Estudo de escalabilidade de servidores baseados em eventos em sistemas multiprocessados: um estudo de caso completo

Daniel de Angelis Cordeiro

DISSERTAÇÃO APRESENTADA
AO INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO
DO GRAU DE MESTRE EM CIÊNCIAS

Área de Concentração : **Ciência da Computação**
Orientador : **Prof. Dr. Alfredo Goldman vel Lejbman**

São Paulo, 27 de outubro de 2006

Este trabalho foi financiado por FAPESP (Processo N. 03/10064-4).

Estudo de escalabilidade em servidores baseados em eventos em sistemas multiprocessados: um estudo de caso completo

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Daniel de Angelis Cordeiro
e aprovada pela Comissão Julgadora.

São Paulo, 27 de outubro de 2006.

Banca Examinadora :

Prof. Dr. Alfredo Goldman vel Lejbman (orientador) – IME-USP

Prof. Dr. Marco Dimas Gubitoso – IME-USP

Prof. Dr. Renato Fontoura de Gusmão Cerqueira – PUC-Rio

à minha mãe Beth, ao meu pai Carlos
e aos meus irmãos Tiago e Rafael

Agradecimentos

À minha família, por todo carinho e incentivo.

Ao meu orientador, professor Alfredo Goldman, pela sua dedicação a este trabalho de mestrado, por sua paciência e incentivo nos momentos difíceis e, principalmente, pela sua amizade. Mais do que um orientador, pude contar sempre com um amigo durante todos esses anos de iniciação científica e de mestrado.

À Dilma Menezes da Silva, por sua participação como co-orientadora deste trabalho. Sua ajuda foi fundamental na definição do projeto e nos estudos iniciais. Além disso, sua determinação em ajudar-nos, mesmo que remotamente, é sem dúvida uma grande inspiração para nós todos.

A todos os amigos do IME, por tornarem os anos que passei aqui muito mais divertidos. Em particular, gostaria de agradecer aos amigos que estiveram mais próximos durante o mestrado: Alex Kusano, Bruno Pera, Dairton Bassi, Daniel Martin, Danilo Sato, Daniel Vaquero, Fabio Nishihara, Fernando Mário, Flavia Rainone, Giuliano Mega, Igor Sucupira, Jan Gentil, João Soares, Julian Monteiro, Kelly Braghetto, Leandro Barion, Marcos Broinizi, Pedro Takecian, Peter Kreslins e Ricardo Abrantes.

À Fundação de Amparo à Pesquisa do Estado de São Paulo, pelo financiamento deste projeto.

Por fim, a todas as pessoas que contribuíram de forma direta ou indireta na realização deste trabalho.

Muito obrigado!

Resumo

O crescimento explosivo no número de usuários de Internet levou arquitetos de software a reavaliarem questões relacionadas à escalabilidade de serviços que são disponibilizados em larga escala. Projetar arquiteturas de software que não apresentem degradação no desempenho com o aumento no número de acessos concorrentes ainda é um desafio.

Neste trabalho, investigamos o impacto do sistema operacional em questões relacionadas ao desempenho, paralelização e escalabilidade de jogos interativos multi-usuários. Em particular, estudamos e estendemos o jogo interativo, multi-usuário, QuakeWorld, disponibilizado publicamente pela id Software sob a licença GPL. Criamos um modelo de paralelismo para a simulação distribuída realizada pelo jogo e o implementamos no servidor do QuakeWorld com adaptações que permitem que o sistema operacional gerencie de forma adequada a execução da carga de trabalho gerada.

Abstract

The explosive growth in the number of Internet users made software architects reevaluate issues related to the scalability of services deployed on a large scale. It is still challenging to design software architectures that do not experience performance degradation when the concurrent access increases.

In this work, we investigate the impact of the operating system in issues related to performance, parallelization, and scalability of interactive multiplayer games. Particularly, we study and extend the interactive, multiplayer game QuakeWorld, made publicly available by id Software under GPL license. We have created a new parallelization model for Quake's distributed simulation and implemented that model in QuakeWorld server with adaptations that allows the operating system to manage the execution of the generated workload in a more convenient way.

Sumário

1. Introdução

A tecnologia de redes de computadores atual possibilita a troca de informações entre milhões de usuários através da Internet. Tal ambiente rapidamente despertou o interesse de diversos prestadores de serviços, que perceberam que a Internet é a maneira mais eficaz atender seus usuários, a custos relativamente baixos.

O crescimento no número de serviços disponibilizados e, principalmente, no número de usuários desses serviços fez com que a escalabilidade de servidores de aplicação se tornasse uma das principais preocupações dos arquitetos de *software*.

Segundo relatório [17] do instituto de pesquisa IBOPE//NetRatings, cerca de 5 milhões de pessoas possuíam acesso domiciliar à Internet no Brasil no ano 2000. No segundo trimestre de 2006, o Brasil respondia por 21,2 milhões dos 472 milhões de pessoas com acesso domiciliar à Internet. O rápido crescimento do número de usuários observado nos últimos anos criou uma demanda sem precedentes por serviços disponibilizados via Internet.

O Yahoo! [31], conhecido portal de serviços na Internet, foi o endereço mais acessado do mundo durante o primeiro semestre de 2006 segundo a Alexa Internet [35], empresa especializada em monitorar a utilização da Internet. De acordo com o relatório financeiro [32] divulgado pelo Yahoo! Inc., durante o segundo quadrimestre de 2006 foram contabilizados, em média, 3,491 bilhões de *page views* – quantidade de vezes que as páginas de um determinado domínio foram visualizadas – por dia[1]. Segundo o relatório, o portal recebe uma média de 412 milhões de visitantes distintos, por dia.

[1] não são contabilizados os acessos ao Yahoo! China e ao Yahoo! Japão

Redes velozes e confiáveis não são o único pré-requisito para a criação de serviços que consigam atender tal demanda. É necessário que as aplicações disponibilizadas sejam capazes de atender a esse grande número de usuários sem apresentarem degradação no desempenho.

O entendimento da escalabilidade de tais serviços é fundamental para que os arquitetos de *software* consigam disponibilizá-los para um grande número de usuários. Ao mesmo tempo em que a escalabilidade de aplicações científicas foi exaustivamente estudada, pouco se conhece sobre a escalabilidade de aplicações comerciais.

Um dos paradigmas bastantes utilizados em sistemas distribuídos escaláveis é o paradigma de programação *event-driven* (baseado em eventos), onde o fluxo de execução do programa é determinado por eventos e o processamento corresponde a respostas a estes eventos. Tipicamente, o aplicativo informa ao despachante de eventos quais os eventos em que está interessado e o despachante notifica o programa sempre que o evento ocorre.

Duas classes de programas comerciais de grande interesse teórico baseados em eventos são: (1) servidores web e (2) jogos interativos multi-usuários. A escalabilidade de servidores web é discutida em vários trabalhos [9, 16, 15], assim como o impacto do sistema operacional na aceleração de seu desempenho [52, 62, 61, 19, 44, 18, 54].

Neste trabalho, investigamos o impacto do sistema operacional em questões relacionadas ao desempenho, paralelização e escalabilidade de jogos interativos multi-usuários. Em particular, estudamos e estendemos o jogo interativo, multi-usuário, QuakeWorld [28], disponibilizado publicamente pela id Software [25] sob a licença GPL [29]. Criamos um modelo de paralelismo para a simulação distribuída realizada pelo jogo e o implementamos no servidor do QuakeWorld, com adaptações que permitem que o sistema operacional gerencie de forma adequada a execução da carga de trabalho gerada.

1.1. Organização do Trabalho

Esta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta trabalhos na área de escalabilidade de sistemas de computação, questões de escalabilidade relacionadas ao sistema operacional e um estudo sobre as arquiteturas de *software* utilizadas em jogos comerciais. No Capítulo 3, descrevemos o modelo utilizado pelo servidor original do QuakeWorld, nossos esforços para o aumento do número de usuários simultâneos e a caracterização do desempenho do servidor com o aumento do número de clientes. Um novo modelo para a paralelização do QuakeWorld, detalhes de sua implementação e adaptações para tornar seu paralelismo mais eficiente em sistemas operacionais modernos são apresentados no Capítulo 4. A análise dos fatores que afetam o desempenho do servidor, bem como os resultados alcançados com a implementação do modelo proposto são apresentados no Capítulo 5. Por fim, as conclusões obtidas ao longo deste trabalho são apresentadas no Capítulo 6.

2. Trabalhos relacionados

Este capítulo apresenta os trabalhos relacionados que foram estudados e que formam a base teórica e inspiração para este projeto.

Dividiremos este capítulo em quatro partes. Na primeira, mostraremos quais modelos de programação são normalmente utilizados para a obtenção de escalabilidade em servidores que lidam com múltiplas sessões de usuários concorrentes. A segunda parte apresenta uma abordagem para a paralelização de programas baseados em eventos. Em seguida, apresentaremos alguns trabalhos que almejam minimizar os problemas ocasionados pela falta de controle e de flexibilidade dos serviços prestados pelo sistema operacional. Por fim, descreveremos quais arquiteturas de *software* conhecidas são utilizadas atualmente pela indústria de jogos.

2.1. Arquiteturas de sistemas escaláveis

Definir o que é arquitetura de um sistema de computação não é uma tarefa trivial. Não há uma definição do termo "arquitetura de sistemas de computação" ou "arquitetura de *software*" que seja universalmente aceita [34]. A definição que melhor satisfaz o sentido utilizado neste trabalho foi proposta por Thomas Lane, em seu relatório técnico intitulado "*Studying Software Architecture Through Design Spaces and Rules*" [42]:

> "Arquitetura de *software* é o estudo da estrutura em larga escala e do desempenho de um sistema de computação. Aspectos importantes de uma arquitetura de

> sistema incluem a divisão das funções em módulos do sistema, os modos de comunicação entre esses módulos e a representação das informações compartilhadas".

Também não há um consenso entre os pesquisadores sobre a definição de escalabilidade [24]. Normalmente, a noção de escalabilidade é utilizada em sistemas multiprocessados e, intuitivamente, indica que um sistema de computação possui um ganho de desempenho *linear*, proporcional ao número de processadores empregados.

É correto concluir, portanto, que a forma como os componentes de *software* interagem definem a eficiência do programa em um ambiente multiprocessado, ou seja, a arquitetura do sistema de computação define se este é escalável ou não.

A seguir, discutiremos as arquiteturas mais utilizadas por aplicações que precisam lidar com múltiplas sessões de usuários concorrentes, utilizando a classificação proposta em [52]. Serão apresentadas as arquiteturas *Multi-process*, *Multi-thread*, *Single-process event-driven* e *Asymmetric multi-process event-driven*.

2.1.1. Multi-process

Na arquitetura *Multi-process*, o paralelismo é alcançado com a utilização de múltiplos processos do sistema operacional. Cada processo é responsável pela execução de todos os passos necessários para atender uma requisição realizada por um usuário. Terminada a execução de todos os passos, o processo pode atender outra requisição.

A intercalação de atividades como acesso ao disco, processamento em CPU e acesso aos recursos de rede ocorre naturalmente, uma vez que o sistema operacional elege outro processo para usar a CPU sempre que o processo ativo é bloqueado.

2.1.2. Multi-thread

Programas que utilizam a arquitetura *Multi-thread* empregam múltiplas *threads* independentes, mas que utilizam um único espaço de endereçamento de memória compartilhado. Cada

thread realiza todas as etapas do processamento de uma requisição de usuário antes de aceitar uma nova requisição de forma semelhante ao uso de processos empregada pela arquitetura *Multi-process*.

