Universidade Federal de Pernambuco

Graduação em Ciência da Computação

Centro de Informática

FFORCE

Framework for Collaborative Environments

Aluno: Gabriel Fernandes de Almeida (gfa@cin.ufpe.br)

Orientadora: Judith Kelner (jk@cin.ufpe.br)

Co-Orientadora: Veronica Teichrieb (vt@cin.ufpe.br)

Recife, outubro de 2006.

Resumo

Atualmente, há grande demanda por sistemas de colaboração *on-line*, nas mais variadas áreas de trabalho, principalmente as que lidam com domínios específicos e grande número de variáveis. A colaboração também tem papel fundamental na indústria do entretenimento *on-line*, destacando os jogos que se baseiam no conceito de *Massively Multiplayer Online Game*, ou MMOG, que permite que milhares de pessoas interajam simultaneamente.

O Trabalho de Graduação aqui apresentado tem como objetivo contribuir, neste cenário previamente descrito, com uma pesquisa que engloba as áreas de Colaboração em Tempo Real, Sistemas Distribuídos e Realidade Virtual Colaborativa. Neste trabalho será especificada a arquitetura de um *framework*, denominado FFORCE (*Framework for Collaborative Environments*), e descrita a implementação de um protótipo para este arcabouço. Sendo o intuito da versão piloto deste projeto a colaboração entre aplicações de Realidade Virtual, algumas operações primitivas serão introduzidas, tais como mudança de posição e orientação de objetos da cena.

Agradecimentos

Aos meus pais, pela coragem que tiveram ao me colocar neste mundo louco, e por terem educado quatro filhos com sucesso.

Aos meus irmãos, pelo companheirismo e pelas brigas.

À Taciana, pelo apoio incondicional e compreensão.

Aos colegas do Colégio de Aplicação, minha segunda família.

Aos companheiros do GPRT/GRVM: Joma, Mouse (pelas consultorias), Arthur, Márcio, João Grandão, Curupa, e todos com que eu tenho prazer em conviver. Agradeço especialmente à professora Judith, minha orientadora, e a Veronica, minha co-orientadora, pelas orientações, conselhos e direcionamento.

Índice

Índice de Figuras

1 Introdução

Sistemas de colaboração *on-line* [1] têm uma grande demanda em diversas áreas de atuação, principalmente as que lidam com domínios muito específicos e grande número de variáveis. Dentre estas áreas, pode-se incluir a indústria de *software* e a medicina. Nelas, existem dificuldades em reunir *experts* para realizar tarefas especializadas, em um único local de trabalho.

Um cenário típico de uma aplicação que pode se beneficiar com a utilização de colaboração *on-line* é aquele que simula ambientes reais, como o canteiro de alguma obra em construção, por exemplo. Um grupo de usuários, situados em locais diferentes, acessa um ambiente virtual colaborativo para discutir sobre uma possível alteração ao projeto original, onde estas mudanças serão realizadas por algum usuário e refletidas nos dados a serem apresentados aos demais. Além disso, é importante que estas mudanças sejam feitas em tempo real, um requisito essencial para quebrar o fator distância.

As demandas impostas por sistemas colaborativos e tele-imersivos, tais como grande largura de banda, baixa latência e baixa variação de latência (*jitter*), tornam essas aplicações um dos grandes desafios na área de redes. Por exemplo, numa aplicação que utiliza recursos de áudio e transmite gestos virtuais de usuários, é requerida baixa latência, assim como no caso das que necessitam distribuir atualizações de estado do mundo virtual [2].

A colaboração também tem papel fundamental na indústria do entretenimento *on-line*, destacando os jogos que se baseiam no conceito de *Massively Multiplayer Online Game* (MMOG), que permite que milhares de pessoas interajam simultaneamente, compartilhando um único mundo virtual, como visto na Figura 1.1. Para suportar todo o processamento requerido nestas aplicações, são usados vários conceitos de sistemas distribuídos. Este tipo de jogo tem atraído uma quantidade considerável de usuários, como pode ser evidenciado em [3]. Há de se considerar que, além do tempo, os jogadores também investem dinheiro nesse *hobby*, pois alguns desses jogos requerem pagamento de assinatura mensal.

Figura 1.1 - MMOG World of Warcraft [4]

Infelizmente, a literatura disponível é escassa, como também, dificilmente se encontram pesquisas com código aberto disponibilizado para desenvolvimento de aplicativos que utilizam colaboração em tempo real. Existem vários *frameworks* de código fechado, como o *Groove Virtual Office* [5] ou o *BigWorld Technology Suite* [6], sendo este último uma solução completa de infra-estrutura de *software* para desenvolvimento de jogos, baseados no conceito de MMOG. Alguns artigos, desta área, ainda não apresentaram resultados quantificados de desempenho, e sim qualificados, para protótipos em fases iniciais de desenvolvimento [7][8]. Discussões em torno deste tipo de infra-estrutura (física e lógica) ainda se encontram em um estágio preliminar, como se pode notar num dos principais eventos sobre este tópico, o MASSIVE [9], que contemplou assuntos como o desenvolvimento de estruturas de suporte para MMOG's que sejam escaláveis e eficientes [10].

O Trabalho de Graduação (TG) aqui apresentado tem como objetivo contribuir, neste cenário previamente descrito, com uma pesquisa que engloba as áreas de Colaboração em Tempo Real, Sistemas Distribuídos e Realidade Virtual Colaborativa. Neste trabalho será especificada a arquitetura de um *framework*, denominado FFORCE (*Framework for Collaborative Environments*), a fim de contemplar alguns dos requisitos dos enfoques descritos neste texto, como também descrever a implementação de um protótipo para este arcabouço. Finalmente, será utilizado um ambiente distribuído de Realidade Virtual (RV) para validação deste arcabouço. Sendo o intuito da

versão piloto deste *framework* a colaboração entre aplicações de RV, algumas operações primitivas serão introduzidas, tais como mudança de posição e orientação de objetos da cena.

A Realidade Virtual Distribuída (RVD) oferece a possibilidade de se utilizar um único ambiente virtual integrando diferentes computadores ligados em rede. Um ambiente virtual distribuído permite que diversos profissionais, fisicamente distantes, acessem esse ambiente através da rede e o utilizem simultaneamente, interagindo com ele e uns com os outros; diz-se que este ambiente virtual distribuído é colaborativo. A Realidade Virtual Colaborativa (RVC) permite o desenvolvimento deste tipo de ambiente, como na aplicação da Figura 1.2, e torna possível um melhor controle sobre variáveis complexas de determinados domínios, através da visualização [2].

Figura 1.2 - CALVIN: Aplicação de RVC [2]

Para que o *framework* proposto trabalhe de forma distribuída (replicação), é preciso lidar com problemas que envolvem atraso, como o sincronismo de início da aplicação e o sincronismo de ações dos usuários (incluindo problemas de *"lag compensation"*). Por se tratar de uma aplicação distribuída com o objetivo de suportar vários usuários simultâneos, é determinante que ela seja escalável. Esta característica tem impacto direto sobre o desempenho, devendo ser analisado o equilíbrio entre escalabilidade e eficiência, refletindo na escolha de um protocolo de comunicação que favoreça este equilíbrio.

Este trabalho é composto por mais três capítulos, onde o Capítulo 2 explica alguns conceitos relacionados com as áreas de Colaboração em Tempo Real, Sistemas Distribuídos e RV em Ambientes Colaborativos, para embasamento do leitor; no Capítulo 3 será descrita a arquitetura e a implementação do protótipo do FFORCE. Por fim, no Capítulo 4, serão mencionadas as principais conclusões desta pesquisa e possíveis trabalhos futuros.

2 Ambientes Distribuídos e Colaborativos

Neste capítulo serão explorados tópicos que embasam e fundamentam o trabalho desenvolvido para criação do FFORCE, no contexto de ambientes colaborativos e distribuídos. Na primeira seção, os aspectos relativos à colaboração em tempo real serão analisados; na segunda, serão observados conceitos e problemas relacionados aos sistemas distribuídos; na terceira, RVC; e, na quarta, as áreas de aplicação, demandas e os trabalhos que hoje existem e intersectam as três áreas são exploradas.

2.1 Colaboração em Tempo Real

Sistemas colaborativos de tempo real permitem que pessoas trabalhem em conjunto e sincronizadamente, mesmo quando os participantes e o conteúdo dos seus respectivos trabalhos se encontram em lugares fisicamente distantes. Para isso, esses sistemas podem suportar informações que ajudem na interação entre os usuários, e/ou dados sobre os trabalhos compartilhados [11][12], permitindo que o trabalho em questão seja disponibilizado para os usuários. Em poucas palavras, sistemas com suporte à colaboração em tempo real devem lidar com aspectos humanos (como as pessoas colaboram entre si) e técnicos.

2.1.1 Aspectos Humanos

Em conversações e em trabalhos de colaboração, as pessoas usam vários artifícios para se comunicarem; esses artifícios que permeiam as comunicações incluem mudanças no tom da voz, pausas, gestos com o corpo e mãos, contato visual, além do conhecimento e noção da presença, como também a reação de outras pessoas que estão no ambiente. Essas habilidades são usadas para saber quem está falando, quem está ouvindo, estabelecer entendimento numa conversa ou realizar troca de conhecimentos. O objetivo é capturar e transmitir informações relacionadas a essa dinâmica que ocorre entre colaboradores, sejam elas explícitas ou não, o que não é uma tarefa trivial. A transmissão de voz é um exemplo da dificuldade de se transmitir essas dinâmicas, pois a representação digital da voz é feita por amostragem e codificada, para que seja possível transmiti-la digitalmente. Desta forma, boa parte da amplitude, entonação e timbre da voz são perdidas

durante este processo.

Em suma, os fatores humanos relativos à presença demandam atenção especial, mas, ao mesmo tempo, dificilmente serão plenamente satisfeitos. Ao olhar leigo, a presença poderia ser representada até mesmo por vídeo e áudio de baixa qualidade, o que não é verdade; mesmo os melhores sistemas de colaboração desenvolvidos, como protótipos usados para pesquisa, suportam a chamada *tele presença* apenas parcialmente. Por isso, é importante levar em conta os requisitos do sistema em questão, no momento de representar a presença.

Outros tipos de problemas emergem em sistemas colaborativos que suportam troca de dados referentes aos trabalhos compartilhados. Esse tipo de sistema disponibiliza, durante o encontro, materiais como notas, documentos, planos ou desenhos comuns aos participantes do trabalho, e normalmente são implementados de acordo com uma das três formas abaixo descritas:

1. Um sistema de captura de vídeo exibe o trabalho e os objetos como uma imagem para transmissão, e é possível sobrepor as imagens de vários participantes, com a restrição de que um usuário não poderá manipular os objetos exibidos pelo outro;
2. Um sistema de compartilhamento onde todos os participantes vêem a mesma imagem, onde a cada rodada um usuário poderá interagir com a imagem exibida pelo sistema, funcionando de forma análoga ao compartilhamento de um computador por vários usuários, mas impedindo que haja interações simultâneas dos usuários com a interface;
3. Sistemas que são desenvolvidos especificamente para colaboração são cientes de que há um grupo de usuários interagindo, e tratam as entradas de cada participante separadamente. Desta forma, é possível customizar a visão que cada usuário tem do sistema, além de permitir que atividades simultâneas aconteçam.

Assim como para o suporte à presença, os sistemas que realizam troca de dados devem suportar fatores humanos inerentes à interação em grupo.

A maioria dos espaços compartilhados de trabalho (*groupwares* como o visualizado na Figura 2.1) tem características em comum. Em primeiro lugar, as pessoas manipulam objetos nesses espaços; essa manipulação inclui movê-los, modificá-los e removê-los. A implicação é que o espaço deve ser interativo. Segundo, as pessoas normalmente gesticulam para se comunicar com os outros participantes. Gestos frequentemente são associados à fala, como quando uma pessoa

aponta um objeto e diz “isto aqui”. Os sistemas devem fornecer suporte à habilidade de uma pessoa falar e gesticular nesse espaço de trabalho. Em terceiro lugar, as pessoas usam o espaço de trabalho compartilhado como um meio para expressar idéias, onde falam enquanto manipulam objetos. Assim, a manipulação do objeto deve ser visível em todos os locais, sem atraso (*lag*) aparente, se forem usados como artefatos de conversação. Em quarto lugar, as pessoas dividem seu tempo de trabalho colaborativo entre o trabalho individual e o trabalho em grupo. Isto significa que as pessoas devem focalizar sua atenção em partes diferentes do espaço de trabalho, enquanto estão fazendo o trabalho individual. Quinto, as pessoas percebem o que os outros estão fazendo, durante as ações deles. Assim, o espaço de trabalho deve prover informação suficiente para que as pessoas saibam quem está no espaço de trabalho, onde está e o que está fazendo.

