o funcionamento da biblioteca libGlass e aplicações distribuídas neste contexto.

Aos colegas da empresa Corollarium Tecnologia por terem cedido algumas fotos que fizeram parte deste trabalho.

A todos os colegas da nossa turma e em especial aos que participaram dos grupos de estudo: Carlos Eduardo Andrade Oliveira, Eduardo Machado Real, Fabiano da Silva Fernandes, Francisco Sanches, Marcos Guilherme Cassolato de Oliveira, Jair José da Silva e o Waldomiro Sebastião Moreira. Quero destacar o apoio do Denilson Caraça Peramos, do Francisco Fambrini e da Marta Angélica Montiel Ferreira.

Aos colegas de trabalho e professores da FIAP, onde leciono, que me incentivaram na realização deste mestrado.

Aos meus colegas de tecnologia da informação onde desenvolvi minha carreira profissional e aos colegas da Igreja Presbiteriana do Brasil onde construi meus valores pessoais e espirituais.

A minha querida esposa, Isabel Maciel Fernandes Contri, e ao meu filho, Gabriel de Oliveira Contri, pelo amor, carinho, amizade e palavras de motivação permanentes nestes anos de mestrado.

Ao meu pai, Acticlinio Contri, "*in memoriam*" que sempre acreditou no meu potencial e foi um incentivador dos meus estudos. A minha amorosa mãe, Genny Apparecida Barbieri Contri, que sonhava em ter um filho e aguardou doze anos para eu vir a este mundo.

"O coração do homem pode fazer planos, mas a resposta certa dos lábios vem do SENHOR."

Provérbios 16:1

Muito Obrigado,

Luiz Francisco Contri

Resumo. *Os avanços tecnológicos dos últimos anos têm impulsionado o desenvolvimento de aplicações distribuídas, dentre elas, as simulações com alto grau de imersão e interação construídas utilizando as técnicas de Realidade Virtual. Essas soluções buscam usufruir de características como escalabilidade, alta capacidade de processamento e tolerância a falhas. Porém, o desenvolvimento é complexo e exige ferramentas especializadas. Esta dissertação tem como objetivo modelar o comportamento das aplicações distribuídas, em especial as de Realidade Virtual, de tal forma que permita a criação de um analisador de mensagens que facilite a verificação comportamental das mensagens trocadas num aglomerado. Além disso, apresenta o GTracer, que é uma ferramenta que faz a verificação de comportamento das aplicações de realidade virtual distribuída desenvolvidas com a biblioteca libGlass.*

Palavras chaves: *Realidade Virtual; Sistemas Distribuídos; Aplicações Distribuídas; Análise do Comportamento; Depurador de Mensagens; Troca de Mensagens.*

***Abstract**: The technological advances of recent years have boosted the development of distributed applications, among them, the simulations with high immersion degree and interaction built using the techniques of Virtual Reality. These solutions seek to take advantage of features such as scalability, high processing capacity and fault tolerance. However, development is complex and requires specialized tools. This proposal aims to model the behavior of distributed applications, in particular the ones of the Virtual Reality, that allows the creation of an analyzer messages, to facilitate behavioral verification of messages exchanged in a cluster. Furthermore, it shows the GTracer, a tool that checks the behavior of distributed virtual reality applications developed with the library libGlass.*

***Keywords:** Virtual Reality; Distributed Systems; Distributed Applications; Behavior Analysis; Debbuger Messages; Message Passing.*

Sumário

Glossário

API - *Application Programming Interface*

CAVE - *CAVE Automatic Virtual Environment*

JNI - *Java Native Interface*

LCD - *Liquid Crystal Display*

LED - *Light Emitting Diod*

MPI - *Message Passing Interface*

OpenGL - *Open Graphics Library*

OpenSG – *Open Scene Graph*

PC - *Personal Computers*

PDA - *Personal Digital Assistant*

PVM – *Parallel Virtual Machine*

RV – *Realidade Virtual*

TCP - *Transmission Control Protocol*

UDP - *User Datagram Protocol*

VIA – *Visual Interface Architecture*

VR – *Virtual Reality*

Lista de Tabelas

Lista de Figuras

1. Introdução

Aplicações distribuídas tornaram-se o modelo adotado por muitos sistemas modernos (redes sociais, serviços *web*, bancos e outros) (Sedayao, 2008; Balasubramanian *et al.* 2010; Pallot *et al.* 2013; Wang *et al.* 2013). Existem diversas motivações para o uso desse modelo, dentre elas é que essas aplicações intrinsecamente distribuídas necessitam de alto poder computacional que pode ser alcançado através do uso dos diversos computadores e a possibilidade de utilização de computadores ociosos (Tanenbaum e Steen, 2007).

As aplicações de Realidade Virtual (RV) com alto grau de imersão e interação adotaram nos últimos anos esse modelo, exigindo também escalabilidade. Essa demanda por aplicações distribuídas ocorre porque para gerar e manipular os mundos sintéticos, explorando todos os sentidos fundamentais do corpo humano, como a visão, a audição, o tato e o olfato, torna-se necessário utilizar os mais diversos tipos de dispositivos de entrada e saída, que geralmente estão acoplados a diversos tipos de computadores.

A Figura 1 mostra um exemplo de aplicação de RV distribuída. Nela é simulada a visualização de uma arcada dentária que é executada em um *cluster* para tratar todo o processamento do ambiente e apresentada em um mini-CAVE com três telas. Além disso, faz parte dela um sistema de detecção de gestos para o tratamento das interações.

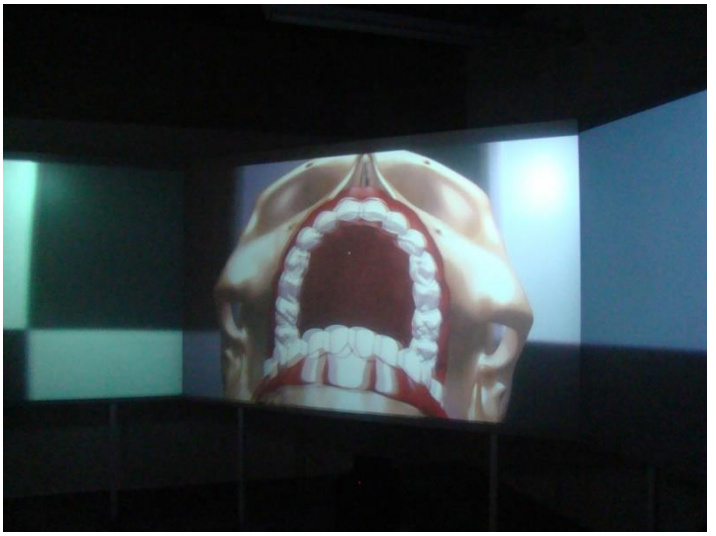

Figura 1 – Aplicação de Realidade Virtual Distribuída (Dias, 2011)

Apesar do estado atual da tecnologia de desenvolvimento de aplicações distribuídas permitir que estas estejam presentes em diversos contextos, ainda existem muitos pontos a serem aprimorados, dentre eles a depuração. De forma geral essas aplicações são vários processos sequenciais que são executados em diversos computadores, que se comunicam via rede (Tanenbaum e Steen, 2007). Por isso, os diversos problemas de desenvolvimento enfrentados para criar as aplicações centralizadas, também fazem partes das distribuídas.

Por exemplo, tanto os depuradores sequenciais quanto os distribuídos devem interferir de forma mínima na aplicação e devem apresentar o estado (contexto) da aplicação no momento. Contudo, as aplicações distribuídas tratam outros pontos também, como a complexidade maior, visto que os processos estão em computadores diferentes; que a quantidade de dados envolvidos é maior, pois podem existir milhares de processos; e que existem efeitos anômalos causados por problemas de sincronização e comunicação (KranzlMüller, 2000).

Este trabalho possibilita a modelagem do comportamento das aplicações de RV distribuídas de tal forma que seja possível criar um analisador que valide tal comportamento. Por exemplo, ao modelar o comportamento esperado das barreiras de sincronização, torna-se então possível confrontar a execução de uma aplicação com o esperado e relatar se está ocorrendo conforme o esperado. Para isso, analisa-se as funcionalidades que são específicas de tais aplicações e que dependem da troca de mensagens. Além disso, o GTracer, a partir de sua versão original (Guimarães, 2004), evoluiu e recebeu alterações e novas funcionalidades que permitem realizar a análise comportamental das aplicações de RV desenvolvidas com a biblioteca libGlass, emitindo um diagnóstico para o desenvolvedor poder verificar se o comportamento da aplicação, para uma determinada funcionalidade, foi executado conforme o esperado ou apresentou problemas e neste caso servir de auxílio para a devida correção.

Para a formalização do comportamento das aplicações de RV distribuídas utiliza-se conceitos de Redes de Petri e Teoria de *Bag*. Redes de Petri possibilita visualizar comportamentos relacionados com sincronização de dados ou quadros e também compartilhamento de recursos, bem como concorrência e paralelismo. Teoria de *Bag*

possibilita extrair das Redes de Petri as pré-condições e pós-condições necessárias para a execução das funcionalidades.

Essa dissertação engloba diversas áreas do conhecimento, como: redes de computadores, engenharia de *software*, computação gráfica, teste de *software* e sistemas distribuídos. O envolvimento delas foi necessário para formalizar o comportamento das aplicações de RV distribuídas, tornando-se então possível a implementação da verificação do comportamento baseada em técnicas de depuração e análise das trocas de mensagens no GTracer. Essa dissertação não visou substituir os depuradores de programa tradicionais. Porém, procura facilitar o desenvolvimento das aplicações de RV distribuídas e, além disso, facilitar a aprendizagem de criação de tais aplicações e a adição de novas funcionalidades na biblioteca libGlass. Apesar das técnicas terem sido implementadas no GTracer, elas foram projetadas para serem implementadas em qualquer ferramenta voltada para as aplicações alvo deste trabalho.

1.1 Objetivo

Essa dissertação tem como objetivo a modelagem do comportamento das aplicações distribuídas, em especial as de RV, de tal forma que permita a criação de um analisador de mensagens que facilite a verificação comportamental das mesmas. Este analisador foi implementado na ferramenta de depuração GTracer.

1.2 Estrutura da Dissertação

O Capítulo 2 apresenta a fundamentação teórica dessa dissertação. Para isso, mostra detalhes das aplicações alvo desse trabalho e a libGlass, que é a biblioteca que possibilitou a implementação das técnicas desenvolvidas. Além disso, como a especificação comportamental utilizou Redes de Petri e a Teoria *Bag*, então essas também são apresentadas.

O Capítulo 3 mostra técnicas e ferramentas de depuração de troca de mensagens existentes. Além disso, posiciona o GTracer, que implementa o modelo de verificação desenvolvido nesse trabalho, em relação às outras ferramentas.

O Capítulo 4 mostra os modelos criados que permitiram a especificação comportamental das aplicações de RV distribuídas, que iniciou-se com a especificação

das funcionalidades das aplicações de RV distribuídas e finalizou-se em pré-condições e pós-condições, que representam as situações nas quais as funcionalidades ocorrem durante a execução das aplicações.

O Capítulo 5 apresenta a implementação da especificação comportamental no GTracer e os resultados obtidos.

No Capítulo 6 são apresentadas as considerações finais.

2. Fundamentação Teórica

A RV é uma área que agrega conhecimentos e tecnologias que lidam com mundos sintetizados por computadores, que podem ser explorados e manipulados em tempo de execução em ambientes tridimensionais (Kirner e Tori, 2004). Destacam-se nestes mundos o alto grau de imersão, que proporciona o envolvimento do usuário no ambiente; e a interação, que são as respostas geradas a partir das ações realizadas pelos usuários (Mori, 1994).

Este capítulo apresenta inicialmente uma visão geral das aplicações de RV distribuídas e, em seguida, a libGlass, que é a biblioteca para computação distribuída utilizada neste trabalho e, consequentemente, a fonte geradora de mensagens para o GTracer. Logo após, apresenta uma visão geral das Redes de Petri, pois estas serviram como base para formalizar o modelo de execução das aplicações envolvidas neste trabalho. Por fim, mostra a Teoria *Bag*, que permitiu, a partir da modelagem construída pela Redes de Petri, especificar as situações que as funcionalidades são executadas nas aplicações de RV distribuídas.

2.1 Aplicações de Realidade Virtual Distribuídas

As aplicações de RV estão cada vez mais sofisticadas, proporcionando alto grau de imersão e interação para os usuários, e mais abrangentes, simulando desde conteúdos educacionais, médicos, industriais até esportivos (Abulrub, 2011; Souza, 2012; Kai-Hu e Xin-Jian, 2013). Tradicionalmente, utiliza-se computadores com memória compartilhada para a execução destas aplicações, que são de alto custo e com escalabilidade limitada (*e.g.* memória, placas, dispositivos de saída), o que reflete na melhora de desempenho.

A tendência atual é utilizar sistemas computacionais com memória distribuída (*e.g. clusters*), que são de menor custo, escaláveis e com alta capacidade de processamento (Khan e Ali, 2012). Os sistemas de memória compartilhada se distinguem dos sistemas com memória distribuída fundamentalmente pelo modo de distribuição e sincronização dos dados. No caso dos sistemas com memória compartilhada, o *hardware* e a maioria dos *softwares* foram projetados para eles; por

isso, a distribuição e sincronização dos dados são realizadas automaticamente pelo sistema operacional, via barramento interno; no caso dos *clusters*, a distribuição e a sincronização dos dados são feitas por *softwares* de comunicação via rede, sendo a latência maior. Contudo, quando os *clusters* são utilizados com *software* apropriado, eles propiciam uma alta escalabilidade, podendo ter um número ilimitado de nós, porém, isso não garante o ganho de desempenho das aplicações, pois existe a dependência de elementos como latência de comunicação e dependência entre as tarefas (Khan e Ali, 2012).

As vantagens e desvantagens do uso de *clusters* de computadores na área de Realidade Virtual já foi abordada por vários trabalhos (SGI, 2001; Zuffo *et al.* 2001; Klosowski, 2002; Lin *et al.* 2002; Zuffo *et al.* 2002; Schaeffer e Goudeseune, 2003; Guimarães, 2004), o que resultou em diversas soluções, dentre elas os *VRClusters* (Zuffo *et al.* 2001), que são *clusters* compostos por computadores pessoais que dispõem de *hardware* dedicado a esse tipo de problema, como placas gráficas e dispositivos específicos de interação.

As aplicações de RV tratadas neste trabalho disponibilizam para os usuários um alto grau de imersão e interação, então são projetadas para executarem em ambientes de multiprojeção, como os CAVEs (*CAVE Automatic Virtual Environment*) (Cruz-Neira *et al.* 1992), e suportam várias formas de interação, como *trackers*, dispositivos móveis e interfaces naturais (Harper *et al.* 2008). A Figura 2 exemplifica uma dessas aplicações multiprojetada em um mini-CAVE com três telas. O usuário pode navegar nela utilizando gestos (interface natural).

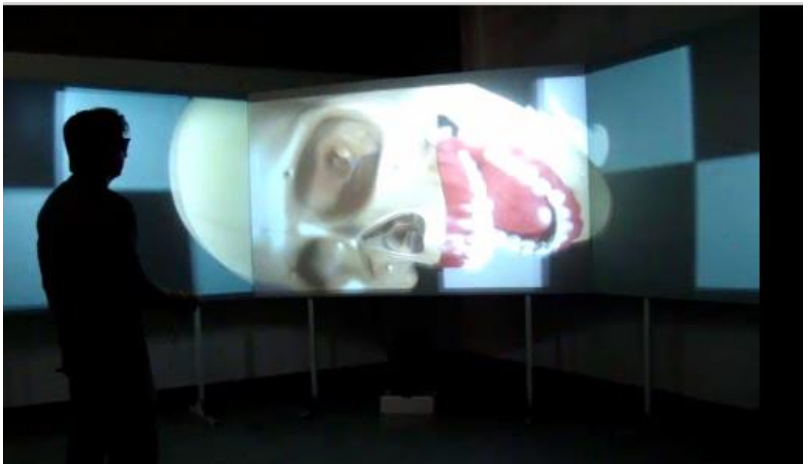

Figura 2 – Aplicação executando em um mini-CAVE com interação via gestos (Dias, 2011)

Outra característica das aplicações envolvidas neste trabalho é que utilizam *clusters* como plataforma de execução, que podem ou não serem desenvolvidas de maneira automática. Quando construídas de maneira automática, elas são desenvolvidas como se fossem executadas em um sistema computacional com memória compartilhada; e uma ferramenta, como um compilador, a transforma para ser executada no *cluster*. O ambiente Chromium (Humphreys *et al*., 2001) é um exemplo de ferramenta que implementa essa solução. Ele transforma comandos OpenGL de uma aplicação centralizada em fluxos de comandos que são enviados para os nós dos *clusters* gerarem as imagens. Apesar de simples, pois não há necessidade de prever os problemas referentes a um sistema distribuído durante a modelagem e implementação, a construção de uma ferramenta eficiente de transformação é difícil. Além disso, a aplicação estará limitada aos recursos oferecidos por tal ferramenta de transformação.

Outra abordagem, que gera melhores resultados do que a anterior, é o uso de comandos que possibilitam a troca de informações e sincronização entre os nós do *cluster* (Guimarães, 2004). Essa alteração do código fonte (abordagem não automática) é uma desvantagem, pois deverá ser levada em conta durante a modelagem e implementação. Exemplos de bibliotecas que podem utilizar esta estratégia incluem a libGlass (Guimarães *et al*., 2003), Net Juggler (Allard *et al.*, 2002) e OpenSG (OpenSG, 2013).

Na abordagem não automática, a estratégia de distribuição dos dados pode ser a de distribuição de estímulos ou a de cálculo centralizado dos resultados e distribuição (Guimarães, 2004). A primeira costuma ser simples, consistindo na adição de comandos no código fonte que distribuam os eventos de entrada (*input*) (*e.g. tracker*, teclado e *mouse*) para os outros nós. Contudo, pode trazer alguns problemas, já que não existe nenhuma sincronia de dados. Isso significa que, se por alguma razão um dos nós sair de sincronia (perdendo um evento, processando-o incorretamente ou fora de ordem), o aplicativo não se recuperará.

A abordagem de cálculo centralizado dos resultados e distribuição é menos suscetível a este tipo de problema, e em geral, mais difícil de implementar. Nela, o programador trata todos os dados a serem compartilhados ou sincronizados, como a

posição e orientação do observador, os estados internos do aplicativo e as opções de visualização (*e.g wireframe* ou sólido), entre outros. Os tipos de dados que podem mudar em cada quadro são os de: controle, que estabelecem o que desenhar (por exemplo, a direção do ponto de vista do observador em cada nó); e os de mudança do conjunto dos dados (por exemplo, o modelo deve ser atualizado, pois ocorreu uma mudança na textura de um objeto). As alterações desses dados, como as ocorridas devido à entrada de dados de uma luva, faz com que seja necessária a sincronização para que imagens consistentes sejam geradas (Guimarães *et al*., 2013).

O modelo adotado para o desenvolvimento das aplicações não influencia a forma de depuração, pois ela foca na verificação de funcionalidades específicas, como a sincronização e o compartilhamento de uma variável. Contudo, devido a robustez, a maioria das aplicações envolvidas neste trabalho adotaram o modelo de Cálculo centralizado dos resultados e distribuição, implementado utilizando uma arquitetura mestre/escravo com replicação (Guimarães, 2004). Neste tipo de aplicação os dados são replicados em diversos nós do *cluster*, o que reduz a necessidade de transmissão de dados. Existem três tipos de nós nesse modelo: o coordenador, que assegura a distribuição e sincronização das informações; os nós de interface, que recebem as interações dos usuários, como, por exemplo, as entradas de gestos; e os nós de saída, que processam os dados e enviam as imagens para os dispositivos de saída. Essa divisão de funcionalidades entre os nós permite a atribuição de tarefas conforme as características de cada um. Por exemplo, um dispositivo móvel, que tem baixa capacidade de processamento, pode ser responsável apenas pela entrada de dados, como, por exemplo, comandos de voz.

A Figura 3 ilustra este modelo de Cálculo Centralizado dos Resultados e Distribuição (arquitetura mestre/escravo com replicação), onde alguns nós tratam o recebimento de interações; o servidor coordena a sincronização e distribuição dos dados; e os outros nós processam os dados e projetam nos dispositivos de saída.

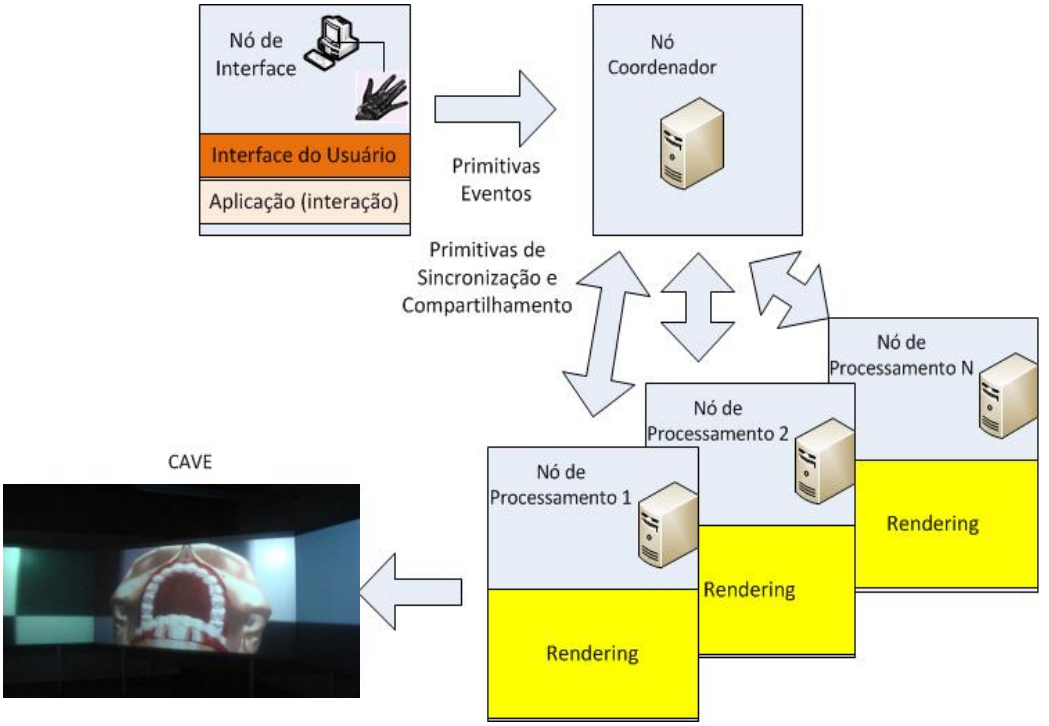

Figura 3 – Modelo de Cálculo Centralizado dos Resultados e Distribuição

2.2 libGlass

Aplicações de RV distribuídas implementadas com o modelo Cálculo centralizado dos resultados e distribuição (arquitetura mestre/escravo com replicação) fornecem um alto grau de imersão e interação. A análise comportamental dessas aplicações foi implementada na libGlass, que tem características, como Guimarães (2004) aponta:

- Transparência: o desenvolvedor não precisa conhecer detalhes de baixo nível em relação a troca de mensagens;
- Facilidade de uso e porte: permite que tanto aplicações novas quanto as já desenvolvidas sejam portadas;
- Desempenho: fornece desempenho suficiente para as aplicações de interesse deste trabalho;
- Independência de protocolos: ela suporta diversos protocolos, como TCP (*Transmission Control Protocol*) e UDP (*User Datagram Protocol*);

- Independência de dispositivos: pode ser executada em diversos tipos de dispositivos (desde *smartphones* até supercomputadores);
- Multiplataforma: pode ser executada em Windows, Linux e Macintosh. Além disso, suporta C/C++ e Java, e
- *Threads*: fornece suporte nativo a *threads*.

Segundo Guimarães (2004), a libGlass consiste de um conjunto escalável de componentes que podem ser utilizados pelas aplicações. Uma característica de destaque delas é o fato de visar a fácil utilização, tanto para o desenvolvimento de novas aplicações quanto para manutenção das existentes, pois a maioria das soluções disponíveis necessita de um grande número de alterações no código fonte e até mesmo mudanças na arquitetura da aplicação.

A libGlass é tolerante a falhas e permite que nós sejam instanciados e removidos a qualquer momento. Na possibilidade de um nó parar de responder a biblioteca detecta e trata esta falha e a aplicação não é interrompida, prevenindo *deadlocks*. Características como essas são fundamentais para a biblioteca e a distingue de outras soluções, como NetJuggler (Allard *et al*., 2002) e OpenSG (OpenSG, 2013). Assim, a verificação de funcionalidade da libGlass realizadas pelo GTracer é muito importante, pois auxiliará a garantir que a biblioteca está funcionando quando ocorrer alterações na mesma. Então, por exemplo, o GTracer ao alertar que um comportamento não está sendo atendido, torna possível ao desenvolvedor notar que está ocorrendo um comportamento indevido, como por exemplo, um *deadlock*.

A Figura 4 apresenta uma visão geral da arquitetura da libGlass e seus componentes. Ela é composta por um *framework*, que provê funcionalidades como a de sincronização; pelo componente Protocolo, que encapsula diversos protocolos de comunicação de redes; e pelas aplicações de Suporte, que são programas que auxiliam o desenvolvimento das aplicações (Guimarães, 2004).