Atualmente, alguns trabalhos [65, 66] não fazem mais distinção entre as arquiteturas *Multi-process* e *Multi-thread* por considerarem que *threads* estão disponíveis em todos os sistemas operacionais modernos e que a diferença entre os processos leves (*threads*) e processos convencionais – tão marcante em sistemas operacionais antigos – é cada vez menor. No sistema operacional Linux, por exemplo, processos e *threads* são implementados da mesma forma, exceto pela questão do espaço de endereçamento de memória. Nesse sistema operacional, *threads* são processos que compartilham todas as páginas de memória com o processo que as criou.

A utilização de *threads* como única forma de obtenção de escalabilidade é fonte de diversas discussões. Em sua apresentação intitulada "*Why threads are a bad Idea*" [51], Ousterhout argumenta que *threads* devem ser utilizadas apenas quando paralelismo real é necessário. Para Ousterhout, utilizar *threads* é muito difícil, pois:

- o programador é obrigado a coordenar o acesso a regiões de memória compartilhada; tarefa complicada mesmo para programadores mais experientes;
- torna a depuração mais complicada, pois *threads* introduzem dependências temporais de modo que a ordem em que as *threads* foram escalonadas se torna decisiva para a detecção de *bugs*;
- podem impedir que o sistema seja dividido em módulos independentes ou que utilize métodos de *callback*, pois estes podem introduzir *deadlocks*;
- podem reduzir o desempenho do sistema. Granularidade fina na utilização de *locks*, em geral, reduz o desempenho e aumenta a complexidade do programa. Além disso, a sobrecarga utilizada por um grande número de *threads* (trocas de contexto, manutenção de várias pilhas de execução, escalonamento, etc.) pode comprometer o desempenho.

Por outro lado, outros trabalhos [6, 63, 64] argumentam que o uso de *threads* é a melhor forma de abstração para a criação de programas que executam operações concorrentes. O que falta, entretanto, é uma biblioteca de *threads* em nível de usuário que tenha maior controle sobre itens como o escalonamento interno das *threads*, mecanismos de sincronização mais leves e que permitam, inclusive, a criação de compiladores especializados que realizem otimizações como, por exemplo, pilhas de tamanho variável e remoção de dados desnecessários sobre o estado da *thread* da pilha antes da realização de chamadas de funções.

Para que a implementação de uma biblioteca assim seja possível, é necessário que o sistema operacional forneça para os desenvolvedores mecanismos que permitam maior flexibilização do controle de *kernel threads*.

2.1.3. Single-process event-driven

Os problemas de desempenho e limitações da arquitetura *Multi-thread* levaram os desenvolvedores a evitar o seu uso e aplicar uma abordagem baseada em eventos.

Na arquitetura *Single-process event-driven*, um único processo servidor, o despachante de eventos, permanece em um laço onde, indefinidamente, processa diferentes tipos de eventos armazenados em uma fila. Esses eventos indicam que algum tipo de processamento deve ocorrer e podem ser gerados pelo sistema operacional ou como resultado do processamento de outros eventos.

Nesta abordagem, o sistema computacional é representado como uma máquina de estados finita, onde cada estado é uma etapa do processamento de uma requisição de usuário. Dessa forma, divide-se o programa em pequenas tarefas, ou eventos, que serão processados seqüencialmente pelo despachante de eventos em seu laço principal.

Utilizando-se de chamadas assíncronas de sistema, o servidor é capaz de intercalar o processamento de eventos de diversas conexões enquanto a execução da chamada assíncrona não termina, multiplexando, dessa forma, o processamento de múltiplas requisições sem deixar a

CPU ociosa.

Teoricamente, isso deveria ser suficiente para processar múltiplas requisições no contexto de uma única *thread* de controle por CPU, evitando os custos adicionais decorrentes da utilização das arquiteturas *Multi-process* e *Multi-thread* como trocas de contexto e sincronização de *threads*.

Porém, na prática, problemas intrínsicos à implementação de algumas chamadas de sistema assíncronas em sistemas de arquivos faz com que algumas operações, supostamente assíncronas, acabem bloqueando o processo que as chamou até que a operação de E/S termine. Por esse motivo, o trabalho de Pai et al. [52] propõe a arquitetura apresentada em 2.1.4.

2.1.4. Asymmetric multi-process event-driven

A arquitetura *Asymmetric multi-process event-driven* (AMPED) combina a abordagem *Single-process event-driven* com a utilização de *threads* ou processos ajudantes.

Nesta arquitetura, quando o despachante de eventos precisa processar um evento que faz uso de uma das chamadas de sistema de acesso ao disco que, mesmo sendo chamadas assíncronas, podem bloquear, o despachante instrui uma de suas *threads* ou processos ajudantes para realizar a operação potencialmente bloqueante. Assim que a operação termina, o ajudante notifica o processo principal (através de um canal de comunicação inter-processo).

A arquitetura AMPED almeja preservar a eficiência da arquitetura baseada em eventos tradicional em operações que não envolvam acesso ao disco e evita os problemas decorrentes da má implementação de chamadas assíncronas de muitos sistemas operacionais.

De um modo geral, a utilização de arquiteturas baseadas em eventos mostra-se mais eficiente do que o uso de um grande número de *threads* para atingir o nível de concorrência necessária na multiplexação do processamento de um grande número de requisições simultâneas [52, 65].

2.2. Paralelização de programas baseados em eventos

A utilização da arquitetura *Single-process event-driven*, vista na Seção 2.1.3, mostrou-se muito interessante para a obtenção de boas taxas de utilização de CPU. Entretanto, esse modelo não é adequado para programação em ambientes multi-processados.

Uma prática comum de desenvolvedores que querem utilizar programas baseados em eventos em ambientes multiprocessados é utilizar múltiplas cópias do mesmo programa que executam independentemente entre si. Cada uma dessas cópias utiliza da melhor forma possível um dos processadores disponíveis.

Esta solução mostrou-se eficiente em alguns casos bem específicos, como, por exemplo, servidores web. Tais casos têm como ponto em comum o fato que o tratamento de eventos vindos da mesma fonte (do mesmo cliente, por exemplo) é independente do tratamento de eventos vindos de outras fontes. Essa abordagem não é adequada se o programa possui um estado global mutável que é compartilhado entre múltiplos clientes ou múltiplas requisições.

A necessidade de transformar programas que utilizam o paradigma de programação baseada em eventos em programas multi-processados levou Zeldovich et al. a proporem [69] uma biblioteca que facilitasse a paralelização desses programas.

Zeldovich observou que a maior parte do esforço realizado para transformar um programa baseado em eventos em um programa paralelo consiste em determinar quais eventos podem ser processados concorrentemente e quais devem ser mantidos seqüenciais. Com base nessa observação, desenvolveram a biblioteca *libasync-smp*, baseada na biblioteca *libasync* desenvolvida como parte da implementação de um sistema de arquivos seguro denominado SFS [49].

Um servidor que utiliza-se da *libasync-smp* é composto de um processo que contém uma *thread* auxiliar por processador disponível. Cada *thread* escolhe um evento para ser processado de uma lista de eventos disponíveis e executa o método que trata esse evento.

Para que a divisão dos eventos entre as *threads* não provoque modificações concorrentes ao estado global da aplicação, a biblioteca associa cores à cada método tratador de eventos.

A biblioteca assegura que durante a execução da aplicação dois métodos com a mesma cor não serão executados paralelamente. É responsabilidade do desenvolvedor atribuir as cores aos métodos. Se nenhuma cor for atribuída, todos os métodos terão a mesma cor e serão executados seqüencialmente.

Utilizar tal mecanismo é equivalente a utilizar um mesmo *lock* que proteja a execução de todos os métodos de tratamento de eventos de mesma cor. Os autores afirmam que mesmo que este método permita apenas uma paralelização com granularidade mais grossa do que normalmente é possível com a utilização de *locks*, ele é suficiente para obter um ganho de desempenho considerável em computadores multiprocessados.

Os resultados de seus experimentos mostram ganhos consideráveis de desempenho com pouco esforço de desenvolvimento. Para adaptar o sistema de arquivos SFS a um ambiente multiprocessado foi necessário alterar apenas 90 das 12.000 linhas de código da aplicação. Tal alteração garantiu que o SFS rodasse 2,5 vezes mais rápido em um computador com 4 processadores.

Mesmo aplicações que não possuem tarefas computacionalmente intensas conseguiram ganhos com a utilização da *libasync-smp*. A paralelização de um servidor HTTP baseado em eventos realizada pelos autores alcançou um ganho de 1,5 vezes ao utilizar-se 4 processadores.

2.3. Questões relacionadas ao sistema operacional

Um dos grandes problemas no projeto de sistemas escaláveis é a falta de controle sobre os recursos gerenciados pelos sistemas operacionais atuais.

Historicamente, os sistemas operacionais possuem como objetivo permitir que diversos processos tenham acesso a diferentes recursos computacionais de uma maneira segura e eficiente. Para isso, o sistema operacional apresenta os recursos computacionais disponíveis de forma virtualizada, onde cada aplicação acessa os recursos como se esse acesso fosse exclusivo. Por conta disso, os sistemas operacionais atuais não permitem que as aplicações influenciem as decisões tomadas no gerenciamento de recursos, nem expõem dados suficientes sobre os re-

cursos de forma a permitir que as aplicações se adaptem e mudem o comportamento. Essa virtualização esconde das aplicações o fato de que os recursos são limitados e compartilhados.

Por esse motivo, os estudos mais recentes na área de sistemas de computação escaláveis progridem junto com as pesquisas feitas na área de sistemas operacionais.

Nesta seção veremos alguns dos problemas decorrentes da virtualização de recursos do sistema operacional e algumas das soluções propostas que foram estudadas no decorrer deste trabalho.

2.3.1. Aceitação de novas conexões

A aceitação de novas requisições de usuários possui grande influência no desempenho de sistemas que atendem a um grande número de usuários simultaneamente, como mostra Brecht et al. em [12].

Para entender essa influência, precisamos lembrar que, para que um cliente possa enviar uma requisição para o servidor, é necessário que uma conexão TCP seja estabelecida. Isso é feito através do protocolo *TCP three-way handshake* [50]. Assim que o *handshake* é concluído, o sistema operacional adiciona um novo soquete à *accept queue* que, no Linux, possui no máximo 128 posições por soquete de escuta. Toda vez que a aplicação executar a chamada de sistema `accept()`, um soquete será removido do início da fila e será associado a um descritor de arquivos, que é devolvido à aplicação.

Se a aplicação aceitar as conexões em uma velocidade menor do que a da chegada de novas conexões, em algum momento a *accept queue* ficará cheia. Quando a estiver ficar cheia, todas as conexões que chegarem serão descartadas, uma vez que não haverá espaço para elas na fila. Esses descartes são um problema sério tanto para o cliente quanto para o servidor. Por um lado o cliente não poderá enviar mais requisições para o servidor e será forçado a iniciar o protocolo novamente; por outro, o servidor investiu recursos para completar o *three-way handshake* para descobrir, no fim, que a conexão deve ser descartada. Por isso os descartes de

conexões devem ser evitados sempre que possível.

O trabalho descrito em [12] mostra que não basta aceitar todas as conexões da *accept queue* de uma única vez (o que obviamente diminuiria o número de conexões perdidas). É necessário encontrar um equilíbrio entre aceitar novas conexões e processar as já existentes, de forma a aumentar a vazão (*throughput*) do servidor.

Brecht et al. concluíram que cada servidor deve possuir uma política de aceitação de recursos mais adequada para o tipo de trabalho que realiza e que não há uma regra que sirva para todos os servidores encontrarem esse equilíbrio. Mas ressaltam que com acesso a mais informações sobre a execução da aplicação seria possível implementar aplicações com políticas dinâmicas de aceitação de novas requisições.

2.3.2. Gerenciamento de threads

Na Seção 2.1.2 apresentamos diversos problemas relacionados ao uso de *threads* em aplicações escaláveis. As limitações causadas pela falta de integração entre a aplicação e as *kernel threads* são conhecidas há muito tempo pelos pesquisadores.