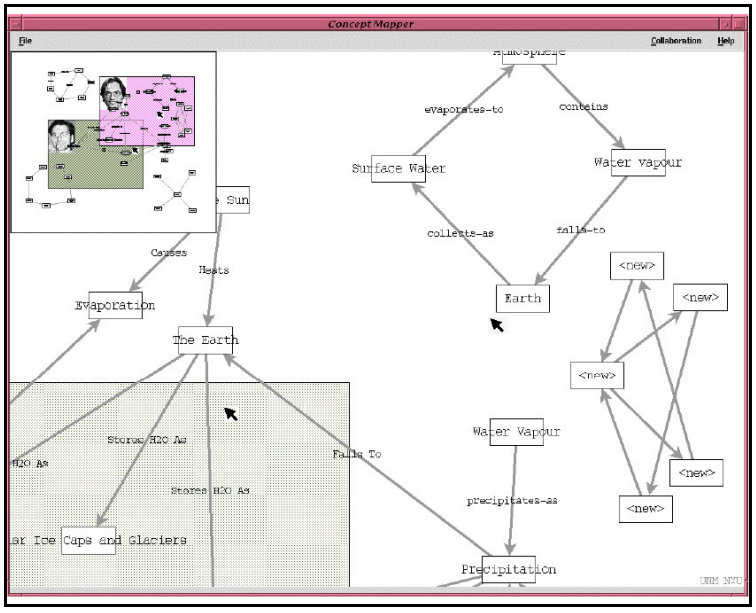

Figura 2.1 - Aplicação de *Groupware* para mapas mentais [2]

2.1.2 Aspectos Técnicos

Outros aspectos importantes em ambientes colaborativos de tempo real são os técnicos, entre eles: gerenciamento de sessão, segurança e privacidade, controle de acesso, tolerância a falhas, sincronismo, compensação de atraso, e controle de concorrência [11]. Este último aspecto será explorado mais detalhadamente nesta subseção por intersectar o escopo do projeto desenvolvido.

Os tópicos que falam sobre sincronização e controle de falhas serão mais explorados na Seção 2.2, sobre sistemas distribuídos.

O *gerenciamento de sessão* permite que usuários criem conexões, e façam reuniões e encontros com outros participantes. Os controladores da sessão frequentemente são apresentados através de metáforas. A metáfora de um telefone, por exemplo, representa pessoas "chamando" outras para iniciar uma sessão. Da mesma forma, a metáfora de um espaço implica que as pessoas podem navegar por um espaço, ver quem está ao seu redor, e iniciar uma conversação com pessoas que lá se encontram. Em suma, o gerenciamento de sessão ajuda na interação dentro do ambiente compartilhado, realizando, de forma transparente, conexões entre usuários e controlando a conexão dos mesmos com as aplicações do espaço compartilhado.

A *segurança* e a *privacidade* são problemas a serem considerados pelos *groupwares,* restringindo o acesso ao ambiente, apenas, para usuários com permissão. Dependendo do escopo da aplicação, devem também garantir que as transmissões realizadas durante a sessão sejam confidenciais. Como um ambiente colaborativo executa ações em vários locais diferentes, os participantes necessitam de garantias de que o *groupware* não comprometerá a integridade do seu sistema local.

O *controle de acesso* determina quem pode acessar um objeto compartilhado do *groupware* e quando. O controle de acesso pode ser requerido quando os usuários desejam ter seus próprios objetos, ou seja, objetos que somente eles podem manipular, e deve ser implementado evitando interferências na interação entre os usuários.

Tolerância a falhas é um dos pontos que devem ser tratados pelas aplicações colaborativas de tempo real, de forma a manter o serviço num patamar aceitável, ou seja, manter a qualidade do serviço, procurar rotas de comunicação alternativas quando um canal não é mais adequado, além de criar pontos de verificação para que erros possam ser contornados. Maiores detalhes sobre tolerância a falhas serão discutidos na Subseção 2.2.1, que fala sobre replicação.

Técnicas de *lag compensation* estão intimamente ligadas aos problemas de sincronização abordados na Subseção 2.2.2, e são utilizadas para homogeneizar o estado global da aplicação colaborativa, de forma a simular a simultaneidade das ações dos usuários, mascarando a latência da rede. Estas técnicas de compensação normalmente são usadas em arquiteturas cliente/servidor centralizadas, pois implementá-las em ambientes distribuídos requer abordagens complexas. Na

Figura 2.2, é possível visualizar uma tela do jogo *on-line* War Rock [13] com os tempos de latência em destaque; esse tipo de jogo requer pequenos índices de latência para que o entretenimento do usuário não seja comprometido.

Bernier [14] descreve três técnicas de *lag compensation*:

1. *client side extrapolation* é baseada na predição de estados futuros dos objetos, e pode introduzir inconsistências no caso de ações mais complexas;
2. *client side interpolation* consiste em exibir um estado passado, e os objetos que não são controlados pelos usuários são representados pela seguinte fórmula: estado atual deste objeto subtraindo-se a latência do usuário. Desta forma o estado passado será apresentado ao usuário, e este terá a impressão de simultaneidade das suas ações;
3. *server side latency compensation* consiste em calcular cada ação do usuário no exato contexto em que ela ocorreu, usando o estado armazenado para este usuário combinado com a latência calculada para o mesmo.

Figura 2.2 - Jogo War Rock com tempos de latência em destaque [13]

O *controle de concorrência* é requerido para evitar inconsistências, e lidar com ações conflitantes. Entretanto, este controle num ambiente colaborativo deve ser feito de forma diferente dos métodos tradicionais, porque o usuário é uma parte ativa do processo. Por exemplo, as pessoas que realizam atividades com alto grau de interatividade, não tolerarão atrasos introduzidos por esquemas conservadores de travamento (*locking*) e serialização. Similarmente, os usuários devem compreender os efeitos de alguns mecanismos requeridos para reparar inconsistências no ambiente interativo, como refazer e desfazer operações. Finalmente, os usuários podem solucionar alguns conflitos através da interação entre eles. Para que esta solução funcione, alguma indicação de conflito deve ser mostrada na interface do sistema.

Existem várias abordagens para solucionar problemas relacionados ao controle de concorrência em ambientes colaborativos de tempo real; dentre elas: travamento de dados (*locking*), transações, único participante ativo, detecção de dependência, e execução reversível.

Travamento de dados é uma abordagem para o controle de concorrência que simplesmente bloqueia os dados antes que eles sejam modificados. Vários problemas emergem ao se usar esse tipo de controle. Existem várias técnicas que diminuem a probabilidade de que um pedido de *locking* seja recusado, como liberar o pedido a um recurso que já estava travado, se o atual detentor do bloqueio está inativo. Uma outra técnica é fornecer aos participantes indicadores visuais dos recursos bloqueados, e assim diminuir a probabilidade de pedidos serem emitidos para objetos já travados.

Há três problemas principais com travamento de dados [15]: primeiramente há o *overhead* do pedido e obtenção do bloqueio, o que pode incluir um tempo de espera se os dados já estiverem travados; de qualquer forma haverá uma degradação no tempo de resposta da aplicação. Segundo, há a questão da granularidade. Pode-se tomar como exemplo um editor de texto compartilhado; neste caso, não é claro o que deve ser bloqueado quando um usuário move o cursor para o meio de uma linha e introduz um caractere. Deve-se bloquear o parágrafo inteiro, a sentença, apenas a palavra, ou somente o caractere? Alta granularidade traz menos restrições às ações dos usuários, mas gera uma sobrecarga muito maior no sistema. O terceiro problema está em determinar quando devem ser feitos os pedidos e as liberações de bloqueios. Usando o exemplo acima, o bloqueio deve ser pedido quando o cursor é movido ou quando a tecla é pressionada? O sistema não deve incomodar o usuário com estas perguntas, mas é difícil embutir o travamento automático

na aplicação.

Uma segunda abordagem para o controle de concorrência é o *uso de transações*. Os mecanismos de controle baseados em transações vêm sendo usados em sistemas multi-usuários interativos [16] que têm menos demanda por um curto tempo de resposta, o que não é o caso de sistemas colaborativos de tempo real. Para esses sistemas há um grande número de problemas: primeiro, há a complexidade dos algoritmos de controle e processamento distribuído de transações, com custo subseqüente no tempo de resposta; segundo, se o uso de transações for implementado usando travamentos, há os problemas mencionados anteriormente, e se algum outro método estiver sendo usado, como *timestamps*, as ações dos usuários podem ser abortadas pelo sistema. Geralmente, o uso de transações não é recomendado em sistemas interativos; por exemplo, um usuário com duas transações ativas para o mesmo objeto em janelas separadas: seria apresentado a dois estados diferentes dos mesmos dados – quando melhor seria que as janelas mostrassem o mesmo estado. O uso de transações esbarra numa questão filosófica, pois são usadas em sistemas de bancos de dados para dar ao usuário a ilusão de que ele é o único usuário do sistema, em oposição ao que os sistemas colaborativos requerem, que é tornar visíveis as ações do usuário a outros.

Outra abordagem para o controle de concorrência é permitir que apenas *um usuário por vez participe* da interação com o ambiente virtual. A "entrega do bastão" ao usuário da vez, para que o mesmo tenha controle da interação pode ser feita por *software* ou por algum acordo prévio. O maior problema é que, esta técnica só se aplica em situações onde a dinâmica da sessão não depende do paralelismo nos gestos dos participantes, já que pode inibir o fluxo de informações entre os usuários. Além disso, caso haja falha no funcionamento do protocolo (principalmente se ele depender de ações dos usuários), conflitos podem facilmente ocorrer.

Detecção de dependência faz uso de *timestamps* para detectar operações conflitantes, que devem ser resolvidas manualmente; a grande vantagem deste método é que não é necessária a sincronização, e operações não-conflitantes podem ser executadas assim que requisitadas. O problema existente é que qualquer operação que dependa de intervenção do usuário para assegurar a integridade dos dados está sujeita a erros humanos.

Na *execução reversível*, as operações são efetuadas imediatamente, mas as informações sobre as mesmas são mantidas para que seja possível desfazê-las depois, caso seja necessário. Esse

mecanismo requer algum sistema de serialização, ou o uso de *timestamps*, mas permite respostas imediatas. As desvantagens são: a necessidade de algum mecanismo de controle central, para, no caso de operações serem executadas fora de ordem, desfazê-las e tornar a executá-las em ordem, além da possibilidade de uma operação desse tipo confundir o usuário ao aparecer e desaparecer na tela.

2.2 Sistemas Distribuídos

Um sistema distribuído, segundo a definição de Andrew Tanenbaum, é uma "coleção de computadores independentes que se apresenta ao usuário como um sistema único e consistente" [17]. Sistemas distribuídos consistem de uma coleção de computadores autônomos ligados por uma rede de comunicação. Suas vantagens incluem a possibilidade de seu crescimento incremental (ou seja, novos computadores e linhas de comunicação poderem ser acrescidos ao sistema), a possibilidade de implementação de aplicações inerentemente distribuídas, tolerância a falhas e distribuição do processamento.

Assim, a computação distribuída consiste em adicionar o poder computacional de diversos computadores (ou vários processadores na mesma máquina) interligados por uma rede de computadores, para processar colaborativamente determinada tarefa de forma coerente e transparente, ou seja, como se apenas um único computador estivesse executando a tarefa.

Alguns dos fatores que justificam o uso de sistemas distribuídos são: baixo custo dos processadores; custo por instrução em um processador de menor porte é inferior ao custo em um computador de grande porte; desejo de maior participação por parte dos usuários finais; necessidade de maior disponibilidade do sistema; facilidade para interligar sistemas aplicativos distintos; tecnologia de rede disponível; necessidade de compartilhamento de recursos caros; segurança e confiabilidade, devido à distribuição do sistema; opção de implementação de balanceamento de carga entre os processadores; e possibilidade de crescimento incremental de poder de processamento.

Sistemas de computação distribuída podem diferir bastante entre si, dependendo do modo como seus processadores estão conectados. Dentre as várias arquiteturas que utilizam múltiplos processadores têm-se, por exemplo: computadores vetoriais; multiprocessadores;

multicomputadores; e sistemas compostos por várias estações de trabalho conectadas por uma rede local ou por uma rede de longa distância. Além disso, a distribuição do sistema pode ser feita por *hardware* e/ou *software* [18].

No tocante ao hardware, um sistema é distribuído quando não existe memória primária compartilhada pelos elementos processadores, e não-distribuído no caso contrário. Quanto ao *software*, pode-se dizer, de modo geral, que um sistema é distribuído quando seus processos se comunicam através de mecanismos baseados em troca de mensagens. Isso porque se supõe, a princípio, que o *hardware* hospedeiro é distribuído; portanto, processos residentes em elementos processadores diferentes podem trocar informações somente via rede de comunicação, por meio de envio e recebimento de mensagens. Analogamente, um sistema não é distribuído (via *software*) quando seus processos se comunicam através de dados compartilhados.

Existem basicamente quatro esquemas de distribuição, decorrentes da combinação de *hardware* e *software* distribuídos ou não-distribuídos, os quais são [18]:

1. *software* distribuído sendo executado em um *hardware* distribuído: esse esquema caracteriza um sistema, no qual os processos estão sendo executados em processadores separados e se comunicando através da troca de mensagens sobre uma rede local ou uma rede de longa distância;
2. *software* distribuído sendo executado em um *hardware* não distribuído: aqui a troca de mensagens entre os processos é simulada através do uso de memória compartilhada;
3. *software* não distribuído sendo executado em um *hardware* distribuído: aqui o objetivo é esconder a distribuição física, fazendo com que o sistema aparentemente possua memória compartilhada;
4. *software* não distribuído sendo executado em um *hardware* não distribuído: esquema que caracteriza os sistemas convencionais que são ditos fortemente acoplados.