O *framework* é composto por:

- Instanciação: tem como objetivo inicializar as aplicações conforme arquitetura interna da libGlass, que possibilita selecionar a aplicação como cliente ou servidor, e

- *Plugins:* tem como responsabilidade gerenciar os componentes internos da biblioteca, que são os seguintes: transmissão de eventos - fornecido pelo *plugin* Eventos; compartilhamento de dados – disponibilizado pelo *plugin* Compartilhamento; sincronização de barreiras – implementado pelo *plugin* Barreiras; e o de associação de funções – fornecido pelo *plugin Alias* (Pseudônimo). A libGlass não está restrita somente a esses *plugins*, outros podem ser adicionados conforme a necessidade, sem modificação da estrutura interna.

O componente Protocolo esconde as diferenças entre os protocolos de comunicação suportados, como o TCP (*Transmission Control Protocol*), UDP (*User Datagram Protocol*), MPI (*Message Passing Interface*), VIA (*Virtual Interface Architecture*) (Kim e Jung, 2001) ou outros. Esse componente suporta todos os tipos básicos de dados como *integer*, *float*, *string* e outros, além disto permite a definição e criação de outros tipos de dados.

O outro componente da libGlass são as aplicações de Suporte constituídas pelo GTracer, pelo GEditor e pelo GVoicer. O GEditor é uma ferramenta para a geração da interface gráfica para PDAs. A facilidade de utilização de PDAs em ambientes de multiprojeção já foi estudada em diversos projetos, como Hartling *et al*. (2002) e Hill e Cruz-Neira (2000). O GVoicer é um mecanismo que possibilita a criação do controle das aplicações via comando de voz, onde cada comando é associado a um evento; e um leitor que recebe e trata os comandos de voz e que transmite os eventos associados para um nó do aglomerado gráfico, o qual receberá e executará a rotina apropriada. O GTracer é o analisador de comportamento das aplicações. As ferramentas de suporte auxiliam o desenvolvimento das aplicações libGlass.

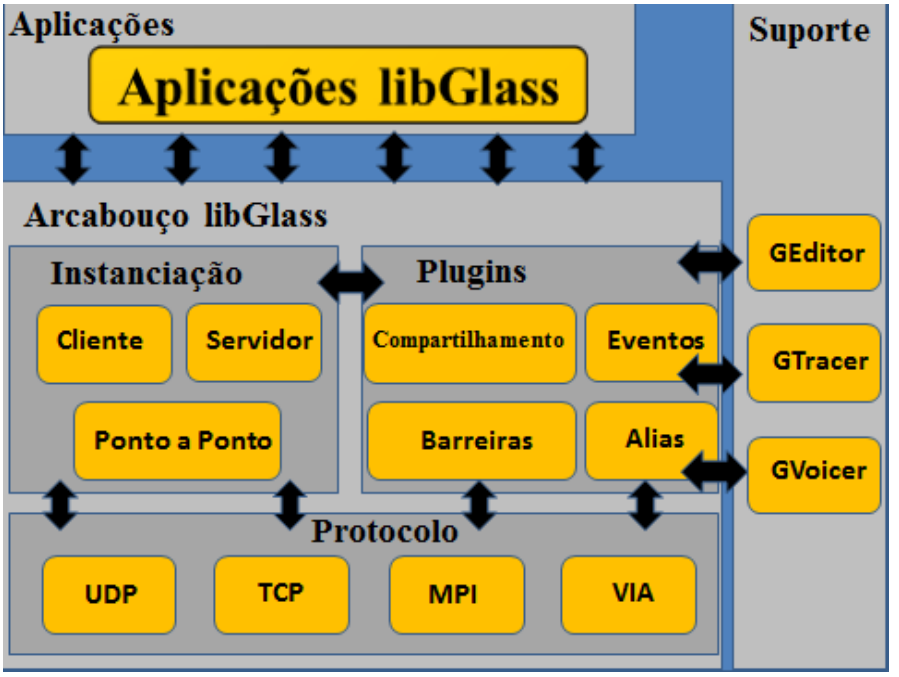

Figura 4 – Arquitetura libGlass (Guimarães, 2004)

As Aplicações libGlass são aplicações de RV baseadas em *VRClusters* e podem ser desenvolvidas em C/C++ ou Java, estando a integração com Java disponível por meio da utilização de JNI (*Java Native Interface*) (JAVA, 2014). As Aplicações libGlass foram testadas nas arquiteturas PC, Silicon e PDA (Guimarães, 2004).

2.3 Redes de Petri

Utilizou-se Redes de Petri neste trabalho para modelar o comportamento das aplicações de RV distribuídas, porque elas são capazes de especificar, por exemplo, concorrência, paralelismo, sincronização e compartilhamento de recursos. Além disso, oferecem uma representação gráfica, bem como o comportamento de cada processo envolvido, a comunicação entre eles, os estados possíveis (condições) e também as ações realizadas (eventos). Assim, a modelagem das aplicações com essa ferramenta foi um passo para possibilitar a extração de pré-condições e pós-condições que possibilitaram identificar as situações de uso de cada funcionalidade durante a execução das aplicações.

A teoria sobre Redes de Petri surgiu com Carl Adam Petri em sua tese de doutoramento, denominada *Kommunikation mit Automaten*, apresentada em 1962 na faculdade de Matemática e Física da Universidade de Darmstad, Alemanha, com o

objetivo de estabelecer relações entre condições e eventos ao trabalhar com protocolos de comunicação entre componentes assíncronos (Petri, 1962).

Redes de Petri é uma técnica de especificação formal para modelagem e análise de sistemas. Pode ser utilizada em qualquer tipo de sistema, porém seu principal foco de atuação está em sistemas que executam em seu processamento atividades paralelas, concorrentes, assíncronas e não-determinísticas. Destacam-se como principais áreas para aplicabilidade desta técnica: Ciência da Computação, Engenharias Eletrônica e Química, além de Administração de Empresas. Em Ciência da Computação e Engenharia Eletrônica tem sido muito utilizada para avaliação de desempenho, especificação de protocolos de comunicação, diagnóstico de falhas e especificação de projetos de *software* e *hardware* (Maciel, Lins e Cunha, 1996).

Atua como uma importante ferramenta matemática e gráfica, possibilitando um ambiente uniforme para modelagem, análise e projeto de sistemas, simulação de sistemas a eventos discretos. Utiliza-se as Redes de Petri para modelar a estrutura do sistema e seu comportamento. Como ferramenta gráfica possibilita a visualização simultânea da estrutura do sistema e seu comportamento dinâmico modelado (Zuruwaski e Zhou, 1994).

Uma Rede de Petri é um tipo particular de grafo dirigido bipartido formado por três componentes que são lugares, transições e arcos dirigidos. Arcos dirigidos conectam lugares a transições ou transições a lugares (Wang, 2007). Os componentes lugares são os componentes passivos da rede e correspondem as variáveis de estado (condições). As transições são os componentes ativos da rede e correspondem as ações (eventos) realizadas pelo sistema.

Redes de Petri modelam dois aspectos importantes presentes nos sistemas que são as condições e as ações. As condições habilitam a realização de ações (eventos). Desta forma em cada estado do sistema, verificam-se determinadas condições que podem possibilitar a ocorrência de ações (eventos), que por sua vez podem modificar o estado do sistema (Peterson, 1977).

Uma ação que é representada graficamente na Rede de Petri pelo componente transição pode ser executada a partir de algumas pré-condições definidas em algumas variáveis de estado, que por sua vez são representadas na rede pelos componentes

lugares. Portanto, existe uma relação direta entre lugares e transições que possibilitam a realização de uma ação. Do mesmo modo, após a realização de uma ação, os lugares terão suas informações alteradas (pós-condições).

Segundo Peterson (1981), formalmente uma Rede de Petri pode ser definida como uma 5-tupla N=(P,T,I,O,Mo), onde:

(1) P={p1,p2,p3,...,pm} é um conjunto finito de lugares;
(2) T={t1,t2,t3,...,tm} é um conjunto finito de transições, $P \cup T \neq \emptyset$, e $P \cap T = \emptyset$;
(3) I: $P \times T \rightarrow N$ é uma função de entrada que define arcos dirigidos dos lugares para as transições, onde N é um conjunto inteiro não negativo;
(4) O: $T \times P \rightarrow N$ é uma função de saída que define arcos dirigidos das transições para os lugares, e
(5) Mo: $P \rightarrow N$ é a marcação inicial.

A marcação de uma Rede de Petri é a atribuição de *tokens* ou fichas aos lugares. *Tokens* residem nos lugares. O número e a posição dos *tokens* podem mudar durante a execução da rede. Os *tokens* são usados para definir a execução da Rede de Petri.

A execução de uma Rede de Petri é controlada pelo número e distribuição dos *tokens*. Pela mudança na distribuição dos *tokens* nos lugares pode-se representar a ocorrência de eventos ou execução de operações e desta forma estudar o comportamento dinâmico do sistema modelado. Uma Rede de Petri é executada pelo disparo de transições. As regras de habilitação e disparo de uma transição estão destacados a seguir:

(1) Regra de Habilitação: uma transição t esta habilitada se cada lugar de entrada p de t, contém no mínimo um número de *tokens* igual ao peso do arco dirigido que conecta p a t, isto é, $M(p) \geq I(p,t)$ para transições habilitadas t.
(2) Regra de Disparo: somente transição habilitada pode ser disparada. O disparo de uma transição habilitada remove o número de *tokens*, igual ao peso do arco dirigido de todos os lugares de entrada pi de t, e são criados *tokens* nos lugares de saída. O número de *tokens* criados em cada lugar de saída da transição t é igual ao peso do arco de saída.

Matematicamente, disparando uma transição t, em M é produzida uma nova marcação: $M'(p) = M(p) - I(t,p) + O(t,p)$, para transições habilitadas t (Wang, 2007).

Uma rede pode representar diversos aspectos da modelagem de um sistema. Por exemplo, através da representação gráfica por Redes de Petri é possível modelar um sistema distribuído e representar os processos e a comunicação entre eles.

Algumas abordagens matemáticas tem sido propostas, entre estas uma que trata Redes de Petri do ponto de vista da algebra matricial (Brams, 1983), outra conforme a Teoria *Bag* (Peterson, 1981) e uma terceira baseada em relações (Gomide, 2011).

2.4 Teoria *Bag*

Nesta dissertação utiliza-se a Teoria *Bag* porque possibilita representar matematicamente as transições de uma Rede de Petri com mais de uma entrada ou saída relacionada a um lugar. Essas entradas e saídas especificadas serviram como base para o entendimento das situações em que as funcionalidades são acionadas nas aplicações de RV distribuídas.

Definição *Bag*: define-se um *bag* M sobre um conjunto não-vazio C, por uma função M: C→ N, onde M(C) representa o número de ocorrência do elemento C em M.

Utiliza-se o símbolo [] para denotar os *bags* e {}, para os conjuntos.

Define-se a estrutura de uma rede de Petri R, como uma quíntupla R=(P,T,I,O,K), onde P={p1,p2,...,pn} é um conjunto finito não-vazio de lugares, T={t1,t2,...,tm} é um conjunto finito não-vazio de transições. I:T → P∞ é um conjunto de *bags* que representa o mapeamento de transições para lugares de entrada. O:T→ P∞ é um conjunto de *bags* que representa o mapeamento de transições para os lugares de saída. K:P → N é o conjunto das capacidades associadas a cada lugar, podendo assumir um valor infinito (Maciel, Lins e Cunha, 1996).

2.5 Notas Finais

As aplicações de RV distribuídas agregam *hardware* e *software* específicos, além disso, questões referentes a interface de usuários, fatores humanos, percepção e outros. O desenvolvimento delas exige ferramentas que facilitem o desenvolvimento.

Este capítulo mostrou inicialmente uma visão geral dessas aplicações, incluindo os princípios de funcionamento e desenvolvimento. Além disso, apresentou a libGlass, que é uma ferramenta que busca esconder a complexidade de desenvolvimento dessas aplicações. O modelo comportamental que permite a analisar as aplicações foi implementado nessa biblioteca, via a ferramenta GTracer. Por fim, expôs as Redes de Petri e a Teoria *Bag*, que são recursos de formalização que são utilizados nesse trabalho respectivamente para especificar o comportamento das aplicações estudadas e para determinar as situações de uso das funcionalidades .

3. Depuração de Troca de Mensagens

Uma das dificuldades no processo de desenvolvimento de aplicações distribuídas é a tarefa de encontrar erros, que é essencial, pois um programa é útil somente se executa corretamente e se é razoavelmente confiável. A depuração pode auxiliar a melhoria do desempenho da aplicação, pois aponta o comportamento da mesma. Acompanhar a execução de diversos processos espalhados por vários nós não é uma tarefa trivial, pois cada nó possui o contexto de execução próprio. Assim, torna-se necessário o apoio de ferramentas de análise capazes de representar de maneira gráfica o grande volume de dados gerados (KranzlMüller, 2000).

Este capítulo apresenta as principais técnicas para depuração de aplicações e as ferramentas existentes. Além disso, posiciona o GTracer em relação a essas ferramentas. Por fim, realiza a comparação entre as técnicas existentes.

3.1 Introdução

A depuração de *software* pode representar dois significados distintos: depuração de programas e depuração de mensagens. No contexto de engenharia de *software* depurar significa a tarefa de localização e remoção de defeitos (Araki, Furukawa e Cheng, 1991), que pode também resultar em outros benefícios, como a melhora do desempenho. Depurar aplicações distribuídas é mais difícil do que as sequenciais. Segundo KranzlMüller (2000), elas são mais complexas, geram um maior volume de dados e sofrem de efeitos anômalos, como *deadlocks*.

Existem diversas ferramentas de depuração voltadas para atender sistemas distribuídos de 1 até n nós. Algumas exigem uma participação e esforço maior do desenvolvedor, pois dependem muitas vezes da alteração do código fonte da aplicação, inserindo pontos de controle e verificação, como o comando *printf* (Reynolds, 2006) ou adição de primitivas específicas de sinalizações no código. Por outro lado, outras fornecem recursos funcionais para que o programador não precise modificar ou realizar inserções manuais de código na aplicação em questão.

A Figura 5 apresenta o gráfico de posicionamento do GTracer comparativamente com as outras ferramentas mencionadas neste trabalho. O GTracer é uma ferramenta de

depuração que verifica se as aplicações atendem determinados comportamentos. O eixo das abscissas indica o porte do sistema, desde um sistema de um único nó, passando por sistemas de porte médio, como sistemas em rede local e atingindo sistemas de grande porte como nós, distribuídos na internet. O eixo das ordenadas indica o esforço do desenvolvedor, participando da depuração manualmente e agindo sobre a ferramenta de depuração.

O uso de comandos como o *printf* e de ferramentas como o gdb (Reynolds, 2006) exigem um esforço maior do programador. A Project 5 (Reynolds, 2006) é um exemplo de ferramenta que não exige alteração de código, ou seja, o comportamento da aplicação é verificado sem a necessidade de modificações na aplicação (caixa preta). O GTracer exige a coleta dos dados dos nós – atualmente é manual, mas pode ser implementada de maneira automática. Os detalhes das ferramentas apresentadas na Figura 5 serão abordados na próxima subseção.

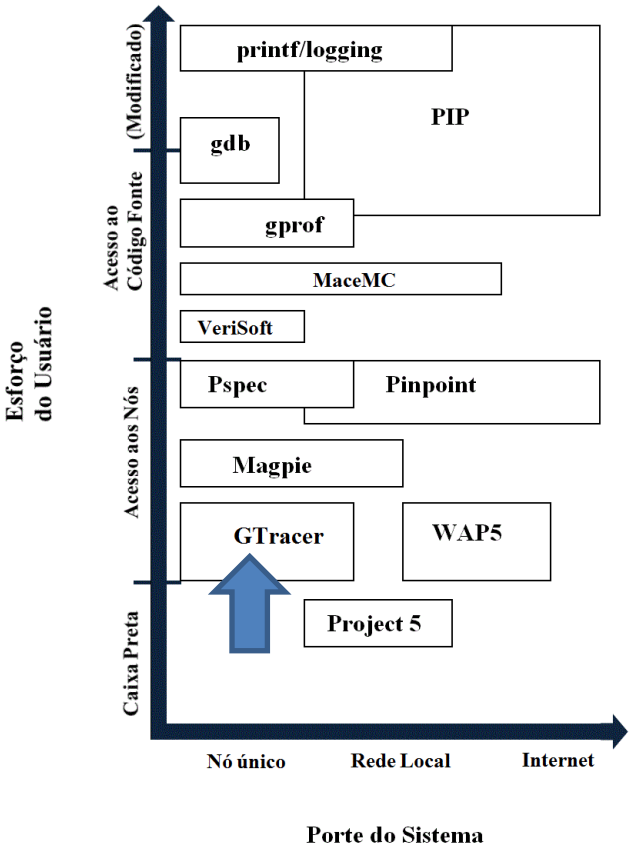

Figura 5 – Posicionamento do GTracer em relação a outras ferramentas de depuração (adaptado de Reynolds, 2006)

Segundo (XupingTu *et al*., 2010), a principal diferença entre um sistema distribuído e o centralizado é que o primeiro introduz a troca de mensagens via rede. Essas mensagens de rede podem proporcionar ganhos como o de escalabilidade e de desempenho, contudo aumentam a complexidade da aplicação. Todos os tipos de mensagens de rede são transmitidos a partir de alguns nós para outros para notificar os nós receptores, informando que eles devem completar alguma tarefa, e então todos podem colaborativamente completar uma tarefa maior. Como as mensagens podem às vezes chegar atrasadas, perder-se no caminho, não serem enviadas ou mal utilizadas, então erros podem ser gerados. Além disso, podem ocorrer apenas depois de uma determinada seqüência de eventos, geralmente envolvendo máquina ou falhas de rede, que são muitas vezes difíceis de reproduzir.

Os possíveis erros na troca de mensagens podem tornar o desenvolvimento, testes e depuração das aplicações distribuídas uma tarefa difícil, o que impulsiona ferramentas como o GTracer. No contexto deste trabalho, depuração significa rastrear troca de mensagens de diversos nós via rede que são armazenadas em *logs* locais e verificar se atendem um comportamento pré-especificado, identificando possíveis problemas, como erros de sincronização de dados compartilhados e de comunicação. Esses erros podem afetar diretamente o resultado desejado, na medida que os nós poderão estar esperando por alguma resposta, como a permissão para transferir a imagem de um *buffer* para um dispositivo de saída.

3.2 Técnicas e Ferramentas de Depuração de Mensagens

Existem diversos projetos e pesquisas relacionadas a depuração de mensagens de aplicações distribuídas. A seguir são apresentados projetos correlacionados com essa dissertação, como o PIP (Reynolds *et al.*, 2006) e VeriSoft (Godefroid, 2005); e as principais técnicas de depuração de troca de mensagens, como Caminho Causal e Verificação de Modelo (Clarke, Emerson e Sifakis, 2009).

3.2.1 Caminho Causal (*Causal Path*)

O comportamento das aplicações distribuídas pode ser representado através da técnica de depuração denominada caminho causal (Reynolds, 2006). Um caminho causal começa normalmente com alguma requisição de um usuário e representa um conjunto de situações de comportamento apresentadas pela aplicação, indicando os caminhos previstos e não previstos pelo desenvolvedor.

A Figura 6 representa uma aplicação distribuída em três camadas com o caminho causal indicado pelas setas mais largas, onde o cliente envia uma requisição para o servidor *web*, que envia uma mensagem para o servidor de autenticação, que lê uma informação do servidor do banco de dados, que contata um servidor de aplicação que contata outro servidor de aplicação e que contata outro servidor de banco de dados e cada servidor responde por sua vez e a resposta é enviada ao cliente.

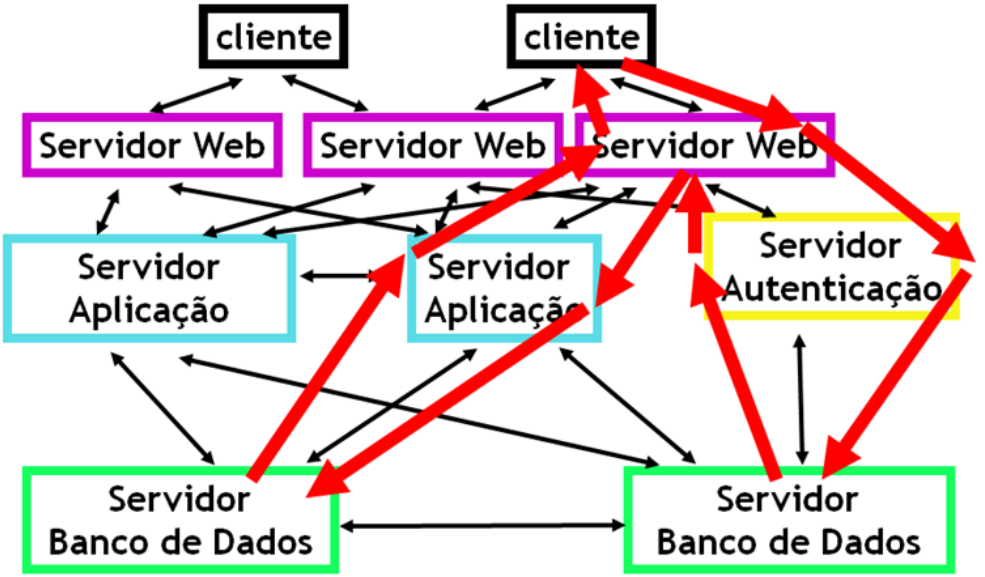

Figura 6 – Sistema de três camadas com caminho causal indicado (adaptado de Reynolds, 2006)

Segundo Reynolds (2006), "um caminho causal é um conjunto parcialmente ordenado de eventos relacionados em um ou mais nós". É parcialmente ordenado pois para certos elementos existe uma relação de precedência, mas não para todos. Neste sentido, o primeiro evento deste conjunto não tem ninguém que o precede. Para os

próximos elementos do conjunto de eventos tem-se um elemento que o precede. O caminho pode ser dividido por estes eventos interligados.

Muitos erros acontecem por divergências entre o que foi especificado pelo desenvolvedor e o comportamento real apresentado pela aplicação. A técnica do caminho causal auxilia aos programadores a depurar o sistema através da identificação e segregação dos caminhos com problemas e que apresentam comportamento inesperado.

Depuração baseada em caminhos pode permitir aos programadores encontrar caminhos defeituosos ou anormais e permitem otimizar tanto a taxa de transferência como a latência entre os nós (XupingTu *et al.*, 2010). A seguir, alguns exemplos de ferramentas que implementam essa técnica:

- Pip: tem como objetivo controlar de forma automática o comportamento da aplicação, realizando verificações de resultados contra as expectativas do desenvolvedor. Essa ferramenta tem como objetivo identificar tanto erros estruturais da aplicação quanto de desempenho. Um erro estrutural resulta no processamento ou comunicação acontecendo no lugar errado ou na sequência errada. Um erro de desempenho pode resultar de chamadas em excesso ou poucas chamadas para um recurso importante do sistema (Reynolds *et al*., 2006);
- Magpie: coleta os eventos dos vários nós de uma aplicação e extrai os caminhos determinados pelo desenvolvedor e constrói um modelo probabilístico do comportamento solicitado. Esta ferramenta é capaz de coletar eventos gerados pelo *kernel*, *middleware* e componentes da aplicação em ambiente reais (Barham *et al.*, 2004) e também não requer modificação da aplicação que está sendo analisada, mas requer a construção de um programa específico e escrito por um especialista para rastrear a informação através do caminho causal (Barham *et al.*, 2003);
- Pinpoint: tem como foco principal a identificação de erros de componentes. Ela utiliza conceitos de probabilidade para detectar anomalias em uma base de eventos de erros identificados (*logs*) ao invés de analisar caminhos inteiros (Chen *et al*., 2002);

- Project 5: permite a análise de aplicações fechadas que não podem ter o código fonte alterado. Infere caminhos a partir do rastreamento de comunicação através de um ou mais *sniffers* de rede. Em seguida, a ferramenta agrega estes caminhos e exibe para o usuário procurar por problemas de desempenho ou comportamento inesperado (Reynolds, 2006), e
- WAP5: permite o mapeamento da estrutura de comunicação e identifica caminhos inesperados, possibilitando a visualização de cada etapa e apontando as possíveis causas de atraso**.** É composta por um conjunto de ferramentas para coleta de *logs* por processos, através de uma API (*Application Programming Interface*) específica (Reynolds *et al*., 2006).

A técnica de Caminho Causal pode ser utilizada em diversos contextos, como no caso de aplicações de Internet de larga escala. Contudo, demanda a especificação prévia dos caminhos possíveis, o que limita a possibilidade de depuração. Em contrapartida, expande a limitação das técnicas de depuração tradicional, que não conseguem tratar de múltiplos processos e nós, não cuidam das diversas fontes de atraso na transmissão de redes, incluindo a latência de rede, a largura de banda limitada, a falta de confiabilidade dos nós e a programação paralela.

3.2.2 Verificação de Modelo (*Model Checking*)

Esta técnica de depuração distribuída foi proposta por Edmund M. Clarke e E. Allen Emerson (Clarke e Emerson, 1981). Ela visa verificar de forma automática as propriedades e comportamentos apresentados pelas aplicações distribuídas, identificando de forma exaustiva os estados que são possíveis de acontecer (Clarke, Emerson e Sifakis, 2009).