Em 1992, Anderson et al. mostraram [6] que apesar do desempenho de *kernel threads* ser, em geral, uma ordem de magnitude melhor do que o desempenho de processos tradicionais, é uma ordem de magnitude pior do que o desempenho de bibliotecas de *threads* em nível de usuário. Propuseram, então, um mecanismo que promove a integração entre escalonador de *threads* do *kernel* e a biblioteca de *threads* em nível de usuário. Este mecanismo foi denominado *scheduler activations*.

A proposta consiste na introdução de um mecanismo de *callback* no *kernel* que permita que a biblioteca de *threads* em nível de usuário seja notificada da ocorrência de eventos relacionados ao escalonamento das *kernel threads*. Na ocorrência desses eventos o *kernel* notificaria a biblioteca de *threads*, que por sua vez poderia tomar a atitude mais adequada em vista do estado atual da aplicação.

Por exemplo, se uma das *threads* ativas executasse uma operação bloqueante, o *kernel* notificaria a biblioteca de usuário que, por sua vez, poderia escolher qual *thread* disponível é a mais adequada para a execução no momento. Essa escolha poderia ser simplesmente baseada na prioridade das *threads* disponíveis, ou poderia utilizar dados mais sofisticados pertencentes ao domínio da aplicação, como qual das *threads* disponíveis poderia executar sem entrar em uma seção crítica cujo *lock* pertence a uma das *threads* suspensas.

2.3.3. Acompanhamento do estado das conexões

Aplicações que fazem uso intensivo de comunicação utilizam-se de operações de acesso a rede não-bloqueantes para evitar que o seu processo seja bloqueado sem precisar utilizar um grande números de *threads* de controle que potencialmente afetariam o desempenho da aplicação.

Essas operações não-bloqueantes, por sua vez, tornam necessários mecanismos para o acompanhamento das conexões que permitam que a aplicação só execute uma operação de E/S quanto tiver certeza de que esta não será bloqueada.

Os mecanismos tradicionais de acompanhamento da conexão definido pelos padrões POSIX (as operações `select()` e `poll()`) recuperam o estado da conexão do *kernel* utilizando duas trocas de contexto e duas operações de cópias de dados. A quantidade de dados copiados é diretamente proporcional ao número de conexões abertas naquele momento. Algumas propostas recentes [43, 45] tentam diminuir a quantidade de dados copiados do *kernel* para a aplicação e da aplicação para o *kernel*. Entretanto, essas novas propostas pecam pelo fato de aumentarem o número de chamadas de sistema, que prejudicam especialmente os soquetes com menor tempo de vida.

Para reduzir o número de chamadas de sistema realizadas pelas aplicações para o acompanhamento do estado de conexões, Marcel-Catalin Rosu e Daniela Rosu propuseram [54] o método *user-level connection tracking*, que consiste em utilizar uma área de memória alocada pela aplicação mas utilizada também pelo *kernel* para manter o estado de cada conexão. Dessa

maneira, a aplicação precisaria realizar um único acesso comum à memória, sem a necessidade de realizar trocas de contexto nem cópias de dados entre o *kernel* e a aplicação.

2.4. Arquiteturas utilizadas em jogos comerciais

A importância comercial de jogos massivamente multi-usuários é indiscutível. Segundo a Associação Internacional de Desenvolvedores de Jogos (IGDA [10]), em 2004 havia aproximadamente 2 milhões de assinaturas de serviços de jogos massivamente multi-usuários em todo o mundo.

Infelizmente, as empresas produtoras dos jogos de maior sucesso não divulgam detalhes da implementação de seus servidores por questões mercadológicas.

Alguns trabalhos [36, 40, 37] se propuseram a catalogar o que se conhece sobre arquiteturas de jogos massivamente multi-usuário e seus resultados serão apresentados nesta seção.

2.4.1. Componentes básicos

Praticamente todos os jogos massivamente multi-usuários são divididos em alguns componentes que, juntos, compõem os serviços presentes em um servidor de jogos comercial. São eles:

contas de usuários: é necessário um componente para armazenar as informações sobre os usuários e permitir que um jogador seja autenticado e associado a uma conta existente;

estado do mundo: um componente deve ser responsável por manter e persistir o estado atual do mundo virtual;

inteligência artificial: a simulação de jogadores não-humanos deve ser realizada por um componente do sistema que, para isso, deve implementar algum tipo de inteligência artificial;

ações do jogador: um dos componentes do sistema deve ser responsável por simular as ações realizadas pelos jogadores e seus efeitos sobre o mundo virtual;

chat: todo jogo multi-usuário possui um componente que permite que os jogadores troquem mensagens entre si durante uma sessão do jogo;

camada de rede: responsável por receber os dados dos clientes e enviar notificações de alteração do estado do mundo virtual do jogo.

2.4.2. Cliente-servidor, terminal burro

Nesta arquitetura, o programa cliente age simplesmente como um coletor de dados de entrada e visualizador do estado do jogo. Os dados de entrada (eventos de teclado, mouse, etc.) são coletados pelo cliente e enviados através da rede para o servidor – que pode ser formado por uma única máquina ou por um aglomerado (*cluster*) de máquinas.

O servidor recebe estes eventos, utiliza-os como parâmetros para calcular o próximo estado da simulação do mundo virtual do jogo e, então, envia os dados relevantes sobre o novo estado do mundo virtual a cada jogador.

Determinar o que é relevante para cada jogador é feito de diversas maneiras. Alguns sistemas determinam qual a possível área de atuação de um jogador baseados em sua localização e objetos 3D em sua volta. Outros utilizam o padrão *Publisher-Subscriber* [13], onde cada cliente informa ao servidor a lista de objetos em que está interessado e o servidor o notifica sempre que um desses objetos sofrer alguma alteração de estado.

Por fim, o programa cliente utiliza os dados enviados pelo servidor para criar uma representação do estado do mundo virtual visível por este cliente. O jogo Quake [28], apresentado na Seção 3.1, utiliza-se desta arquitetura.

2.4.3. Peer-to-peer

Jogos que utilizam a arquitetura *Peer-to-peer* (P2P) [39] utilizam um servidor centralizado apenas para armazenar os dados sobre a conta do usuário. Os clientes são responsáveis por atualizar o estado do jogo, além de ler os dados de entrada dos usuários e desenhar a representação

do mundo virtual.

Nesta arquitetura, o estado global do jogo é distribuído entre os participantes do mesmo. A simulação é realizada pelo próprio cliente que gerou o evento e seu resultado é então distribuído – em geral através de comunicação *multicast* – ou é realizada por grupos de clientes formados de acordo com a localização dos jogadores no mapa virtual do jogo, como no caso do jogo SimMud [39], desenvolvido na Universidade de Pennsylvania.

A arquitetura P2P apresenta alguns problemas que a impedem de ser adotada em larga escala. É difícil, por exemplo, impedir que jogadores mal intencionados tentem modificar o estado do jogo, já que o estado é distribuído e é responsabilidade dos clientes simulá-lo. Outro problema recorrente é a latência de comunicação. A comunicação em redes P2P tendem a possuir maior latência devido às operações de roteamento de pacotes que ocorrem nos participantes da rede.

2.4.4. Shards e servidores distribuídos

Para possibilitar um maior número de usuários, alguns jogos dividem o mapa virtual do jogo em vários *shards*[1] (ou fragmentos) e utilizam múltiplos servidores (ou aglomerados de servidores) onde cada servidor é utilizado para realizar o processamento de cada um desses fragmentos, independentemente.

A divisão em *shards* é feita de forma que um jogador em um dos *shards* interaja o mínimo possível com jogadores em outros *shards*. Por exemplo, jogadores em *shards* distintos talvez não possam se ver, mas possam trocar mensagens entre si.

Outra forma de dividir o processamento entre os servidores distribuídos é dividir os componentes do sistema entre as máquinas disponíveis, ou seja, cada máquina disponível é responsável pelo processamento de um conjunto de funcionalidades do jogo. Isto é, um servidor realiza as simulações, outro servidor é responsável por receber e retransmitir as mensagens enviadas entre os usuários, outro servidor simula a inteligência artificial de personagens não-humanos,

[1] o termo *shards* é muito usado neste contexto, logo mantivemos o termo original

etc. Essa é a arquitetura utilizada por alguns *middlewares* como o TeraZona [33].

Também há variações dessa arquitetura no que diz respeito à arquitetura de comunicação entre os servidores. O *middleware* de desenvolvimento de jogos massivamente multi-usuário Butterfly.net [14], utiliza o Globus Toolkit para compor os servidores em grades computacionais.

2.4.5. Bancos de dados

Apesar de parecer estranho em um primeiro momento, certos jogos utilizam bancos de dados para permitir que um grande número de usuários utilizem seus servidores simultaneamente.

Os gerenciadores de banco de dados mais modernos permitem que o desenvolvedor de jogos realize operações de acesso a disco de forma muito eficiente. Além disso, o uso ou não de múltiplos processadores em sistemas SMP (*Symmetric MultiProcessor*) depende apenas da licença adquirida.

Criar servidores que rodam em aglomerados também não é um problema, já que os gerenciadores de bancos de dados mais poderosos do mercado permitem a construção de aglomerados de bancos de dados e automaticamente distribuem a carga entre os computadores que formam o aglomerado.

Até onde conhecemos, essa arquitetura é aplicada apenas em jogos do tipo MMORPG (*Massively Multiplayer Online Role Playing Game*) [36, 37], onde o estado do mundo virtual deve ser sempre persistido, ao contrário de jogos de ação em primeira pessoa, onde normalmente os dados sobre o jogador são descartados no momento de sua desconexão.

Após a visão geral das arquiteturas de comunicação e de jogos multi-usuários vista nesse capítulo, estudaremos a seguir, em detalhes, a arquitetura do Quake.

3. O servidor do quake

Realizamos um estudo minucioso do modelo utilizado pela id Software [25] no código-fonte do jogo QuakeWorld [28] para que fosse possível caracterizar e propor melhorias ao seu funcionamento.

Neste capítulo descreveremos a arquitetura e modo de funcionamento da versão original do QuakeWorld, resultados de nossa análise de desempenho e caracterização do uso dos recursos do sistema operacional pela versão original. Apresentaremos, também, a proposta de paralelização e resultados do trabalho de Abdelkhalek et al [4, 5, 3].

3.1. Estudo do modelo utilizado no código-fonte do Quake

O Quake é um jogo de ação interativo, multi-usuário, desenvolvido e distribuído pela id Software [25]. Seu lançamento, em 31 de maio de 1996, foi considerado um marco na história dos jogos pois introduziu grandes avanços nos jogos de computador como, por exemplo, a utilização de modelos tri-dimensionais para representação das entidades que compõem o jogo. Até então utilizavam-se modelos bi-dimensionais e o desenho era feito utilizando projeções ortogonais. Em dezembro do mesmo ano foi lançada uma atualização com melhorias no modo de jogo multi-usuário, que proporcionou melhor desempenho em jogos via Internet. Essa nova versão foi denominada QuakeWorld. O restante do texto utiliza os termos Quake e QuakeWorld indiscriminadamente para referenciar essa nova versão.

No Quake, a comunicação entre clientes e servidores segue a conhecida arquitetura *cliente-*

servidor. Os jogos que seguem essa arquitetura delegam para o servidor a tarefa de manter a consistência da simulação do jogo e coordenar todos os clientes. Já os clientes são responsáveis pela renderização gráfica e pela interação com os usuários. Mais especificamente, um conjunto de jogadores (ou clientes) se conectam a um servidor de jogo centralizado (ou simplesmente servidor), juntam-se a uma sessão do jogo e participam até que o jogador deixe a sessão ou que esta seja terminada. Os clientes se comunicam apenas com o servidor e é responsabilidade do servidor processar as ações enviadas pelos clientes e notificá-los das ações realizadas por outros clientes.