Em termos de distribuição no nível de componentes de *software*, existem três aspectos que podem estar distribuídos: dados, programas e controle [18].

No tocante a distribuição de dados, é possível enumerar os sistemas de arquivos distribuídos e os sistemas de banco de dados distribuídos.

Já no que diz respeito a programas, podem-se ter: programas centralizados e programas distribuídos. Um programa centralizado é executado em uma arquitetura na qual cada um dos

processadores pode executar qualquer instrução desse programa. Já um programa distribuído é aquele que se encontra "particionado" em várias memórias primárias, sendo que cada uma é acessada por um processador diferente que executa a parte do programa que se encontra na memória a ele associada.

A distribuição do controle é a que distingue um sistema distribuído de um sistema de arquitetura clássica. O controle é centralizado quando a execução de um programa, em qualquer instante, está sob os cuidados de um único elemento processador. Quando o controle é distribuído, a execução de um programa está sob os cuidados de mais de um elemento processador.

Abaixo serão explorados dois conceitos-base dos sistemas distribuídos: replicação e sincronização.

2.2.1 Replicação

A replicação é usada para alcançar vários dos objetivos dos sistemas distribuídos, como a tolerância a falhas e a capacidade de incrementar facilmente o poder de processamento do sistema.

Para aumentar o poder de processamento do sistema, existem várias abordagens [19]. Alguns sistemas adotam mecanismos de replicação para todos os objetos distribuídos; sua maior desvantagem é controlar a consistência dos estados dos objetos distribuídos, além de usar mais memória no sistema (somando-se todas as réplicas do sistema). Outras abordagens para replicação de objetos distribuídos envolvem particionamento dos dados, usando modelos hierárquicos coordenados ou semi-coordenados. Neste caso diferentes partes do sistema são distribuídas dependendo da lógica da aplicação; num jogo, por exemplo, essa divisão poderia seguir a geografia do espaço apresentado ao usuário. Cada servidor (ou conjunto de servidores) ficaria responsável por partes da cena e o controle dos estados delas poderia ou não ser distribuído. Essa abordagem ajuda bastante no balanceamento da carga entre os servidores que processam os objetos distribuídos, além de cada servidor necessitar de menos memória e poder de processamento. Porém, o controle de falhas se torna muito mais complexo.

Quando se fala de tolerância a falhas, duas técnicas de replicação emergem [20]: a primeira, chamada *Primary-backup replication*, usa um servidor primário; os outros servidores servem como cópias de *backup*, e não interagem com os clientes. Pode-se visualizar na Figura 2.3 o

funcionamento desta técnica, onde o servidor primário atualiza as cópias de *backup* e depois de receber a confirmação de que todas as cópias estão atualizadas, envia a confirmação da operação ao cliente. Outra técnica existente é chamada de *Active replication*, onde todas as réplicas têm o mesmo papel, sem existir uma entidade centralizadora. Neste caso, como pode-se visualizar na Figura 2.4, o cliente deve atualizar todas as réplicas, e após receber a primeira resposta de que a atualização foi bem-sucedida, deve continuar seu processamento.

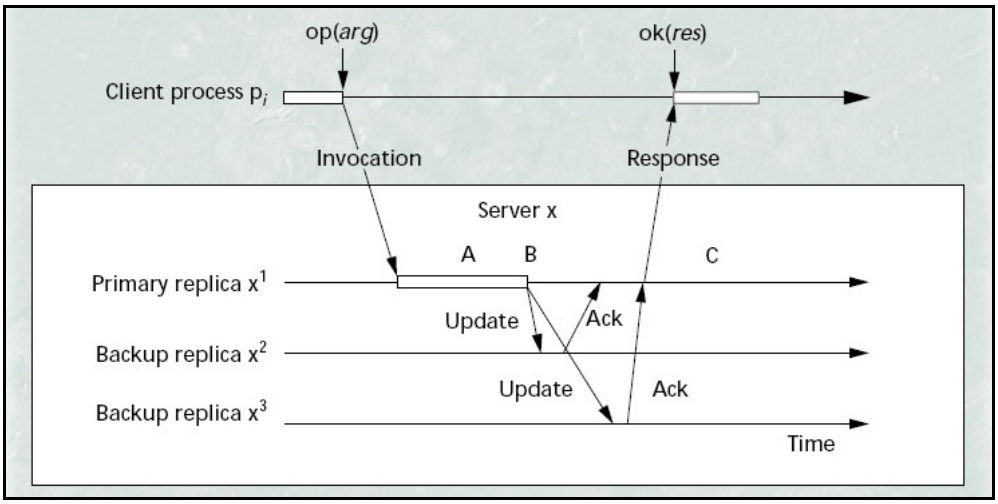

Figura 2.3 - Técnica P*rimary-backup replication* [20]

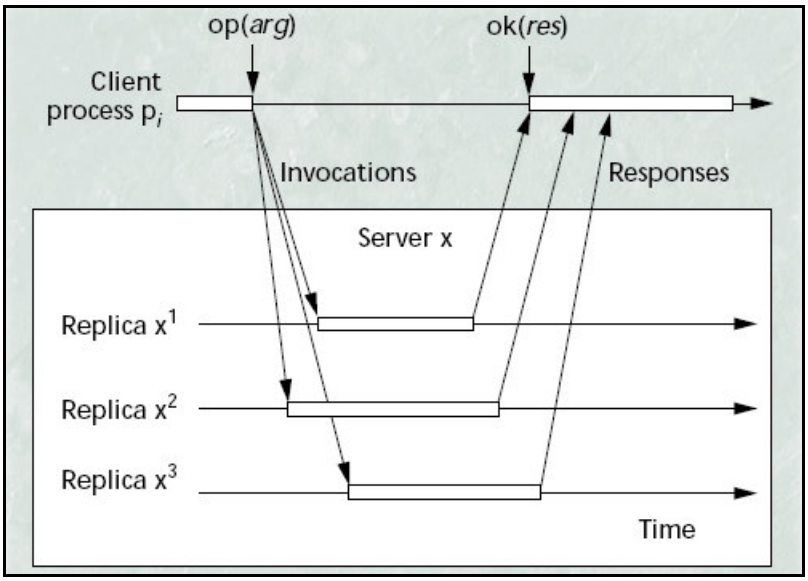

Figura 2.4 - Técnica *Active-replication* [20]

2.2.2 *Sincronização*

Vários problemas de sincronização podem ser enumerados quando usuários acessam um sistema distribuído: sincronismo de início e término de sessão, sincronismos de ações dos clientes (que foram apresentados na Subseção 2.1.2), dentre outros. Problemas de sincronização estão diretamente ligados ao controle de concorrência, como mencionado na Subseção 2.1.2 no segmento que aborda controle de concorrência. Algumas nuances específicas dos sistemas distribuídos serão discutidas na seqüência, como sincronização de relógios e relógios lógicos, algoritmos de *token ring*, eleição, e exclusão mútua.

Lamport [21] diz que a *sincronização de relógios* dos processos não precisa ser absoluta; o que importa não é que todos os processos concordem com o exato tempo em que os eventos aconteceram, mas que concordem com a ordem de ocorrência dos eventos (seus *clocks* lógicos devem convergir). A consistência interna é o que importa, e, para isso, *clocks* físicos não precisam ser sincronizados de forma exata.

No algoritmo de *token ring* (Figura 2.5), é construído um anel lógico por *software* no qual é atribuída a cada processo uma posição no anel. Quando o anel é inicializado, o processo 0 (inicial) ganha o *token*, que circula no anel (passa do processo k para o k+ 1). Quando o processo recebe o *token*, verifica se ele quer entrar na região crítica (onde o processo executar suas tarefas), e em caso positivo, entra na região, realiza o seu trabalho e ao sair passa o *token* para o elemento seguinte do anel. Não é permitido ao processo entrar em uma segunda região crítica com o mesmo *token*.

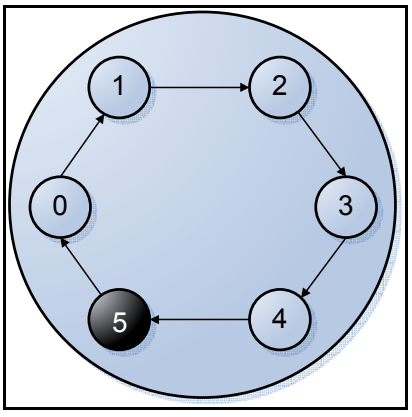

Figura 2.5 - Algoritmo do *token ring*; neste exemplo, o processo 5 está com o *token*

Alguns algoritmos distribuídos requerem um coordenador dos seus processos. Para isso, uma *eleição* é realizada. Para realização da eleição podem ser usados dois algoritmos: um baseado em *token ring* e outro no algoritmo *bully*, de Garcia-Molina [22]. O algoritmo baseado no *token ring* faz o uso do anel, mas sem o *token*. Quando um processo nota que o coordenador não está funcionando, ele envia uma mensagem de ELEIÇÃO para o seu sucessor contendo o seu número de processo. A mensagem dá a volta no anel, sendo o novo coordenador determinado (que é o processo com o número mais alto na lista) e, então, outra mensagem circula informando quem é o novo coordenador, e depois é retirada do anel. O algoritmo *bully* funciona da seguinte forma: quando um processo **P** nota que o coordenador não está respondendo a uma requisição, ele inicia uma eleição. **P** envia uma mensagem de ELEIÇÃO para todos os processos com números maiores que o seu; se nenhum responde, **P** ganha a eleição e se torna coordenador, e se um processo com número maior responde, este último irá assumir a coordenação.

O sistema de *exclusão mútua* [23] também se baseia no conceito de região crítica, mas pode ser implementado usando um sistema centralizado ou distribuído. Na abordagem centralizada, um processo é eleito como coordenador. Quando um processo quer entrar na região crítica, envia uma mensagem fazendo uma requisição ao coordenador, e se nenhum outro processo está na região, a permissão é dada ao solicitante, e o processo então entra na região crítica. Supondo que um processo peça permissão para entrar na região crítica, e o coordenador já tem conhecimento que outro processo está na região, ele não enviará uma resposta, bloqueando o processo até que o primeiro se retire da região crítica. Na abordagem do algoritmo distribuído, quando um processo quer entrar na região crítica, constrói uma mensagem contendo o nome da região, o número do processo e o tempo atual, e a envia para todos os outros processos. Quando um processo recebe uma mensagem de requisição de outro processo, uma das seguintes opções pode ocorrer:

1. Se o receptor não está na região crítica nem quer entrar, envia de volta uma mensagem de OK;
2. Se o receptor já está na região, ele não responde e coloca a requisição na fila;
3. Se o receptor quer entrar na região crítica, mas ainda não o fez, ele compara o tempo da mensagem que chegou com o tempo da mensagem que ele enviou para os outros processos, e o menor tempo (ou ordenamento, dependendo do mecanismo adotado) vence. Se a mensagem que chegou tem tempo menor, ele envia de volta um OK. Se sua própria

mensagem possui o menor tempo, ele coloca na fila a requisição que chegou e não envia resposta, bloqueando o processo.

Após pedir permissão, um processo espera até que todos tenham dado a sua permissão, e somente quando todas as permissões chegam, o processo pode entrar na região crítica. Ao sair da região crítica, ele envia uma mensagem de OK para todos os processos na sua fila.

Na Figura 2.6 é possível visualizar o funcionamento do algoritmo distribuído de exclusão mútua. No exemplo, os processos 0 e 2 querem entrar na região crítica, com o primeiro enviando uma mensagem no tempo 8, e o segundo com o tempo 12. Como o processo 0 tem o tempo menor, entra na região crítica e só responde ao processo 2 depois de sua saída, realizando o desbloqueio do último.

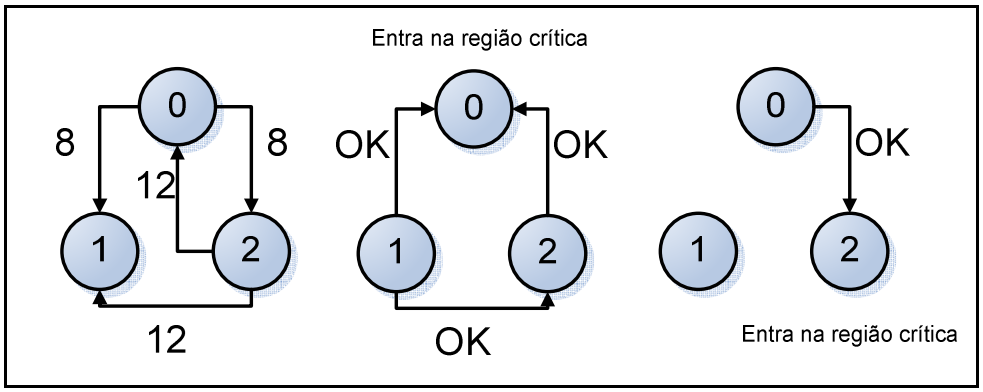

Figura 2.6 - Algoritmo distribuído de exclusão mútua

2.3 Realidade Virtual em Ambientes Colaborativos

A RVD oferece a possibilidade de se utilizar um mesmo ambiente virtual a partir de diferentes computadores ligados em rede, ao mesmo tempo. Quando um ambiente virtual distribuído também permite que diversos usuários, fisicamente distantes, acessem esse ambiente através da rede e o utilizem simultaneamente interagindo com ele e uns com os outros, diz-se que o ambiente virtual distribuído é colaborativo. A RVC permite o desenvolvimento deste tipo de ambiente [2].