Esta técnica visa analisar automaticamente o espaço de estados finitos de uma aplicação baseando-se em propriedades comportamentais, como segurança, atingibilidade (*reachability*), razoabilidade (*rairness*), ausência de *deadlock* e vivacidade (*liveness*). Geralmente, esta técnica é implementada adicionando no código as primitivas de sinalização que indicam que aquele código foi executado, o que altera o estado. Essa técnica sofre de problemas como o de explosão de estados e tradicionalmente funciona somente para estados finitos.

A Figura 7 representa as etapas de execução do sistema de verificação de modelo. A implementação dela exige a execução das seguintes etapas:

- Modelagem: constrói-se o modelo formal do sistema conforme os seus requisitos da aplicação;
- Especificação: especifica-se o comportamento esperado da aplicação, ou seja, as propriedades dela, que pode ser definida através de lógicas temporais ou máquinas de estado, e
- Verificação: realiza-se um confronto entre a Modelagem e a Especificação através de uma ferramenta de verificação denominada verificador de modelos (*model checker*). O resultado desta ferramenta é um valor que indica se a especificação, que reflete o comportamento esperado, foi atendida ou não. Em caso negativo o verificador devolve um conjunto de estados que podem ser alcançados, denominado de contra-exemplo, demonstrando que a especificação não é válida junto aos requerimentos do sistema ou modelo.

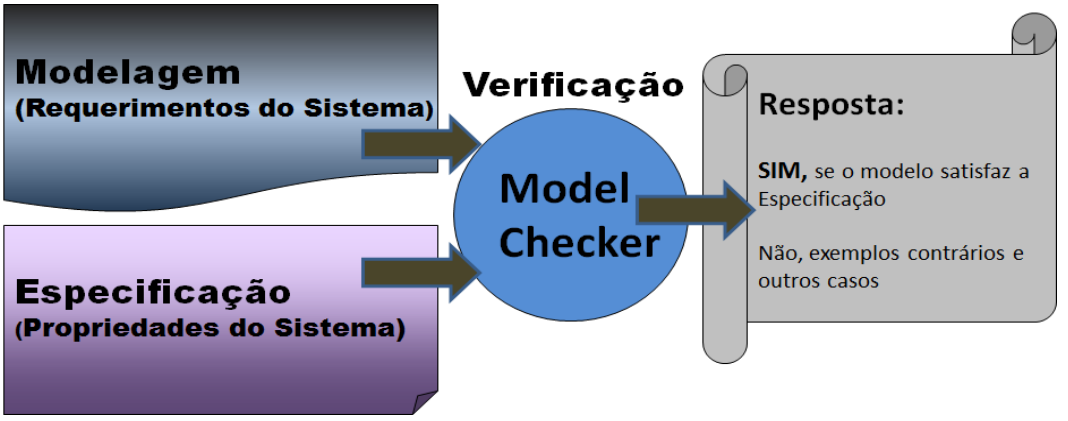

Figura 7 – Diagrama de Verificação de Modelos

A seguir alguns exemplos de ferramentas que implementam essa técnica:

- MaceMc: é uma ferramenta que realiza a Verificação de Modelo. Além da técnica Verificação de Modelo, ela é também capaz de simular Caminhos Causais. Ela utiliza como base um template que permite o compilador realizar uma validação de alto nível dos serviços providos pelos objetos (Killian et al., 2007), e

- Verisoft: a maioria das ferramentas de Verificação de Modelos realiza a checagem tendo como base as propriedades especificadas. Esta ferramenta busca validar o sistema por intermédio da execução de diversos processos para arbitrariamente validar a aplicação (Godefroid, 2005).

3.2.3 Comparação das Técnicas de Depuração

Existem outras ferramentas e técnicas de depuração. Por exemplo, a ferramenta e linguagem Pspec implementa e realiza verificação do desempenho por intermédio de asserções (*performance assertion checking*). Nela o desenvolvedor adiciona asserções no código que são posteriormente coletadas para analisar o desempenho esperado (Sharon e Weihl, 1993).

A Tabela 1 faz um mapeamento das técnicas de depuração apresentadas anteriormente, descrevendo as respectivas ferramentas e cenários utilizados.

Técnica de Depuração	Ferramenta	Cenário
Caminho Causal	PIP	Sistemas de grande porte, com código fonte disponível e problemas de desempenho
Caminho Causal	Project 5	Sistemas locais do tipo Intranet e com problemas de desempenho em caminhos
Caminho Causal	WAP5	Sistemas de Internet com problemas de desempenho em caminhos
Caminho Causal	Pinpoint	Falhas em nós ou componentes em sistemas distribuídos
Caminho Causal	Magpie	Erros bem identificados em aplicações Windows
Verificação de Modelo	MaceMc e Verisoft	Sistemas de pequeno porte com dificuldades para reproduzir os erros
Asserções	Pspec	Analisa o *log* gerado pela aplicação e verifica se o desempenho é o esperado

Tabela 1 – Mapeamento Técnicas de Depuração x Ferramentas

3.3 Implementação da Depuração de Mensagens

O ideal é que o processo de depuração não interfira na codificação das aplicações. Algumas ferramentas buscam não exigir alteração, coletando dados usando recursos nativos do sistema, como chamadas do sistema operacional e/ou monitorando o trafego da rede utilizando *sniffers*. Apesar de atender alguns casos, não é suficiente robusta para capturar detalhes das aplicações.

Outro ponto desejável é que a depuração não afete o desempenho das aplicações em execução. Se a abordagem de monitoramento não for no modo promíscuo, provavelmente o código deverá gerar saídas que diminuirá o desempenho da aplicação. Por isso, a depuração geralmente é habilitada somente em momentos de verificação do ambiente. Há uma variedade de pesquisas que se baseiam em registros para análise *post-mortem*, incluindo análises estatísticas de utilização de recursos (Aguilera *et al*., 2003), correlacionando eventos (Barham *et al.*, 2004), seguindo as dependências entre os componentes (Fonseca *et al.*, 2007) e verificação de caminhos causais (Reynolds *et al.* 2006). Fundamentalmente, o registro de *logs* pode verificar todas imposições de sobrecarga em tempo de execução e expor estas informações para análise, inclusive considerando milhares de nós (Verbowski *et al*., 2006).

A Figura 8 é um exemplo genérico da técnica de depuração com análise de *logs*, onde é possível visualizar três etapas do processo:

- Coletar *Logs:* nesta etapa do processo busca-se os registros com as mensagens de *logs* gravadas na rede distribuída;
- Analisar *Logs*: na análise dos *logs*, os registros coletados são lidos e comparados com comportamentos esperados e possibilita a identificação de situações com problemas, e
- Exibir Resultados: é responsável pela visualização dos resultados coletados e analisados em um formato que permita ao observador identificar situações esperadas e situações com problemas a serem solucionadas.

Registrar *logs* das aplicações altera o desempenho das mesmas, contudo é escalável, pois somente no momento de coleta ocorre a comunicação entre os diversos nós e o servidor responsável pela análise – que também pode ser implementada de forma

distribuída, ou seja, os servidores podem ser divididos em níveis, no qual cada nível é responsável por um determinado pré-processamento.

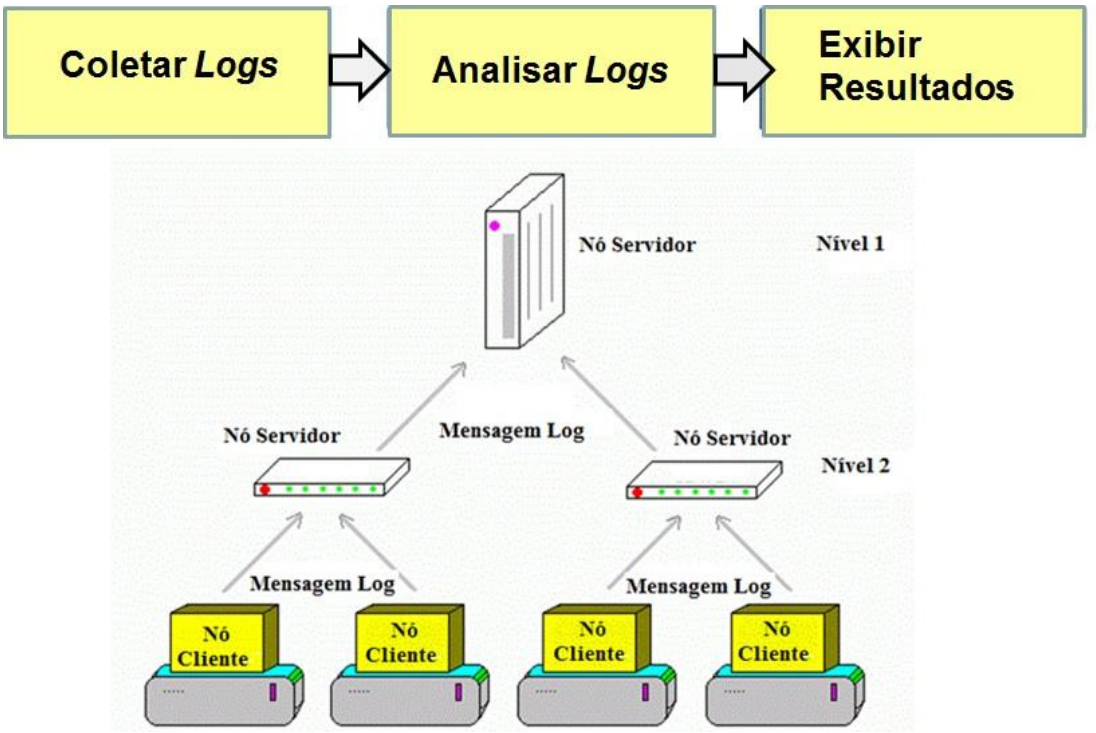

Figura 8 – Esquema genérico de uma ferramenta de coleta de *logs*

Mesmo com a coleta habilitada somente durante um período suficiente para reproduzir a situação desejada, geralmente o volume de dados é grande o suficiente para inviabilizar a análise sem auxílio de ferramentas apropriadas. Dessa forma, é desafiadora a tarefa de construir uma ferramenta capaz de verificar se o comportamento da aplicação em execução está conforme o esperado, pois, é tarefa árdua identificar, por exemplo, se um nó não liberou uma barreira de sincronização devido a tarefa que o mesmo já foi executado ou se não liberou devido a uma falha. Por isso, em muitas situações, o comportamento do depurador é dependente das decisões de projetos implementadas pela biblioteca de programação distribuída adotada.

Uma proposta para minimizar o trabalho dos programadores, que é a adotada pela libGlass, é das próprias bibliotecas conterem os mecanismos de rastreamento de

comportamento. Essa abordagem facilita a depuração dos serviços providos pela biblioteca, contudo não trata dos erros de alto nível da aplicação.

3.4 Notas Finais

A depuração identifica falhas nas aplicações apontando falhas em tempo de execução. Esse tipo de ferramenta é indispensável para o desenvolvimento dos *softwares*. Embora, seja um assunto pesquisado por muitos, ainda é um desafio, principalmente quando se trata de aplicações distribuídas, pois, enquanto na programação sequencial a prática comum de depuração limita-se a estabelecer *breakpoints*, reinicializar a aplicação e verificar os valores nesses *breakpoints*, o que acaba atribuindo o ônus de identificação do problema ao desenvolvedor. Nas aplicações distribuídas essa abordagem não é o suficiente devido ao fato de envolver diversos nós e processos com contextos de execução próprios, e de gerar grandes volumes de dados.

Esse capítulo apresentou diversas técnicas, ferramentas e abordagens de implementação de depuração de aplicações distribuídas, posicionando o GTracer em relação as ferramentas apresentadas. Neste contexto exige que o ônus de verificação de erros seja automatizada de tal forma que o esforço do desenvolvedor seja minimizado, não alterando código e com respostas precisas e em alto nível de abstração.

4. Comportamento das Aplicações de Realidade Virtual Distribuídas

O desenvolvimento de aplicações de RV distribuídas não é considerado uma tarefa trivial devido ao envolvimento de diversos nós computacionais que trocam informações via rede. Independentemente do tipo da aplicação, ela deve atender às propriedades necessárias para o seu correto funcionamento. Para a realização da verificação do atendimento dessas propriedades é preciso conhecer o modelo comportamental das mesmas, que pode, por exemplo, ser descrito por um modelo matemático, criando-se um modelo com o comportamento do sistema e a especificação das propriedades ou pode-se criar uma prova de teorema, no qual o comportamento é descrito através de axiomas e regras, utilizando-se lógica para deduzir as propriedades do sistema.

Este capítulo tem como objetivo mostrar como a partir das Aplicações de RV distribuídas e da modelagem delas, é possível criar uma base de especificação capaz de ser implementada em uma ferramenta de depuração para a verificação do comportamento das mesmas.

4.1 Introdução

As aplicações de RV distribuídas utilizam a troca de mensagens para a comunicação e sincronização dos dados entre os diversos nós. Contudo, existe um conjunto de funcionalidades de alto nível que são comuns a elas, como a liberação das barreiras de sincronização para apresentação das imagens e a sincronização das variáveis para a geração das imagens. Então, cabe a um depurador para essas aplicações a tarefa de verificar se essas funcionalidades estão sendo executadas conforme o previsto. Para isso, torna-se necessário especificá-las e formalizá-las de tal maneira que seja possível construir uma ferramenta de verificação.

A Figura 9 mostra os passos desenvolvidos neste trabalho para o entendimento e formalização do comportamento esperado das funcionalidades das aplicações de RV distribuídas. Isso resultou em diversos modelos que são independentes da biblioteca de baixo nível usada para a implementação da troca de mensagens, como MPI (*Message Passing Interface*) (MPI, 2013), PVM (*Parallel Virtual Machine*) (PVM, 2013) ou libGlass. Cada um desses modelos contribuíram para o entendimento de um aspecto

dessas aplicações. Iniciou-se pelo levantamento das funcionalidades que dependem da biblioteca de troca de mensagens e finalizou-se com as pré-condições e pós-condições para determinar a implementação da ferramenta de verificação. Os detalhes destes modelos são apresentados nas próximas seções.

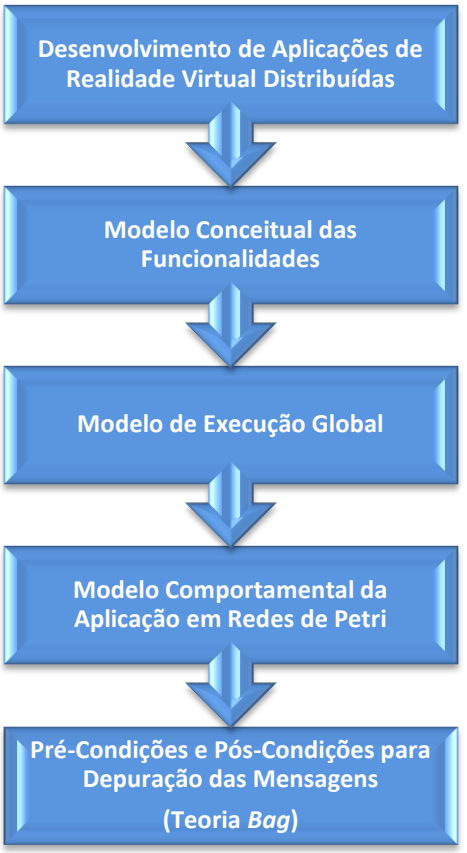

Figura 9 – Passos para a criação das pré-condições e pós-condições

4.2 Desenvolvimento de Aplicações de Realidade Virtual Distribuídas

Independentemente do modelo de desenvolvimento adotado (automático ou não), é necessário que haja sincronia dos dados (*datalock*) e a dos *frames* (*framelock*), pois são eles que garantem a coerência de imagens em um ambiente de RV imersivo (multiprojetado). A sincronia de dados deve ser implementada de tal forma que ocorra antes do processamento dos dados, assim a geração das imagens irá considerar, por exemplo, um novo ponto de vista do usuário; e a sincronia dos *frames* deve ocorrer antes da troca dos *buffers* de imagem, dessa forma as imagens são enviadas para os dispositivos de saída ao mesmo tempo (Guimarães, 2004).

A abordagem não automática em que os desenvolvedores implementam o modelo de Cálculo centralizado dos dados e distribuição é a que mais demanda conhecimento de programação, pois é necessário identificar os dados a serem compartilhados e os pontos de adição das barreiras de sincronização. Contudo, implementadores bem treinados são capazes de desenvolver com facilidade, pois mesmo em aplicações aparentemente muito diferentes, as ações são as mesmas. Neste contexto de aprendizagem de distribuição da aplicação um depurador pode atuar no ensino das ações necessárias, pois os aprendizes conseguem entender visualmente as ações a serem implementadas. Independentemente da biblioteca adotada e da aplicação, o desenvolvedor deve sempre realizar as tarefas descritas a seguir, pois geralmente, possuem geometria estática, requerendo apenas a sincronia dos dados de controle e da posição e orientação do observador (Guimarães, 2004):

- Inserção das barreiras (sincronização de barreiras): são as primitivas que criam pontos de sincronização, em que os processos ficam aguardando no estado de espera até que todos os outros tenham alcançado o mesmo ponto do programa, para que então, sejam desbloqueados e continuem o processamento. Comumente, são adicionadas antes da troca dos *buffers* de imagem;
- Criação das funções para a geração do ponto de vista e associação das mesmas a nós (Associação): cada nó é encarregado de um ponto de vista da imagem de acordo com a sua tela. Por exemplo, em um CAVE um nó deve gerar o ponto de vista da direita, outro da esquerda, e assim por diante. Assim, torna-se necessária a criação de funções capazes de calcular o ponto de visão de cada nó. A associação da função específica a cada nó é realizada durante a inicialização dos mesmos ou no momento de execução;
- Adição das variáveis compartilhadas (compartilhamento de variáveis): a geração coerente das imagens exige que os dados sejam sincronizados antes de cada imagem gerada. Então, o desenvolver deve determinar quais as variáveis serão compartilhadas e sincronizadas. Assim, por exemplo, se a iluminação está habilitada, então todas as imagens das várias telas devem ser geradas levando em consideração esse fato, e

- Tratamento das entradas dos dispositivos de interação (enfileiramento de eventos): os nós que recebem os dados de entrada dos dispositivos devem recebê-los e enviá-los para que os nós de processamento gerem as imagens. Como um *cluster* é composto por diversos nós, então é comum que exista vários nós de entrada de dados, então, por exemplo, um nó pode tratar gestos, enquanto um outro é responsável pelos dados do *tracker*. Embora a geração dos dados de entrada ser assíncrona, o tratamento deles no momento de geração das imagens é síncrona, ou seja, cada entrada é tratada de maneira sequencial.

Apesar da necessidade de poucas alterações para a codificação da distribuição da aplicação no *cluster*, se não for realizada de maneira apropriada, a tarefa de depurar o programa sem uma ferramenta apropriada é complexa, pois depende de dados oriundos de diversos nós. Assim, a maneira visual que o depurador representa a transmissão de dados e sincronia é importante para os desenvolvedores.

4.3 Modelo Conceitual das Funcionalidades

Levando-se em conta as funcionalidades apresentadas na seção anterior e especificando outras, como a inicialização/finalização (funcionalidade Inicialização/Finalização) dos nós, construiu-se o Modelo Conceitual das Funcionalidades. Os detalhes de cada funcionalidade são apresentados nas subseções seguintes.

Cabe a cada funcionalidade padronizar as mensagens que registram as suas ações, por exemplo, quando um dos nós é inicializado, esse envia uma mensagem "*init*" para o servidor do ambiente. Todas mensagens deverão ser armazenadas em arquivos de *logs* para que sejam posteriormente coletadas, analisadas e os resultados sejam exibidos.

Em um sistema centralizado sempre é possível determinar a ordem em que dois eventos aconteceram devido ao fato do sistema ter uma memória comum e um único relógio. Em sistemas distribuídos, onde este modelo comportamental se insere, não temos uma memória e um relógio comum, às vezes é impossível saber a sequência de eventos (Silberschatz *et al.*, 2004).

As próximas sub-seções detalham os cenários das funcionalidades realizadas via troca de mensagens pelas aplicações distribuídas de RV.

4.3.1 Funcionalidade Inicialização/Finalização

Essa funcionalidade tem como objetivo controlar a inicialização e finalização dos nós. Ela é importante porque o nó servidor precisa conhecer os nós que estão ativos ou não e em qual momento. Além disso, o servidor utiliza primitivas dessa funcionalidade para finalizar todos os nós. A Figura 10 ilustra o cenário em que ela é acionada. Inicialmente, os nós C1 e C2 são inicializados (*init*) e informam o servidor (S). Logo após, quando o usuário solicita a finalização, um dos nós, no caso o C1 envia uma solicitação (*killall*) para que o servidor S finalize todos os nós, para isso ele envia uma mensagem (*kill*) para todos os nós ativos.

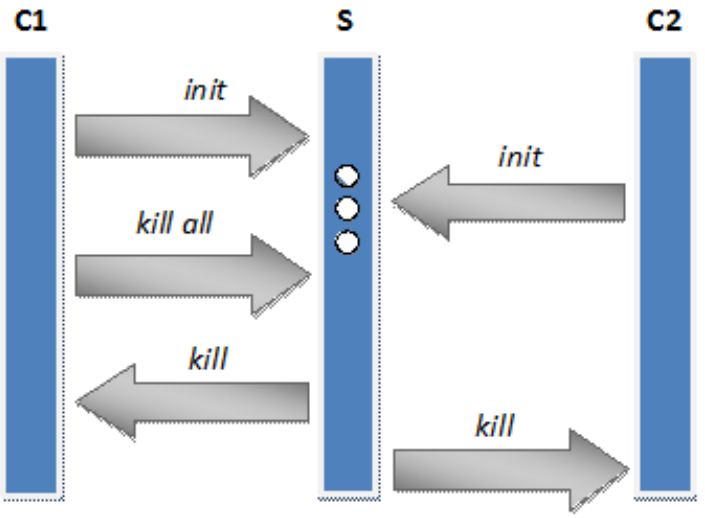

Figura 10 – Funcionalidade Inicialização/Finalização

Cabe ao depurador verificar se o comportamento esperado de cada primitiva está ocorrendo. Para isso, deve responder questões como:

- Qual o momento da inicialização de cada nó?
- Qual o momento da finalização de cada nó?
- Qual nó solicitou a finalização?
- Quais são os nós que estão ativos em determinado momento?

Para realizar a verificação comportamental e responder essas questões, cada mensagem (primitiva) deve fornecer os seguintes dados:

Formato da mensagem: [Id] [Tipo da Mensagem] [Funcionalidade] [Data/Horário] [Origem/Destino] [Mensagem]

- Id: Identificador da mensagem
- Tipo da Mensagem: Indica se é uma mensagem enviada ou recebida
- Funcionalidade: Inicialização/Finalização
- Data/Horário: Momento em que a mensagem é gerada
- Origem/Destino: Indica o número do nó no qual a mensagem se originou e o nó para o qual foi enviada.
- Mensagem: Conteúdo da mensagem trafegada entre nós.
    - *Init* - Pedido para inicialização de um nó.
    - *Kill* - Pedido para finalização de um nó.
    - *KillAll* - Pedido para finalização de todos os nós.

4.3.2 Funcionalidade Compartilhamento de Variáveis

Essa funcionalidade possibilita o compartilhamento de dados entre os nós. A Figura 11 representa um cenário exemplo. O nó C1 cria (*create*) a variável A do tipo compartilhada (*shared*), que é registrada no servidor S. Em seguida, o nó C2 registra (*create*) o interesse na variável – como a variável já existe no servidor, por isso é registrado o interesse. Logo após, o valor da variável é alterado (valor 214) por C1 e é enviado (*sendUpdate*) para o servidor S. Por fim, os nós C1 e C2 solicitam (*getUpdate*) o valor da variável para o servidor S. Como o servidor poderá ter recebido alterações de diversos nós, ele deverá seguir uma regra interna, como uma fila de prioridade, e determinar qual é o valor atual da variável.

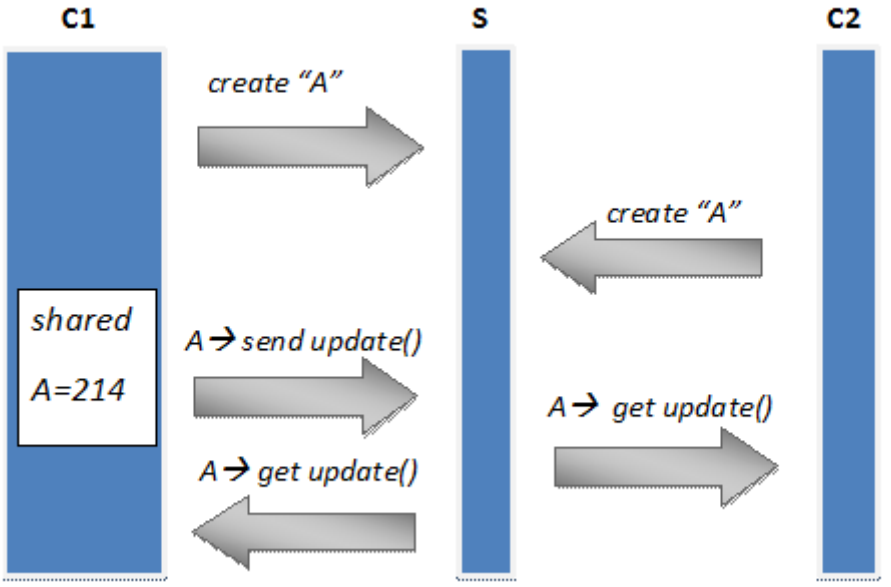

Figura 11 – Funcionalidade Compartilhamento de Variáveis

Além de verificar se o comportamento de compartilhamento de variáveis está ocorrendo conforme o especificado, ele deve responder questões como:

- Qual o valor de uma variável em determinado instante?
- Qual foi o nó que gerou o valor da atualização?
- Quais nós enviaram alterações de variáveis?
- Quais são os nós que compartilham uma determinada variável?