O processamento no servidor é dividido em quadros (*frames*). Em cada quadro, o servidor evolui o modelo de simulação, processa os dados enviados pelos clientes e responde aos clientes com os dados relevantes.

O primeiro passo realizado em um quadro é verificar se há clientes que perderam a comunicação com o servidor. Após 65 segundos de inatividade, o servidor remove o jogador inativo da sessão do jogo. O próximo passo é evoluir o modelo de simulação do mundo virtual do Quake. O modelo utilizado pelo servidor pode ser dividido em duas partes: *modelo físico* e *modelo de jogo*.

O modelo físico descreve e simula a mecânica física das entidades autônomas do jogo. O código do servidor que simula essas características físicas é conhecido no meio de desenvolvimento de jogos como *game engine*. Utilizando o modelo físico, o jogo determina, a cada quadro, qual a nova velocidade, aceleração, direção e posição absoluta do jogador levando em conta se ele está andando em uma superfície plana, se está no meio de um salto (e, portanto, deve-se considerar o efeito da gravidade) ou se está na água (que além da gravidade que puxa o jogador para o fundo do lago, deve-se considerar o atrito provocado pela água). O *game engine* é considerado parte vital de um jogo pois, junto com os gráficos do cliente, é ele quem dá a sensação de realidade do jogo.

O modelo de jogo descreve como os elementos do mundo virtual se comportam no jogo, por exemplo, o que acontece quando uma bala atinge o seu alvo ou qual o próximo passo

que um "monstro" irá tomar (utilizando técnicas de inteligência artificial). O modelo de jogo é descrito em uma linguagem desenvolvida pela id Software denominada *QuakeC*. O código nessa linguagem é compilado e tem seu `bytecode` interpretado pelo servidor no momento da simulação do modelo.

O terceiro passo executado pelo servidor durante a execução de um quadro é processar as mensagens enviadas pelos clientes. Há várias mensagens relacionadas ao controle da sessão do jogo mas, tipicamente, o cliente enviará uma mensagem do tipo `MOVE`. Uma mensagem do tipo `MOVE` informa o servidor que o jogador executou um movimento. Essa é a ação mais importante e que tem mais conseqüências para a simulação do jogo. Ao receber uma mensagem `MOVE`, o servidor evolui o modelo físico da entidade que representa o personagem do jogador e determina quais outras entidades da simulação *potencialmente* serão afetadas por esse movimento.

O servidor mantém uma estrutura de dados denominada árvore *Areanode* que permite que o servidor localize rapidamente quais entidades estão próximas a um determinado jogador. Cada nó da árvore representa as entidades presentes em uma região do mapa. O filho esquerdo de um nó representa todas as entidades presentes na metade esquerda dessa região e o filho direito representa as entidades da metade direita da região. A raiz da árvore *Areanode* representa todas as entidades presentes no jogo. Note, entretanto, que apenas a localização da entidade é considerada.

O cálculo dessas regiões ocorre no momento da carga da mapa. Para a criação de cada região, o servidor escolhe um dos três eixos cartesianos (x, y ou z) e um ponto neste eixo que, juntos, definem um plano perpendicular ao eixo escolhido. A região definida à esquerda do plano será a região representada pelo filho esquerdo da raiz e a região definida à direita do plano será o filho direito da raiz. Cada uma das regiões será, então, dividida novamente até que a árvore *Areanode* tenha altura igual a 32 e seja uma árvore perfeitamente balanceada.

Uma busca na árvore *Areanode* não leva em consideração o labirinto 3D que representa o mapa, isto é, elementos do labirinto como portas e paredes não são considerados.

A Figura 3.1 mostra um exemplo simplificado de uma árvore *Areanode*. À esquerda temos a

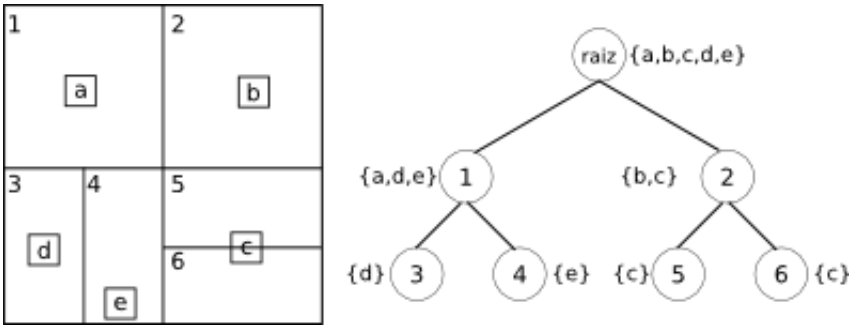


Figura 3.1.: Exemplo de regiões definidas pelo algoritmo de construção das árvores *Areanode* em um mapa e sua árvore correspondente

representação de um mapa que foi dividido em quatro regiões e as entidades presentes nessas áreas. À direita temos a árvore *Areanode* correspondente e o conjunto de entidades alcançadas por uma busca na árvore em cada nó.

Dado que uma entidade pode estar presente em mais de uma folha ao mesmo tempo, ao determinar a lista de entidades próximas ao jogador é necessário percorrer a árvore de uma das folhas para a raiz, até encontrar um nó que englobe totalmente a entidade. A lista de elementos que estão presentes nesse nó são, então, marcadas como "potencialmente modificadas".

No final do quadro, o servidor constrói e envia as mensagens com as atualizações do estado da simulação apenas para os clientes que enviaram mensagens durante o quadro, ou seja, o servidor assume que os clientes ativos estão freqüentemente enviando novos dados para o servidor.

Todas as mensagens são enviadas através do protocolo UDP (*User Datagram Protocol*) e são classificadas em mensagens *confiáveis* e *não-confiáveis*. Todo cliente deve notificar o recebimento de uma mensagem confiável no envio da próxima mensagem para o servidor. Se o servidor não receber a notificação, retransmitirá a mensagem juntamente com a nova mensagem. Mensagens confiáveis são utilizadas para notificar o cliente sobre alterações nas posições das entidades do jogo e mensagens não-confiáveis são utilizadas para atualizações de dados menos importantes, como, por exemplo, mudanças na iluminação ambiente.

Figura 3.2.: Diagrama de atividades da execução de um quadro de simulação do Quake

A construção dessas mensagens, entretanto, não é trivial. Para que as mensagens consumam pouca banda – um requisito essencial em 1996 – o servidor envia apenas os dados das entidades que foram alteradas durante o quadro e que estão na possível área de alcance do jogador (*Possible Visible Set* ou *(PVN)*). O cálculo da possível área de alcance do usuário requer o uso intensivo dos recursos de processamento, mas é realizado de forma eficiente graças ao uso de uma estrutura de dados chamada *Binary Space Tree* [56].

As árvores BSP [56] permitem representar subdivisões de um espaço n-dimensional em subespaços convexos. Árvores BSP foram muito estudadas na área de computação gráfica pois possuem aplicações interessantes como remoção de superfícies escondidas e *ray tracing*. No Quake, uma árvore BSP é utilizada para representar o labirinto 3D que forma o mapa do jogo. Com isso, os clientes podem determinar rapidamente quais áreas do mapa devem desenhar na tela de acordo com a posição do jogador e o servidor pode determinar rapidamente quais entidades estão na possível área de atuação do jogador, isto é, quais entidades não estão escondidas atrás de nenhum obstáculo.

A Figura 3.2 exibe o diagrama de atividades da versão original do servidor do Quake para a execução de um quadro de simulação.

3.2. Versão multithreaded do Quake

Ahmed Abdelkhalek et al. iniciaram um esforço para caracterização dos requisitos computacionais e paralelização do servidor do QuakeWorld.

O trabalho de Abdelkhalek pode ser dividido em duas fases. A primeira fase consiste na caracterização dos requisitos computacionais e análise de desempenho do servidor do Quake [4, 5]. Nessa etapa, a preocupação maior foi estudar como os requisitos computacionais da versão original do servidor do Quake variam com a variação do número de clientes simultâneos.

A segunda fase [3], apresenta seu esforço na paralelização do trabalho e na implementação de uma versão multithreaded do servidor do Quake.

Ambas serão discutidos nas próximas seções.

3.2.1. Caracterização da utilização de recursos

A caracterização dos requisitos computacionais do servidor do Quake foi realizada por Abdelkhalek em [4, 5]. O ambiente de testes utilizado foi composto por computadores Pentium II 400 MHz, com dois processadores dual, 512 MB de memória RAM interligados através de uma rede Ethernet de 100 Mbits/s. O sistema operacional utilizado foi o Windows NT.

Seus experimentos mostraram que:

- cerca de 830 bytes/s são trocados entre cada cliente e o servidor, independentemente do número de usuários da sessão. Uma rede Fast Ethernet de 100 Mbits/s teoricamente pode comportar mais de 1.000 usuários simultâneos;

- apesar do limite de 32 usuários por jogo, o servidor do Quake não perde desempenho mesmo com, aproximadamente, 90 usuários. Com mais usuários o processamento realizado no servidor torna-se um gargalo no sistema;

- o servidor poderia utilizar melhor o poder de processamento do Pentium II. O servidor é capaz de realizar uma média de 0,416 instruções por ciclo de processamento, enquanto

que o Pentium II permite que sejam realizadas até 4 instruções por ciclo. Isso indica que a CPU fica ociosa na maior parte do tempo e mostra um potencial maior ainda se considerarmos processadores mais modernos.

3.2.2. Proposta de paralelização

A percepção de que o processamento, e não o consumo de banda de rede, é o principal gargalo do sistema levou Abdelkhalek a desenvolver uma proposta de paralelização para o servidor do Quake [3].

Sua proposta consiste de uma versão *multithreaded* do servidor do Quake com atribuição estática de tarefas. Na inicialização do servidor são criadas tantas *threads* quanto processadores disponíveis. Os clientes são associados às *threads* no início da execução, manualmente. Cada *thread* utiliza uma porta UDP própria, por onde os clientes associados a ela se comunicam.

Todo o processamento realizado para a simulação de um mesmo cliente é feito sempre pela mesma *thread*. Os autores afirmam que é possível aplicar políticas dinâmicas de atribuição de clientes a *threads* e sugerem que uma política que levasse em conta a localização dos jogadores seria interessante. Essa idéia foi tratada como um trabalho futuro, pois seriam necessárias alterações mais intrusivas no código do servidor. Nosso modelo de paralelismo, apresentado no capítulo 4 mostra que é possível implementar distribuição dinâmica de tarefas sem a necessidade de alterações tão intrusivas.

3.2.3. Paralelização de um quadro do servidor

A fim de manter a corretude da simulação realizada pelo servidor, dois invariantes foram impostos pelos autores:

- cada fase do processamento do servidor é distinta e não deve se sobrepor às outras;
- cada fase deve ser executada em sua ordem original: simulação do modelo físico, processamento da requisição e processamento da mensagem de resposta.

Dessa forma, as mesmas etapas de processamento da VERSÃO-ORIGINAL – descritas na Seção 3.1 – são mantidas nesta proposta de paralelização.

No início da execução do servidor, todas as *threads* são criadas e ficam suspensas em uma chamada à `select()`. Assim que a primeira mensagem de um cliente chega ao servidor, a primeira *thread* a receber uma mensagem retorna da chamada à `select()` e é eleita como coordenadora deste quadro de simulação.

A *thread* eleita como coordenadora é responsável por executar todas as etapas seqüenciais do servidor. Em particular, é ela que realiza a etapa de simulação física do mundo assim que um novo quadro de simulação é iniciado. As *threads* que possivelmente retornaram da chamada à `select()` durante a simulação do modelo físico ficam suspensas em uma barreira de sincronização.