O ambiente virtual colaborativo pode ser usado para discutir sobre um conjunto de dados que está sendo visualizado. Este conjunto de dados pode ser originário de uma base de dados,

pode estar sendo computado em um supercomputador, e, além disso, os dados podem ser reais ou não.

Por exemplo, um modelo de elevação digital, originalmente armazenado em um banco de dados, é visualizado por alguns usuários, que analisam o modelo com o objetivo de detectar erros no mesmo. Na medida em que as análises são feitas, os usuários interagem entre si para validar suas opiniões, ou para buscar informações adicionais sobre o modelo. Constatado que um suposto erro necessita ser corrigido, o modelo é então alterado e atualizado no banco de dados, e na representação do terreno no ambiente virtual, mantendo a consistência em todos os pontos do sistema.

Outra categoria de ambientes virtuais é composta por ambientes de tele-imersão, os quais não visam apenas reproduzir, nos mínimos detalhes, um encontro face-a-face entre pessoas, mas também definir a "próxima geração" de interfaces. Estas interfaces devem possibilitar que usuários em locais geograficamente distribuídos possam colaborar em tempo real em um ambiente híbrido, simulado e compartilhado, como se estivessem no mesmo espaço físico. Ela será a mais nova síntese de tecnologias de mídia (ambientes 3D, tecnologias de projeção e de visualização, tecnologias de rastreamento, tecnologias de áudio, robótica e *haptics*) usando redes de alta velocidade, ou outra tecnologia de transmissão de dados.

2.4 Áreas de Aplicação

As aplicações que podem ser implementadas como sistemas distribuídos são inúmeras. Existem quatro vantagens básicas de se projetar uma aplicação para um sistema distribuído, a saber: diminuição do tempo de execução da aplicação; aumento do grau de confiabilidade e disponibilidade da aplicação; uso de partes do sistema para fornecer serviços especializados; e, a inerente distribuição da aplicação [18].

A *diminuição do tempo de execução da aplicação* pode ser alcançada através do uso do paralelismo existente em um sistema distribuído. Alguns programas terão seus tempos de execução reduzidos se partes deles forem executadas em processadores diferentes, ao mesmo tempo. Um exemplo disso é a compilação paralela de módulos de um dado programa em máquinas diferentes, além da implementação de algoritmos de busca heurística.

Sistemas distribuídos são potencialmente *mais confiáveis*, pois se os processadores são autônomos, a falha em um não afeta o funcionamento correto dos demais. Portanto, a confiabilidade do sistema pode ser aumentada ao se replicar funções e/ou dados da aplicação nos vários processadores. Desse modo, se alguns processadores falharem, os demais poderão continuar o serviço. Um exemplo clássico de aplicação tolerante a falhas é o controle de uma fábrica automatizada.

Além de todos os fatores supramencionados, a implementação de uma aplicação como um conjunto de *serviços especializados* amolda-se adequadamente no ambiente de um sistema distribuído. Um exemplo é um ambiente que forneça diversos serviços, entre eles um servidor de arquivos, um servidor de impressão, processos (uma máquina mais potente que compartilha seus recursos), um servidor de tempo, um *gateway*, entre outros. Cada um desses serviços pode usar um ou mais processadores dedicados, garantindo bom desempenho e alta confiabilidade, além da facilidade em adicionar novos processadores para atender a novos serviços ou para aumentar a capacidade computacional de serviços já existentes. Os servidores podem trocar requisições entre si através da rede, tornando recursos especiais do sistema passíveis de compartilhamento.

Uma outra vantagem do uso de sistemas distribuídos é na implementação de aplicações que sejam *inerentemente distribuídas*. Nos ambientes colaborativos de tempo real e nas aplicações de RVC podem ser aproveitadas as vantagens dos sistemas distribuídos. Atualmente, como já foi citado, existe uma categoria de aplicações que está na interseção dos três tópicos explorados nas Seções 2.1, 2.2 e 2.3 e está crescendo amplamente [3], que é o MMOG; esse tipo de aplicação recria um mundo virtual compartilhado por milhares ou centenas de milhares de usuários simultaneamente. Em alguns deles, até o conceito de tempo que se passa no ambiente virtual emula o tempo real [4], sendo possível marcar horários para eventos no mundo virtual.

Os MMOGs são, atualmente, as aplicações que exploram amplamente os conceitos apresentados neste capítulo [19][24][25]. Elas permitem o acesso de muitos usuários ao mesmo tempo em um único mundo virtual, em alguns casos, de grande dimensão; para isso, devem lidar com problemas de limitação de banda, latência (e diferença de latência entre usuários), processamento de cena, balanceamento de carga, conflitos entre ações dos usuários, consistência entre objetos distribuídos, entre outros. Além disso, alguns deles ainda demandam um grande poder de processamento gráfico nas máquinas dos usuários [26].

Além dos MMOGs, pode-se citar algumas aplicações colaborativas que vem surgindo, como o Microsoft Office Live [27] e o Sharepoint Portal Server [28], que têm como objetivo colaboração síncrona e assíncrona para criação, gerenciamento e modificação de conteúdo, além de possibilitar encontros virtuais. Essas aplicações nascem de um desejo cada vez maior de evitar viagens e diminuir custos das empresas. Por outro lado, essas aplicações trazem outras necessidades dessas aplicações são relativas a vários aspectos técnicos, como controle de versão de documentos compartilhados, gerenciamento de grandes volumes de dados, tolerância a falhas e qualidade de serviço.

3 FFORCE

Este capítulo descreve em detalhes o *framework* FFORCE para desenvolvimento de ambientes colaborativos, em tempo real, e o protótipo desenvolvido como prova de conceito.

3.1 Descrição

FFORCE deriva de *Framework for Collaborative Environments*, e foi concebido com o propósito de auxiliar no desenvolvimento de aplicações colaborativas em tempo real, como, por exemplo, aplicações que têm o intuito de utilizar a *expertise* de profissionais fisicamente distantes uns dos outros sem a necessidade de viagens. Além disso, foi motivado pelo surgimento de sistemas colaborativos de grande escala [6], como os MMOGs, pela evolução das tecnologias necessárias para desenvolvimento de tais sistemas.

O escopo inicial do FFORCE é voltado para aplicações que façam uso da RV, auxiliando o desenvolvimento de aplicações nessa área, mas também visando outros tipos de aplicações gráficas que possuam representações de câmeras e hierarquia de objetos, como, por exemplo, jogos *online*. Há uma vasta gama de projetos que foram desenvolvidos na área de RVD, grande parte usando VRML (*Virtual Reality Markup Language*), embora, atualmente, existam outras opções de renderização e visualização de mundos 3D bastante sofisticadas [29][30].

O FFORCE foi desenvolvido em dois módulos, cliente e servidor, ambos construídos sobre a biblioteca de comunicação XCServices (maiores detalhes na Subseção 3.2.1), desenvolvida pelo autor.

Por servidor, entenda-se toda a parte de processamento da cena, que pode ser distribuída entre instâncias que fazem parte do *core* (o *core* – ou núcleo, é o conjunto de servidores que se comunicam e recebem conexões dos clientes) do sistema, que recebe as conexões dos clientes e realiza todo o controle de conflitos e distribuição de cópias. A replicação das unidades centrais de processamento, além de facilitar a implantação de um sistema de tolerância a falhas, tem por objetivo implementar numa futura versão, adicionar módulos de processamento especializado, como o de comportamento físico. Isso possibilitaria que, possuindo placas de processamento físico, como a Ageia Physx [31], apenas nos computadores que compõem o *core* do sistema, diversos

clientes tivessem em seu terminal a exibição de comportamentos de simulação física. O uso de *hardware* de processamento físico incrementa o realismo das cenas apresentadas ao usuário, como pode ser visto na Figura 3.1. Sem este módulo estes comportamentos só seriam reproduzidos, caso todos os clientes possuíssem as placas, recursos estes que ainda são custosos.

Figura 3.1 - Cena de um jogo que faz uso de processamento físico [31]

O módulo cliente realiza a comunicação com o *core*, transmitindo as ações do usuário e recebendo as mudanças de posição dos objetos, calculadas para esses movimentos (maiores informações ver Subseção 3.3.2).

3.2 Ferramentas e Bibliotecas Utilizadas

O FFORCE foi desenvolvido em linguagem C e C++, utilizando o ambiente de desenvolvimento Microsoft Visual Studio 2005. Para auxiliar no desenvolvimento do protótipo, foram utilizadas três bibliotecas, uma para o módulo de comunicação (XCServices), um *engine* gráfico (OGRE [30]) e uma biblioteca de auxílio ao uso de *threads* em C++ (Libcppmt). Estas bibliotecas são detalhadas na seqüência.

3.2.1 Biblioteca de Comunicação XCServices

A maior parte da camada de comunicação do FFORCE foi escrita na linguagem C, aproveitando a biblioteca XCServices que já havia sido desenvolvida, por este autor, para codificação e transmissão de tipos básicos, como s*trings*, *ints*, *shorts*, *booleans*, *arrays* de *bytes* e *doubles*. Inicialmente, a função da biblioteca XCServices era somente prover interoperabilidade com um *middleware* escrito em linguagem JAVA, o XPeer [32], mapeando também as mensagens do protocolo do mesmo.

Todas as funções de transmissão da biblioteca XCServices haviam sido implementadas, apenas, para envio de dados sobre *sockets* do tipo *Transmission Control Protocol* (TCP), mas foram estendidas no projeto FFORCE para enviar dados sobre *User Datagram Protocol* (UDP), assim como, para codificar e decodificar estes dados para a formação de pacotes a serem transmitidos. A codificação/serialização para os tipos mencionados anteriormente segue o padrão descrito na *Application Programming Interface* (API) da linguagem JAVA [33], e encontra-se descrita na Tabela 3.1.

Tipo	Descrição
byte	Leitura e escrita de um *byte,* representado como o tipo primitivo *unsigned char* da linguagem C.
boolean	Utiliza um *byte* com valor 0 para representação do valor "falso" e 1 para "verdadeiro".
int	Representa números inteiros através de 4 *bytes*, e para transmissão e recepção utiliza o formato *big endian*, onde o *byte* mais significativo é transmitido primeiro.
short	Representa números inteiros utilizando 2 *bytes*, e também faz uso da codificação *big endian* para transmissão e recepção.
array de *bytes*	Codifica e transmite seqüências de até 65.536 *bytes*. Para transmissão, utiliza um *boolean*, para indicar se o *array* é nulo e então, um número inteiro positivo de 2 *bytes*, para o tamanho da seqüência, que é, então, transmitida.
array extendido de *bytes*	Utiliza a mesma técnica aplicada para *arrays* de *bytes*, mas pode transmitir seqüências de até 4.294.967.296 *bytes*, utilizando uma representação de 4 *bytes* para o tamanho.
string	A representação de *strings* utiliza o formato *modified* UTF-8 (*unicode*) [33], melhor explicador a seguir. Para transmissão, é utilizado um *boolean*, indicando se a string é nula e, então, um *short* indicando a quantidade de *bytes* a serem lidos para recompor a *string*, que é, então, transmitida.

Tabela 3.1 - Codificação dos tipos suportados na biblioteca de comunicação

O formato de codificação de caracteres da linguagem JAVA (*modified* UTF-8 [33]), e utilizado no FFORCE, merece uma explicação mais detalhada. Ele utiliza um, dois ou três *bytes* para transmitir cada caractere que originalmente tem dois *bytes*. Os caracteres na faixa entre '\u0001' e '\u007F' são representados usando um *byte*, como pode ser visto na figura abaixo:

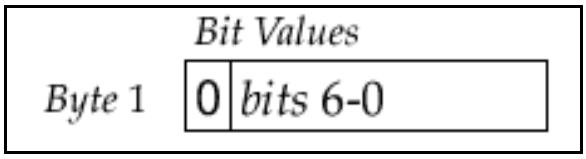

Figura 3.2 - Representação de caractere usando um *byte* [33]

Os caracteres na faixa entre '\u0080' e '\u07FF' e o caractere nulo '\u0000' são representados usando dois *bytes* (Figura 3.3), e os caracteres entre '\u0800' e '\uFFFF' são representados utilizando três *bytes* (Figura 3.4).