Para realizar a verificação do compartilhamento de variáveis e responder essas questões, cada mensagem deve fornecer os seguintes dados:

Formato da mensagem: [Id] [Tipo da Mensagem] [Funcionalidade] [Data/Horário] [Origem/Destino] [Mensagem] [Variável] [Valor]

- Id: Identificador da mensagem
- Tipo da Mensagem: Indica se é uma mensagem enviada ou recebida
- Funcionalidade: Compartilhamento de Variável
- Data/Horário: Momento em que a mensagem é gerada
- Origem/Destino: Indica o número do nó no qual a mensagem se originou e o nó para o qual foi enviada.

- Mensagem: Conteúdo da mensagem trafegada entre nós.
    - *Create* - indica mensagem de criação da variável.
    - *SendUpdate* - indica o envio do valor de uma variável de um nó para o servidor
    - *GetUpdate* - retorna o valor atualizado do servidor.
- Variável: Nome que identifica o nome da variável.
- Valor: Conteúdo da variável.

4.3.3 Funcionalidade Sincronização de Barreiras

Barreiras de sincronização são pontos na execução nos processos nos quais uns esperam pelos outros. A Figura 12 representa o cenário previsto em uma sincronização de barreiras no tempo t1, no qual estavam ativos apenas o servidor S e o nó C1, que já possui a barreira b1 criada. Nesse momento, o C1 envia uma mensagem de sincronização (*b1.sync*) para o servidor, que libera a barreira b1 (*free b1*). No momento t2 (t2 > t1), o nó C2 torna-se ativo, solicita a sincronização dessa barreira (b1.sync) e fica bloqueado. Como os nós C1 e C2 são os nós que estão trabalhando a barreira b1, então enquanto o C2 não enviar o comando de sincronização (*b1.sync*) para o servidor e liberar todos (*free b1*), o C1 continuará aguardando.

Nas aplicações de RV distribuídas geralmente existem duas barreiras, a de liberação de esvaziamento do *buffer* de imagem (*framelock*) e a de sincronização dos dados (*datalock*), que garante que as imagens serão geradas a partir do mesmo valor. Embora no mesmo programa possam existir diversas barreiras, cada uma deve ser tratada de forma independente, ou seja, uma barreira não pode bloquear outra. Caso contrário, poderá ocorrer situações de *deadlock*.

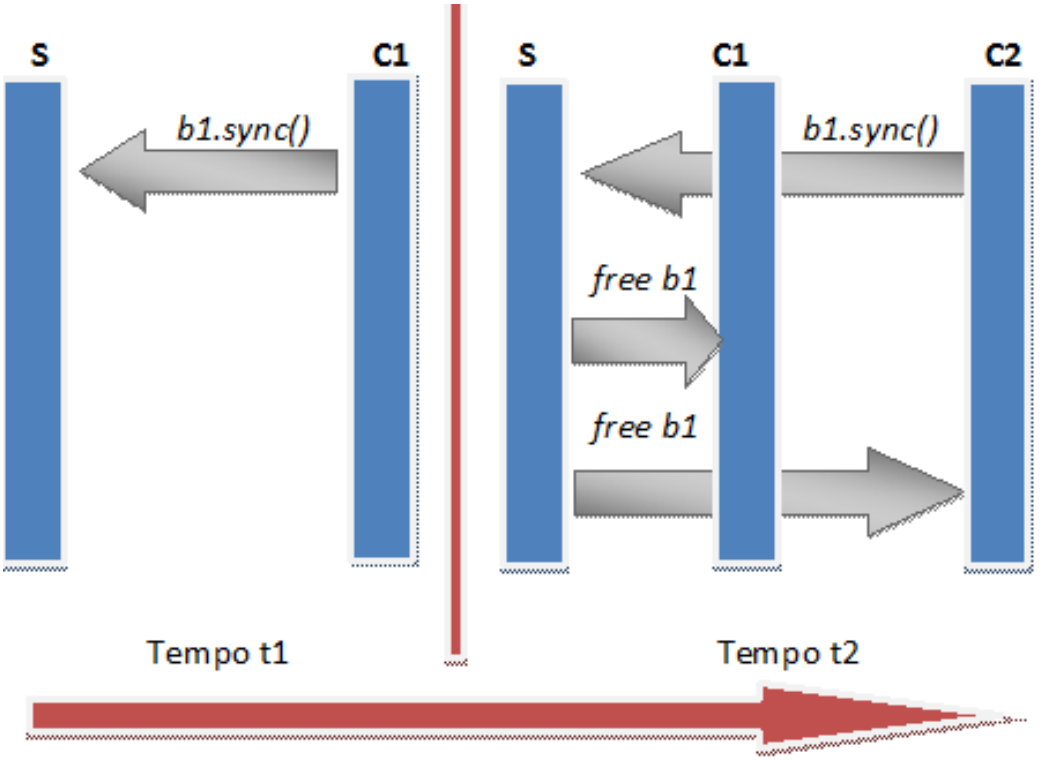

Figura 12 – Funcionalidade Sincronização de Barreiras

Além de verificar se ocorreu o bloqueio e liberação das barreiras, é desejável que as seguintes questões sejam respondidas:

- Quais são os interessados em determinada barreira em dado momento?
- Quais são os nós bloqueados em determinado momento?

Para realizar a verificação da sincronização de barreira e para responder essas questões, cada mensagem deve fornecer os seguintes dados:

Formato da mensagem: [Id] [Tipo da Mensagem] [Funcionalidade] [Data/Horário] [Origem/Destino] [Mensagem] [Variável]

- Id: Identificador da mensagem
- Tipo da Mensagem: Indica se é uma mensagem enviada ou recebida
- Funcionalidade: Barreira
- Data/Horário: Momento em que a mensagem é gerada
- Origem/Destino: Indica o número do nó no qual a mensagem se originou e o nó para o qual foi enviada.

- Mensagem: Conteúdo da mensagem trafegada entre nós.
    - *Create* – cria ou registra interesse na barreira
    - *Sync* – solicita sincronização da barreira ou libera a barreira.
- Variável: Nome que identifica a barreira.

4.3.4 Funcionalidade Associação de Funções

Permite que funções (procedimentos) sejam associados a nós em tempo de execução. A Figura 13 representa o cenário do uso de associação de funções. Inicialmente o nó cliente C1 associa uma função a ele, no exemplo, a função *Frente()* – que calcula a imagem apresentada na frente do usuário em um CAVE. Em seguida, o nó C2 associa outra função a ele, no exemplo, a função *Esquerda()* – que calcula a imagem apresentada da tela da esquerda do usuário em um CAVE. Em continuação, antes da geração da imagem a respectiva função de cada nó é chamada. Se, por exemplo, o C2 deixar de funcionar, a função *Esquerda()* pode ser associada em tempo de execução para outro nó.

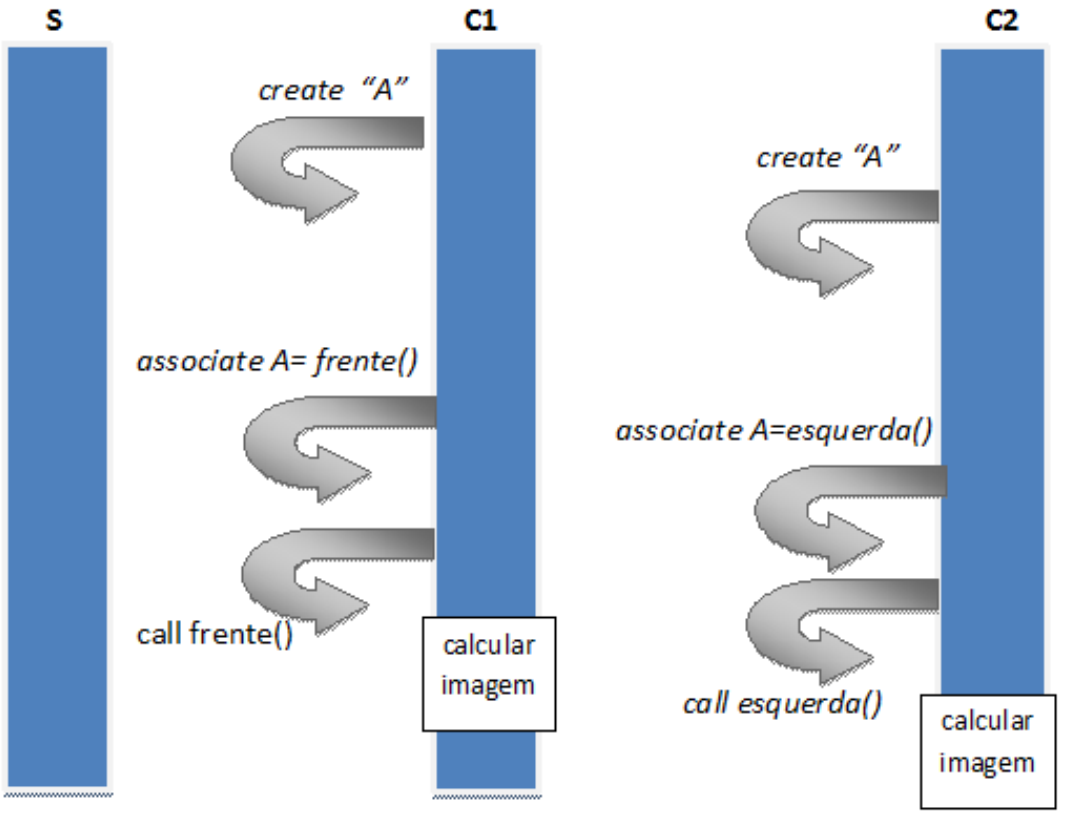

Figura 13 – Cenário com Associação de Funções com recurso *Alias*

A verificação provida pelo depurador deve ser capaz de responder questões como:

- Qual nó chamou qual função?
- O nó está associado a alguma função?

Para realizar a verificação de comportamento da associação de funções e para responder essas questões, cada mensagem deve fornecer os seguintes dados:

Formato da mensagem: [Id] [Tipo da Mensagem] [Funcionalidade] [Data/Horário] [Origem/Destino] [Mensagem] [Função]

- Id: Identificador da mensagem
- Tipo da Mensagem: Indica se é uma mensagem enviada ou recebida
- Funcionalidade: Associação de Funções
- Data/Horário: Momento em que a mensagem é gerada
- Origem/Destino: Indica o número do nó no qual a mensagem se originou ou nó para o qual foi enviada.
- Mensagem: Conteúdo da mensagem trafegada entre nós.
    - *Associate* - associa uma função a um nó
    - *Call* - executa a função associada
- Função: Nome da função associada.

4.3.5 Funcionalidade Enfileiramento de Eventos

Possibilita a transmissão de dados assíncronos entre os nós, o que facilita então o tratamento dos dados dos dispositivos de interação. A Figura 14 ilustra essa funcionalidade. Nesse exemplo, existem três celulares (Celular 1, Celular 2 e Celular 3) que servem como dispositivo de interação com o ambiente, sendo que um dos nós do *cluster* é responsável pelo recebimento desses dados. Então, tanto os celulares quanto o nó receptor registram interesse em uma variável do tipo evento. Todos os dados gerados (*enqueue*) pelos celulares são enfileirados e ficam disponíveis para o nó receptor. Em uma simulação de RV, o receptor faz a leitura dessa fila (*unqueue*) e informa aos outros nós do *cluster* que uma interação foi realizada, como, por exemplo, mudança de ponto de vista. Esse envio de informações para os outros nós é geralmente realizado via compartilhamento de variáveis.

Os três dispositivos celulares e os 4 computadores (C1, C2, C3 e C4) compõem uma rede de 7 nós com capacidade de processamento diferentes entre si. Cada celular está gerando uma variável do tipo evento e cada uma destas variáveis poderá ter um conteúdo distinto entre si. Em C1 ocorrerá o recebimento dos eventos transmitidos, A=”ESQUERDA”, A=”FRENTE” e A=”DIREITA” respectivamente dos celulares 1, 2 e 3, através de troca de mensagens. Esquematicamente, destaca-se a sequência de operações, representadas pelas linhas de 1 até 8, acontecendo no receptor C1:

1 Celular 1 envia A=”ESQUERDA”

2 Celular 2 envia A=”FRENTE”

3 Celular 3 envia A=”DIREITA”

4 No receptor C1 temos a Fila A → ESQUERDA←FRENTE←DIREITA

5 C1 *unqueue* S=A, a variável S receberá o primeiro elemento da Fila A, “ESQUERDA”

6 C1 *sync* S=”ESQUERDA”

7 C1, C2, C3, C4 a partir da sincronização recebem S=”ESQUERDA”

8 C1, C2, C3, C4 calcula (S=”ESQUERDA”)

As linhas 5 até 8 correspondem a um ciclo de linhas de comando que se repetem para cada elemento da Fila A, na sequência será recebido o elemento “FRENTE” e o processamento continua até o último elemento ser recebido.

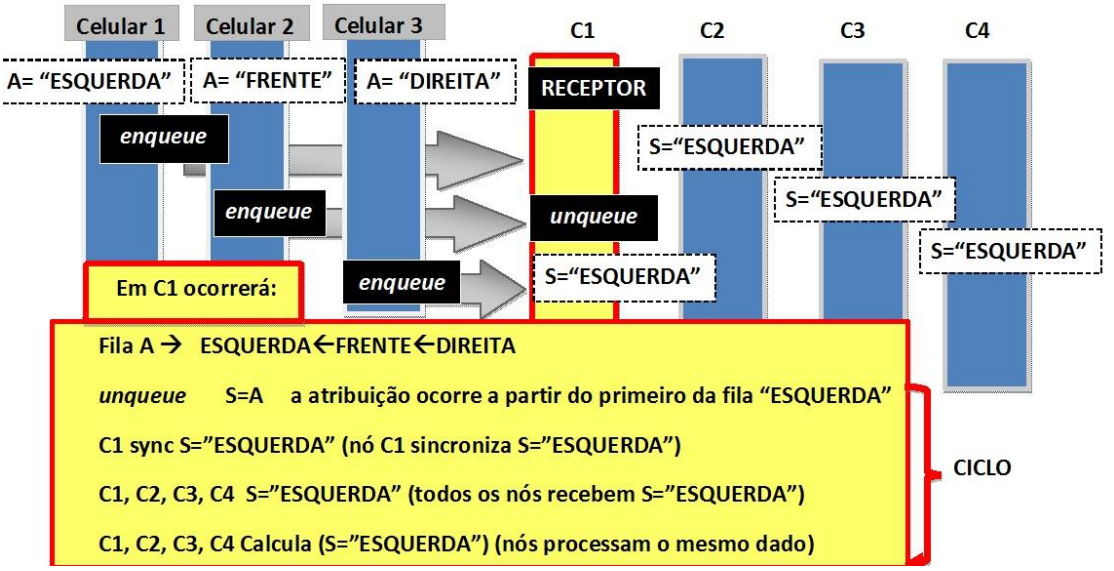

Figura 14 – Funcionalidade Enfileiramento de Eventos

Além de verificar o comportamento dessa funcionalidade, é desejável que o depurador seja capaz de responder os seguintes questionamentos:

- Qual a ordem de recebimento dos eventos de cada nó?
- Qual foi o nó que gerou um determinado evento?

Para realizar a verificação do enfileiramento de evento e para responder essas questões, cada mensagem deve fornecer os seguintes dados:

Formato da mensagem: [Id] [Tipo da Mensagem] [Funcionalidade] [Data/Horário] [Origem/Destino] [Mensagem] [Variável] [Valor]

- Id: Identificador da mensagem
- Tipo da Mensagem: Indica se é uma mensagem enviada ou recebida
- Funcionalidade: Evento
- Data/Horário: Momento em que a mensagem é gerada
- Origem/Destino: Indica o número do nó no qual a mensagem se originou ou nó para o qual foi enviada.
- Mensagem: Conteúdo da mensagem trafegada entre nós.
    - *Create* – cria e/ou registra interesse em uma variável do tipo fila
    - *EnQueue* – adiciona um elemento na fila
    - *Unqueue* – retorna o primeiro elemento da fila.

- Variável: Nome que identifica o nome da variável.
- Valor: Conteúdo da variável.

4.4 Modelo de Execução Global

Levando-se em conta as funcionalidades apresentadas na seção anterior e o comportamento geral das aplicações de RV distribuídas, construiu-se o Modelo de Execução Global (Figura 15). Ele é independente do *hardware* e do *software* utilizado, pois o que difere nas simulações é a distribuição das tarefas entre os nós, onde cada um processa tarefas conforme a sua característica. Por exemplo, os nós com baixa capacidade de processamento gráfico podem ser responsáveis pelo tratamento das interações do usuário ou pelo som. O sistema de multiprojeção adotado também não influencia nesse modelo, pois cada tela adicionada representa apenas um ou mais de um processo no sistema. Contudo, a heterogeneidade de *hardware* e *software* pode influenciar no desempenho do sistema e, consequentemente, na qualidade da imersão e interação.

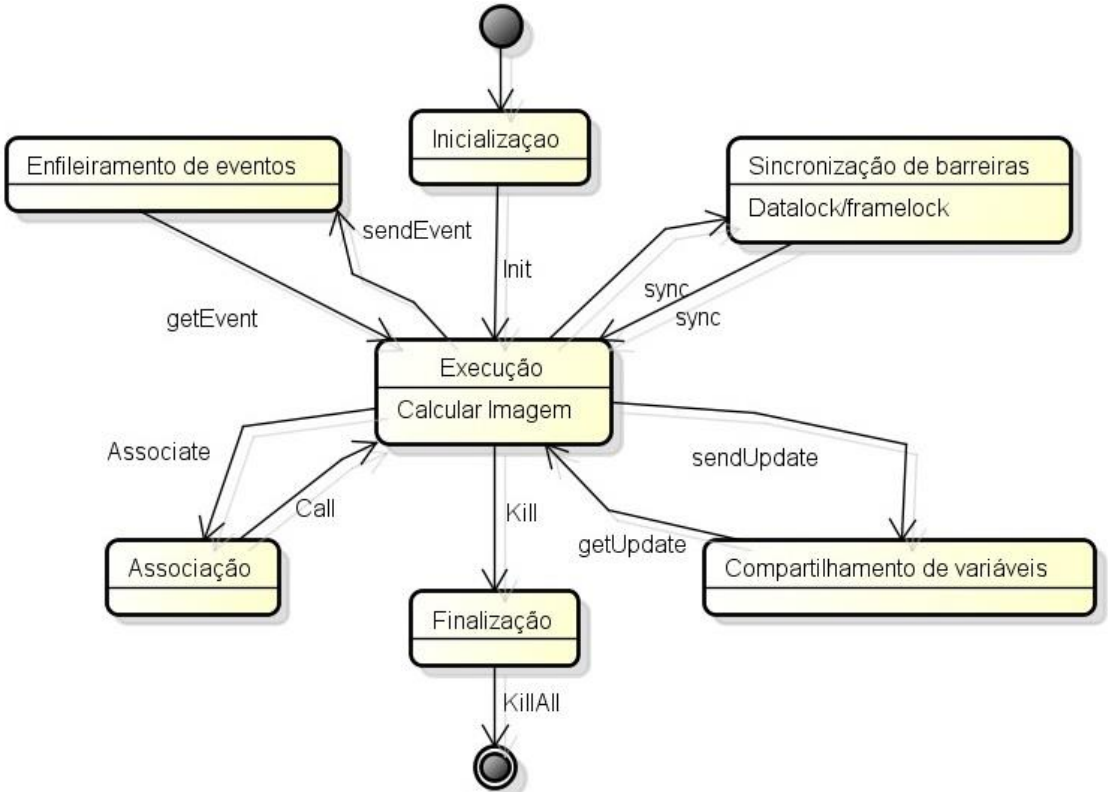

Figura 15 – Modelo de execução global das aplicações de Realidade Virtual Distribuídas

O comportamento das aplicações distribuídas como um todo influencia diretamente nas funcionalidades que o sistema de troca de mensagens deve oferecer. No caso das aplicações de RV distribuídas, para que o sistema gere as saídas, que podem ser, por exemplo, imagens e/ou sons, é necessário que várias outras tarefas, que estão espalhadas entre os nós, funcionem de forma adequada. Para isso é preciso compartilhar e sincronizar os dados de controle, os quais determinam o que desenhar (por exemplo, a direção do ponto de vista do observador em cada nó); e os de mudança do conjunto dos dados (por exemplo, o modelo do objeto deve ser atualizado, pois ocorreu uma mudança na textura). Sempre que um dado compartilhado é alterado (funcionalidade Compartilhamento de variáveis), esse deve ser enviado (*sendUpdate*) para o sistema para que essa alteração seja enviada para todos os nós (*getUpdate*).

Barreiras de sincronização (funcionalidade Sincronização de barreiras) são pontos nos programas que fazem com que os mesmos fiquem parados até que todos tenham chegado a esse ponto (Toscani, 2003; Guimarães, 2004). Antes de gerar qualquer imagem, os dados devem ser sincronizados (*datalock*), pois todos os nós devem levar em conta a mesma informação, por exemplo, a posição do usuário, para que a geração seja coerente. Após a geração das imagens, há também a necessidade de sincronização dos nós a fim de que a exibição das imagens ocorra no mesmo momento (*framelock*). Então, quando um nó que tem função de processamento gráfico acaba de gerar (*render*) uma imagem, esse envia uma mensagem de sincronização para todos os outros nós (*sync*) e espera que todos os outros façam os mesmo. Quando o sistema nota que todos solicitaram a sincronização, então é enviada uma mensagem *"free"* a todos para continuarem as atividades.

As interações dos usuários alteram diretamente os dados compartilhados, então o sistema recebe esses eventos (funcionalidade Enfileiramento de eventos), os enfileira e, por fim, envia (*sendEvent*) de forma sequencial para que todos os nós processem os dados relacionados a ele (*getEvent*). Como existem vários nós envolvidos no sistema, cada um pode executar tarefas diferentes e em momentos diferentes. Assim, em um momento qualquer um dos nós pode ser associado a função de geração da imagem frontal em um sistema de multiprojeção e outro a uma imagem lateral. Em qualquer tempo de execução essa associação pode ser atribuída (*associate*) e o nó pode executá-la (*call*) - (funcionalidade Associação). Outro comportamento global ocorre quando é

gerada a mensagem de finalização da aplicação (*kill*), a qual é enviada para todos os nós (*killAll*) (funcionalidade Inicialização/Finalização).

4.5 Modelo Comportamental da Aplicação em Redes de Petri

O Modelo de execução apresentado anteriormente foi capaz de especificar o funcionamento global das aplicações de RV distribuídas, porém ainda não foi o suficiente para formalizar. Por isso, utilizou-se as Redes de Petri com o objetivo de obter a modelagem comportamental destas aplicações como passo para obter as pré-condições e pós-condições que permitam o entendimento de uso de tais funcionalidades. O uso destas redes permitiu aprofundar na descrição e análise de todos as funcionalidades envolvidas, pois elas são capazes de representar todos os elementos envolvidos destas aplicações, como concorrência, sincronização e compartilhamento.

4.5.1 Inicialização/Finalização

A Figura 16 mostra a Rede de Petri que representa a funcionalidade inicialização. Na possibilidade de um determinado nó emissor, por exemplo C1, necessitar entrar em atividade, este poderá enviar um comando para inicialização para o nó receptor, por exemplo S1, que passará a exercer o papel de servidor. Neste momento, o nó S1 recebendo este comando de inicialização estará ciente da entrada de C1 em execução.

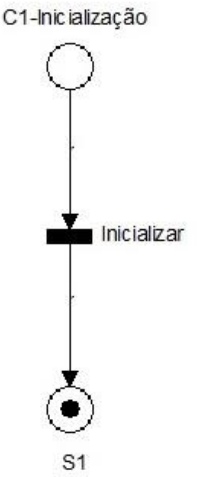

Figura 16 – Modelo Comportamental da Aplicação – Inicialização

A Figura 17 mostra a Rede de Petri que representa a funcionalidade finalização. Nela a partir de um nó emissor qualquer, no exemplo foi escolhido o nó C1, é possível comandar os processos de finalização que devem ser executados em todos os nós pertencentes a rede. Esta rede é composta por 4 lugares (C1, C2, Cn, S1) e duas transições (Finalização, Finalizar). Os lugares representam variáveis de estado e as transições representam ações realizadas pelo sistema em estudo.

A simulação desta rede (Figura 17) inicia-se quando um determinado nó emissor, por exemplo C1, envia a qualquer outro nó receptor, que poderá funcionar como nó servidor, por exemplo S1, um comando para a finalização de todos os outros nós componentes da rede. Imediatamente o nó receptor S1, que estará funcionando como servidor, estará distribuindo informação para todos os nós da rede (C1 até Cn) para que estes possam estar prontos para executar os procedimentos de finalização.