Terminada a simulação do modelo físico, a *thread* coordenadora notifica as demais *threads* para prosseguirem com a execução. Após a notificação, se houver alguma *thread* que ainda está suspensa em uma chamada à `select()`, ou seja, que não possui nenhuma mensagem em sua fila de mensagens, então esta não será executada neste quadro e permanecerá suspensa na barreira de sincronização até o próximo quadro de simulação.

A próxima etapa é o processamento das mensagens enviadas. Durante essa etapa, cada *thread* entra em um laço responsável por consumir e processar todas as mensagens contidas em seu soquete de mensagens. A *thread* fica neste laço até que todas as mensagens tenham sido processadas. Terminado o laço, todas as *threads* esperam em uma barreira de sincronização antes de iniciarem a próxima fase.

A última fase é a fase de processamento de respostas. Cada *thread* computa e envia uma mensagem de resposta para todos os clientes que tiveram uma mensagem processada neste quadro de simulação. Assim que cada *thread* termina de enviar as mensagens uma chamada à `select()` é realizada e o quadro de simulação é terminado. A *thread* coordenadora envia um sinal para as *threads* que não participaram do quadro para indicar o término do quadro.

3.2.3.1. Sincronização

Dado que as fases de simulação são separadas por barreiras de sincronização, o servidor só precisa realizar sincronizações intra-fases para acessar estruturas de dados globais. Apesar da etapa de simulação do modelo físico manipular estruturas de dados globais, esta é realizada apenas pela *thread* coordenadora de forma seqüencial e, por isso, não requer sincronização intra-fase.

A etapa de criação e envio de respostas não apresenta nenhum conflito de leitura e escrita nas estruturas de dados compartilhadas. O processo de criação das mensagem de resposta envolve a leitura dos dados globais. Mas só há escrita nas estruturas de dados locais da *thread*. Como todas as *threads* só realizam a leitura das estruturas de dados globais, não há necessidade de sincronização nesta etapa.

A etapa de processamento das mensagens enviadas pelos clientes, entretanto, é mais complicada. A simulação dos comandos enviados nas mensagens envolvem a atualização das características das entidades e, por isso, deve realizar algum mecanismo de sincronização.

O mecanismo proposto utiliza a semântica da árvore *Areanode*, uma das estruturas de dados globais atualizadas nesta etapa. A árvore *Areanode* é descrita com mais detalhes na Seção 3.1.

A sincronização das entidades do jogo consiste em proteger com um *lock* o nó da árvore que contém todas as entidades relevantes para a atualização da estrutura de dados. Antes de atualizar uma entidade, o servidor tenta colocar um *lock* na folha da árvore que contém o elemento. Todos os nós são percorridos até que seja encontrado um ancestral que contenha todos os objetos que potencialmente serão afetados pela atualização. Assim que encontrado, esse ancestral também é protegido com um *lock*.

Os resultados dos experimentos – descritos na Seção 3.2.3.2 – mostraram que o maior gargalo da Versão-Multithreaded proposta por Abdelkhalek é justamente o tempo gasto com a sincronização. Ao utilizar oito *threads*, o tempo gasto com sincronização chega à 70% do tempo total. 30% do tempo total foi gasto com obtenção de *locks* e 40% com barreiras de

sincronização global.

Abdelkhalek aprimorou a técnica e desenvolveu uma nova estratégia para a utilização de *locks*. A otimização consiste em utilizar a semântica dos objetos para a realização dos *locks*.

Os objetos do servidor foram classificados em dois tipos distintos, de acordo com o impacto que a simulação de cada objeto exerce sobre outros objetos do servidor. Objetos como projéteis, por exemplo, que são criados no momento do processamento da mensagem, mas que serão simulados somente na etapa de simulação física, afetarão apenas clientes que estão muito próximos do jogador que disparou o projétil e, portanto, a manipulação de seus dados pode ser feita em segurança colocando o *lock* em uma área bem menor do mapa. Objetos como jogadores, que podem afetar outros objetos em uma área bem maior durante a etapa de processamento da mensagem, precisam de um *lock* de uma área maior. Tal otimização reduziu o tempo gasto com obtenção de *locks* de 30% para 20% do tempo total da simulação.

3.2.3.2. Resultados

Os novos testes foram realizados em um computador Pentium III 1.4 GHz com quatro processadores, todos com *Hyper-threading* habilitados.

A paralelização da etapa de criação de mensagens de resposta escala muito bem para diferentes números de jogadores simultâneos. O recebimento das mensagem, parte da etapa de recebimento e processamento de mensagens, também escala bem com o aumento do número de jogadores. Isso ocorre porque nessas duas etapas não há escrita em estruturas de dados globais.

Apesar da carga total de processamento do servidor ser bem balanceada entre as *threads*, a carga de cada quadro de simulação, independentemente, mostrou-se desbalanceada. Por isso o tempo de espera em barreiras de sincronização pode chegar a 40% do tempo total da execução.

Além disso, o tempo com sincronização para atualização de estruturas de dados globais pode chegar a até 35% do tempo total. Com a otimização que leva em consideração a semântica dos objetos, esse tempo pode ser reduzido para 20% do tempo total.

Com relação ao número de jogadores, a versão paralela consegue simular 25% mais jogadores do que a versão seqüencial usando 4 *threads*, mesmo com o aumento super-linear causado pelo aumento da interação entre os jogadores. O servidor fica saturado com 128 jogadores ao utilizar 2 *threads*, com 144 ao utilizar 4 *threads* e com 160 ao utilizar 8 *threads*.

3.3. Experimentos com a versão original

Simultaneamente ao estudo do código-fonte do Quake, cujo funcionamento foi apresentado na Seção 3.1, realizamos diversos experimentos a fim de estender os resultados obtidos por Abdelkhalek e Bilas em [4].

3.3.1. Objetivos

Em seu primeiro trabalho sobre a utilização de recursos do Quake [4], Abdelkhalek caracterizou a utilização de recursos computacionais exigidos pelo servidor do Quake em termos de tamanho das mensagens trocadas entre clientes e servidor (para a caracterização de uso da banda de rede), utilização da CPU (o que incluiu caracterizar o tempo para processar as mensagens dos clientes, tempo para processar as mensagens de resposta e ociosidade do processador) e dados internos do processador, tais como predição de desvios (*branch predictions*), ausências no *cache* (*cache misses*), busca de instruções (*instruction fetches*) e conflito de recursos (*resource conflicts*).

Nosso objetivo com a investigação preliminar apresentada nesta seção é caracterizar o comportamento do servidor em termos do uso de CPU para a realização das tarefas desempenhadas pelo servidor e, também, identificar quais recursos do sistema operacional são necessários e qual o impacto da utilização desses recursos no desempenho do servidor.

3.3.2. Ambiente de testes

Todos os nossos testes foram realizados em 6 computadores com processador AMD Athlon^{TM} XP 1700+ (1,47 GHz), 1 GB de memória RAM, conectados através de uma rede Ethernet dedicada de 100 Mbit/s. Os testes foram realizados em máquinas dedicadas, utilizando-se o sistema operacional Linux, versão 2.6.9.

3.3.3. Ferramentas utilizadas

3.3.3.1. GNU Profiler

Para a caracterização das tarefas realizadas pelo servidor do ponto de vista funcional utilizamos a ferramenta *GNU Profiler*, ou simplesmente `gprof` [20].

O `gprof` é uma ferramenta utilizada para medir o desempenho de um programa (*profiling*). Com ele é possível medir o tempo gasto na execução de cada função do programa. Com o `gprof`, o programador pode detectar problemas de desempenho (uma função que demora mais do que deveria) e até candidatos a possíveis otimizações no código. Entretanto, deve-se notar que o `gprof` só produz informações sobre o tempo em que o processo efetivamente ocupa a CPU. Ele é incapaz de apontar problemas de desempenho relacionados a operações que bloqueiam o processo, onde o tempo é gasto em espaço de *kernel*.

Outra funcionalidade muito interessante do `gprof` é a construção, durante a execução do programa, do grafo de chamadas de funções. Dada uma função, o grafo de chamadas de funções permite que o programador descubra quais funções se utilizam dela e, junto com as informações de consumo de tempo das funções produzidas pelo `gprof`, qual o impacto da utilização dessa função na execução das outras funções.

A utilização do `gprof` é simples. Primeiramente, deve-se incluir a biblioteca do *gprof* durante a link-edição do programa. Após a execução do mesmo, será gerado um arquivo binário com os resultados das medições. A ferramenta `gprof` lê o arquivo gerado e exibe um relatório contendo o "perfil plano", que contém os dados sobre o tempo total gasto e número de chamadas feitas

por cada função, e o grafo de chamadas das funções utilizadas no programa.

O `gprof` foi imprescindível tanto para a caracterização do desempenho do servidor, quanto para o entendimento do código do mesmo. A versão utilizada do `gprof` em nossos testes foi a 2.15.

3.3.3.2. Linux Trace Toolkit

O *Linux Trace Toolkit* [68,30] (LTT) é um conjunto de ferramentas e uma adição ao *kernel* do Linux que permite acompanhar todos os eventos gerados no sistema operacional durante a execução de qualquer programa.

O LTT oferece informações detalhadas sobre o momento da ocorrência e duração de eventos como, por exemplo, chamada e retorno de uma chamada de sistema, início e fim de tratamento de interrupções, mudanças geradas pelo escalonador de processos, eventos de manipulação de processos (`wait()`, `fork()`, `exit()`, etc.), operações de E/S em sistemas de arquivos e em soquetes de comunicação e muitos outros eventos. De acordo com os testes de desempenho realizados por seus autores em [68], sua utilização implica em um pequeno custo computacional adicional de, aproximadamente, 2,5% no desempenho do sistema.

Como os eventos são capturados em espaço de *kernel*, não é necessário recompilar o programa a ser analisado. É preciso, apenas, montar um sistema de arquivos chamado *RelayFS*, utilizado pelo LTT para transferir grandes quantidades de dados do espaço de *kernel* para espaço de usuário de forma eficiente.

Um programa auxiliar, o *tracedaemon*, lê todos os dados armazenados no *RelayFS* a partir do momento de sua execução e gera um arquivo binário contendo todos os dados sobre os eventos gerados pelo *kernel* no fim de sua execução. O pacote do LTT também contém um programa chamado *TraceViewer* que exibe graficamente a lista de eventos gerada, imprescindível para a visualização da grande de quantidade de dados coletados durante a execução do *tracedaemon*.

Graças ao LTT, a caracterização do servidor do Quake do ponto de vista do consumo de recursos do sistema operacional foi possível.

3.3.4. Experimentação

O primeiro passo do nosso trabalho de experimentação com o servidor do Quake foi automatizar a execução de seus clientes.

Para que isso fosse possível, modificamos o código-fonte do Quake para que a entrada do usuário fosse dada de forma automatizada. Utilizando-se da estrutura fornecida pelo próprio Quake, adicionamos ao programa eventos de movimentação e de ação gerados de forma pseudo-aleatória. Seu funcionamento é simples. A cada dez quadros de execução do cliente, é enviado para o servidor um evento de início de movimentação. Após cinco quadros do início do movimento, o cliente envia um evento de ação, que pode ser um evento informando o salto do jogador, um tiro ou a mudança de direção do movimento do usuário. Após mais cinco quadros, com o envio do evento de término de movimentação pelo cliente, o servidor termina a simulação de movimento do usuário e o cliente dá início a um novo ciclo.

Tomamos cuidado para que a taxa de envio de eventos simulasse adequadamente a taxa de envio de um jogador humano, de modo que a carga sofrida pelo servidor correspondesse a carga de um servidor em uma situação normal de utilização. Os intervalos de tempo entre o envio dos eventos foram determinados a partir da observação de um jogo real e acreditamos que seja uma boa aproximação, adequada para nossos requisitos iniciais.