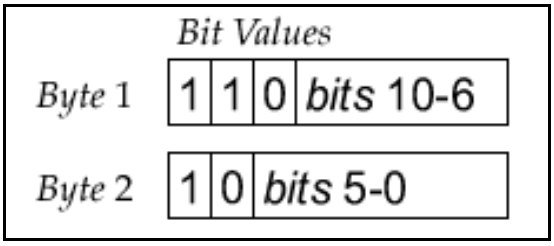

Figura 3.3 - Representação de caractere usando dois *bytes* [33]

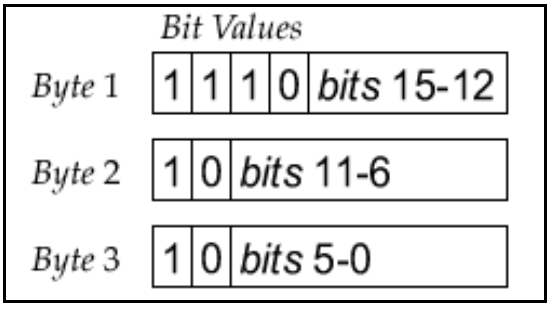

Figura 3.4 - Representação de caractere usando três *bytes* [33]

3.2.2 OGRE

O OGRE (*Object-oriented Graphics Rendering Engine*) é um *engine* gráfico de código aberto que provê uma vasta gama de *plugins*, ferramentas e *add-ons*, facilitando a criação de vários tipos de aplicações gráficas [30][34].

Esse *engine* funciona adequadamente em inúmeras configurações de *hardware* com aceleração gráfica 3D (*Graphics Processing Unit* - GPUs) e plataformas disponíveis no mercado. A interface de programação oferecida nativamente pelo OGRE é escrita em C++, mas existem alguns *wrappers* para o OGRE em Java, .NET e Python, ainda em fase de desenvolvimento [30].

O propósito do OGRE não é ser apenas um *game engine*; ele é um *rendering engine* genérico que pode ser incorporado a bibliotecas de tratamento de entradas, de processamento de som e as plataformas que disponibilizem algoritmos de inteligência artificial, compondo assim um *kit* de desenvolvimento mais completo [34].

No protótipo do FFORCE, o OGRE foi utilizado para representar os cenários, exibi-los, realizar operações sobre os objetos e captar os movimentos que os usuários realizam na cena compartilhada. Ou seja, o OGRE foi usado para implementar as operações primitivas básicas de RV que são executadas no protótipo do sistema. Cada objeto da cena compartilhada é associado a um identificador de dois *bytes* no formato *unsigned short* (ou seja, cada cena pode ter até 65.536 objetos) para ser transmitido.

3.2.3 Libcppmt

A libcppmt, desenvolvida no Grupo de Pesquisa em Redes e Telecomunicações (GPRT) do CIn-UFPE, é uma biblioteca de auxílio ao desenvolvimento de aplicações *multi-thread* em linguagem C++; ela acrescenta uma camada de abstração para o uso de *threads* em aplicações, evitando chamadas a funções do sistema operacional e dispensando a necessidade de gerenciar diretamente os *handles* dessas *threads*. Essas funcionalidades da biblioteca foram utilizadas no protótipo do FFORCE.

3.3 Arquitetura e Funcionamento

Como mencionado anteriormente, o FFORCE foi dividido em dois módulos: cliente e servidor. A arquitetura de software do protótipo desenvolvido pode ser visualizada na Figura 3.5.

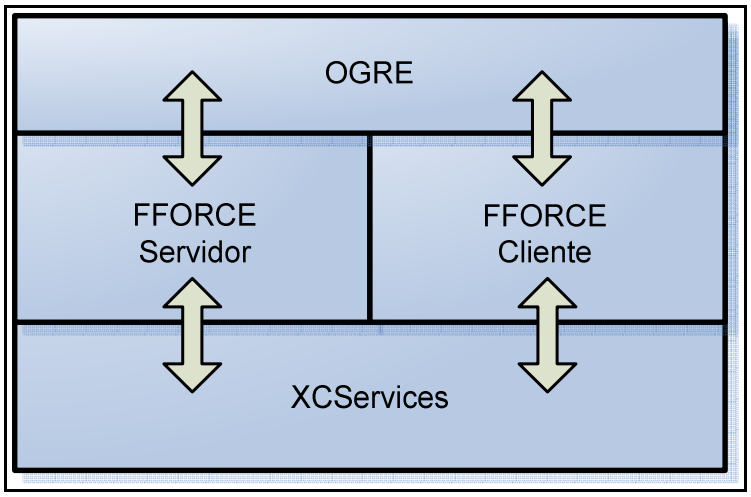

Figura 3.5 - Arquitetura de *software* do protótipo do FFORCE

Na camada mais alta, encontra-se a lógica específica da aplicação (no caso o protótipo), onde, através do OGRE, há a representação da cena que está sendo compartilhada por todos os integrantes da sessão. Caso o usuário estivesse usando outras ferramentas para representar a sua cena, seria nesta camada, acima de todos os módulos que ele deveria inserir a lógica da sua aplicação.

A arquitetura do módulo servidor, representada na Figura 3.6, foi dividida em 3 partes. A primeira, (na parte superior da imagem), representa logicamente a cena. A segunda, executa as operações aplicadas aos elementos contidos na cena, e a terceira, contém a lógica de comunicação (que utiliza a biblioteca de comunicação) com os clientes e outros servidores. É importante mencionar que esta divisão favorece a especialização do processamento que é feito no servidor. Seria possível, por exemplo, adicionar um módulo de processamento físico, acoplado a uma placa de *hardware* específica para execução de tal tarefa.

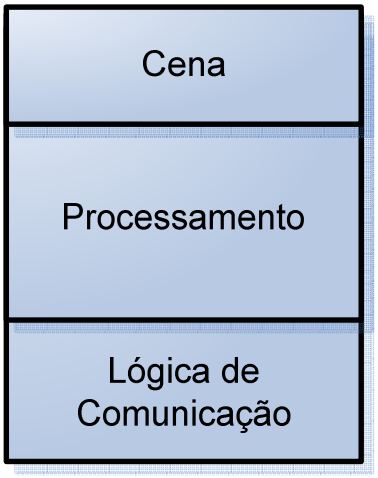

Figura 3.6 - Arquitetura de *software* do módulo servidor

Nas próximas subseções explica-se como o FFORCE foi estruturado e seu funcionamento; os detalhes do protocolo de comunicação que suporta a lógica da aplicação serão descritos na Seção 3.4.

3.3.1 Restrições

Como mencionado na Seção 2.1, operações comuns sobre objetos virtuais compartilhados incluem movê-los, alterá-los e removê-los. O FFORCE, atualmente, suporta apenas que o usuário mova objetos. Porém, como é possível criar hierarquias de objetos para representar dados mais complexos, quando um objeto é movido, ele pode modificar uma estrutura mais sofisticada da qual faz parte.

Atualmente, a cena representada na sessão é de conhecimento prévio de todos os participantes, sejam eles clientes ou servidores. Esta restrição diminui consideravelmente a quantidade de dados a serem transmitidos pela rede, já que não é necessário enviar dados sobre os objetos representados na cena; somente suas posições e forças aplicadas a eles são transmitidas pela rede.

O FFORCE foi desenvolvido para plataforma Windows 32 *bits*, e o protótipo ainda não se encontra numa versão estável, para disponibilização de uso por outros desenvolvedores, devido às restrições de cronograma do projeto.

3.3.2 *Cliente*

O módulo cliente é responsável por conectar-se a uma das instâncias do *core* do sistema, exibir a cena compartilhada na tela do terminal, capturar as interações do usuário com a cena, enviar dados sobre a movimentação aplicada a um objeto e receber atualizações das posições dos objetos da cena.

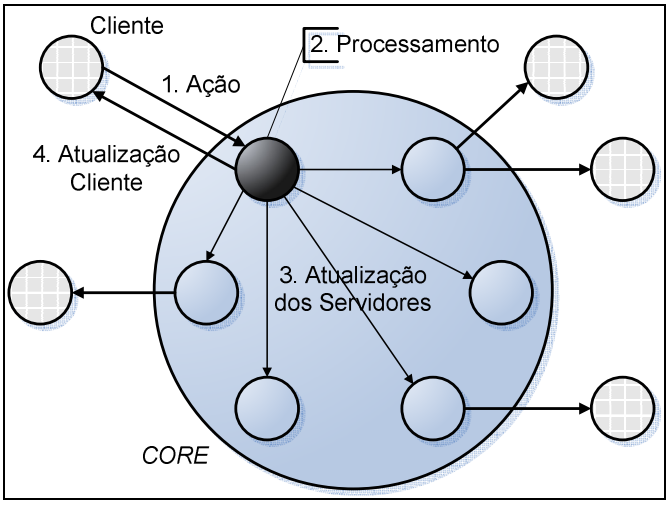

Figura 3.7 - Processo de atualização da cena após ação do usuário

Ao conectar-se ao servidor, caso a instância selecionada esteja com seu *pool* de conexões preenchido, o cliente recebe uma lista dos servidores que atualmente fazem parte do *core* do sistema. O cliente é responsável por tentar conexão com os outros servidores, até um número limite estabelecido pelo usuário; caso o usuário deseje, o *default* pode ser o total de instâncias do *core*. Após uma conexão ser estabelecida com sucesso, o cliente pode iniciar a interação com a cena, enviando ações e recebendo atualizações, como pode ser visualizado na Figura 3.7.

No módulo cliente, o OGRE é usado para representar informações sobre a cena e exibi-la no terminal, como visto na Figura 3.8, além de capturar as interações do usuário com a cena; essa interação é feita na forma de vetores aplicados a objetos. Esses vetores são capturados, quadro-a-quadro, e transmitidos à instância do *core* ao qual o usuário está conectado. O *core* então realiza as operações de cálculo e retorna aos usuários as posições atualizadas dos objetos alterados por aquele movimento.

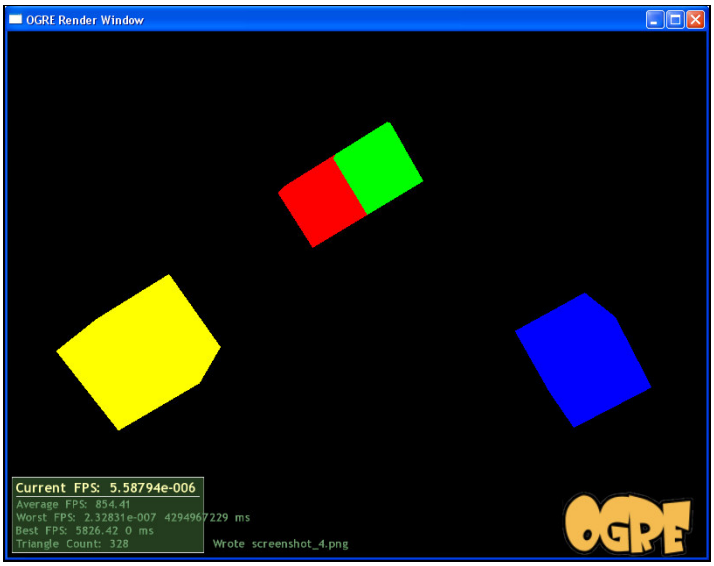

Figura 3.8 - Cena visualizada no cliente através do *engine* OGRE

Para continuar participando da sessão, o cliente deve enviar mensagens de *keep-alive* somente ao servidor ao qual está conectado, como visto na Figura 3.9. No momento em que o usuário desejar finalizar sua participação, pode simplesmente fechar a conexão, forçando o servidor a descobrir que as mensagens de *keep-alive* deixaram de ser enviadas por este usuário. Uma forma elegante de desconexão é informar ao servidor da sua saída da sessão.

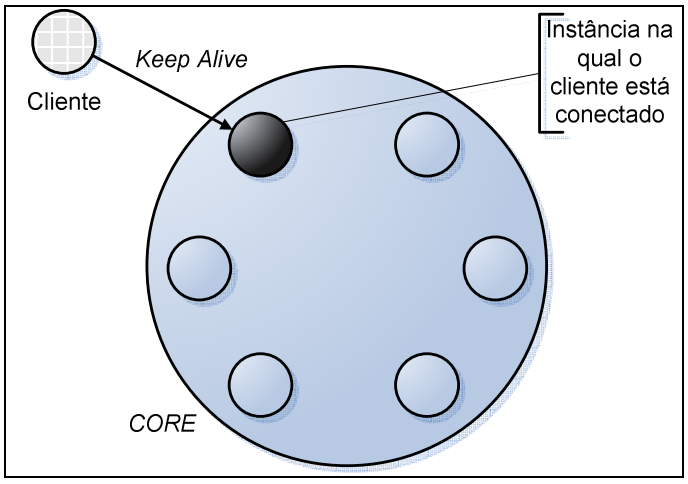

Figura 3.9 - Mensagens de *keep alive* mantêm o usuário na sessão

3.3.3 Servidor

No módulo servidor encontra-se a lógica para resolução de conflitos, distribuição de cópias, sincronização de cena e tolerância a falhas. Cada instância de servidor que é iniciada, caso queira fazer parte de um *core* existente, deve conhecer ao menos um endereço de uma instância que já foi iniciada; dessa forma, ao comunicar-se com essa instância, receberá a lista de todas as instâncias iniciadas para aquela sessão, sincronizará as posições dos objetos da cena e passará também a receber conexões de clientes.