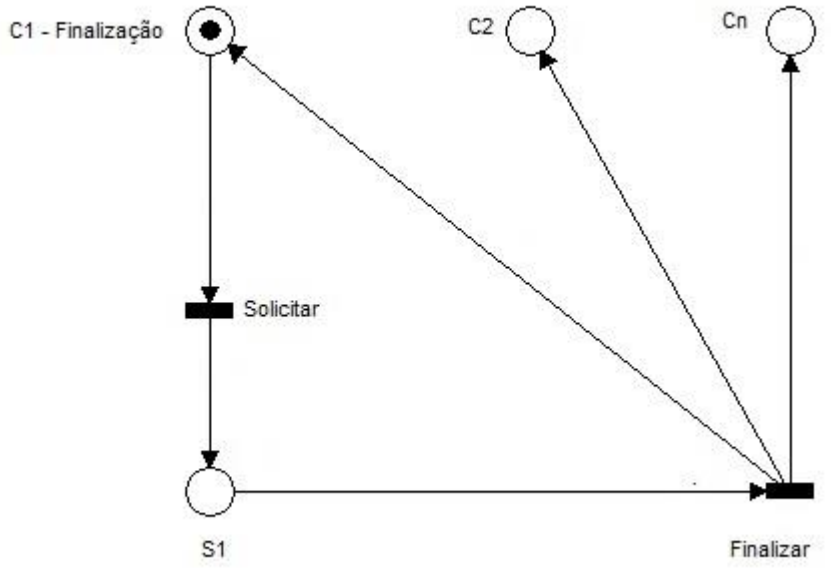

Figura 17 – Modelo Comportamental da Aplicação – Finalização

4.5.2 Ciclo de Geração de Imagens

A Figura 18 mostra o modelo comportamental desenvolvido, que representa o ciclo de geração de imagens de uma aplicação de RV distribuída, que é projetada em

várias telas (multiprojetada) e que recebe interações de diversos dispositivos de entrada. Essa rede é composta por nove lugares (Dispositivo 1, Dispositivo 2, Dispositivo n, Fila gerada, Variáveis dessincronizadas, Variáveis sincronizadas, Visão definida, Imagem no *buffer*, *Buffer* vazio) e seis transições (Enfileirar eventos, Compartilhar variáveis, Fazer o *Datalock*, Associar ponto de vista, Calcular imagem, Fazer o *Framelock*), sendo que os lugares definem variáveis de estado e as transições definem ações realizadas pelo sistema. Uma ação acontece relacionada a algumas pré-condições (variáveis de estado). Portanto, lugares e transições são interligados por estas pré-condições que possibilitam a realização de uma determinada ação. Após a execução de uma transição os lugares terão suas informações alteradas (pós-condições).

A simulação dessa rede (Figura 18) inicia-se com os Dispositivos 1 até n (luvas, *mouse*, rastreadores e outros) gerando eventos (mudança de ponto de vista, por exemplo), que disparam o enfileiramento (Enfileirar eventos) e resulta em uma fila (Fila gerada). Assim que existe algum dado enfileirado, é acionada ação de compartilhamento do valor (Compartilhar variáveis) com todos os nós responsáveis pela geração das imagens, resultando então em valores dessincronizados (Variáveis dessincronizadas). Em sequência, ocorre a sincronização dos dados (Fazer o *Datalock*), o que resulta em variáveis com valores iguais em todos os nós (Variáveis sincronizadas). Desse modo, cada nó torna-se apto a executar (Variáveis sincronizadas) a sua função de geração de imagens, assim o nó ficará responsável pela imagem de uma das telas de projeção (Visão definida). Em seguida, cada nó calcula a sua imagem (Calcular imagem) e a armazena no *buffer* da placa de vídeo (Imagem no *buffer*). Logo após, ocorre a sincronização de quadros (Fazer o *Framelock*), que bloqueará todos os nós até que estes estejam com as suas respectivas imagens geradas no *buffer*. Por fim, as imagens dos *buffers* são enviadas para os dispositivos de saída (*Buffer* vazio), o que habilitará a execução do ciclo novamente. Se nesse novo ciclo não houver valores na fila, então as imagens são geradas com os valores atuais.

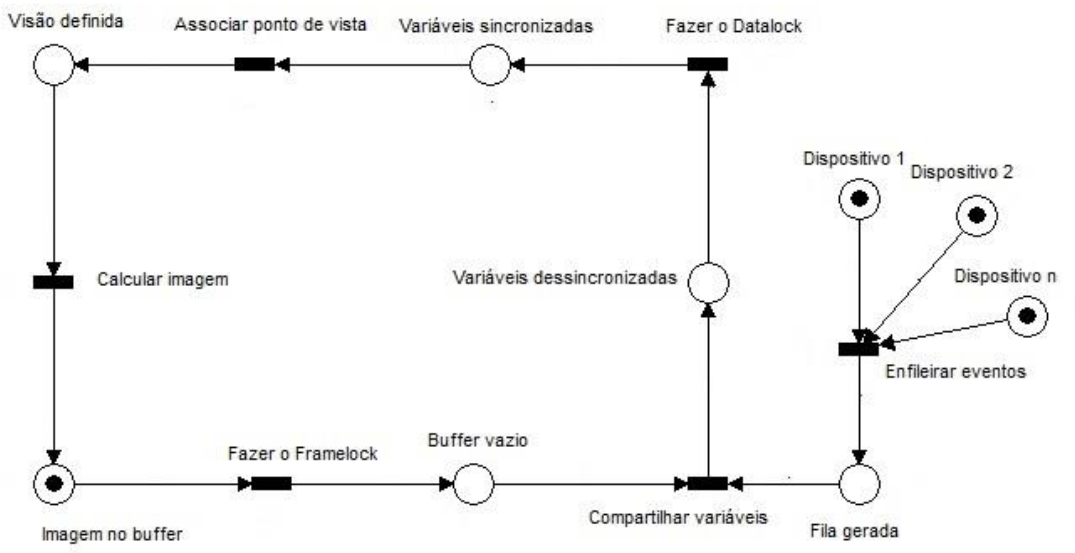

Figura 18 – Modelo Comportamental da Aplicação – Ciclo de Geração das Imagens

4.6 Pré-Condições e Pós-Condições para a Depuração das Mensagens

A ideia de criar pré-condições e pós-condições é permitir especificar formalmente o comportamento que o depurador deve verificar. Enquanto as pré-condições indicam o que deve ocorrer antes que uma função seja chamada, as pós-condições indicam o que acontecerá depois que a função for chamada. As Redes de Petri, apresentadas anteriormente, podem ser definidas formalmente de diversas formas, dentre elas, com a Teoria *Bag*, que é uma generalização do conceito de conjunto, possibilitando a repetição de elementos. Nesse trabalho optou-se por essa teoria porque ocorrem repetições de elementos e portanto ela atende plenamente a necessidade para a representação formal. Dessa forma, tem-se o mapeamento das transições para os lugares (*Bags*). Para outras situações diferentes deste trabalho (onde não ocorrem repetições de elementos) existe a possibilidade de Redes de Petri serem representadas por notação matricial ou pode-se realizar a definição através de relações matemáticas e pesos associados a estas relações (Maciel, Lins e Cunha, 1996).

4.6.1 Inicialização/Finalização

4.6.1.1 Inicialização

A rede da Figura 16, que representa a funcionalidade inicialização, é composta por 2 lugares (C1, S1) e uma transição (Inicializar). Os lugares representam variáveis de estado e a transição representa a ação realizada pelo sistema.

Define-se a quíntupla Rinicializacao = (P,T,I,O,K) onde o conjunto de lugares P é P={C1-Inicialização, S1}, o conjunto de transições T é T={Inicializar}, o conjunto de *bags* de entrada I é I={I(Inicializar) = [C1]}, o conjunto de *bags* de saída O é O={O(Inicializar) = [S1]}, e o conjunto de capacidades de lugares é k={kC1 = 1, KS1 = 1}.

Através do modelo comportamental da aplicação em Redes de Petri e pelo formalismo matemático pela Teoria *Bag* (Peterson, 1981) extrai-se a seguinte pré-condição e pós-condição necessária para o cenário de inicialização:

Pré-Condição: I(Inicializar)=[C1]

Pós-Condição: O(Inicializar)=[S1]

A pré-condição e a pós-condição associadas à funcionalidade inicialização possibilitam identificar as situações de uso desta funcionalidade durante a execução da aplicação. A pré-condição para a realização de determinada ação é representada pelos *bags* $I(t_i)$, ou seja, para que a transição “Inicializar” possa acontecer, é necessário que a condição “C1” seja verdadeira, dado que I(Inicializar)=[C1]. O lugar que é pós-condição da transição t_i é representado pelos *bags* $O(t_i)$. A transição “Inicializar” tem como lugar de saída O(Inicializar)=[S1].

Desta forma, em tempo de execução da aplicação, para que a funcionalidade inicialização seja executada é necessário que um determinado nó cliente solicite a inicialização. Esta mensagem é enviada para qualquer outro nó receptor que funcionará como servidor e procederá a execução desta solicitação e este nó (servidor) estará ciente de que o nó cliente solicitante está ativo.

4.6.1.2 Finalização

A rede da Figura 17, que representa a funcionalidade finalização, é composta por 4 lugares (C1, C2, Cn e S1) e duas transições (Solicitar, Finalizar). Os lugares representam variáveis de estado e as transições representam ações realizadas pelo sistema.

Define-se a quíntupla Rfinalizacao = (P,T,I,O,K) onde o conjunto de lugares P é P={C1, C2, Cn, S1}, o conjunto de transições T é T={Solicitar, Finalizar}, o conjunto de *bags* de entrada I é I={I(Solicitar) = [C1], I(Finalizar) = [S1]}, o conjunto de *bags* de saída O é O={O(Solicitar) = [S1], O(Finalizar)=[C1,C2,Cn]}, e o conjunto de capacidades de lugares é k={kC1 = 1, kC2 = 1, kCn = 1, KS1=1}.

Através do modelo comportamental da aplicação em Redes de Petri e pelo formalismo matemático pela Teoria *Bag* (Peterson, 1981) extrai-se as seguintes pré-condições e pós-condições necessárias para o cenário de finalização:

Pré-Condição: I(Solicitar)=[C1]

Pós-Condição: O(Solicitar)=[S1]

Pré-Condição: I(Finalizar)=[S1]

Pós-Condição: O(Finalizar)=[C1,C2,Cn]

As pré-condições e pós-condições associadas à funcionalidade finalização possibilitam identificar as situações de uso desta funcionalidade durante a execução da aplicação. As pré-condições para a realização das ações são representadas pelos *bags* $I(t_i)$, ou seja, para que a transição "Solicitar" possa acontecer, é necessário que a condição "C1" seja verdadeira, dado que I(Solicitar)=[C1]. Da mesma forma, para que "Finalizar" possa acontecer, é necessário que a condição "S1" seja verdadeira, dado que I(Finalizar)=[S1].

Os lugares que são pós-condições das transições t_i são representados pelos *bags* $O(t_i)$. A transição "Solicitar" tem como lugar de saída O(Solicitar)=[S1] e a transição "Finalizar" tem como lugar de saída O(Finalizar)=[C1,C2,Cn].

Desta forma, em tempo de execução da aplicação, para que a funcionalidade finalização seja executada é necessário que um determinado nó cliente solicite a

finalização. Esta mensagem é enviada para qualquer outro nó receptor, que funcionará como servidor, e este procederá a execução dos procedimentos de finalização ("Finalizar") para todos os outros nós da rede (C1, C2, Cn).

4.6.2 Ciclo de Geração de Imagens

O Modelo Comportamental da Figura 18, representado por uma Rede de Petri (Maciel, Lins e Cunha, 1996) que é formada por nove lugares e seis transições, que representam respectivamente as variáveis de estados e as ações realizadas pelo sistema. As pré-condições para a realização das ações são representadas pelos *bags* $I(t_i)$, ou seja, para (Associar ponto de vista), é necessário que a condição (Variáveis sincronizadas) seja verdadeira, dado que I(Associar ponto de vista)=[Variáveis sincronizadas]. Os lugares que são pós-condições das transições ti são representados pelos *bags* $O(t_i)$. A transição (Associar ponto de vista) tem como lugar de saída O(Associar ponto de vista)=[Visão definida].

Define-se a quíntupla Rgeracaodeimagens = (P,T,I,O,K) onde o conjunto de lugares P é P={Dispositivo 1, Dispositivo 2, Dispositivo n, Fila gerada, Variáveis dessincronizadas, Variáveis sincronizadas, Visão definida, Imagem no *buffer*, *Buffer* vazio}, o conjunto de transições T é T={Enfileirar eventos, Compartilhar variáveis, Fazer o *Datalock*, Associar ponto de vista, Calcular imagem, Fazer o *Framelock*}.

O conjunto de *bags* de entrada I é I={I(Enfileirar eventos) = [Dispositivo 1, Dispositivo 2, Dispositivo n], I(Compartilhar variáveis) = [Fila gerada, *Buffer* vazio], I(Fazer o *Datalock*) = [Variáveis dessincronizadas], I(Associar ponto de vista) = [Variáveis sincronizadas], I(Calcular imagem) = [Visão definida], I(Fazer o *Framelock)* = [Imagem no *buffer*]}.

O conjunto de *bags* de saída O é O={O(Enfileirar eventos) = [Fila gerada], O(Compartilhar variáveis) = [Variáveis dessincronizadas], O(Fazer o D*atalock*)=[Variáveis sincronizadas], O(Associar ponto de vista) = [Visão definida], O(Calcular Imagem) = [Imagem no *Buffer*], O(Fazer o *Framelock*) = [*Buffer vazio*]}.

O conjunto de capacidades de lugares é k={kDispositivo 1 = 1, kDispositivo 2 = 1, kDispositivo n = 1, kFila gerada = 1, kVariáveis dessincronizadas = 1, kVariáveis sincronizadas = 1, kVisão definida = 1, kImagem no *buffer* = 1, k*Buffer* vazio = 1}.

4.6.2.1 Compartilhamento de Variáveis

Neste cenário de compartilhamento de variáveis o depurador deverá apresentar a troca de mensagens entre os nós e possibilitar responder perguntas que identificam o valor de uma variável em determinado instante ou o nó que gerou o valor da atualização.

Através do modelo comportamental da aplicação em Redes de Petri e pelo formalismo matemático pela Teoria *Bag* (Peterson, 1981) extrai-se as seguintes pré-condição e pós-condição necessárias para o cenário de Compartilhamento de Variáveis:

Pré-Condição: I(Compartilhar variáveis) = [Fila gerada, *Buffer* vazio],

Pós-Condição: O(Compartilhar variáveis)=[Variáveis dessincronizadas]

A pré-condição e pós-condição associadas à funcionalidade compartilhamento de variáveis possibilitam identificar as situações de uso dela durante a execução da aplicação. A pré-condição para a realização das ações é representada pelos *bags* $I(t_i)$, ou seja, para que "Compartilhar variáveis" possa acontecer, é necessário que as condições "Fila Gerada" ou "*Buffer* vazio" sejam verdadeiras, dado que I(Compartilhar Variáveis)=[Fila Gerada, *Buffer* vazio]. O lugar que é a pós-condição da transição t_i é representado pelos *bags* $O(t_i)$. A transição "Compartilhar variáveis" tem como lugar de saída O(Compartilhar Variáveis)=[Variáveis dessincronizadas], ou seja, após ocorrer a ação de compartilhamento de variáveis na rede, as variáveis ficam num estado sem sincronização.

Desta forma, nesta instância, em tempo de execução da aplicação, para que a funcionalidade compartilhar variáveis seja executada é necessário que uma das duas condições tenham acontecido ou que a fila tenha sido gerada ou a outra situação de *buffer* vazio tenha ocorrido. Após a funcionalidade compartilhar variáveis ter sido executada torna-se necessário a execução da sincronização de barreiras. No modelo comportamental do ciclo de geração de imagens (Figura 18), na sequência da transição "Compartilhar variáveis", a próxima transição "Fazer o *Datalock*" que está interligada por uma seta com o lugar de entrada "Variáveis dessincronizadas" realizará a sincronização dos dados.

4.6.2.2 Sincronização de Barreiras

Neste cenário de sincronização o depurador deverá apresentar os nós que fazem parte da sincronização de computadores através da utilização do recurso barreira e possibilitar responder perguntas indentificando se ocorreu *deadlock* ou se todos os nós estão esperando *sync()* para dar continuidade.

Através do modelo comportamental da aplicação em Redes de Petri e pelo formalismo matemático pela Teoria *Bag* (Peterson, 1981) extrai-se as seguintes pré-condições e pós-condições necessárias para o cenário de sincronização de dados (*Datalock*) e sincronização de imagens (*Framelock*):

Pré-Condição: I(Fazer o *Datalock*)=[Variáveis dessincronizadas]

Pós-Condição: O(Fazer o *Datalock*)=[Variáveis sincronizadas]

Pré-Condição: I(Fazer o *Framelock*)=[Imagem no *buffer*]

Pós-Condição: O(Fazer o *Framelock*)=[*Buffer* vazio]

As pré-condições e pós-condições associadas à funcionalidade sincronização de barreiras possibilitam identificar as situações de uso desta funcionalidade durante a execução da aplicação. As pré-condições para a realização das ações são representadas pelos *bags* I(t_i), ou seja, para que "Fazer o *Datalock*" possa acontecer, é necessário que a condição "Variáveis dessincronizadas" seja verdadeira, dado que I(Fazer o *Datalock*)=[Variáveis dessincronizadas]. Da mesma forma, para que "Fazer o *Framelock*" possa acontecer, é necessário que a condição "Imagem no *buffer*" seja verdadeira, dado que I(Fazer o *Framelock*)=[Imagem no *buffer*].

Os lugares que são pós-condições das transições t_i são representados pelos *bags* O(t_i). A transição "Fazer o *Datalock*" tem como lugar de saída O(Fazer o *Datalock*)=[Variáveis sincronizadas] e a transição "Fazer o *Framelock*" tem como lugar de saída O(Fazer o *Framelock*)=[*Buffer* vazio].

Desta forma, nesta instância, em tempo de execução da aplicação, para que a funcionalidade sincronização de barreiras seja executada é necessário para haver a sincronização de dados (Fazer o *Datalock*) que tenha ocorrido o compartilhamento de variáveis e as variáveis estejam no estado "dessincronizadas". Para haver a

sincronização de quadros (Fazer o *Framelock*) é necessário que a imagem esteja no *buffer*. No modelo comportamental do ciclo de geração de imagens (Figura 18), na sequência da transição "Fazer o *Datalock*", a próxima transição "Associar ponto de vista" que está interligada por uma seta com o lugar de entrada "Variáveis sincronizadas" realizará a associação de funções.

4.6.2.3 Associação de Funções

Neste cenário de associação de funções a verificação provida pelo depurador deverá ser capaz de responder questões que identifiquem o nó chamador de uma função ou se determinado nó está associado a alguma função.

Através do modelo comportamental da aplicação em Redes de Petri e pelo formalismo matemático pela Teoria *Bag* (Peterson, 1981) extrai-se as seguintes pré-condições e pós-condições necessárias para o cenário de associação de funções, nas transições associar ponto de vista e calcular imagem:

Pré-Condição: I(Associar ponto de vista) = [Variáveis sincronizadas]

Pós-Condição: O(Associar ponto de vista)=[Visão definida]

Pré-Condição: I(Calcular imagem)= [Visão definida]

Pós-Condição: O(Calcular Imagem)=[Imagem no *buffer*]

As pré-condições e pós-condições associadas à funcionalidade associação de funções possibilitam identificar as situações de uso desta funcionalidade durante a execução da aplicação. As pré-condições para a realização das ações são representadas pelos *bags* $I(t_i)$, ou seja, para que "Associar ponto de vista" possa acontecer, é necessário que a condição "Variáveis sincronizadas" seja verdadeira, dado que I(Associar ponto de vista)=[Variáveis sincronizadas]. Da mesma forma, para que "Calcular imagem" possa acontecer, é necessário que a condição "Visão definida" seja verdadeira, dado que I(Calcular imagem)=[Visão definida].

Os lugares que são pós-condições das transições t_i são representados pelos *bags* $O(t_i)$. A transição "Associar ponto de vista" tem como lugar de saída O(Associar ponto de vista)=[Visão definida] e a transição "Calcular imagem" tem como lugar de saída O(Calcular imagem)=[Imagem no *buffer*].

Desta forma, nesta instância, em tempo de execução da aplicação, para que a funcionalidade associação de funções seja executada na transição "Associar ponto de vista" torna-se necessário que as variáveis estejam no estado de "sincronizadas" e na transição "Calcular imagem" que o estado de "Visão definida" seja verdadeiro. No modelo comportamental do ciclo de geração de imagens (Figura 18), na sequência da transição "Calcular imagem", a próxima transição "Fazer o *Framelock*" interligada por uma seta com o lugar de entrada "Imagem no *buffer*" realizará a sincronização de quadros.

4.6.2.4 Enfileiramento de Eventos

Neste cenário de enfileiramento de eventos o depurador deverá apresentar a troca de mensagens entre os nós e possibilitar responder questões que identifiquem a ordem de recebimento dos eventos de cada nó ou que nó gerou um determinado evento.

Através do modelo comportamental da aplicação em Redes de Petri e pelo formalismo matemático pela Teoria *Bag* (Peterson, 1981) extrai-se as seguintes pré-condição e pós-condição necessárias para o cenário de enfileiramento eventos:

Pré-Condição: I(Enfileirar eventos) = [Dispositivo 1, Dispositivo 2, Dispositivo n]

Pós-Condição: O(Enfileirar eventos) = [Fila gerada]

A pré-condição e a pós-condição associadas à funcionalidade enfileiramento de eventos possibilitam identificar as situações de uso desta funcionalidade durante a execução da aplicação. A pré-condição para a realização de determinada ação é representada pelos *bags* $I(t_i)$, ou seja, para que a "Enfileirar eventos" possa acontecer, é necessário que uma das condições "Dispositivo 1", ou "Dispositivo 2" ou "Dispositivo n" seja verdadeira, dado que I(Enfileirar eventos)=[Dispositivo 1, Dispositivo 2, Dispositivo n]. O lugar que é pós-condição da transição t_i é representado pelos *bags* $O(t_i)$. A transição "Enfileirar eventos" tem como lugar de saída O(Enfileirar eventos)=[Fila gerada].

Desta forma, nesta instância, em tempo de execução da aplicação, para que a funcionalidade enfileiramento de eventos seja executada na transição "Enfileirar eventos" torna-se necessário que pelo menos um dos dispositivos (1, 2 ou n) seja

verdadeiro, ou seja, esteja ativo. No modelo comportamental do ciclo de geração de imagens (Figura 18), na sequência da transição “Enfileirar eventos”, a próxima transição “Compartilhar variáveis” interligada por uma seta com o lugar de entrada “Fila gerada” realizará o compartilhamento de variáveis.

4.7 Notas Finais

Aplicações de RV baseadas em *cluster* possuem grande quantidade de troca de mensagens entre seus nós, pois durante sua execução, qualquer modificação efetuada no ambiente deve ser compartilhada. O sincronismo entre os nós também deve ser realizado a cada frame desenhado. Visto que uma aplicação desta gera no mínimo 90 frames por segundo (Regan e Pose, 1994) , então pode-se afirmar que o sincronismo é uma tarefa de extrema importância. Se houver alguma inconsistência neste tipo de mensagem, a aplicação pode não ser executada da maneira esperada.

Esse capítulo identificou e formalizou as funcionalidades específicas das aplicações de RV distribuídas, através da geração de modelos independentes do depurador a ser desenvolvido para verificação das trocas de mensagens. Estes modelos resultaram em pré-condições e pós-condições necessárias para a construção deste depurador. Neste trabalho estes modelos permitiram a construção do depurador GTracer.

5. GTracer: analisador comportamental

Neste capítulo é apresentada a ferramenta de suporte que analisa o comportamento das aplicações de RV distribuídas desenvolvidas com a libGlass, que é o GTracer. É demonstrada a verificação das funcionalidades especificadas no capítulo anterior. Para isso, foram criados diversos cenários de testes.

Dessa forma, essa ferramenta reforça um dos pilares da libGlass, que é de facilitar o desenvolvimento de aplicações distribuídas.

5.1 Introdução

Como em qualquer tipo de desenvolvimento de *software*, as aplicações baseadas em *clusters* também são propícias a erros. No desenvolvimento de aplicações com memória compartilhada, a identificação de possíveis erros lógicos de programação pode ser facilmente realizada com o auxílio de um depurador.

As linguagens de programação, geralmente, possuem algum tipo de depurador para auxiliar o desenvolvedor na identificação dos erros. No caso das aplicações distribuídas, as mensagens trafegadas entre os nós não podem ser visualizadas em depuradores convencionais, pois estes se preocupam com a estrutura da linguagem e não com mensagens de comunicação e sincronização.

Existem alguns depuradores com foco no desenvolvimento de aplicações baseadas em *grid* e *clusters*, contudo, a maioria deles visam realizar a verificação de código e processos distribuídos. Hood e Jost (2000) desenvolveram um depurador para ambientes baseados em *grid*. Esse depurador pode encontrar processos distribuídos, mesmo que estes não tenham sido criados sob o controle do depurador, porém, não apresentam as informações relacionadas a troca de mensagens.

O desenvolvimento ou suporte de aplicações com a libGlass não é dependente de processos distribuídos, e sim, de troca de mensagens. Portanto, um depurador que consiga apresentar informações a respeito das mensagens é de extrema importância.

As mensagens das aplicações desenvolvidas com a libGlass podem ser visualizadas graficamente por meio do GTracer na sua versão inicial apresentada em 2004 (Guimarães, 2004). Essa versão restringia-se a apenas visualizar as mensagens que

trafegam entre os nós de um *cluster*, apresentando informações como tipo da mensagem, tipo de dado, origem e destino.