Para permitir a utilização de múltiplos clientes em um mesmo computador, modificamos o cliente para que fosse eliminada a etapa de desenho do estado atual do jogo na tela. Ao invés de realizar as operações de desenho, o cliente executa um comando `sleep()`, de forma que código simule um cliente que consiga atingir uma taxa de execução de trinta quadros por segundo, número máximo permitido pelo código de desenho e de simulação do Quake.

O servidor do Quake possui uma limitação no número máximo de usuários que podem participar de uma sessão do jogo. Por padrão, essa limitação é de 16 jogadores por sessão, mas esse número pode ser elevado para 32 sem nenhuma modificação no código-fonte do programa.

Alteramos o código-fonte do servidor para que fosse possível remover essa limitação no

número de usuários. Para isso, foi necessário muito estudo do código-fonte disponibilizado e o entendimento de diversas sutilezas que estão presentes neste domínio de aplicação.

Apesar da aparente modularização do código-fonte, o servidor do Quake foi otimizado para que fosse o mais eficiente possível em ambientes monoprocessados e mono-tarefas – o sistema operacional alvo para a execução do Quake, em 1996, era o MS-DOS. Por isso, em prol da eficiência, optou-se por quebrar o encapsulamento entre seus módulos. Muitas vezes os dados necessários para a realização de uma tarefa estão distribuídos entre os módulos, de forma que cria-se uma dependência na ordem de utilização dos módulos.

Fatores como inversão de controle durante a execução de *scripts* em QuakeC, inter-dependência implícita entre constantes definidas em diferentes módulos e, principalmente, ausência de documentação no código-fonte tornaram a alteração e entendimento do Quake um grande desafio.

Além disso, foi necessário entender algumas características da aplicação que não se refletem diretamente no código. Os mapas utilizados para representar o labirinto 3D, por exemplo, influem de forma determinante no desempenho do servidor. Mapas compostos por pequenas salas ou por diversos corredores induzem a formação de mensagens de resposta pequenas, porém a etapa de construção de mensagens para os clientes torna-se mais lenta. Os mapas possuem, também, elementos próprios de simulação, como alavancas ou plataformas móveis, que influenciam a etapa de simulação da mesma forma que um novo jogador e que podem aumentar consideravelmente o tamanho das mensagens se o mapa apresentar um grande número desses elementos. Em todos os testes desta etapa, utilizamos um mapa desenvolvido e disponibilizado publicamente pela mesma empresa que desenvolveu o Quake, denominado *death32c* [27].

Sucintamente, as alterações necessárias envolveram o aumento do número máximo de entidades que podem ser simuladas pelo servidor (de 768 para 1536), aumento do tamanho máximo das mensagens trocadas entre os clientes e o servidor (de 1450 bytes para 7250 bytes), extensão do mecanismo de envio de mensagens *confiáveis*, para que o aumento do tamanho das mensagens pudesse ser aproveitado para a retransmissão de mensagens confiáveis e, finalmente, a

elevação do número máximo de clientes simultâneos permitidos pelo servidor.

Após essas alterações, conseguimos elevar o número de usuários simultâneos de 32 para 160. Números maiores do que 160 induzem a desconexões prematuras dos clientes devido à limitações no protocolo de troca de mensagens implementado pelo Quake. Denominamos essa versão do código do Quake de VERSÃO-ORIGINAL-ESTENDIDA.

Analisamos o desempenho e funcionamento das duas versões do servidor: VERSÃO-ORIGINAL, assim como distribuída pela id Software, e a VERSÃO-ORIGINAL-ESTENDIDA, modificada para comportar 160 usuários. O resultado dessa análise será apresentado a seguir.

3.3.5. Resultados obtidos

Nossos resultados[1] corroboram os resultados obtidos por Abdelkhalek em [4].

Com os dados do GNU *Profiler* verificamos que no servidor da VERSÃO-ORIGINAL, com limite de 32 usuários simultâneos, 8,85% do tempo é gasto com a simulação do modelo físico do jogo. 52,15% do processamento é utilizado durante o processamento de mensagens enviadas pelos clientes e 34% do tempo é utilizado para a construção das mensagens de resposta para os clientes. O restante do processamento, cerca de 5%, é utilizado em tarefas como *log* e verificação da atividade dos clientes.

Com o servidor da VERSÃO-ORIGINAL-ESTENDIDA, com 160 usuários simultâneos, 17,5% do tempo foi gasto com a simulação do modelo físico, 53,5% do tempo foi utilizado para o processamento de mensagens dos clientes, 27% para a construção das mensagens de resposta para os clientes e os outros 2% foram utilizados pelas outras tarefas.

Os resultados das medições feitas com o auxílio do *Linux Trace Toolkit* mostram claramente que o servidor do Quake fica ocioso a maior parte do tempo. O servidor realiza computação durante cerca de 40 segundos, dos 3 minutos de duração do teste. No restante do tempo o processo fica esperando por novas mensagens de usuário, suspenso devido a uma chamada de `select()`. Isso nos mostra que Abdelkhalek estava certo ao concluir que o servidor do Quake

[1] valores médios obtidos após 10 simulações

possui grande potencial para melhorias.

Os resultados com o servidor da Versão-Original-Estendida indicam que o processo utilizou a CPU por cerca de 90% do tempo. Praticamente todo o restante do tempo foi consumido pelas chamadas de sistema `select()` e `socketcall()`. A primeira é utilizada no acompanhamento das conexões para a realização de chamadas de E/S assíncronas. A segunda é a operação que efetivamente realiza o envio e leitura de dados na rede.

A quantidade de chamadas de sistemas utilizadas é pequena. Além das chamadas de sistema `select()` e `socketcall()`, apenas a `gettimeofday()` é utilizada com freqüência. A chamada `gettimeofday()` devolve a hora do sistema, com precisão de milisegundos, e é utilizada durante a simulação do modelo físico

Também é notável o fato de que não há ocorrências de falhas de página de memória, indicando um baixo consumo de memória RAM, como de fato foi constatado em [4].

Os resultados dos experimentos com a Versão-Original, juntamente com as conclusões sobre a caracterização e sobre a proposta de paralelização de Abdelkhalek orientaram as decisões tomadas para a criação de um novo modelo de paralelismo para o servidor do Quake, que será apresentado no próximo capítulo.

4. Metodologia de paralelização

Neste capítulo apresentamos a proposta e implementação de um modelo de paralelização para o servidor do Quake que utiliza-se da semântica do jogo para realizar balanceamento de carga dinamicamente.

Utilizamos o modelo de programação BSP [59] aliado a uma extensão da arquitetura baseada em eventos para ambientes multiprocessados para obter um modelo de paralelização que necessita de apenas um ponto de sincronização entre os processadores. Neste modelo não é necessário, portanto, sincronizar o acesso à memória compartilhada durante a etapa paralela de processamento.

No final do capítulo, apresentamos detalhes sobre a implementação do modelo proposto e uma adaptação que permite a execução eficiente em sistemas operacionais que utilizam o conceito de afinidade entre processos e processadores para realizar o escalonamento dos processos.

O código-fonte da versão paralela do servidor está publicamente disponível na Internet, no endereço `http://grenoble.ime.usp.br/~gold/orientados/`, sob a licença GPL [29].

4.1. Motivação

4.1.1. Análise da versão multithreaded do Quake

A versão *multithreaded* do servidor do Quake implementada por Abdelkhalek [3] (Versão-Multithreaded) e descrita na Seção 3.2 apresentou graves problemas de desempenho em

seu estágio de desenvolvimento inicial. Tais problemas foram ocasionados, principalmente, por problemas de contenção de *locks*.

A estratégia utilizada por Abdelkhalek para a divisão de tarefas consiste em atribuir cada jogador a uma *thread* no momento em que o jogador se conecta ao servidor. Cada *thread* é selecionada através de um escalonamento do tipo *round-robin*. A partir daí, toda computação necessária para a evolução do modelo físico e do modelo de jogo é realizada por essa *thread*.

Essa distribuição de tarefas não leva em consideração o modelo de sincronização de *threads* utilizado. Apesar da sincronização *inter-thread* ser realizada utilizando-se as informações contidas na árvore *Areanode* e, conseqüentemente, levar em conta as informações sobre o posicionamento das entidades no mapa virtual do jogo, a distribuição de tarefas não é atualizada para refletir as modificações de posicionamento das entidades. Isso causa um desbalanceamento de carga e eleva o problema das contenções de *locks*.

Em testes realizados pelos autores em uma máquina SMP com quatro processadores (onde todos tinham a funcionalidade *Hyper-Threading* ativada) a versão *multithreaded* com 8 *threads* gastava 35% do tempo com a sobrecarga devida à contenção de *locks*. Em 40% do tempo, as *threads* ficavam suspensas esperando que outras *threads* terminassem alguma tarefa antes de poderem continuar.

Nota-se que além da falta de investigação mais aprofundada sobre o problema de balanceamento de carga, houve uma falha na definição de uma política eficiente de utilização de *threads*. O artigo analisa o desempenho do servidor quando é utilizada uma *thread* por cliente, por exemplo, o que vai contra os resultados da área de escalabilidade de sistemas como os apresentados na Seção 2.1.

Além disso, o problema de contenção de *locks* causou um grande impacto no desempenho da versão *multithreaded* do servidor. A análise do problema fez com que os autores percebessem que utilizar a semântica das entidades do jogo para a obtenção dos *locks* possuía um grande potencial para melhorar esses problemas.

De fato, ao criarem uma política diferenciada [3] para proteger as folhas da árvore *Areanode*

que levava em conta o potencial de alteração do estado da simulação durante o processamento de determinadas entidades, o tempo com contenção de *locks* baixou de 35% para 20% do tempo total do quadro do servidor.

Por estas razões, acreditamos que uma proposta para um novo modelo de paralelismo para o servidor do Quake deve:

- levar em conta a semântica da simulação para distribuição da carga;
- considerar o problema de contenção de *locks*;
- fazer um mapeamento mais preciso entre a abstração de múltiplos processos utilizada (*threads* ou outros processos criados via `fork()`) e os processadores disponíveis.

4.1.2. Resultados dos experimentos iniciais

A análise dos resultados dos experimentos com a versão seqüencial do Quake apresentados na Seção 3.3 permitiu-nos concluir que durante a execução do servidor do Quake há poucas intervenções realizadas pelo sistema operacional.

A Versão-Original-Modificada gasta aproximadamente 10% do tempo em chamadas de sistemas do sistema operacional. O restante do tempo é utilizado exclusivamente pela CPU. Dado o baixo consumo de memória do servidor, não há falhas de páginas de memória.

A utilização eficiente da CPU decorre do fato da implementação do servidor do Quake seguir o modelo *single process event-driven* que descrevemos na Seção 2.1.3. Graças ao uso de chamadas assíncronas para leitura e escrita nos soquetes de rede, o servidor garante a utilização quase que ininterrupta da CPU da máquina que o executa.

Desse modo, acreditamos que o método de paralelização para programas baseados em eventos proposto por Zeldovich et al. descrito na Seção 2.2 é adequado para a implementação de uma versão paralela do servidor do Quake.

Além disso, a análise do consumo de CPU da versão seqüencial modificada para aceitar até 160 clientes mostrou que as etapas de processamento das mensagens recebidas e construção

e envio de respostas para os clientes são boas candidatas à paralelização. Juntas, respondem por cerca de 80% do tempo total de utilização da CPU.

A implementação que descreveremos a seguir utiliza-se desta análise do trabalho desenvolvido por Abdelkhalek. Sua observação sobre como a utilização da semântica do jogo pode ser importante para o desenvolvimento de um método de paralelização eficiente é utilizada como base para uma nova proposta para paralelização e para distribuição dinâmica de carga de processamento.

4.2. Metodologia

4.2.1. Visão geral

Conforme visto na Seção 2.2, Zeldovich et al. observaram que grande parte do esforço realizado para transformar um programa baseado em eventos em um programa paralelo consiste em determinar quais eventos podem ser processados concorrentemente.