3.3.3.1 Iniciando um Novo Servidor

Ao se iniciar uma instância de servidor, há duas opções: a primeira é iniciar um novo *core*; a segunda é ingressar num *core* existente (usando um endereço de instância conhecido, como citado anteriormente). Cada cena possui uma identificação (*scene id*), evitando que servidores com cenas diferentes façam parte de um mesmo conjunto, evitando, também, inconsistências com os usuários conectados.

Caso a nova instância queira fazer parte de um núcleo existente, após conectar-se a um servidor integrante do *core*, a instância recebe uma lista de endereços e portas dos participantes do núcleo atual; somente então, inicia conexões TCP com as demais instâncias. Caso isto não seja possível, serão feitas novas requisições da lista de servidores (seguidas de novas tentativas de conexões), em intervalos de tempo definidos. No caso de falhas sucessivas, a tentativa de ingresso no *core* será abortada.

3.3.3.2 Sistema de Sincronização e Convergência de Cópias

O sistema de sincronização de cópias do FFORCE é baseado na premissa de que os participantes possuem conhecimento da cena completa. Desta forma, só é necessário enviar as posições de objetos que tiveram posições alteradas por movimentos. Isso acontece também nas sincronizações iniciais, tanto para clientes como para servidores, com a diferença de que na sincronização inicial, dependendo da versão atual da cena, todas as posições dos objetos são enviadas, ou nenhuma delas. Neste caso, se a versão do cliente ou servidor que está se conectando for igual à versão atual

da cena, nenhuma posição será enviada ao mesmo. Isso pode ocorrer em dois casos: quando ocorreu alguma perda de conexão do cliente e, até o momento da reconexão, a cena não foi modificada ou, quando a cena inicial não foi alterada por nenhum participante da sessão. Se a versão da cena do cliente ou servidor ao se conectar for diferente da atual, todos os objetos da cena terão suas posições atualizadas. A Figura 3.10 ilustra o processo de sincronização entre um cliente que se conecta e o servidor.

Para evitar inconsistências, são mantidos dez estados anteriores da cena nas instâncias do *core*, para que seja possível desfazer e refazer operações em casos de conflitos.

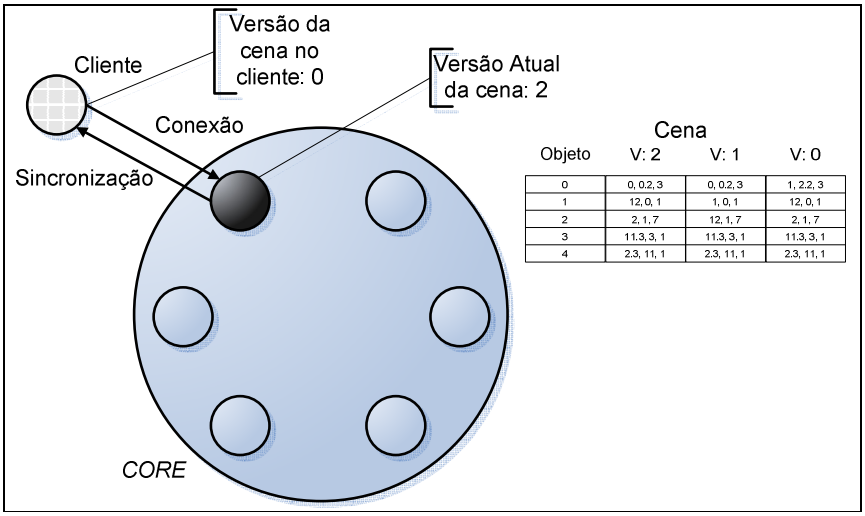

Figura 3.10 - Sincronização da cena entre cliente e servidor

Para a serialização de operações e conseqüente convergência das cópias, foi implementado um algoritmo baseado no GOT (*Generic Operation Transformation*), de conformidade com o descrito por Sun et al [35]. Este algoritmo foi escolhido porque, segundo Vidot et al [36], que comparou os algoritmos dOPT, ORESTE, adOPTed, GOT, SOCT2, GOTO [37][38][39][40][41], e propôs dois novos algoritmos, SOCT3 e SOCT4, ele é o que mais se adequa as necessidades do FFORCE. Este algoritmo usa vetores para manter os estados das interações passadas, através de um ordenamento global (realizado no FFORCE através das prioridades de cada instância do *core*). Somente parte do algoritmo foi implementada, incluindo as operações de UNDO (desfazer) e REDO (refazer).

3.3.3.3 Controle e Resolução de Conflitos

Como explicado anteriormente, cada instância do servidor possui uma lista com as dez últimas versões da cena exibida, para efeito de controle de conflito. O identificador de versão da lista é um número (inteiro positivo) randômico que a cada nova cena adicionada é incrementado em uma unidade. Essa identificação é importante, pois é transmitida entre os servidores e também é usada para detecção de conflitos.

A detecção do conflito é feita quando um *core* recebe de um cliente uma movimentação para um objeto da cena de uma versão que já foi modificada (e ainda não foi atualizada em todo o *core*) ou que está sendo modificada no momento do recebimento da ação. Na versão atual do protótipo, caso aconteça algum conflito entre movimentos aplicados pelos clientes, não haverá aviso aos participantes. O *core* resolverá internamente quaisquer conflitos entre movimentos de usuários e informará aos mesmos as novas posições dos objetos alterados pelos movimentos.

Para resolver conflitos, a cada instância do *core* é designada uma identificação (ID), representando sua prioridade no caso de conflitos; esse ID inicia com 0 (zero) para a instância inicial do *core*, que tem a maior prioridade e é incrementada em 1 (um) para cada instância que passa a participar do *core*. No caso de movimentos conflitantes, quando há a sincronização entre as instâncias após o cálculo das posições, o movimento que foi calculado pela instância com menor prioridade é descartado pelas outras instâncias. Após o descarte, os servidores que entraram em conflito trocam de prioridade para impedir que um mesmo servidor tenha sempre prioridade em relação aos demais. Essa troca de prioridade é coordenada pela instância com a maior prioridade, de forma a evitar que ocorram inconsistências quando acontecem conflitos entre vários servidores. É fácil entender o algoritmo de troca de prioridade através do exemplo explicado na seqüência de figuras abaixo.

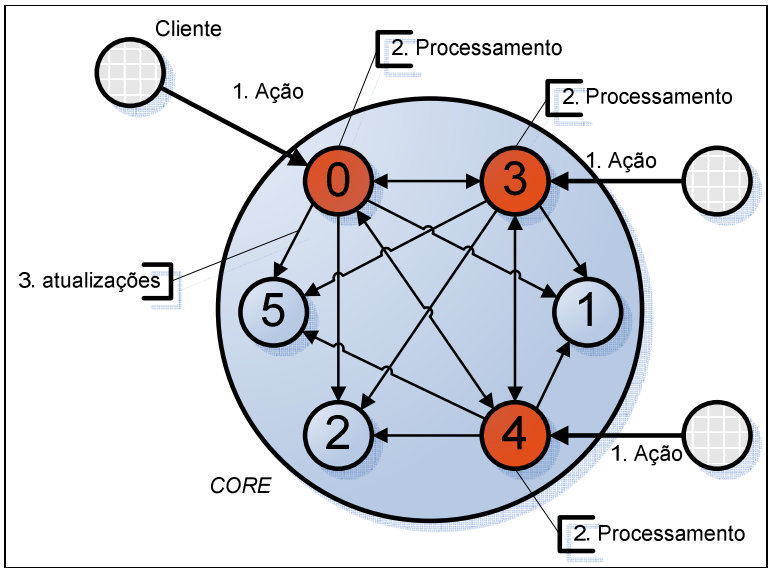

Figura 3.11 - Operações conflitam entre as instâncias 0, 3 e 4

Como pode ser visto na Figura 3.11, clientes conectados aos servidores de prioridade 0, 3 e 4 movimentam um mesmo objeto ao mesmo tempo, gerando atualizações conflitantes. As instâncias 0, 3 e 4 enviam suas atualizações a todos os outros, e neste momento, o conflito é detectado por eles. O descarte (Figura 3.12) das versões enviadas por 3 e 4 é relativamente simples, já que, independentemente da ordem de recebimento, a versão enviada pelo servidor com maior prioridade (no exemplo, 0) sempre deve ser mantida.

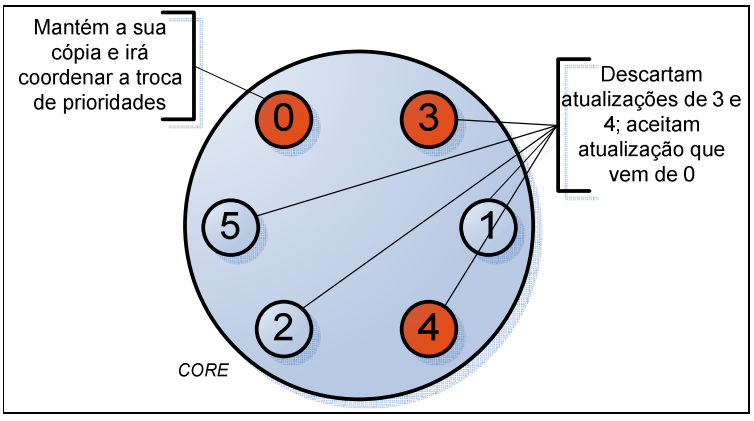

Figura 3.12 - Instâncias 1, 2 e 5 aceitam atualização enviada por 0

Após o descarte das operações efetuadas por 3 e 4, a instância com prioridade 0 inicia a troca de prioridades com as instâncias conflitantes; a instância de maior prioridade atribui a si própria a menor prioridade envolvida no conflito; os outros servidores receberão um "*upgrade*" nas suas ordens de preferência, como pode ser visto na Figura 3.13. Neste exemplo, a instância 0 cai para o nível 4, o servidor de prioridade 3 recebe 0 e o elemento de nível 4 sobe para 3.

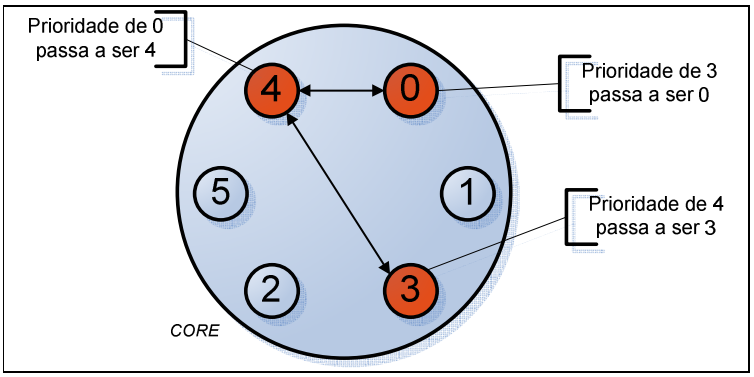

Figura 3.13 - Instância 0 coordena troca de prioridades

No escopo desse trabalho, a abordagem usada para resolução dos conflitos foi baseada no trabalho descrito por Ellis e Gibbs [16] e Suleiman et al [42], divergindo das abordagens citadas no seguinte aspecto: as operações possuem prioridades, que podem ser modificadas de acordo com filas implementadas nos servidores. Na abordagem do FFORCE são as instâncias que possuem prioridades atribuídas, de modo a facilitar a implementação. No algoritmo usado no FFORCE, a causalidade das operações não é respeitada (pois as mesmas podem ser desfeitas sem enviar notificação ao usuário), em detrimento da simplicidade das operações; é fácil perceber que o ordenamento das operações, caso necessário, também poderia ser feito de acordo com as prioridades das instâncias, para serem transpostas no caso de conflito.

3.4 Protocolo de Comunicação

O protocolo de comunicação desenvolvido para ser usado pelo FFORCE não visa a interoperabilidade com outros sistemas de colaboração. O foco deste protocolo consiste em ser simples e conciso. Não há, por exemplo, sistema de autenticação para restringir a entrada de

pessoas não-autorizadas na sessão, pois segurança não foi considerada como um dos requisitos iniciais do projeto.

A codificação usada nas mensagens do protocolo de comunicação segue o padrão já explanado na Seção 3.2.1 e contempla os seguintes tipos básicos: *strings*, *ints*, *shorts*, *booleans*, *arrays* de *bytes* e *doubles*.

Todas as mensagens do protocolo têm cabeçalho de um *byte*, onde os 3 *bits* de maior valor são 011, usados apenas para efeito de identificação de mensagem do protocolo FFORCE e os 5 *bits* de menor valor indicam o código da mensagem. O tamanho do conteúdo da mensagem é variável.

Na Tabela 3.2 tem-se as mensagens do FFORCE, que serão usadas para explicar o funcionamento das principais funcionalidades do protocolo. Ao lado delas, são apresentados os códigos correspondentes.