A versão atual do GTracer, implementada neste trabalho, permite a análise do comportamento das aplicações, tendo como entrada arquivos de *logs* e saídas gráficas com o diagnóstico. Esse diagnóstico é capaz de garantir a integridade da biblioteca a cada nova implementação, já que é possível comparar o comportamento da biblioteca antes e após o desenvolvimento.

A Figura 19 mostra o modelo de trabalho desenvolvido que permitiu construir o GTracer:

- Desenvolvimento de Aplicações de Realidade Virtual Distribuídas: levantou-se os requisitos gerais dessas aplicações, como, por exemplo, a necessidade de sincronia dos dados (*datalock)* e de frames (*framelock*);
- Modelo Conceitual das Funcionalidades: definiu-se as funcionalidades específicas: inicialização/finalização, compartilhamento de variáveis, sincronização de barreiras, associação de funções e enfileiramento de eventos.
- Modelo de Execução Global: nessa etapa identificou-se como as funcionalidades trabalham em conjunto para a execução das aplicações. Essa fase resultou no levantamento das primitivas trocadas entre os nós;
- Modelo Comportamental da Aplicação em Redes de Petri: especificou-se todo o ciclo de execução das aplicações, considerando desde o momento de recebimento de um dado de entrada até a apresentação das imagens nos dispositivos de execução;
- Pré-Condições e Pós-Condições para Depuração de Mensagens (Teoria *Bag)*: chegou-se nessa etapa nas condições de entrada e saída que ilustram as situações de uso das funcionalidades;
- GTracer: construiu-se a ferramenta capaz de fazer a análise de comportamento das aplicações de RV distribuídas (foco desse capítulo).

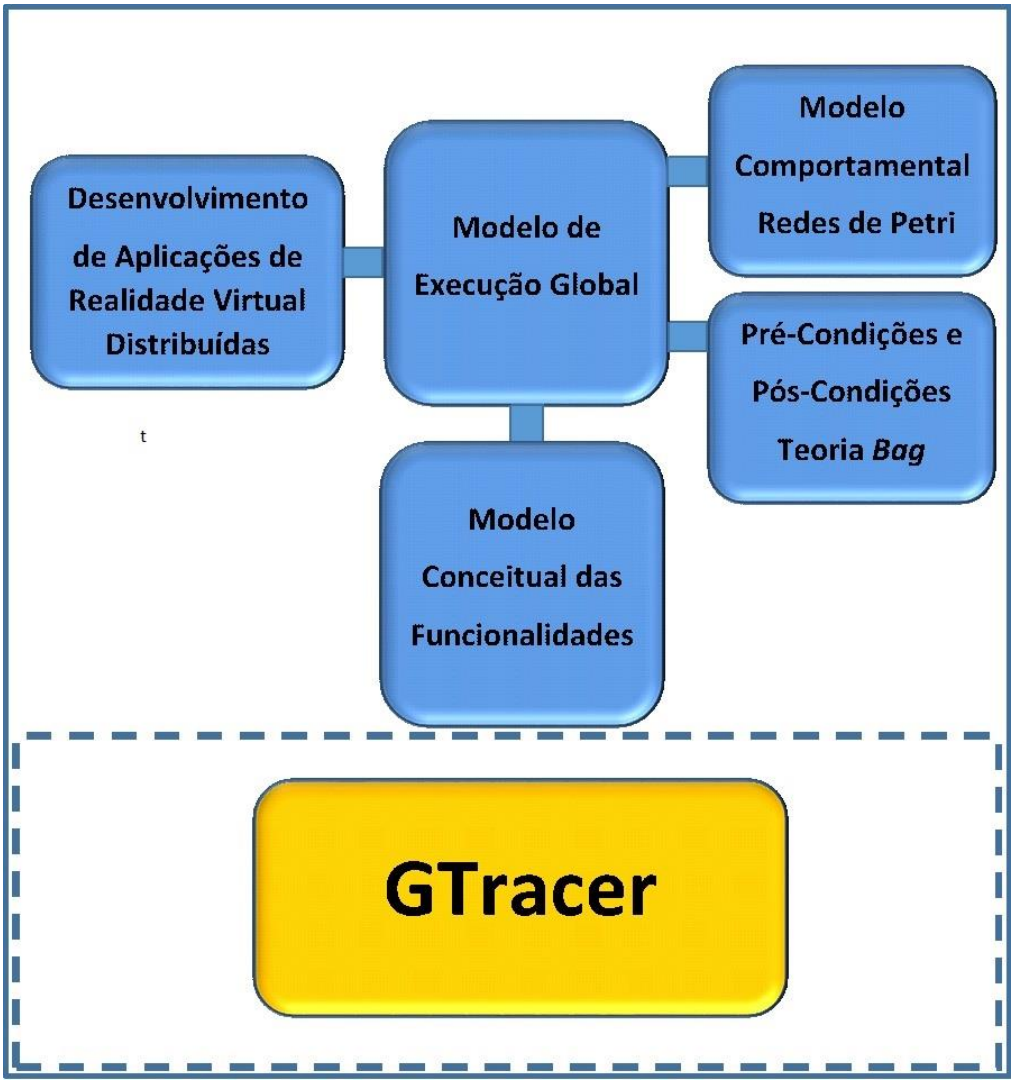

Figura 19 – Modelo de Trabalho Desenvolvido

As funcionalidades identificadas podem ser implementadas por qualquer biblioteca de troca de mensagens. Porém, para a validação das especificações realizadas neste trabalho, foi necessário a escolha de uma biblioteca específica, no caso a libGlass. Essa biblioteca permite o armazenamento de todas as mensagens enviadas/recebidas pelos nós em arquivos de *log* em tempo de execução. A geração desses arquivos é somente realizada quando é habilitada em tempo de compilação. Se não houver interesse na depuração, esta deve estar desabilitada, pois consome tempo de gravação em disco, o que afeta o desempenho da aplicação, afetando diretamente o grau de imersão e interação. Geralmente, ela é habilitada para a detecção de erros durante o desenvolvimento das aplicações ou para fins de aprendizagem.

A Figura 20 apresenta a interface do GTracer, nela o desenvolvedor escolhe o arquivo a ser depurado. Cada arquivo representa um cenário de teste a ser analisado. O nome do arquivo é formado pela cadeia de caracter "teste_2.0" seguida do tipo de cenário a ser analisado (*shared*, *barrier*, *event*, etc), e com a extensão *"log"*, por exemplo, *"teste_2.0_Shared.log"*, *"teste_2.0_barrier.log"* e *"teste_2.0_event.log"*. Cada arquivo é originário de um nó. Estes arquivos são coletados manualmente dos nós remotos ou podem ser visualizados, durante a execução da aplicação, em um nó local.

Figura 20 – Escolha do cenário de teste no GTracer

Cada arquivo é composto por diversas mensagens, sendo que cada uma representa uma primitiva enviada/recebida entre nós (o formato da mensagem segue o padrão definido no capítulo 4. Internamente, implementou-se um *parser* capaz de classificar todas as mensagens.

O volume de mensagens é grande, porque toda vez que uma primitiva é executada, esta é registrada no arquivo. Assim, é impraticável a análise dos dados no modo texto. Por exemplo, é inviável verificar no modo texto a sequência de comunicação entre os nós a fim de analisar o valor de uma variável compartilhada.

5.2 GTracer

A Figura 21 mostra a interface principal do GTracer, que permite a visualização e análise das mensagens. Nesse caso, apresenta-se a troca de mensagens entre cinco nós, no caso o servidor (Master 0 - coordenador) e quatro outros nós clientes (Slaves 1, 2, 3 e 4). Cada seta direcional indica o nó de origem da mensagem e o respectivo destino. Cada tipo de mensagem é representada por uma cor. Na interface é possível a qualquer momento, inicializar o processo, interromper e inicializar a leitura dos *logs*. Assim, a aplicação alvo e o GTracer podem ser inicializados, e após um período a coleta interrompida, o que permite uma análise parcial da mensagem. Faz parte da ferramenta um filtro de mensagens, então, por exemplo, o desenvolvedor pode aplicar um filtro para visualizar somente as mensagens de sincronização (Figura 22).

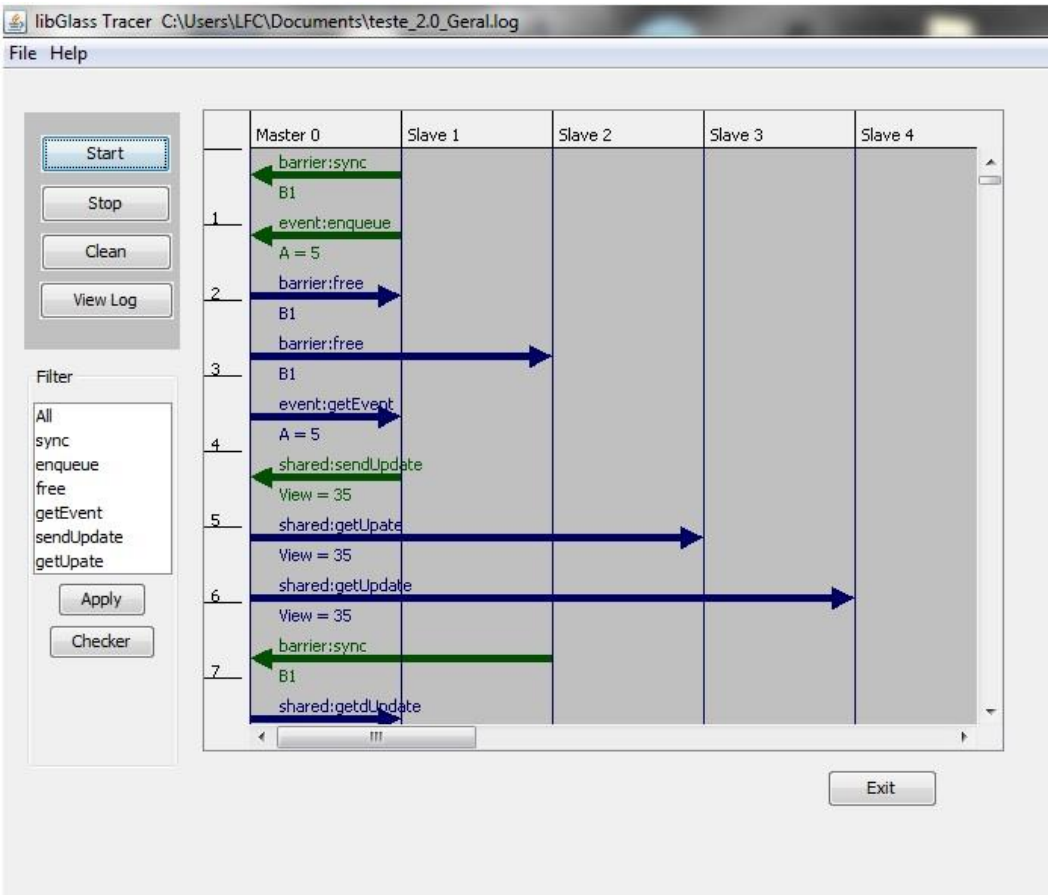

Figura 21 – Interface principal do GTracer

Quando o *parser* analisa um *log*, automaticamente todos os tipos de dados presentes são classificados e transformados em filtros para visualização, e os nós que receberam/enviaram mensagens são adicionados na interface. A Figura 22 mostra a visualização de um *log* no GTracer com a aplicação de um filtro. Foi selecionada a opção *"sync"*, na janela *"Filter"* e na sequência acionado o botão *"Apply"*. No caso apresentado, as mensagens filtradas referem-se a todos os sinais de sincronização de barreiras recebidos pelo nó servidor (Master). Então, é possível notar que os nós Slaves número 1 e 2 enviaram mensagens.

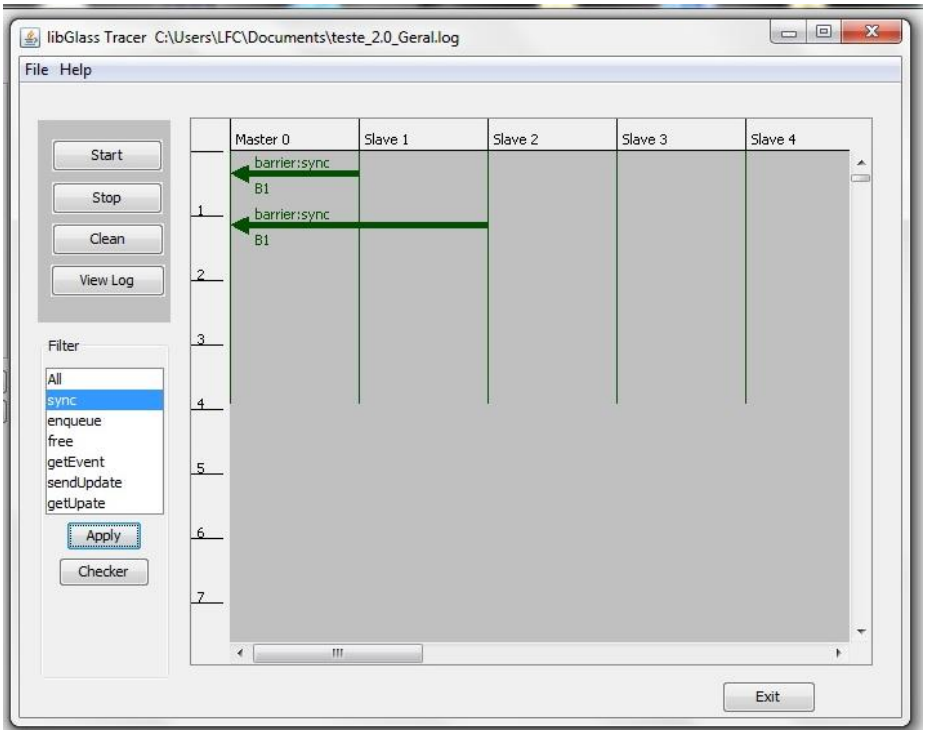

Figura 22 – Aplicação de filtros no GTracer

Faz parte dessa interface o botão *"Checker"*, que dispara a verificação comportamental das aplicações tendo como base as funcionalidades identificadas no Modelo Conceitual das Funcionalidades. Cada funcionalidade é implementada na libGlass via *plug-ins.* Cabe a funcionalidade de análise do GTracer comparar o comportamento atual com as pré-condições e pós-condições de cada funcionalidade.

O GTracer diferencia-se de outras ferramentas em alguns pontos importantes, como, por exemplo: não exige alteração de código fonte; representação visual das mensagens; e pela validação comportamental das aplicações.

As próximas seções apresentam exemplos de verificação realizada pelo GTracer, como o Geral, onde tem-se várias situações juntas em um único cenário, como por exemplo, finalização, compartilhamento de variáveis, sincronização de barreiras e enfileiramento de eventos. Posteriormente, a implementação de cada funcionalidade no GTracer é discutida e apresentada.

5.2.1 Cenário Geral

A Figura 23 apresenta a troca de mensagens entre cinco nós, no caso o servidor (Master 0 - coordenador) e quatro outros nós (Slaves 1, 2, 3 e 4), sem aplicação de filtros, ou seja, as mensagens referem-se a diversas funcionalidades. Neste caso, a mensagem inicial representa o nó Slave 1 solicitando (*sync*) a sincronização da barreira B1 para o nó servidor Master 0; em seguida, o mesmo nó solicita o enfileiramento (*enqueue*) do valor 5 na fila denominada "A". Neste momento, o servidor Master 0 libera a barreira de sincronização B1 para os nós Slave 1 e Slave 2 (note que o nó foi liberado sem solicitar a sincronização previamente). Em seguida, o Master devolve a atualização da fila "A" para o nó Slave 1 (*getEvent*) e, assim, por diante.

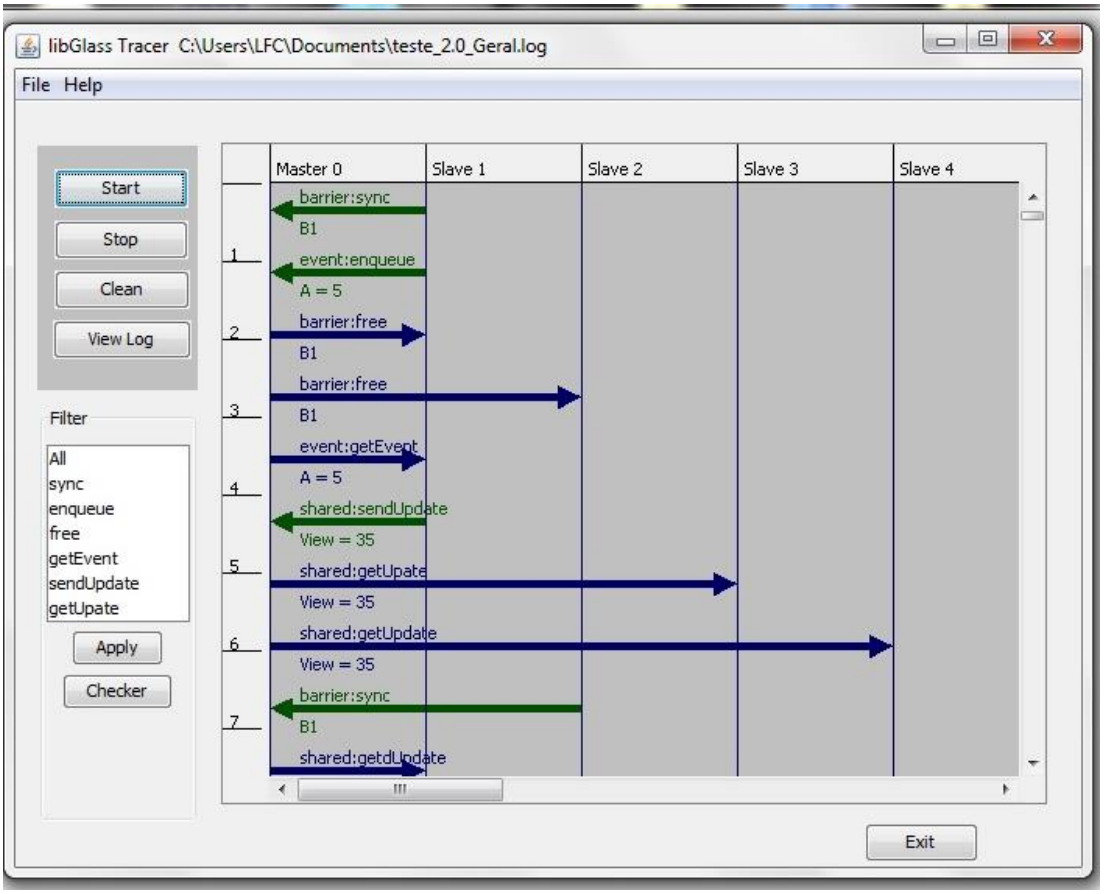

Figura 23 – Exibição do Cenário Geral pelo GTracer

Na Figura 24 é possível visualizar a verificação realizada. No caso indicou diversos problemas *("Failed"*), dentre eles a liberação da barreira B1 para o nó Slave 2, sem que o servidor Master 0 ter recebido o pedido de sincronização. Isso indica que a biblioteca não está controlando as barreiras conforme o comportamento esperado. Em situações como essa é possível a análise de múltiplas combinações de situações de troca de mensagens, independente da ordem em que estes eventos aconteceram ou da origem solicitante de um recurso, simulando com eficácia este ambiente, e possibilitando ao desenvolvedor, tomar decisões no sentido de entender o que está ocorrendo e corrigir os problemas detectados.

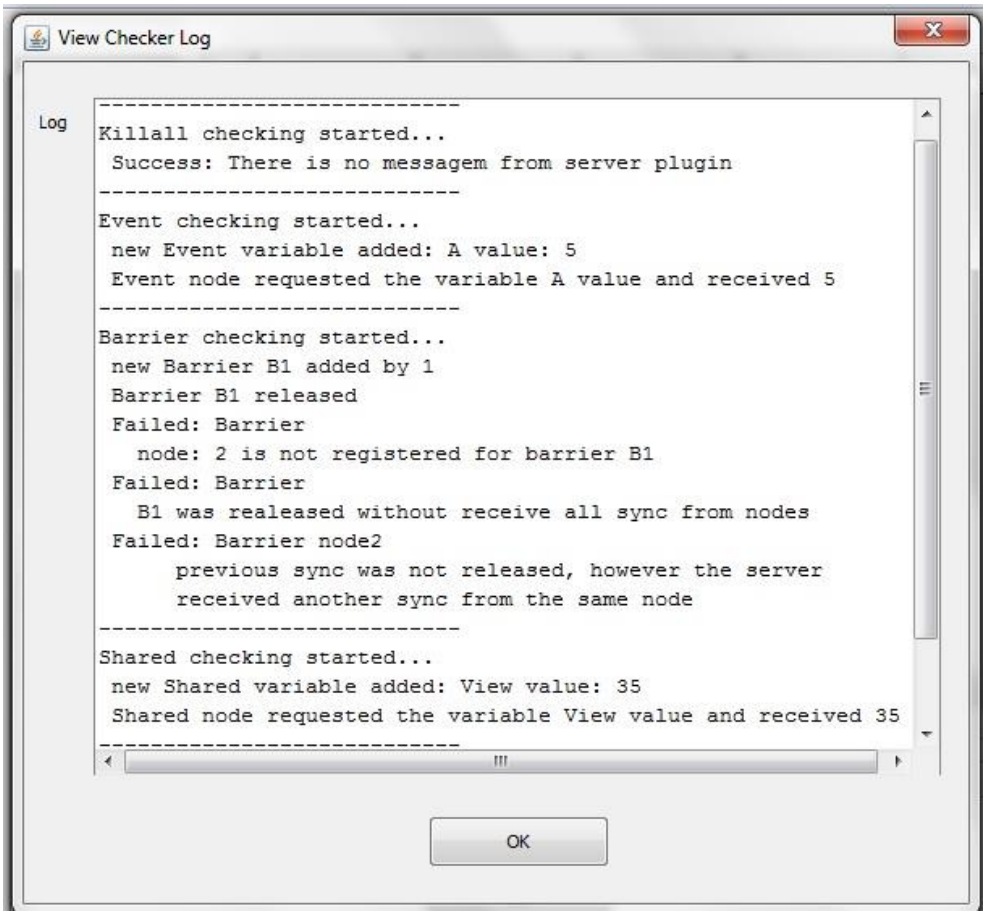

Figura 24 – Exibição do Diagnóstico detalhado para o Cenário Geral

5.2.2 Verificação da Funcionalidade Finalização

O Modelo Conceitual dessa funcionalidade esclareceu que quando o nó mestre receber um pedido de finalização, esse disparará primitivas para os outros nós solicitando que todos terminem. O Modelo de Execução global mostrou que ela é independente das outras funcionalidades, porém é capaz de finalizar as aplicações, sem a necessidade de aguardar a conclusão de uma tarefa específica, como, por exemplo, a liberação de uma barreira. Por fim, as pré-condições e pós-condições extraídas da Rede de Petri, deixou claro o cenário no qual essa funcionalidade pode ser disparada, que é a qualquer momento.

A Figura 25 mostra a troca de mensagens referentes a pedidos de finalização entre três nós, no caso o servidor (Master 0 - coordenador) e dois outros nós (Slaves 1 e 2). Nesse caso, o Slave 1 enviou um pedido (*"killall"*) de finalização das aplicações

para o servidor, que, em seguida, retornou a solicitação (*"kill"*) para os nós slave 1 e slave 2.

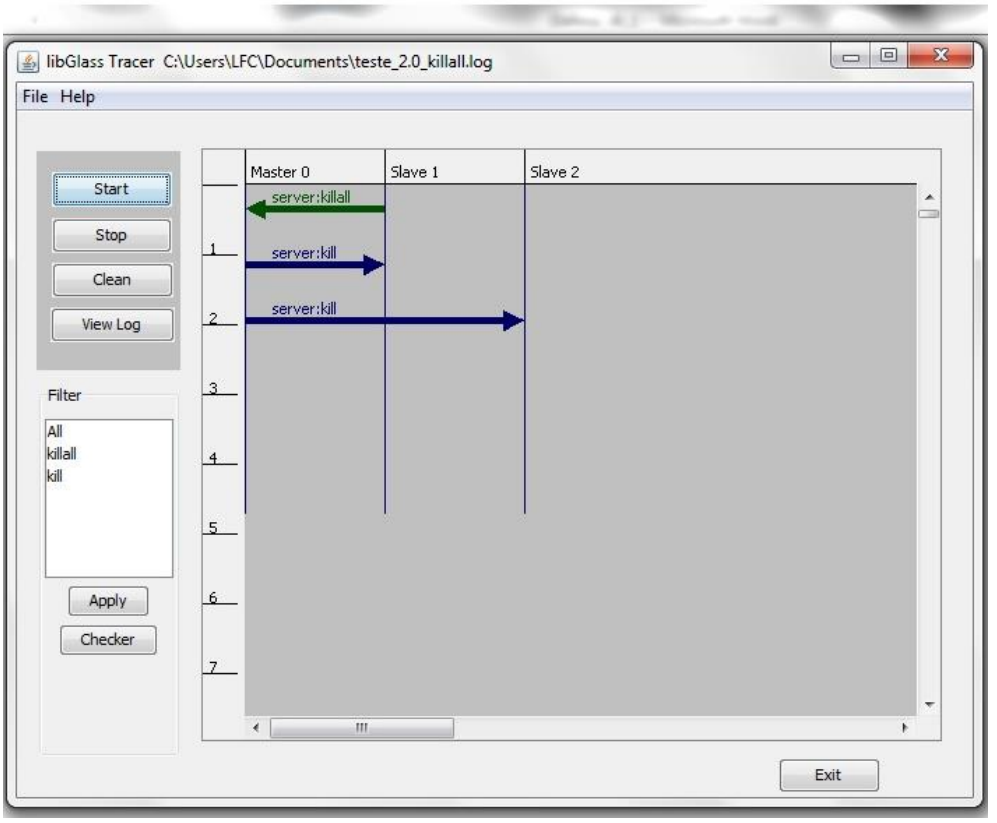

Figura 25 – Comportamento da funcionalidade Finalização como especificado

A Figura 26 mostra o diagnóstico realizado pelo GTracer. Neste caso, a aplicação se comportou conforme o esperado, ou seja, o servidor enviou a primitiva de finalização para os nós após receber uma solicitação de finalização. Situações fora desse comportamento são apontadas pela ferramenta.