No caso do servidor do Quake, a natureza da simulação executada pelo servidor define quais eventos podem ser tratados de forma concorrente e quais devem ser tratados seqüencialmente. Em geral, para executar a simulação de uma mesma entidade do jogo, a ordem de tratamento dos eventos deve ser seqüencial e deve respeitar a ordem previamente definida pela versão seqüencial do servidor do Quake. Entretanto, se a simulação de uma entidade não interferir na simulação de outra entidade, então a execução dos eventos das mesmas podem ser multiplexadas pelo servidor sem alteração nos efeitos da simulação do servidor.

Tendo isso como base, a versão paralela do servidor – denominada doravante como VersãoParalela foi dividida em quatro fases:

1. simulação do modelo físico e tarefas administrativas;
2. distribuição dinâmica de tarefas;
3. tratamento paralelo da fila de eventos do servidor;

4. sincronização global entre os processadores.

As fases 1 e 2 são executadas por um processo coordenador em um momento em que não há paralelismo no servidor. A terceira fase é executada concorrentemente e a última corresponde a uma barreira de sincronização. Esta representa o fim da execução paralela de um quadro do servidor.

O modelo de paralelização desenvolvido para o servidor do Quake segue o modelo *Bulk Synchronous Parallel* (BSP) [59] de computação paralela. Cada quadro de simulação é um super-passo, onde as fases 1 e 2 formam a *input phase* do modelo BSP. A computação local é realizada durante a terceira fase. Por fim, a *output phase* é realizada pela quarta fase da versão paralela.

4.2.2. Simulação do modelo físico

A primeira fase, simulação do modelo físico e distribuição de tarefas, é realizada por um processo coordenador e é executada em uma fase em que não há paralelismo no servidor. O modelo físico é o momento em que o código do servidor executa o interpretador da linguagem QuakeC para realizar a simulação física de todos os elementos da simulação, inclusive do próprio mapa virtual do jogo[1].

A inversão de controle na execução do jogo decorrente da interpretação de código externo ao código do servidor torna difícil a paralelização desse estágio do processamento. A paralelização desta etapa não foi investigada em um primeiro momento por corresponder à apenas 17,5% do tempo total da execução da versão seqüencial do servidor modificada para comportar 160 clientes.

Nesta etapa seqüencial da execução do servidor do Quake também são realizadas algumas tarefas administrativas para manutenção da sessão do jogo. Dentre as tarefas, podemos destacar:

[1] associado ao mapa do jogo pode haver lógica de simulação que determina os objetivos daquela fase do jogo

- detecção e remoção automática de clientes inativos;
- manutenção dos *logs* gerados pela simulação;
- execução de comandos de administração digitados no console de administração do servidor (envio de mensagens aos jogadores, término de uma sessão, alteração do mapa utilizado no jogo, etc.).

4.2.3. Distribuição dinâmica de tarefas

Criamos uma estratégia para realizar a distribuição dinâmica de tarefas que utiliza o mecanismo de atualização de estado dos jogadores do servidor do Quake. Esta estratégia utiliza a posição de todas as entidades relevantes ao processamento de um dado quadro do servidor como critério para o balanceamento de carga entre as CPUs disponíveis.

A estratégia consiste em detectar em cada quadro de simulação do jogo todos os grupos formados por entidades que estão próximas entre si e que, portanto, possivelmente podem interagir entre si. São considerados apenas os elementos relevantes à este quadro de simulação. Ou seja, serão considerados para inclusão nesses grupos apenas os elementos que realizarem alguma ação e aqueles que possivelmente serão afetados por alguma ação neste quadro de simulação.

Dois elementos α e β estão próximos entre si se α está na área de atuação de β. Geometricamente, a área de atuação de uma entidade é definida pelo servidor do Quake como sendo o espaço delimitado por um cubo centrado nas coordenadas da entidade com aresta de tamanho 256 unidades de distância. Se as coordenadas de β estão contidas dentro do espaço delimitado pelo cubo centrado nas coordenadas de α, então α e β estão próximos entre si. Note que o conceito de proximidade aqui definido desconsidera elementos definidos pelo cenário, como paredes e portas.

Definimos um grupo de entidades próximas como sendo o conjunto que contém os elementos representados pelos vértices de cada componente conexo do grafo $G = (V, E)$, onde cada

elemento de V representa uma entidade relevante ao quadro e cada aresta (α, β) em E indica que as entidades α e β estão próximas entre si.

Uma propriedade interessante que decorre da lógica de simulação do servidor é que se a execução de uma ação de uma entidade α produz uma alteração no estado de outra entidade β, então α e β estão em um mesmo grupo de entidades próximas.

Dessa forma, foi possível criar um método de paralelização onde não é necessário nenhum tipo de sincronização para o acesso de leitura e de escrita aos atributos das entidades. Basta que todos os elementos de um mesmo grupo sejam simulados por um mesmo processador. Apenas as modificações às estruturas de dados globais, como, à árvore *Areanode*, por exemplo, devem ser sincronizadas.

O balanceamento de carga entre os processadores é realizado utilizando-se o algoritmo de escalonamento de tarefas denominado *Longest Processing Time First* (LPT) [23, 38]. O escalonamento de n tarefas em $m \geq 2$ processadores é conhecidamente um problema NP-difícil. Por isso, optamos pela utilização de uma heurística para a realização do escalonamento das tarefas nos processadores disponíveis.

O algoritmo LPT ordena as tarefas em ordem decrescente de tamanho e aloca as tarefas, em ordem, o quanto antes no processador que estiver menos sobrecarregado de tarefas. Graham provou [23] que o tempo para o término da última tarefa escalonada através da heurística LPT é, no máximo, $\frac{4}{3} - \frac{1}{3m}$ vezes pior do que o tempo para o término da última tarefa em um escalonamento ótimo.

Coffman et al. [38] estenderam o resultado de Graham ao mostrar que para um número grande de tarefas o escalonamento obtido pelo LPT produz um resultado melhor do que o indicado pelo limitante superior. Se o processador que executa a última tarefa a ser concluída executar outras $k - 1$ tarefas além desta, então o escalonamento obtido através do LPT é, na verdade, $\frac{k+1}{k} - \frac{1}{km}$ vezes pior, se $k \geq 3$. Esse resultado, informalmente, diz que se cada processador executar pelo menos k tarefas, então a diferença entre o escalonamento gerado pelo LPT e o ótimo será de, no máximo, $\frac{1}{k}$ do escalonamento ótimo. Portanto, o LPT é uma

opção adequada para o escalonamento de um grande número de grupos de entidades próximas.

No contexto do servidor do Quake, quanto mais entidades estiverem próximas a um jogador, mais custoso será o tratamento das mensagens enviadas por ele e maior será a quantidade de informações que serão enviadas nas mensagens de resposta. Por esse motivo, definimos como tarefa a simulação de todos os eventos de um mesmo grupo e definimos que o tamanho de uma tarefa é o número de elementos de cada grupo.

Entretanto, há que se notar que em certos casos essa definição de tarefa do LPT pode levar a um desbalanceamento de carga nos processadores, mesmo que a divisão dos grupos por seu tamanho seja balanceada. Isso pode acontecer caso haja algum grupo com tamanho muito grande, mas que contenha poucos jogadores ativos em um dado quadro de simulação. Nesse caso, o escalonador de tarefas irá alocar o maior grupo em um processador e os grupos menores em outro processador. Isso pode fazer com que o último realize mais processamento do que o primeiro. Um exemplo de desbalanceamento será mostrado na Seção 5.3.2, onde mostraremos como diferentes cenários podem influenciar o processamento das tarefas criadas.

Divididas as tarefas entre os processadores, o próximo passo da versão paralela do servidor do Quake é o tratamento da fila de eventos do servidor.

4.2.4. Tratamento paralelo de eventos

O tratamento dos eventos é a fase do servidor que é efetivamente paralela. Compreende tanto o tratamento das novas mensagens enviadas ao servidor, quanto a criação e envio das mensagens de resposta para os clientes que estavam ativos nesse quadro, etapas que correspondiam a cerca de 80% do tempo utilizado pela CPU na versão do servidor modificada para 160 clientes.

Após distribuir os grupos de entidades próximas utilizando o algoritmo proposto em 4.2.3, o processo pai ativa todos os processos auxiliares – um processo por CPU disponível – e estes começam a simulação de seus respectivos clientes.

É interessante notar que cada processo auxiliar age como se fosse uma instância independente do servidor do Quake, onde os únicos clientes ativos no atual quadro de simulação são os clientes contidos nos grupos que foram atribuídos à este processo. Portanto, cada processo auxiliar funciona como um servidor *single-process event-driven* independente.

Dessa forma, não há necessidade de realizarmos nenhuma sincronização entre os processos durante toda essa etapa do servidor. Com isso, conseguimos evitar o problema de contenção de *locks* apresentado na implementação de Abdelkhalek. Por outro lado, nesta abordagem há a necessidade de se transmitir as modificações para o processo coordenador. A sincronização desses dados é a próxima fase da execução do servidor.

4.2.5. Sincronização inter-processos

Após o término do tratamento de eventos, cada processo auxiliar inicia uma fase de troca de informações com o processo coordenador para que este tenha a chance de absorver as mudanças ocorridas com cada cliente e atualizar o estado global da simulação.

O processo coordenador precisa obter dos processos auxiliares as novas informações sobre as variáveis específicas da simulação do jogo, sua nova posição e, então, recalcular a nova posição do jogador no mapa virtual e atualizar a árvore *Areanode* para que esta reflita essa nova posição.

Assim que todos os processos auxiliares terminam de trocar informações com o coordenador, este super-passo da execução paralela é terminado e o processo coordenador pode prosseguir para a fase seqüencial da execução e iniciar um novo quadro de simulação do servidor.

4.3. Detalhes de implementação

Nesta seção apresentamos os detalhes de implementação do modelo de paralelização apresentado na Seção 4.2.

A implementação deu-se através de modificações no código original do servidor, disponível

publicamente no sítio do fabricante sob a licença *GNU General Public License* (GPL) versão 2 [29].

4.3.1. Processos auxiliares e comunicação inter-processos

No protótipo de implementação do modelo de paralelização proposto por este trabalho, os processos auxiliares são processos do sistema operacional criados através da chamada de sistema `fork()`. De acordo com a classificação sugerida por Flynn [21], o protótipo segue o modelo de programação paralela conhecido como *Single Program, multiple data* (SPMD).

Após o cálculo dos grupos de entidades próximas formados pelas entidades ativas de um quadro de simulação, o servidor determina quantos processos auxiliares serão necessários para o processamento dos grupos. Será utilizado um número de processos auxiliares menor ou igual ao número de processadores disponíveis no computador menos um, uma vez que o processo coordenador também executa uma parte da simulação.

A criação de apenas um processo auxiliar por processador se deve ao fato do servidor do Quake utilizar a arquitetura *single-process event-driven* em sua plenitude. O servidor não se utiliza de nenhuma chamada de sistema que possa bloquear o processo uma vez que, como visto na Seção 3.3.5, tais chamadas poderiam levar à má utilização da CPU devido a problemas inerentes à implementação destas no sistema operacional. Como observado em [69], essa metodologia de paralelização garante a utilização eficiente das CPUs disponíveis.

Outra decisão que deve ser discutida é a utilização de processos do sistema operacional ao invés de *threads* para a implementação dos processos auxiliares. Duas razões principais levaram a decisão de utilizarmos processos ao invés de *threads*.

A primeira foi a percepção de que a utilização de processos seria um facilitador da implementação do paralelismo no próprio servidor do Quake.

Desde o início, a complexidade do código-fonte do servidor do Quake foi um desafio para a implementação do modelo de paralelismo no próprio código do servidor. Além de possuir pouca

documentação, o Quake possui várias otimizações no código que permitem simular os modelos 3D dos jogadores em computadores domésticos com pouca capacidade de processamento e memória[2].