Tipo da Mensagem	Nome da Mensagem	Código
Entrada num *core* existente.	SERVER_JOIN	11000
	SERVER_JOIN_OK	11001
Entrada de um cliente na rede.	CLIENT_JOIN	11100
	CLIENT_JOIN_OK	11101
Sincronização total da cena.	SYNC_REQUEST	10000
	SYNC_RESPONSE	10001
Envio da lista de servidores que fazem parte do *core*.	SERVER_LIST_REQUEST	01000
	SERVER_LIST	01001
Mensagens de troca de identificação de prioridade.	SERVER_SET_PRIORITY_REQUEST	01010
	SERVER_SET_PRIORITY_OK	01011
Mensagem de envio de movimento aplicado à cena.	SCENE_MOVEMENT	00100
Atualização do posicionamento de objetos da cena.	SCENE_UPDATE	01100
Mensagem de erro.	ERROR_MSG	00001
Keep alive.	KEEP_ALIVE	00010

Tabela 3.2 - Tipos, nomes e códigos das mensagens do protocolo de comunicação

3.4.1 Conexão Inicial de um Cliente

A conexão inicial do cliente a uma instância conhecida do *core* é feita mediante o envio da mensagem CLIENT_JOIN. Caso a instância já tenha alcançado o número máximo de conexões, retorna ao cliente uma mensagem SERVER_LIST (melhor explicada na subseção seguinte) contendo uma lista dos servidores que atualmente fazem parte do *core*, para que o cliente tente uma conexão com algum dos outros servidores. Em caso de sucesso na conexão, uma mensagem

CLIENT_JOIN_OK é enviada ao cliente, que então tem sua cena atualizada usando a mensagem SYNC_REQUEST, também melhor descrita a seguir; depois da etapa de sincronização, o cliente pode, então, iniciar a interação.

3.4.2 Replicação

Como na versão atual o FFORCE suporta a replicação através do prévio conhecimento da cena compartilhada, incrementar o poder do *core* através de uma nova instância é uma tarefa relativamente simples, já que somente é necessário sincronizar as cópias da nova instância e conhecer o endereço de qualquer instância que faça parte do *core*. O protocolo responsável pela sincronização é executado sobre TCP e composto das seguintes partes: um sistema de sincronização total, usado na conexão inicial (e reconexão, no caso de falha) do servidor ao *core* existente; e um sistema de controle dos nós que compõem o *core*.

Para iniciar a entrada num *core* existente, primeiro a nova instância deve iniciar uma conexão TCP com um servidor que já faz parte do *core*; feita a conexão, uma mensagem SERVER_JOIN é enviada composta de 4 *bytes* (que é a identificação da cena), mais uma *string* contendo o endereço onde aquele servidor receberá conexões de novos servidores, e de um tipo *short* (2 *bytes*) indicando a porta para conexão TCP. Somente então serão transmitidos, na mesma mensagem, os dados para que os clientes realizem a conexão, compostos por uma *string* indicando um endereço e mais dois *shorts* indicando as portas TCP e UDP para realização das conexões. Em caso de sucesso, a instância que está ingressando no *core* do sistema receberá uma mensagem SERVER_JOIN_OK, contendo a identificação de prioridade (*short* - 2 *bytes*) que o servidor deverá usar; em caso de falha, uma mensagem do tipo ERROR_MSG é transmitida.

Quando a entrada é confirmada, a lista de servidores que atualmente fazem parte do *core* é enviada, através da mensagem SERVER_LIST_REQUEST. Utilizando SERVER_LIST, a requisição é respondida; após o *byte* de identificação da mensagem, um *short* (2 *bytes*) é transmitindo, indicando quantos servidores estão contidos na lista, excluindo o requisitante. Os dados enviados de cada servidor são: identificação de prioridade (*short*), endereço de conexão (*string*) para clientes e dois *shorts* indicando porta TCP e UDP seguidos da *string* representando o endereço de conexão para outros servidores e um *short* indicando a porta para conexão dos servidores. Ou seja, para

transmitir os dados de apenas um servidor são usadas duas *strings* (tamanho variável) e cinco *shorts* (dez *bytes*), além do *byte* de identificação da mensagem, como pode ser visto na Figura 3.14.

A	B	C	D	E		F	G
01101001	16 *bits*	16 *bits*	variável	16 *bits*	16 *bits*	variável	16 *bits*

A	SERVER_LIST
B	Número de Servidores
C	Prioridade do servidor
D	Endereço para recebimento de conexão dos clientes
E	Portas TCP e UDP para clientes
F	Endereço para recebimento de conexão de outros servidores
G	Porta TCP para servidores

Figura 3.14 - Campos da mensagem SERVER_LIST para envio de dados de apenas um servidor

Após a entrada no *core* e da atualização da lista de servidores, o sistema de sincronização total é iniciado, com uma mensagem SYNC_REQUEST seguida de 4 *bytes*, representando a versão atual da cena que a instância possui. O servidor que recebe esta requisição irá consultar o vetor de estados armazenados. A resposta será a mensagem SYNC_RESPONSE, e no caso de versões idênticas, um *byte* com valor 00000000 é enviado, indicando não haver objetos a serem atualizados. Caso contrário (houve modificação da cena), todos os objetos terão suas posições atualizadas, e a mensagem será adicionada de quatro *bytes* que indicarão a versão a ser transmitida e dois *bytes* que indicarão a quantidade de objetos que devem ser atualizados. Somente então, cada objeto terá sua posição atualizada utilizando-se 14 *bytes* por objeto, onde 2 *bytes* são para identificação, mais 12 *bytes* para representação da posição do mesmo (coordenadas X,Y e Z representadas por *doubles*).

A	B	C	D	E		
01110001	32 *bits*	16 *bits*	16 *bits*	32 *bits*	32 *bits*	32 *bits*

A	SYNC_RESPONSE
B	Versão da cena
C	Número de objetos na cena
D	Identificação do objeto
E	Coordenadas X, Y e Z

Figura 3.15 - Campos da mensagem SYNC_RESPONSE atualizando apenas um objeto

3.4.3 Tolerância a Falhas

O sistema de descoberta de falhas, seja do *hardware* da máquina ou da rede, é bastante simples, baseado num sistema de *keep alive* (usando a mensagem KEEP_ALIVE). Entre os servidores, como há conexões TCP, caso a conexão de um nó com outros falhe, a reconexão é tentada por até 30 segundos; nesse caso, se ainda houverem usuários conectados, uma mensagem do tipo SERVER_LIST será enviada aos mesmos, contendo a lista de servidores disponíveis. Esta mensagem indica aos clientes que eles devem trocar de servidor porque alguma falha ocorreu e o servidor não conseguiu conexão com o *core* do sistema.

Do lado do cliente, as mensagens de *keep alive* são enviadas pelos mesmos aos servidores, com intervalos de 2 segundos, havendo ou não ações dos usuários. Caso o servidor deixe de receber mensagens de um cliente por 5 segundos (sejam de *keep alive* ou de ações), a desconexão do mesmo é realizada.

3.4.4 Ações dos Usuários

Atualmente, as ações dos usuários estão limitadas a vetores simples aplicados à cena, e para enviá-las, o cliente utiliza a mensagem SCENE_MOVEMENT, seguida simplesmente de seis *doubles* (24 *bytes*), para indicar a força aplicada à cena. Após calcular o resultado para o movimento transmitido, o servidor retorna, através da mensagem SCENE_UPDATE, atualizações para os objetos da cena. Após a identificação da mensagem, é transmitida a versão da cena para aquele

objeto (*int*, quatro *bytes*). Como a transmissão dessa mensagem é feita sobre UDP, cada objeto atualizado configura uma mensagem. Ou seja, para cada objeto tem-se o cabeçalho de um *byte*, quatro *bytes* para versão da cena, dois *bytes* para identificação do objeto (*short*) e 12 *bytes* com a posição do mesmo (três *doubles*).

É interessante também acompanhar o efeito de uma ação de um usuário no *core* do sistema; quando do recebimento de um evento na cena, o servidor que recebeu a mensagem realiza o cálculo das novas posições, atualiza a própria cena e a envia para os clientes conectados a ele e para os outros servidores do *core*, sempre utilizando a mensagem SCENE_UPDATE (para os clientes através de UDP e para os outros servidores através de TCP).

3.4.5 Controle e Resolução de Conflitos

Como mencionado na Subseção 3.3.3.3, um conflito é detectado quando o servidor recebe uma atualização, proveniente de outras instâncias, antes que ele termine de executar uma ação para o mesmo objeto. Quando do recebimento de ações simultâneas de usuários conectados ao mesmo servidor, elas serão colocadas em ordem de acordo com a prioridade internamente designada a cada cliente pelo servidor, e executadas nesta ordem. Esta prioridade só é de conhecimento do próprio servidor. As atualizações conflitantes que ocorrerem, são resolvidas da seguinte forma: se a ação conflitante vier de servidores com maior prioridade, caso a atualização ainda não tenha sido enviada, será descartada; caso a ação já tenha sido enviada, ele desfaz a operação, ou seja, volta para a versão anterior da cena.

Em ambos os casos, o servidor aplica as atualizações recebidas através da mensagem SCENE_UPDATE. Em seguida, o servidor de maior prioridade coordena, através da mensagem SERVER_SET_PRIORITY_REQUEST, a troca de identificação de prioridade dos servidores. Esta troca é informada a todos os integrantes do *core*. A mensagem identifica o servidor que terá sua prioridade alterada através da *string* com o endereço de conexão, seguida de dois *bytes* (*short*) que indicam a porta de conexão. Somente então, a identificação de prioridade é enviada, usando-se mais dois *bytes*. Os servidores devem confirmar a troca de prioridades através da mensagem SERVER_SET_PRIORITY_OK.

3.5 Aplicação em Cenários Reais

Como citado no início do documento, áreas de trabalho que lidam com domínios específicos e grande número de variáveis têm grande demanda por ferramentas colaborativas, para ganhar agilidade e reduzir custos. Dentre estas áreas, pode-se incluir o planejamento de poços de petróleo e gás natural. Neste tipo de trabalho existe uma grande dificuldade em reunir todos os especialistas necessários para realizar certas tarefas (por vezes emergenciais) em um mesmo local de trabalho físico. Além disso, é importante que a colaboração seja feita em tempo real, um requisito essencial para quebrar distâncias.

Planejamento e *design* de poços de petróleo e gás é uma área bastante promissora de pesquisa dentro do contexto da RV. Os engenheiros de poços lidam com ferramenta integradas e interativas que utilizam modelos 3D precisos e realistas. Eles necessitam cada vez mais de um suporte computacional moderno e sofisticado para simular e visualizar grandes volumes de dados. Neste contexto, o FFORCE pode ser usado como ferramenta que permita e facilite o trabalho cooperativo à distância, dando suporte aos engenheiros (e demais especialistas) das áreas de petróleo e gás no planejamento e design de poços, aplicando a tecnologia de RV.

O número de variáveis envolvidas no processo de perfuração de um poço é elevadíssimo. O monitoramento dessas variáveis requer o uso de tecnologias de ponta para coletar os dados previamente, como durante a perfuração de um poço. De posse desses dados é que o time de especialistas será capaz de otimizar o planejamento de poços e lidar com problemas durante a sua perfuração e completação [43].

Nesse contexto, a ferramenta Vis-Petro [44], desenvolvida pelo Grupo de Pesquisa em Realidade Virtual e Multimídia (GRVM) do CIn-UFPE, em parceria com a Petrobrás e financiada pelo CNPq, surge como um instrumento de suporte ao planejamento e *design* de poços de petróleo e gás. Esta ferramenta pode ser utilizada em PCs *desktop* (Figura 3.16) ou em ambientes mais sofisticados como aqueles com projeção 3D e/ou CAVEs. Cabe ressaltar que não existem ferramentas nacionais dessa natureza, caracterizando o aspecto inovador desse projeto.

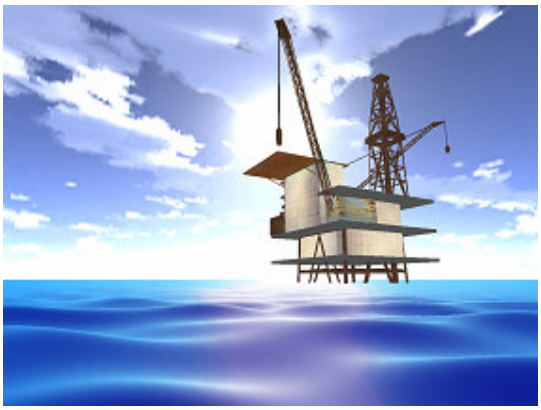

Figura 3.16 - Ferramenta Vis-Petro [44]

Os altos custos envolvidos no processo de perfuração de poços de petróleo e gás, além dos riscos ambientais existentes tanto durante o processo de perfuração como também depois da completação na fase de extração do fluido, justificam o desenvolvimento de uma ferramenta para planejar poços. Ferramentas similares foram desenvolvidas por empresas renomadas internacionalmente no mercado petrolífero como a Landmark [45] e a Schlumberger [46], porém os seus custos são extremamente elevados. Além disso, são ferramentas monousuário e com interface 2D.

É neste cenário que o FFORCE pode ser usado para tornar a ferramenta Vis-Petro uma ferramenta de colaboração; vários fatores contribuem para isso, dentre eles o fato que o Vis-Petro, além de ter sido desenvolvido em C++, fez também uso do *engine* OGRE para visualização e planejamento dos poços, o que facilita sua integração com o FFORCE. Como ainda existe uma complexidade alta para se obter a solução "ótima" na construção de um poço de petróleo e gás, vários benefícios seriam alcançados através da utilização de RVC no planejamento e *design* de poços. Uma vez que esse tipo de planejamento requer modelagem 3D, modelos de simulação precisos e a integração de vários profissionais durante a tarefa de planejamento do poço.