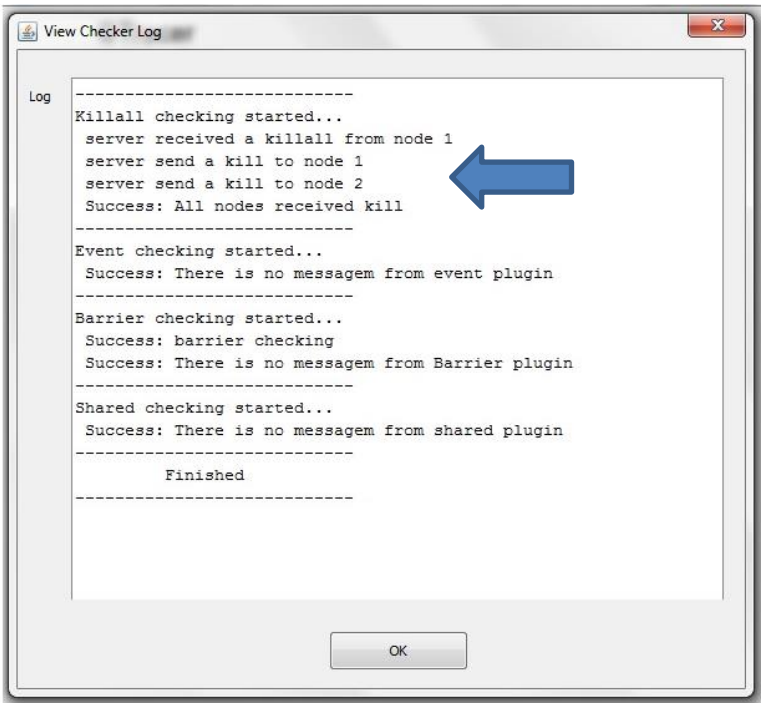

Figura 26 – Diagnóstico em Cenário Finalização sem problemas

A Figura 27 apresenta um cenário no qual o comportamento da aplicação não foi o esperado. O nó slave 1 solicitou a mensagem de finalização de todos os nós para o servidor (*killall*), na sequência o servidor enviou a mensagem *(kill)* de finalização para o nó slave 3, porém não enviou a mensagem de finalização para o nó slave 1.

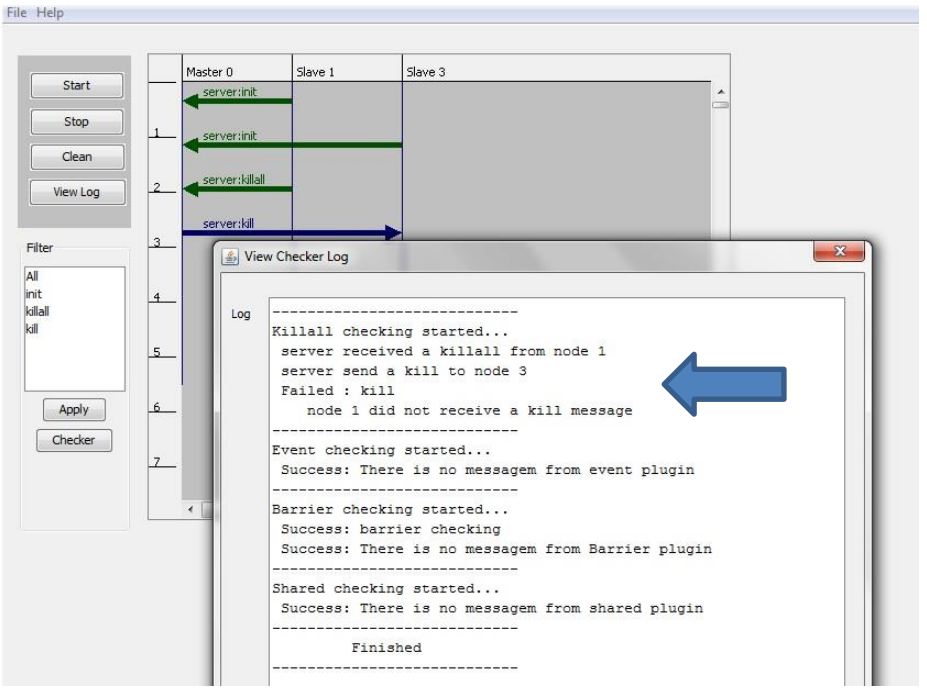

Figura 27 – Funcionalidade Finalização com problemas

5.2.3 Verificação da Funcionalidade Compartilhamento de Variáveis

A Figura 28 mostra um cenário de compartilhamento de variáveis no qual ocorre a troca de mensagens entre quatro nós, no caso o servidor (Master 0) e três outros nós (Slaves 1, 2 e 3). Cada seta direcional indica o nó de origem da mensagem e o respectivo destino. Por exemplo, a seta *sendUpdate* com origem em Slave 1 e destino a Master 0, representa a mensagem de atualização da variável compartilhada "A" com conteúdo igual a "15" para o nó Master 0. Em seguida, o Master 0 atende a solicitação *getUpdate* dos nós clientes Slave 1, 2 e 3 e envia a variável com o valor atualizado, no caso 15. E, assim, por diante.

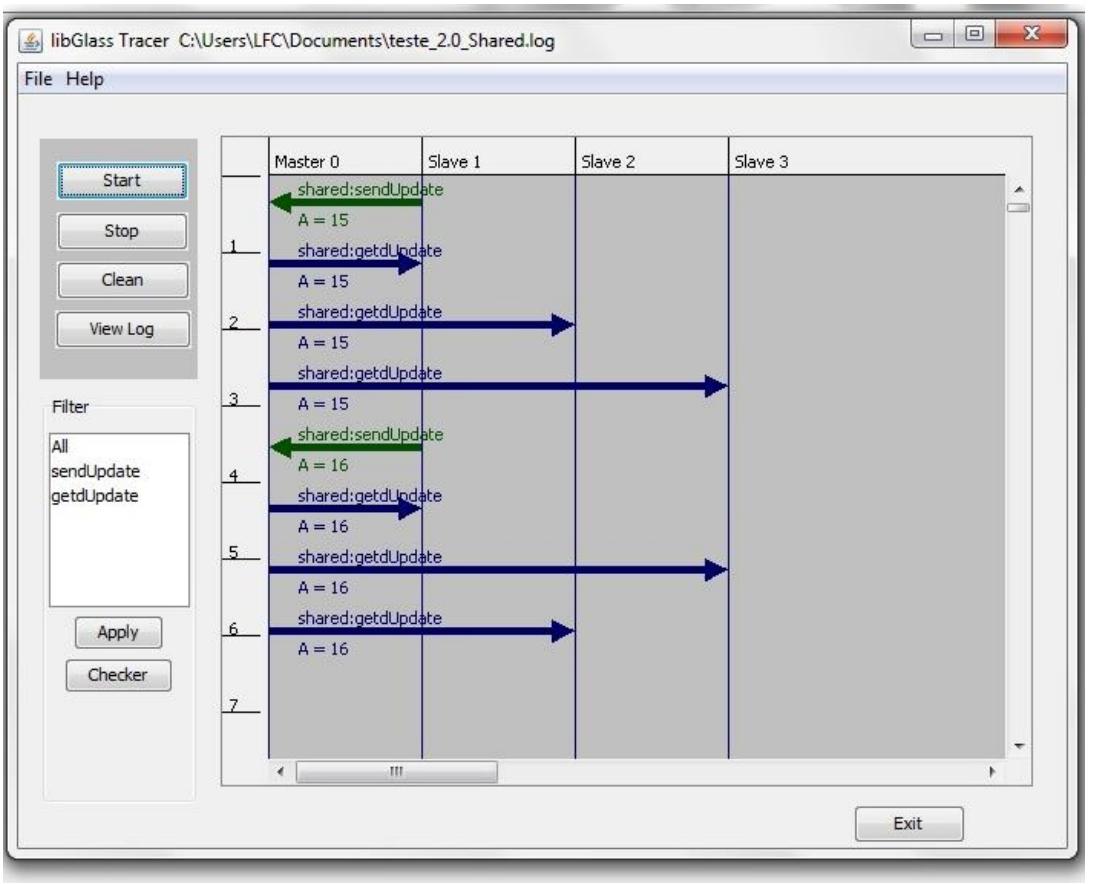

Figura 28 – Cenário de Compartilhamento de Variáveis pelo GTracer

A verificação comportamental dessa funcionalidade extraiu do Modelo Conceitual a sequência de execução dela, no qual um nó envia o dado para o servidor, que é compartilhado entre os outros nós. O Modelo de Execução global mostrou que ela é independente das outras funcionalidades. Por fim, as pré-condições e pós-condições

extraídas da Rede de Petri, deixaram claro que a funcionalidade é disparada quando algum dado sincronizado precisa ser compartilhado entre os nós. Além disso, que mais de um nó pode alterar a mesma variável e compartilhá-la entre outros nós, cabendo então a biblioteca de comunicação implementar um algoritmo para determinar o valor atual.

No caso da libGlass, o valor corrente é estabelecido sempre após um ponto de sincronização, que assume o valor do nó com maior prioridade. Assim, é tarefa do GTracer verificar se o dado que o servidor recebeu é o mesmo que enviou. Se não for, esse fato deve ser informado para o desenvolvedor. Assim, torna-se possível verificar se a biblioteca está respeitando os algoritmos internos de atualização.

Na Figura 29 mostra um exemplo de diagnóstico que não detectou problemas. Nesse caso, o valor atual do servidor, foi o mesmo compartilhado.

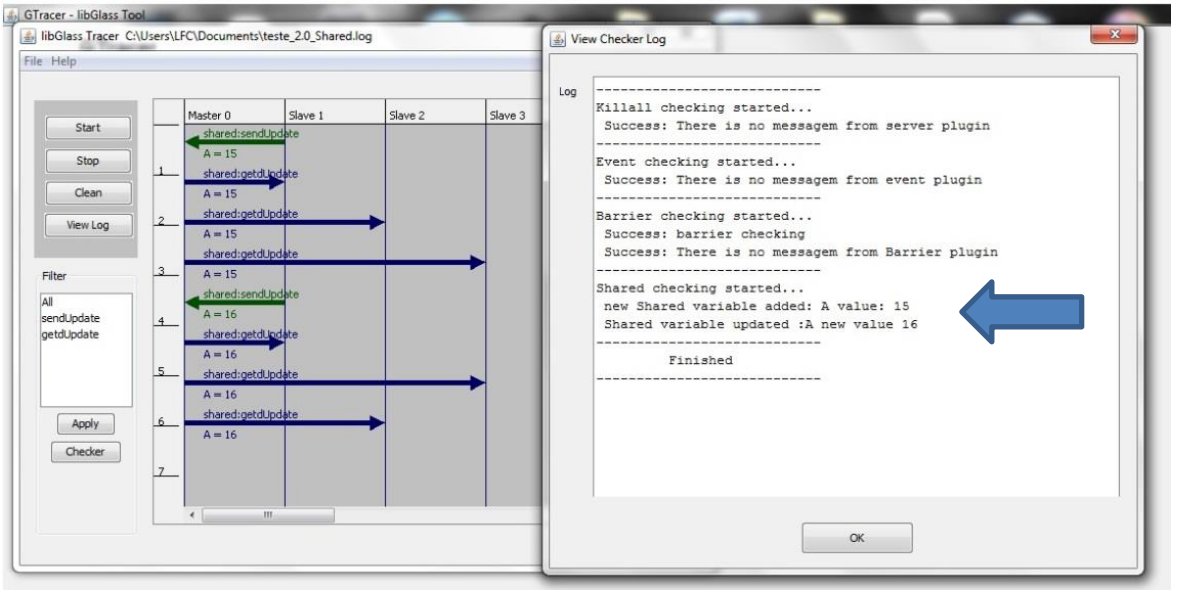

Figura 29 – Comportamento da funcionalidade Compartilhamento como especificado

A Figura 30 aponta um comportamento não esperado no compartilhamento das varíaveis. No caso, o valor inicial da variável A é 15, cujo valor é compartilhado em seguida com o Slave 1. Contudo, em sequência, o Slave 2 recebe o valor 17. A Figura 31 mostra o resultado do diagnóstico realizado, que apontou um comportamento não esperado.

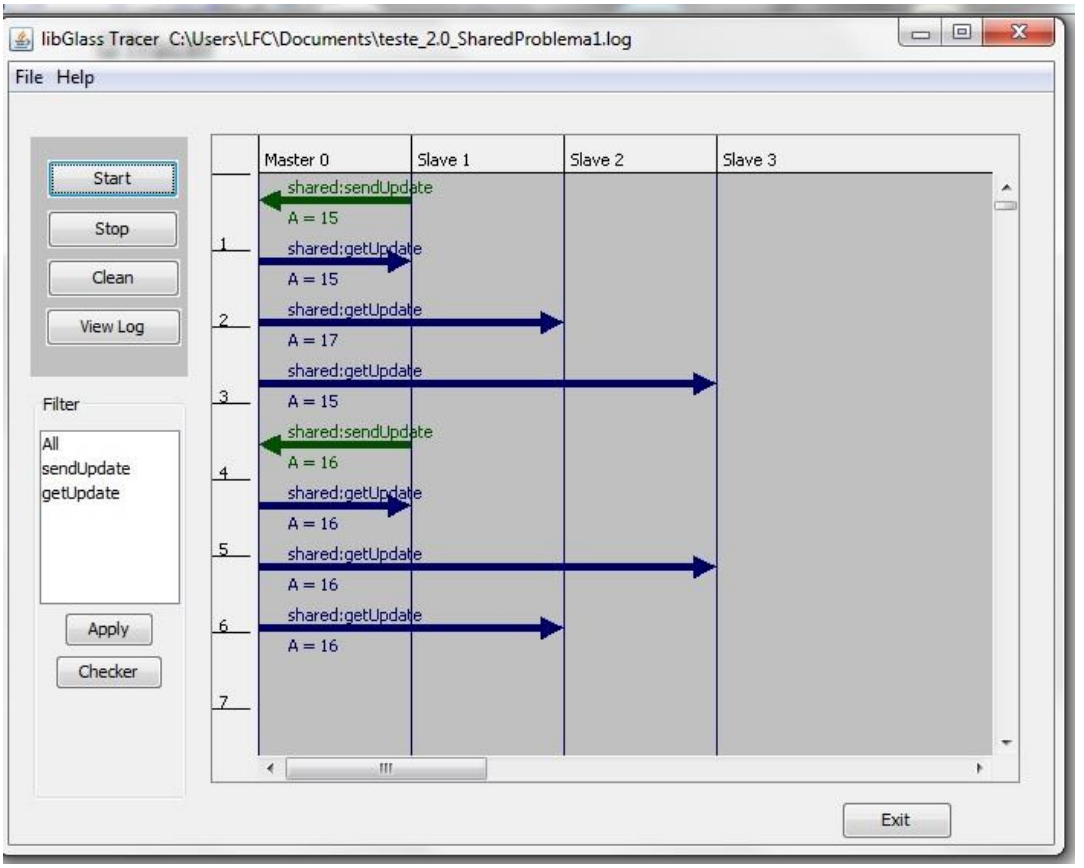

Figura 30 – Cenário Compartilhamento de Variáveis com problemas

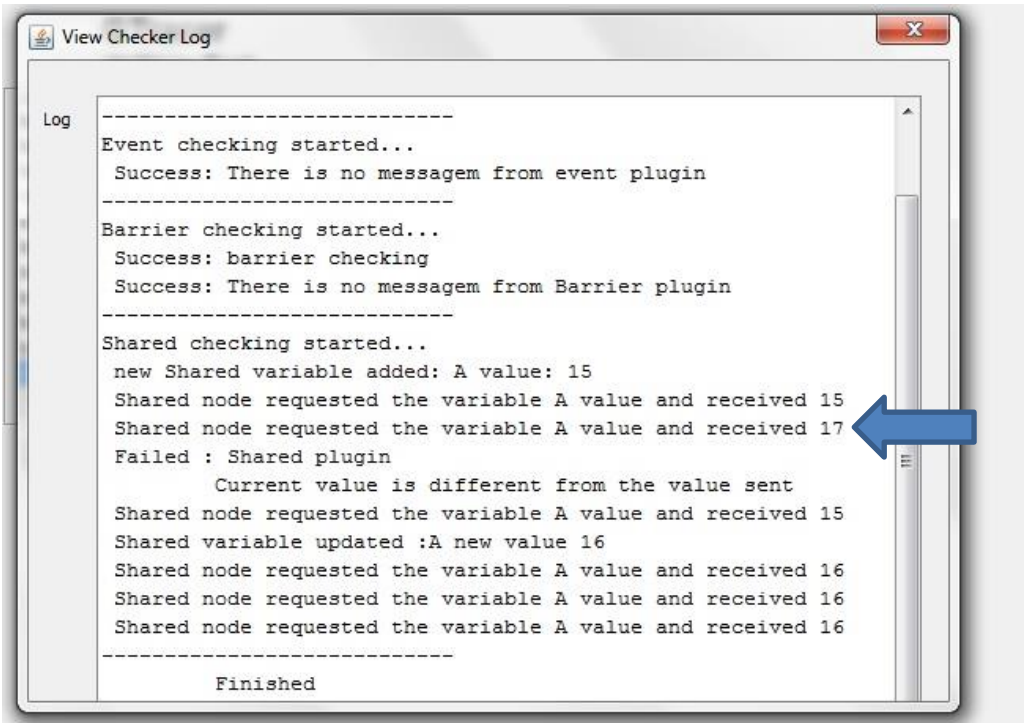

Figura 31 – Diagnóstico Compartilhamento de Variáveis com problemas

5.2.4 Verificação da Funcionalidade Sincronização de Barreiras

A verificação comportamental da sincronização de barreiras extraiu do Modelo Conceitual a sequência de execução, em que um nó solicita a sincronização de uma barreira e somente a libera quando recebe o pedido de todos. O Modelo de Execução global mostrou que ela é independente das outras funcionalidades. Por fim, as pré-condições e pós-condições extraídas da Rede de Petri, deixaram claro que no caso das aplicações de RV distribuídas utiliza-se no mínimo a barreira para realizar o *framelock* e outra para o *datalock*. Dessa forma, o processo de verificação deve levar em conta que pode existir mais de uma barreira em execução e que elas são independentes uma das outras.

A Figura 32 ilustra as mensagens de sincronização no GTracer. Este cenário é composto por três nós, no caso o servidor (Master 0) e dois outros nós (Slaves 1 e 2). Sendo que as duas primeiras mensagens, que são originárias do Slave 1 e 2, são pedidos de sincronização e as duas últimas, são as mensagens de liberação de barreira enviadas pelo servidor. Nesse caso o processo de verificação de comportamento não detecta nenhum comportamento diferente do esperado.

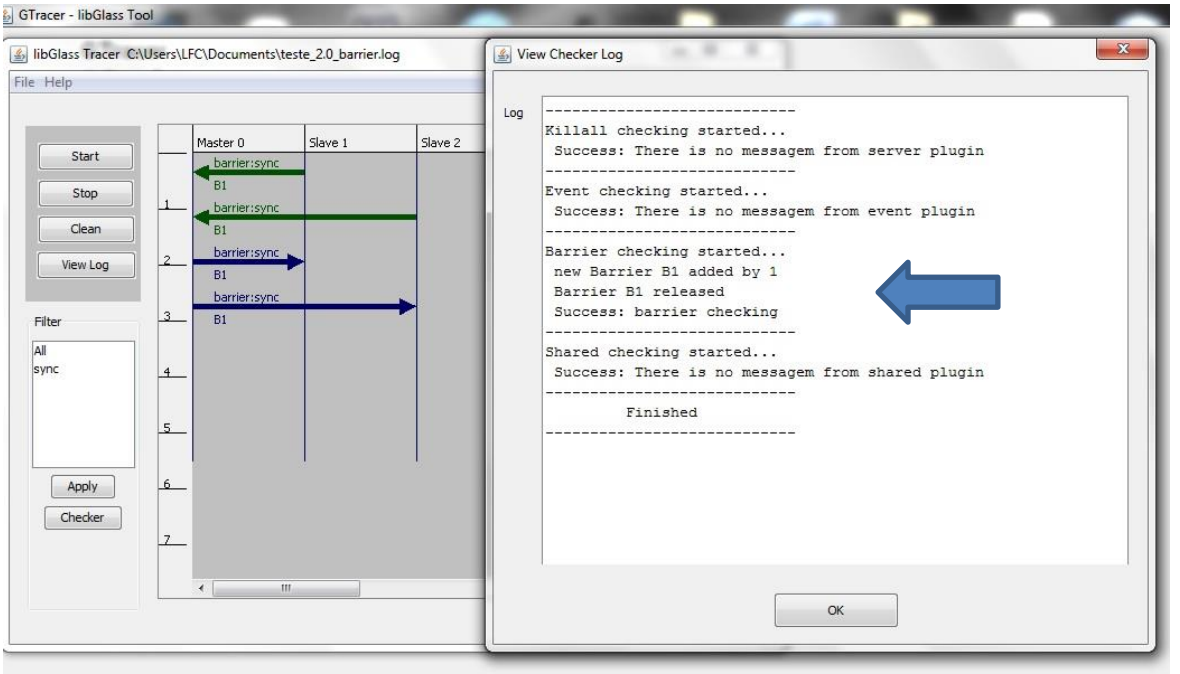

Figura 32 – Comportamento da funcionalidade Sincronização de barreiras como especificado

A Figura 33 mostra um exemplo de mensagens de sincronização que ocorrem em desacordo com o esperado e o resultado da verificação. Nesse caso, o servidor recebe o pedido de sincronização dos Slave 1 e 2. Em seguida, libera a barreira para o Slave 1 e recebe na sequência um novo pedido de sincronização do Slave 2. Esse pedido não está de acordo com o esperado, pois esse nó deveria ainda estar aguardado a liberação do pedido anterior. Assim, a verificação apontou essa discordância de comportamento.

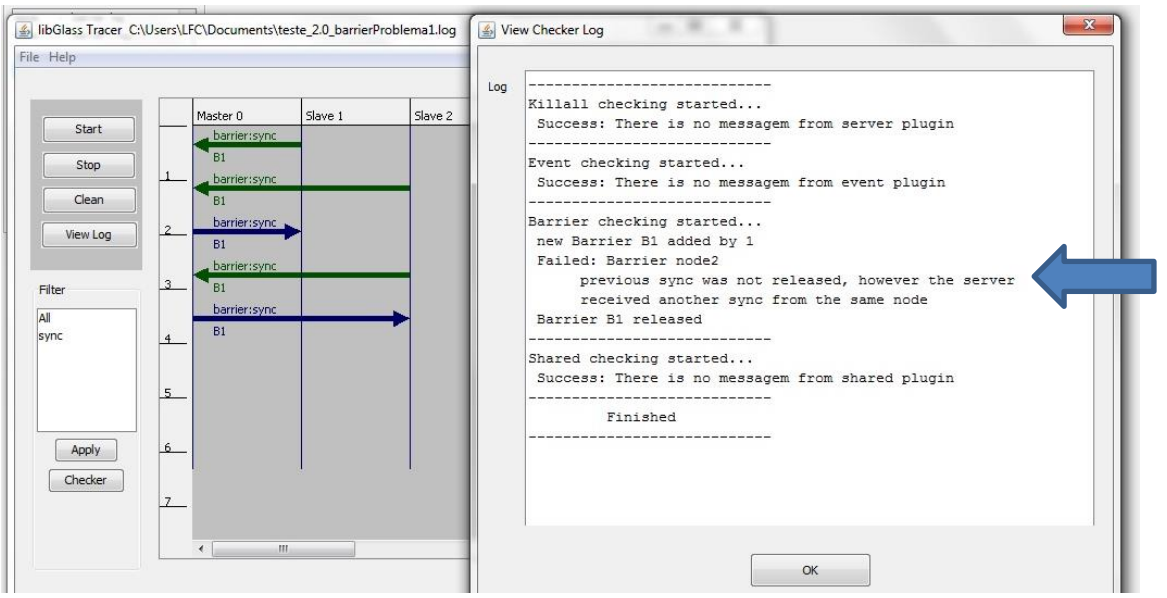

Figura 33 – Sincronização de Barreiras com problemas

5.2.5 Verificação da Funcionalidade Enfileiramento de Eventos

A verificação comportamental dessa funcionalidade baseou-se no Modelo Conceitual para especificar que o nó mestre armazena os valores enviados pelos nós escravos, subscrevendo-os, e os envia para os nós quando recebem um pedido. O Modelo de Execução global mostrou que ela é independente das outras funcionalidades. Por fim, as pré-condições e pós-condições extraídas da Rede de Petri, deixaram claro que esses dados assíncronos podem ter como origem qualquer um dos nós e que cada nó possui uma fila própria para armazenamento. Além disso, que pode existir mais de uma fila por aplicação. Então, cabe ao processo de verificação checar que o último valor recebido de cada fila pelo servidor é o enviado.