A integração do Vis-Petro ao FFORCE deve gerar ganhos para ambos os projetos. Transformar o Vis-Petro em uma ferramenta de colaboração agrega valor à mesma, além de criar demanda por pesquisas em diversas áreas, como análise de interface gráfica (a interface atual não dá suporte à colaboração), análise de desempenho, dentre várias outras. Novas demandas para o FFORCE devem surgir a partir desta integração, já que a arquitetura atual do Vis-Petro, mesmo que se torne

distribuída, não aproveitaria o processamento distribuído, gerando um novo requisito para o FFORCE, que seria a opção de realizar o processamento da cena no cliente, usando o *core* apenas para distribuição e sincronização das cópias apresentadas aos usuários.

4 Conclusão e Trabalhos Futuros

Neste TG, foram pesquisadas diversas áreas de atuação, entre elas: colaboração em tempo real, sistemas distribuídos e RV. O FFORCE, sendo um *framework*, explorou conceitos multi-disciplinares abrindo novos horizontes nas áreas já citadas, que poderão gerar pesquisas relevantes na sua extensão. Principalmente em termos de colaboração, onde existe uma tendência de que, com o constante crescimento de poder de processamento nos clientes e de banda de comunicação [47], a colaboração *on-line* seja uma tarefa comum do dia-a-dia. Além disso, a experiência colaborativa deve tornar-se cada vez mais imersiva, com representações mais refinadas dos usuários e suas ações numa aplicação de interação virtual.

A pesquisa bibliográfica reflete a multi-disciplinaridade e a heterogeneidade do FFORCE: existem referências que foram publicadas desde 1974 em áreas tão diversas como algoritmos de sincronização de relógios lógicos [21] e técnicas de desenvolvimento de jogos massivos (MMOGs) [19]. Vale ressaltar que esta pesquisa de embasamento ainda trará muitas contribuições futuras, para que seja possível retirar as restrições que foram impostas ao FFORCE pelo tempo disponível para pesquisa, especificação e implementação do mesmo. O caráter inovador de simular o comportamento físico do ambiente virtual no servidor também contribui para novas linhas de pesquisa a serem desenvolvidas.

Em termos de trabalhos futuros, muitos tópicos relativos ao desenvolvimento do FFORCE emergem, sendo alguns deles:

- desenvolver uma versão estável do FFORCE, não somente para sua integração ao Vis-Petro (que deve trazer novos requisitos ao projeto), como também, possibilitando sua utilização por terceiros;
- testes de eficiência dos algoritmos de resolução de conflitos e sincronização de cópias e eventual comparação de outros;
- testes de carga utilizando o NIST Net (emulador de rede) [48], para descobrir os limites de carga para a arquitetura do FFORCE.
- estudar a viabilidade de utilizar o FFORCE em outras plataformas, como Linux, Linux 64 bits e Windows 64 bits;

- integração de uma busca otimizada na cena, como *frustum* [49], já que objetos que sejam afetados por possíveis atualizações e que não estão visíveis ao cliente (por causa do posicionamento da câmera) não precisariam ter seu novo posicionamento enviado de imediato;
- adicionar a capacidade de modificar propriedades (como as dimensões) dos objetos da cena, além de removê-los e criar novos;
- adicionar ao módulo de processamento especializado do servidor a simulação do comportamento físico, utilizando bibliotecas como o NxOGRE [30] e a SDK (*Software Development Kit*) da Ageia PhysX [31].

Outra grande vertente de trabalhos surge no contexto do projeto FFORCE por ter escolhido o *engine* gráfico OGRE para implementação do protótipo. Como o OGRE é um *engine* amplamente utilizado, duas novas idéias podem ser adicionadas ao projeto FFORCE: usá-lo como meio de replicação de aplicações de interação baseadas no OGRE e realizar o carregamento dinâmico da cena através da rede. Atualmente, existem *loaders* de cena para o OGRE que fazem o carregamento em tempo de execução, mas apenas localmente.

Este trabalho traz contribuições práticas ao desenvolver um *framework* que poderá ser usado em um projeto em desenvolvimento (Vis-Petro), que deverá migrar para uma versão colaborativa, além de encontrar utilização no ambiente em que foi desenvolvido (GRVM) para aplicações que fazem uso da RV.

5 Referências

5.1 Bibliografia

[1] GOMES, R.; HOYOS-RIVERA, G., COURTIAT, J. *Collaborative Virtual Environments: Going Beyond Virtual Reality*. In: Proceedings of the IEEE International Conference on Multimedia and Expo, 2003. v.2 p.105-113.

[2] LEIGH J.; JOHNSON, A.E.; DEFANTI, T.A. *Issues in the design of a flexible distributed architecture for supporting persistence and interoperability in collaborative virtual environments*. Chicago: University of Illinois at Chicago, 1997. 14p. Relatório técnico.

[3] WOODCOCK, B.S. *An Analysis of MMOG Subscription Growth – Version 19.0*. UR: http://www.mmogchart.com/Analysis.html Consultado em 31 de maio, 2006.

[4] WORLD OF WARCRAFT. URL: http://www.worldofwarcraft.com/ Consultado em 29 de setembro, 2006.

[5] GROOVE NETWORKS. URL: http://www.groove.net/ Consultado em 31 de maio, 2006.

[6] BIGWORLD TECHNOLOGY. URL: http://www.bigworldtech.com/ Consultado em 31 de maio, 2006.

[7] BOSSER, A. *Replication Model for Designing Multi-Player Games Interactions*. In: Proceedings of the ACM SIGCHI International Conference, 2004. p. 263-268.

[8] DOUGLAS, S.; TANIN, E.; HARWOOD, A.; KARUNASEKERA, S. *Enabling Massively Multi-Player Online Gaming Applications on a P2P Architecture*. In: Proceedings of the IEEE International Conference on Information and Automation, 2005. p. 7-12.

[9] MASSIVE UCI, Irvine, EUA. URL: http://www.isr.uci.edu/events/massive/ Consultado em 30 de maio, 2006.

[10] STEELE, M. "Emergent Game Technologies". URL: http://www.isr.uci.edu/events/massive/presentations/MSteele%20-%20MASSIVE%20Presentation%20-%20r1.ppt Consultado em 30 de maio, 2006.

[11] GREENBERG, S. *Real Time Distributed Collaboration*. In: Encyclopedia of Distributed Computing. Estados Unidos: Kluwer Academic Publishers, 1998.

[12] GUTWIN, C.; GREENBERG, S.; ROSEMAN, M. *Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and Evaluation*. In: Proceedings of HCI on People and Computers XI, 1996. p. 281-298.

[13] WAR ROCK WEBSITE. URL: http://www.warrock.net/ Consultado em 29 de setembro, 2006.

[14] BERNIER, Y.W. *Latency compensating methods in client/server in-game protocol design and optimization*. In: Proceedings of the Game Developers Conference, 2000.

[15] GREENBERG, S.; MARWOOD, D. *Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface*. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, 1994. p. 207-217.

[16] ELLIS, C.A.; GIBBS, S.J. *Concurrency Control in Groupware Systems*. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1989. p. 399-407.

[17] TANENBAUM, A.S.; STEEN, M.V. *Distributed Systems: Principles and Paradigms.* Estados Unidos: Prentice Hall, 2001. 803p.

[18] CASETTI, O.; AKAMATU, D.M.; KIRNER, C. *Paradigmas para Construção de Sistemas Distribuídos*. Tematec SERPRO, 1993.

[19] BOSSER, A. *Massively Multiplayer Games: Matching Game Design with Technical Design*. In: Proceedings of the ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, 2004. p. 263-268.

[20] GUERRAOUI R.; SCHIPER, A. *Software-Based Replication for Fault Tolerance*. IEEE Computer, v.30, p. 68-74, 1997.

[21] LAMPORT, L. *Time, Clocks, and the Ordering of Events in a Distributed System*. Communications of the ACM. v.21, n.7, 1978.

[22] GARCIA-MOLINA, H. *Elections in a Distributed Computing System*. IEEE Transactions on Computers, v.C-31, n.1, 1982.

[23] LAMPORT, L. *A new solution for Dijkstra's concurrent programing problem*. Communications of the ACM, v.17, n.8, p.453-455.

[24] BAUER, D.; ROONEY, S.; SCOTTON, P. *Network Infrastructure for Massively Distributed Games*. In: Proceedings of the 1st Workshop on Network and System Support for Games, 2002. p.36-43.

[25] LEE, K.W.; KO, B.J.; CALO, S. *Adaptive server selection for large scale interactive online games*. In: Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video, 2004. p.152-157.

[26] THE ELDER SCROLLS. URL: http://www.elderscrolls.com/ Consultado em 1 de outubro, 2006.

[27] MICROSOFT OFFICE LIVE. URL: http://officelive.microsoft.com/ Consultado em 1 de outubro, 2006.

[28] MICROSOFT SHAREPOINT PORTAL SERVER 2003. URL: http://www.microsoft.com/brasil/sharepoint/ Consultado em 1 de outubro, 2006.

[29] CRYSTAL SPACE 3D. URL: http://www.crystalspace3d.org/ Consultado em 29 de setembro, 2006.

[30] OGRE 3D. URL: http://www.ogre3d.org/ Consultado em 31 de julho, 2006.

[31] AGEIA PHYSX. URL: http://www.ageia.com/physx/index.html Consultado em 29 de setembro, 2006.

[32] ROCHA JR., J.; FIDALGO, J.; DANTAS, R.; OLIVEIRA, L.; KAMIENSKI, C.; SADOK, D. *X-Peer: A Middleware for Peer-to-Peer Applications*. In: 1st Brazilian Workshop on Peer-to-Peer, 2005.

[33] API JAVA 2 PLATFORM SE 5.0. URL: http://java.sun.com/j2se/1.5.0/docs/api/ Consultado em 31 de julho, 2006.

[34] FARIAS, T.; PESSOA, S.; TEICHRIEB, V.; KELNER, J. *O engine gráfico OGRE*. In: Symposium on Virtual Reality, Belém, 2006.

[35] SUN, C.; ZHANG, Y.; JIA, X.; YANG, Y. *A Generic Operation Transformation Scheme for Consistency Maintenance in Real-time Cooperative Editing Systems*. In: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: the Integration Challenge, 1997. p.425-434.

[36] VIDOT, N.; CART, M.; FERRIÉ, J.; SULEIMAN, M. *Copies convergence in a distributed real−time collaborative environment*. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, 2000. p.171-180.

[37] ELLIS, C.A.; GIBBS, S.J. *Concurency Control in Groupware Systems*. In: Proceedings of the ACM International Conference on Management of Data, 1989. p.399-407.

[38] KARSENTY, A.; BEAUDOUIN−LAFON M. *An Algorithm for Distributed Groupware Applications*. In: Proceedings of the 13th International Conference on Distributed Computing Systems, 1993. p.195−202.

[39] RESSEL, M.; NITSCHE−RUHLAND D.; GUNZENHÄUSER R. *An Integrating, Transformation−oriented Approach to Concurrency Control and Undo in Group Editors*. In: Proceedings of the ACM International Conference on Computer Supported Cooperative Work, 1996. p.288−297.

[40] SULEIMAN, M.; CART, M.; FERRIÉ, J. *Serialization of Concurrent Operations in Distributed Collaborative Environment*. In: Proceedings of the ACM International Conference on Supporting Group Work, 1997. p.435-445.

[41] SUN, C.; ELLIS, C.S. *Operational Transformation in Real−Time Group Editors: Issues, Algorithms and Achievements*. In: Proceedings of the ACM International Conference on Computer Supported Cooperative Work, 1998. p.59-68.

[42] SULEIMAN, M.; CART, M.; FERRIÉ, J. *Concurrent Operations in a Distributed and Mobile Collaborative Environment*. In: Proceedings of the 14th International Conference on Data Engineering, 1998. p.36-45.

[43] THOMAS, J.E. *Fundamentos da engenharia de petróleo*. 2ª edição. Rio de Janeiro: Editora Interciência, 2001.

[44] BARROS, P.; PESSOA, D.; LEITE, P.; FARIAS, R.; TEICHRIEB, V.; KELNER, J. *Three-dimensional oil well planning in ultra-deep water*. In: Symposium on Virtual Reality, 2006, Belém. p. 285-296.

[45] LANDMARK. URL: http://www.lgc.com Consultado em 30 de maio, 2006.

[46] SCHLUMBERGER LIMITED. URL: http://www.slb.com Consultado em 30 de maio, 2006.

[47] BURNS, E. *Worldwide Broadband Accounts Continue to Rise*. URL: http://www.clickz.com/showPage.html?page=3574831 Consultado em 29 de setembro, 2006.

[48] NIST NET HOME PAGE. URL: http://www-x.antd.nist.gov/nistnet/ Consultado em 29 de setembro, 2006.

[49] ASSARSSON, U.; MÖLLER, T. *Optimized view frustum culling algorithms for bounding boxes*. Journal of Graphics Tools. v.5, n.1, p.9-22, 2000.

_____

Judith Kelner

_____

Veronica Teichrieb

_____

Gabriel Fernandes de Almeida