A Figura 34 mostra uma situação de uso da funcionalidade Enfileiramento de Eventos que envolve a troca de mensagens entre três nós, no caso o nó mestre recebe o valor 5 na fila denominada A e, em seguida, envia esse mesmo valor para os outros nós.

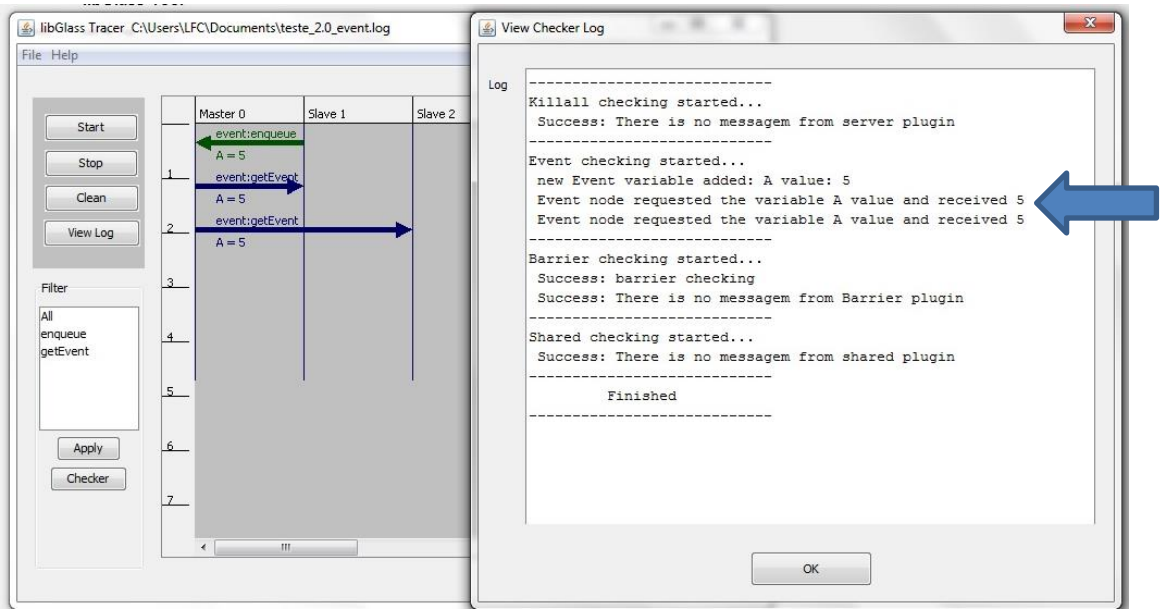

Figura 34 – Comportamento da funcionalidade Enfileiramento de Eventos como especificado

A Figura 35 mostra o resultado de uma análise na qual o valor 7 enviado para um dos nós não é o que está armazenado no servidor, ou seja, não é o comportamento esperado e apresenta o diagnóstico emitido em cenário de enfileiramento de eventos informando que um nova variável do tipo evento foi adicionada com valor igual a 5. Em seguida o nó cliente Slave 1 solicitou a variável A e recebeu o valor 7 e o nó cliente Slave 2 solicitou também a variável A e recebeu o valor 5. O enfileiramento de eventos aconteceu com ocorrência de problemas conforme o diagnóstico apresentado. Assim, a verificação apontou essa discordância de comportamento.

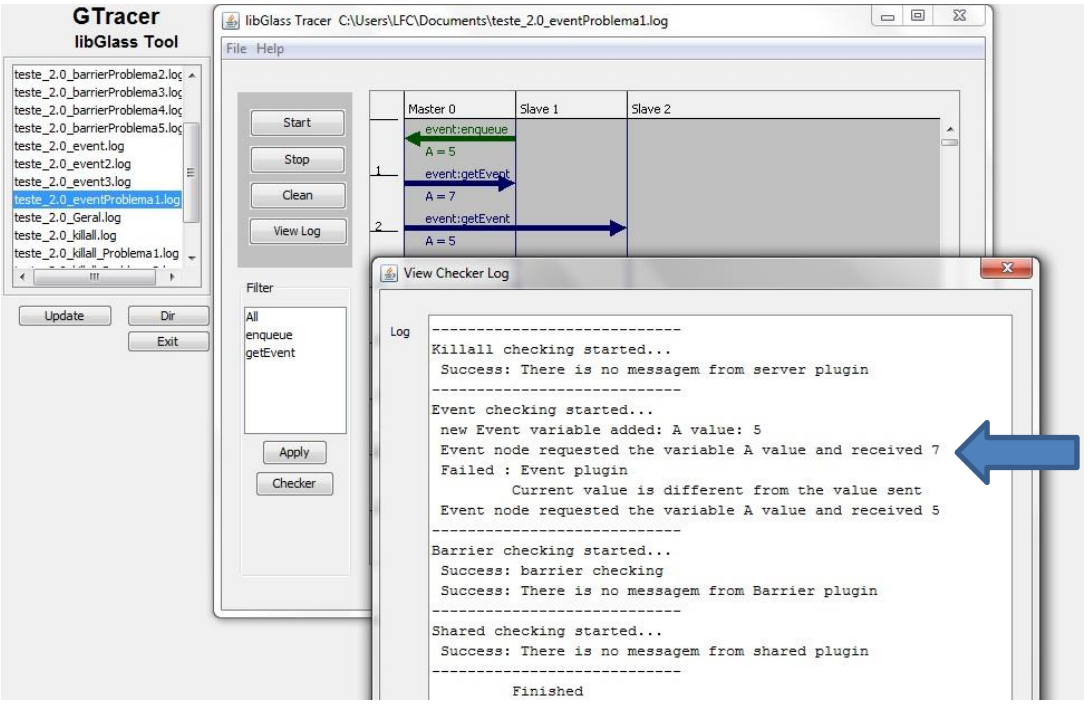

Figura 35 – Enfileiramento de Eventos com problemas

5.3 Notas Finais

Esse capítulo apresentou o GTracer, que implementa a análise de funcionalidades de aplicações de RV Distribuídas. Neste caso, a biblioteca que implementa as funcionalidades é a libGlass. Nela as mensagens enviadas/recebidas pelos nós podem ou não ser armazenadas em arquivos de *log* em tempo de execução. A geração de *logs* é determinada no momento de compilação da aplicação, que deve ser desabilitada se não houver interesse em depuração, pois a geração de *log* consome tempo de gravação de disco, o que afeta queda do desempenho da aplicação, e, consequentemente, interfere no grau de imersão e interação dos usuários. Cada instância da aplicação gera um arquivo de *log*.

Utilizando o GTracer, os desenvolvedores podem visualizar e verificar o comportamento das mensagens trocadas. Assim, o GTracer é capaz de responder questões, como, por exemplo, qual nó solicitou a finalização; qual o valor de uma variável compartilhada em determinado instante; qual nó alterou o valor da variável compartilhada em determinado instante; qual nó está demorando mais para processar e enviar o sinal de sincronização; e qual a ordem de recebimento dos eventos de cada nó.

Faz parte também do GTracer o analisador automático das mensagens trocadas (botão "*Checker"*), que ao ser acionado emite um relatório com o resultado detalhado da análise realizada, considerando todas as funcionalidades, com as conclusões sobre o comportamento das mensagens trocadas entre os nós, informando se ocorreram falhas ou não.

6. Considerações finais

A evolução rápida e contínua da capacidade de processamento, comunicação das redes e do desenvolvimento de *softwares* tem impulsionado a utilização de computação distribuída para os mais diversos problemas, desde os que tratam de armazenamento e compartilhamento de dados até os que demandam poder de processamento. As aplicações de RV com alto grau de imersão e interação têm se beneficiado dessa evolução, utilizando como plataforma de execução *clusters* de computadores (Soares, 2010). Isso ocorre porque essas aplicações são formadas por diversos processos independentes que podem ser distribuídos entre diversos nós, comunicando-se por troca de mensagens via rede de computadores, o que torna a atividade de depuração de sistemas distribuídos um desafio (Bates e Wileden, 1983, Dao *et al.*, 2009).

A libGlass é uma biblioteca para o desenvolvimento de aplicações distribuídas, que permite realizar o desacoplamento de funcionalidades dos nós conforme a característica de cada um (nós de interação, controle e de processamento). O desempenho das aplicações já desenvolvidas com a libGlass teve como limite a capacidade de processamento da placa gráfica utilizada. O desenvolvimento dessa biblioteca teve como uma de suas principais metas, a criação de um ambiente de fácil uso, tanto para o desenvolvimento de novas aplicações quanto para as existentes. Isso se tornou um ponto de destaque, pois a maioria das soluções disponíveis requer um grande número de modificações no código fonte, e, às vezes, na arquitetura da aplicação. Além disso, a libGlass permite a superação de desafios das aplicações com estereoscopia, que é a sincronização de dados (*datalock*) e a de conclusão de frames (*framelock*).

Pelo fato da abordagem de desenvolvimento adotada pela libGlass não ser automática, não limitando os recursos a serem utilizados durante o desenvolvimento, então torna-se necessária ferramentas similares ao GTracer. Esse depurador de mensagens auxilia o desenvolvimento de novas aplicações e facilita o treinamento de desenvolvedores. Depuração de mensagens não é uma atividade fácil dentre as atividades de desenvolvimento de sistemas, especialmente em sistemas distribuídos. O GTracer traz uma solução diferenciada pois não foca na depuração de processos distribuídos, e sim na troca de mensagens entre os nós.

O GTracer possui a possibilidade de visualização de mensagens em modo gráfico. Esta visualização representa a troca de mensagens entre os nós de um sistema distribuído. O *parser* realiza a leitura do arquivo de *log* em sequência e cria a representação gráfica com o comportamento apresentado na aplicação.

Além disso, esta ferramenta contribuiu com um analisador automático de mensagens que é capaz de notar comportamentos referentes a situações como, por exemplo: o comportamento em execução está de acordo com o modelo de execução esperado; ocorrência de falhas de inicialização ou finalização de nós; problemas na atualização das variáveis compartilhadas; falhas na sincronização de barreiras; se as funções associadas aos nós estão sendo chamadas de forma correta; problemas no enfileiramento de eventos e após a verificação emitir um diagnóstico alertando sobre o sucesso ou falha na execução analisada.

6.1 Contribuições da Dissertação

Essa dissertação apresentou a análise de mensagens das aplicações de RV distribuídas e tem como uma das principais contribuições a criação de uma ferramenta de análise de comportamento, independente da biblioteca de troca de mensagem adotada. As principais contribuições desses modelos apresentados neste trabalho são:

- Permite verificar se o *software* segue o modelo esperado;
- Evolução da ferramenta GTracer;
- Os desenvolvedores podem obter um melhor entendimento das funcionalidades através de uma visão de alto nível. Trabalhar com dados de alto nível é mais fácil e o desenvolvedor pode ver instantaneamente o que está errado, e
- Os desenvolvedores podem verificar o estado da aplicação ao longo do tempo.

Esse modelo de funcionalidade resultante é um elemento motivador na criação de novos depuradores para as aplicações de RV distribuídas. Ao implementar nas bibliotecas de trocas de mensagens, possibilita os mais variados cenários de testes para simulação em ambiente de RV.

A partir dos modelos criados, implementou-se o GTracer, cuja versão inicial foi apresentada em 2004 (Guimarães, 2004). Assim, deixou de ser um simples visualizador

de mensagens para um analisador das mesmas, tornando-se uma ferramenta capaz de detectar problemas de implementação nas aplicações e na biblioteca de troca de mensagem adotada. Além disso, pode ser utilizada para o auxílio do ensino de programação distribuída, pois permite aos estudantes visualizarem todas as mensagens de maneira gráfica.

6.2 Trabalhos Futuros

Atualmente, toda a visualização e análise provida pelo GTracer é voltada para um arquivo de cada vez (geralmente do servidor), porém é possível realizar a junção de diversos *logs* e realizar a análise global das mensagens. Isso ampliaria as funcionalidades dessa ferramenta. Por exemplo, permitiria verificar o envio e recebimento das mensagens. Contudo, isso exige novos modelos que permitam a análise cruzada das mensagens.

Além de aprimorar os modelos existentes e criar novos modelos para expandir as funcionalidades, sugere-se como trabalhos futuros o aprimoramento do GTracer para:

- coleta automática dos *logs* dos nós remotos;
- apontamento automático de falhas (e.g. *deadlock*);
- cruzamento dos *logs*, e
- simulador automático de cenários de testes para aplicações de RV.

Além disso, o GTracer também poderá ser utilizado na forma implementada, caso seja adicionado a libGlass o suporte para a interligação de *clusters* remotos.

Referências

Abulrub, A.G. 2011. Virtual reality in engineering education: The future of creative learning, Global Engineering Education Conference (EDUCON), pp. 751-757.

Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P., Muthitacharoen, A. 2003. Performance Debugging for Distributted Systems of Black Boxes, in Proc. SOSP, Bolton Landing, NY.

Araki, K., Furukawa, Z., Cheng J. 1991. A general framework for debugging. IEEE Software, 8(3), pp. 14-20.

Allard, J., Gouranton, V., Lecointre, L., Melin, E., and Raffin, B. 2002. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster, In Proceedings of the IEEE Virtual Reality Conference 2002 (VR '02), IEEE Computer Society, Washington, DC, USA, pp. 273-274.

Balasubramanian, J., Mintz, A., Kaplan, A., Vilkov, G., Gleyzer, A., Kaplan, A., Guida, R., Varshneya, P. and Schmidt, D. C. 2010. Adaptive parallel computing for large-scale distributed and parallel applications. In Proceedings of the First International Workshop on Data Dissemination for Large Scale Complex Critical Infrastructures (DD4LCCI '10). ACM, New York, NY, USA, pp. 29-34.

Barham, P., Isaacs, R., Mortier, R., and Narayanan, D. 2003. Magpie: online modelling and performance-aware systems, In Proceedings of the 9th conference on Hot Topics in Operating Systems (HOTOS ’03), USENIX Association, pp. 85–90.

Barham, P., Donnelly, A., Isaacs, R., Mortier R., 2004. Using Magpie for request extraction and workload modeling. In Proc. OSDI, San Francisco, CA, pp. 259-272.

Bates, P. C. and Wileden, J.C. 1983. High-level debugging of distributed systems: The behavioral abstraction approach. Journal of Systems and Software. Volume 3, Issue 4, December 1983, pp. 255–264.

Brams, G.W. 1983. Réseaux de Petri: Théorie et Pratique, tome 1 e tome 2. Masson Editions.

Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E. 2002. Pinpoint: Problem Determination in Large, Dynamic Internet Services. Computer Science Division, University of California, Berkeley and Stanford University.

Cruz-Neira, C., Sandin, D.J., Defanti, T., Kenyon, R., Hart, J.C. 1992. The CAVE: Audio Visual Experience Automatic Virtual Environment, Communications of the ACM, vol. 35(6), pp. 64–72.

Clarke, E. M., Emerson, E A. 1981. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. Carnegie Mellon University, Computer Science Department. Paper 458.

Clarke, E. M., Emerson, E.A., Sifakis, J. 2009. Model Checking: Algorithmic Verification and Debugging. In Communications of the ACM. November 2009, vol.52(11), pp. 74-84.

Dao, D., Albrecht, J.e, Killian,C. and Vahdat, A. 2009. Live Debugging of Distributed Systems. Lecture Notes in Computer Science Volume 5501, pp. 94-108.

Dias, D. C. 2011. Sistema Avançado de Realidade Virtual para Visualização de Estruturas Odontológicas. Dissertação – Programa de Pós-Graduação em Ciência da Computação, Universidade Estadual Paulista "Júlio de Mesquita Filho". Bauru.

Fonseca, R., Porter, G., Katz, R.H., Shenker, S., Stoica, I. 2007. X-Trace: A pervasive network tracing framework, In Proc. of NSDI’07.

Godefroid, P. 2005. Software model checking: the VeriSoft approach. Formal Methods in System Design, pp. 77-101.

Gomide, A., Stolfi, J. 2011. Elementos de Matemática Discreta para Computação, UNICAMP, pp. 85-110.

Guimarães, M. P., Gnecco, B.B, Cabral, M., Soares, L., Zuffo, M. 2003. Synchronization and data sharing libray for PC clusters, Workshop on Commodity Clusters for Virtual Reality - VR-CLuster'03. Los Angeles, USA.

Guimarães, M. P. 2004. Um Ambiente para o Desenvolvimento de Aplicações de Realidade Virtual baseadas em Aglomerados Gráficos. Tese de doutorado. Escola Politécnica da Universidade de São Paulo.

Guimarães, M. P., Martins, V. F., Dias, D. C, Gnecco, B. B, Contri, L. F. 2013. Desenvolvimento e depuração de aplicações de realidade virtual distribuídas, publicado na 8ª Conferência Ibérica de Sistemas e Tecnologias de Informação, realizada no ISEGI da Universidade Nova de Lisboa, Portugal.

Harper, R., Rodden, T., Rogers, Y., Sellen(eds), A. 2008. Being Human-computer interaction in the year 2020, Publisher: Microsoft Research Ltd. Isbn: 978-0-9554761-1-2.

Hartling, P., Bierbaum, A., Cruz-Neira, C. 2002. Tweek: Merging 2D and 3D Interaction in Immersive Environments. 6th World Multiconference on Systemics, Cybernetics. and Informatics. Orlando. Florida.

Hill, L., Cruz-Neira, C. 2000. Palmtop interaction methods for immersive projection technology systems. Fourth International Immersive Projection Technology Workshop (IPT 2000).

Hood, R., Jost, G. 2000. A debugger for computational grid applications, Heterogeneous Computing Workshop, 2000, (HCW 2000) Proceedings, 9th, pp. 262-270.

Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., Hanrahan, P. 2001. WireGL: a scalable graphics system for clusters, In Proceedings of the 28th annual conference on Computer graphics and interactive techniques (SIGGRAPH '01), ACM, New York, NY, USA, pp. 129-140.

JAVA, Native Interface. Disponível em: https://www.java.com/pt_BR/. [Acessado em 02 janeiro de 2014].

Kai-Hu, Q. Y., Xin-Jian, L. 2013. Application of Computer Virtual Reality Technology in Modern Sports. Third International Confererence in Intelligent System Design and Engineering Applications (ISDEA), pp. 362-364.

Khan, R. Z., Ali, M. F. 2012. Current Trends in Parallel Computing. International Journal of Computer Applications (0975 – 8887) Volume 59– No.2, December 2012.

Killian, C., Anderson, J.W., Braud, R., Jhala, R., and Vahdat, A. 2007. Mace: language support for building distributed systems, In Proc. of PLDI'07.

Kim, J., Kim, K., and Jung, S., 2001. Building a high-performance communication layer over virtual interface architecture on Linux clusters. In Proceedings of the 15th international conference on Supercomputing (ICS '01). ACM, New York, NY, USA, pp. 335-347.

Kirner, C., Tori, R. 2004. Uso de Realidade Aumentada em Ambientes Virtuais de Visualização de Dados. Proc. of VII Symposium on Virtual Reality, SP.

Klosowski, J. T. 2002. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters, In Proceedings of ACM SIGGRAPH, pp. 693-702.

KranzlMüller, D. 2000. Event Graph Analysis for Debugging Massively Parallel Programs, Institut für Technische Informatik und Telematik, Johannes Kepler Universität Linz, Austria.

Lin, P., Pan, Z., Yang, J., ShiI, J. 2002. Implementation of a Low-Cost CAVE system Based on Networked PC, Virtual Environment on a PC Cluster Workshop, Protvino, Russia, pp. 33-40.

Maciel, P.R.M., Lins, R. D., Cunha, P. R. F. 1996. Introdução às Redes de Petri e Aplicações, Departamento de Informática, Universidade Federal de Pernambuco, X Escola de Computação Campinas-SP, julho 1996.

MPI, Message-Passing Interface, 2013. MPI: A Message-Passing Interface Standard. Version 3.0. Disponível em: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. 2012. [Acessado em 10 dezembro de 2013].

Morie, J. F. 1994. Inspiring the future: merging mass communication, art, entertainment and virtual environments. SIGGRAPH Comput. Graph. 28, 2 (May 1994), pp. 135-138

OpenSG, OpenSG Welcome, 2013, Disponível em: https://www.opensg.org/. [Acessado em 10 de janeiro de 2013].

Pallot, M., Eynard, R., Poussard, B., Christmann, O. and Richir, S. 2013. Augmented sport: exploring collective user experience. In Proceedings of the Virtual Reality International Conference: Laval Virtual (VRIC '13). ACM, New York, NY, USA, Article 4, 8 pages.

Peterson, J. L. 1977. Petri Nets, Computing Surveys, Vol. 9, No.3,September.

Peterson, J. L. 1981. Petri Net Theory and the Modeling of Systems. N.J.: Prentice-Hall.

Petri, C. A. 1962. Kommunikation mitAutomaten. Bonn: Institut für Instrumentelle Mathematik, Schriften des NM, (3). Also English Translation, Communication with automata, Griffiss Air Force Base, New York, Technical Report, RADC-TR-65-377, vol1, supplement 1, january 1966.

PVM, Parallel Virtual Machine, 2013. The PVM System. Disponível em: https://www.netlib.org/pvm3/book/node17.html. 2012. [Acessado em 05 de dezembro de 2013].

Regan, M. and Pose, R. 1994. Priority rendering with a virtual reality address recalculation pipeline. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques (SIGGRAPH '94). ACM, New York, NY, USA, pp.155-162.

Reynolds, P.A., 2006. Using causal paths to Improve performance and correctness in distributed systems. Thesis for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University.

Reynolds, P.A., Killian, C., Wiener, J.L., Mogul, J.C., Shah, M.A, Vahdat, A. 2006. Pip: Detecting the unexpected in distributed systems. In Proc. NSDI, San Jose, CA.

Reynolds, P.A., Wiener, J.L., Mogul, J.C., Vahdat, A., Aguilera, M.K. 2006. WAP5: Black-box Performance Debugging for Wide-Area Systems. In International World Wide Web Conference, Edinburgh, Scotland, May 2006.

Schaeffer, B., Goudeseune, C. 2003. Syzygy: Native PC Cluster VR, IEEE Virtual Reality Conference, pp. 25-22. Disponível em: https://www.isl.uiuc.edu/syzygy.html. [Acessado em 10 de janeiro de 2013].

Sedayao, J. 2008. Implementing and operating an internet scale distributed application using service oriented architecture principles and cloud computing infrastructure. *In* Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services (iiWAS '08), Gabriele Kotsis, David Taniar, Eric Pardede, and Ismail Khalil (Eds.). ACM, New York, NY, USA, pp. 417-421.

Sharon E. P., Weihl., W.E. 1993. Performance assertion checking. In Proc. SOSP, pp. 134-145.

Silberschatz, A., Galvin, P.B., Gagne, G. 2004. Fundamentos de Sistemas Operacionais, pp. 352-353.

SGI Graphics Cluster TM, 2001: The Cluster Architecture Challenges, the SGITM Solution, Visualization Solutions Development Group, White Paper.

Souza, A.M.C. 2012. Handcopter Game: A Video-Tracking Based Serious Game for the Treatment of Patients Suffering from Body Paralysis Caused by a Stroke. Virtual and Augmented Reality (SVR), pp. 201-209.

Tanenbaum, A. S., Steen, M. Sistemas Distribuídos: Princípios e Paradigmas. 2ª Edição. Prentice-Hall, 2007.

Toscani, S.S, Oliveira, R.S., Carissimi, A.S. 2003. Sistemas Operacionais e Programação Concorrente. Série Livros Didáticos. Número 14. Instituto de Informática da UFRGS. Editora Sagra Luzzato.

Verbowski, C., Kiciman, E., Kumar, A., Daniels, B., Lu, S., Lee, J., Wang, Y., Roussev, R. 2006. Flight data recorder: Monitoring persistent-state interactions to improve systems management, In Proc. of OSDI'06.

Wang, J. 2007. Petri Nets for Dynamic Event-Driven System Modeling, Department of Software Engineering, Monmouth University.

Wang, Z., Zhu, W., Chen, X., Sun, L., Liu, J., Chen, M., Cui, P., and Yang, S. 2013. Propagation-based social-aware multimedia content distribution. ACM Trans. Multimedia Comput. Commun. Appl. 9, 1s, Article 52, 20 pages.

XupingTu, H. Jin, Fan, X., Ye, J. 2010. Meld: A Real-time Message Logic Debugging System for Distributed Systems. 2010 IEEE Asia-Pacific Services Computing Conference, pp. 59-60.

Zuffo, J. A., Soares, L.P, Zuffo, M. K., Lopes, R.D. 2001. CAVERNA Digital – Sistema de Multiprojecão Estereoscópico Baseado em Aglomerados de PC's para Aplicações Imersivas em Realidade Virtual, In Procedings of the 4th Brazilian Symposium on Virtual Reality - SVR'01, Florianopolis, SC, Brazil, pp. 139–147.

Zuffo, M., Soares, L.P, Bressan, P., Guimarães, M.P. 2002. Commodity Cluster for Immersive Projection Environments, Tutorial. SRV – Symposium on Virtual Reality, Fortaleza.

Zuruwaski, R., Zhou, M. 1994. Petri nets and industrial applications: a tutorial. IEEE Transactions on Industrial Electronics, v. 41, n. 6, pp. 567-583, December.