Apesar do código ser divido em camadas bem definidas, estas são completamente dependentes entre si. Em geral, essa inter-dependência se deve à utilização maciça de variáveis globais e, conseqüentemente, à falta de encapsulamento destas camadas, em decorrência das otimizações feitas no código-fonte do programa.

Isso significa que modificações no código-fonte do Quake exigem um grande cuidado. A troca da ordem de chamada de duas funções que conceitualmente realizam duas tarefas completamente diferentes (às vezes até de duas camadas diferentes) podem introduzir resultados inesperados. Isso deve-se ao fato de que normalmente a execução de uma função do Quake depende de que um conjunto de variáveis globais estejam previamente populadas antes de ser executada. Infelizmente nem sempre é fácil determinar todas as pré-dependências de uma função.

Abdelkhalek utilizou-se de um artifício para minimizar esse problema em sua implementação de uma versão *multithreaded*. Praticamente todas as variáveis utilizadas pelo código do servidor durante a fase de processamento paralelo foram classificadas como sendo variáveis que compõem as estruturas de dados globais e variáveis temporárias, cujo escopo é o processamento dos eventos de cada cliente.

Para as variáveis do primeiro tipo, Abdelkhalek utilizou-se da árvore *Areanode* para sincronizar o acesso de leitura e escrita, conforme explicado na Seção 3.2. Para as demais, Abdelkhalek modificou o código para que cada variável passasse a ter uma dimensão extra, com tamanho igual ao número máximo de *threads* utilizadas. Dessa forma, o que era uma variável do tipo `int` tornou-se uma variável do tipo `int[]`, emulando o que seria uma variável local da *thread*.

A proposta de implementação com `fork()` torna a paralelização transparente para todo o

[2] o Quake foi lançado comercialmente em 1996 e sua execução requer, pelo menos, um sistema com processador Pentium de 75MHz e 8 MB de memória RAM

mecanismo de processamento de eventos – desde a leitura da mensagem até o envio da resposta. Não há preocupação com *locks* nessa etapa e o problema de determinar quais variáveis possuem escopo do tratamento de eventos não existe. Entretanto, há a necessidade de integrar as modificações locais às estruturas de dados globais, como veremos na Seção 4.3.4.

A segunda questão sobre o uso de processos ao invés de *threads* foi a percepção de que após as modificações no código-fonte do Quake o esforço do sistema operacional para criar uma cópia do processo do servidor do Quake através do `fork()` seria relativamente pequeno.

A criação de *threads* no Linux segue os mesmos passos da criação de novos processos, através do uso da chamada de sistema `clone()`. Na verdade, para a criação, execução e escalonamento de tarefas no sistema operacional não há diferenciação entre *threads* e processos comuns. Ambos são representados por uma `task_struct` e criados através de uma chamada à `clone()`; a diferença é que certos recursos são compartilhados entre o processo criado como *thread* e o processo que o criou. Desses recursos compartilhados, talvez o mais importante seja o ponteiro para a tabela de memória virtual, que permite que a memória seja compartilhada entre os processos criados dessa forma, definindo, assim, a semântica de *thread* desses processos.

Quando o novo processo não possui semântica de *thread* e, portanto, não compartilha a memória virtual com o processo pai, é necessário criar uma cópia dos recursos não compartilhados para que as duas linhas de execução possam prosseguir independentemente. O Linux realiza essa cópia de recursos sob demanda, utilizando um mecanismo que é conhecido como *copy-on-write* [11].

O mecanismo de *copy-on-write* utilizado pelo sistema de memória virtual funciona da seguinte forma: quando o processo filho é criado, as páginas de memória do processo pai são marcadas como sendo somente para leitura e a estrutura `task_struct` do processo filho é configurada para apontar para as mesmas páginas do pai. Quando algum dos processos realizar uma operação de escrita em uma das páginas, ocorrerá uma falha de página e o processo fará uma cópia da página para si. Nesse momento, o núcleo do SO marca as duas páginas – a original e a nova cópia – como sendo de leitura e escrita. A partir daí, cada processo utiliza

sua própria cópia da página de memória. Portanto, o sobrecusto de utilização do `fork()` está associado diretamente à quantidade de memória utilizada pelo processo.

No servidor do Quake, a maior parte da memória RAM é consumida na armazenagem das estruturas de dados que contêm as propriedades utilizadas para a simulação de cada entidade do jogo. Tais propriedades são atualizadas em duas situações: na etapa de simulação física do servidor e no processamento de mensagens.

Essas informações precisam ser repassadas aos processos auxiliares após a etapa de simulação física e precisam ser devolvidas para o processo coordenador após o processamento e envio das mensagens. É necessário, então, uma forma de comunicação inter-processos adequada para fazer isso.

Para minimizar a quantidade de páginas de memória copiadas pelo mecanismo de *copy-on-write* do `fork()`, escolhemos utilizar como forma de comunicação inter-processos memória compartilhada.

Dessa forma, o mecanismo de *copy-on-write* não precisa copiar as páginas de memória com as propriedades das entidades, uma vez que estas estão em uma área de memória compartilhada, e nem precisa copiar as páginas de memória que contêm apenas informações que são utilizadas exclusivamente para leitura. Apenas as páginas de memória que armazenam áreas de memória utilizadas no escopo do processamento e envio de mensagens – que não precisam ser sincronizadas com os outros processos auxiliares – é que serão copiadas.

O uso de um *pool* de processos diminuiria o custo da criação de processos, mas não o da cópia das páginas de memória, uma vez que todos os dados da simulação necessariamente precisariam ser copiados do processo coordenador para o filho ao término da fase de simulação física. Além disso, páginas de memória que não fossem utilizadas em um quadro ou que fossem utilizados apenas para leitura teriam que ser copiadas em todos os quadros de simulação e não somente quando fossem necessários, como ocorre com a utilização do mecanismo de *copy-on-write* do sistema operacional.

Essas observações são importantes, pois garantem que a implementação deste protótipo no

próprio servidor do Quake é adequada e garante uma boa aproximação do que seria possível se a implementação utilizasse *threads*.

4.3.2. Distribuição dinâmica de tarefas

A implementação do mecanismo de distribuição dinâmica de tarefas descrito na Seção 4.2.3 exigiu uma modificação no modo como o servidor do Quake consome as mensagens do soquete de rede.

O servidor original do Quake lê uma mensagem do soquete e realiza todo o processamento – da execução dos comandos contidos na mensagem à composição e envio da resposta – antes de ler a próxima mensagem da fila. Esse modo de operação não é adequado em um servidor que processa muitos clientes simultaneamente, já que poderia prolongar em demasia o tamanho de um quadro do servidor. A simulação ficaria comprometida, uma vez que o servidor não teria a oportunidade de executar a simulação física em tempo hábil. O problema do controle da quantidade de mensagens que deve ser processada concorrentemente foi discutido por Brecht et al. e apresentado na Seção 2.3.1.

Para o cálculo do grupo de entidades próximas, o servidor precisa saber antecipadamente quais os clientes ativos, isto é, aqueles que enviaram alguma mensagem durante o último quadro de simulação. A implementação do controle de aceitação de mensagens proposta por Brecht teve como efeito colateral a resolução desse problema.

As mensagens disponíveis no soquete são lidas e armazenadas em uma fila de mensagens. O servidor lê, no máximo, um número de mensagens igual ao número de clientes com que o servidor pode lidar a fim de que o quadro de simulação não fique demasiadamente grande. Durante o processamento das mensagens, estas são lidas da fila de mensagens ao invés de serem lidas diretamente do soquete. Com o enfileiramento prévio das mensagens, determinar quais os clientes ativos neste quadro de simulação é trivial.

Para cada cliente desta fila, são realizados os seguintes passos:

- construção de um `pmove`;
- detecção das entidades na área de atuação do movimento;
- adição dos clientes e das entidades possivelmente afetadas ao grafo de entidades próximas.

No Quake, a estrutura de dados `pmove` representa o movimento a ser realizado pela entidade. Essa estrutura é utilizada em duas situações: na simulação de um movimento, através do método `PlayerMove()`, e na detecção das entidades que possivelmente serão afetadas pelo movimento, através da função `AddLinksToPmove()`.

Antes da execução de um movimento, o servidor constrói a lista de entidades possivelmente afetadas com a chamada à função `AddLinksToPmove()` para que, após a simulação do movimento, seja possível detectar possíveis colisões e seus impactos. Caso haja colisão, o servidor executa o código escrito em QuakeC associado com essas entidades e que dá a semântica de uma colisão para a simulação. Por exemplo, caso a colisão seja entre dois jogadores, então os dois devem simular a física de uma colisão elástica e se afastar um pouco. Se a colisão for entre um jogador e um projétil, então o jogador deve perder "pontos de energia" e o projétil deve ser removido do servidor.

Utilizamos essa mesma funcionalidade para detectar as entidades que serão potencialmente afetadas pelo cliente e que, portanto, deverão ser simuladas no mesmo processador em que esse cliente será simulado.

Construímos um `pmove` igual ao construído pelo servidor para simular uma mensagem do tipo `MOVE`. De todas as mensagens tratadas pelo servidor, esta é a que tem atuação em uma área mais ampla e, portanto, garante que se a simulação da mensagem afetar alguma outra entidade, então essa entidade será incluída no mesmo grupo de entidades próximas que contém o cliente.

Para calcular as entidades que serão possivelmente afetadas pelo cliente, uma chamada à função `AddLinksToPmove()` é realizada. Esse método percorre a árvore *Areanode* e encontra quais os clientes que estão contidos na área do cubo definido acima. A árvore é usada apenas

para realizar uma forma de busca binária no labirinto 3D do servidor, onde a busca leva em consideração a posição das entidades. O uso da árvore *Areanode* torna a construção desta lista muito mais eficiente.

Com esses dados, é possível construir o grafo que representa as entidades próximas. A lista de entidades devolvidas por `AddLinksToPmove()` define as arestas desse grafo. Para cada entidade devolvida na chamada à `AddLinksToPmove()`, criamos uma aresta que liga a entidade com o cliente que enviou a mensagem.

Para calcular efetivamente os grupos de entidades próximas devemos calcular os componentes conexos desse grafo. Por se tratar de um grafo não dirigido, não é necessário criarmos todo o grafo para o cálculo de suas componentes. Utilizamos o conhecido algoritmo de *union-find* com compressão de caminhos descrito em [55], com uma pequena modificação para que a compressão de caminhos seja feita ao fim de cada iteração e não no início. Dessa forma, ao fim de cada inclusão, as informações sobre todas as componentes conexas e seus tamanhos estão atualizadas.

Após aplicar o *union-find* para todos os clientes e seu respectivo conjunto de entidades possivelmente afetadas, é necessário realizar o escalonamento da execução das tarefas nos processadores disponíveis.

Implementamos o algoritmo de balanceamento de carga denominado *Longest Processing Time First* (LPT) [23, 38]. O tamanho de cada grupo foi escolhido como estimativa para o tempo de processamento de cada grupo, uma vez que quanto maior o grupo, possivelmente haverá mais interações entre as entidades e, por conseqüência, mais processamento.

Todos os processadores iniciam o processo com carga total igual a zero. Exceção é feita para o processo coordenador, que inicia com carga igual a um como medida para forçar uma melhor utilização dos processos auxiliares. Para cada grupo da lista ordenada, removemos aquele com o maior tamanho da lista e atribuímos ao processador com menor carga naquele momento. A nova carga do processador escolhido é igual ao valor da carga anterior, acrescido do tamanho do grupo atribuído.

Assim que todos os grupos são atribuídos a algum processador, o servidor pode prosseguir com a criação dos processos auxiliares e com o tratamento paralelo de eventos.

4.3.3. Tratamento paralelo de eventos

A etapa de tratamento paralelo de eventos compreende a leitura e processamento das mensagens enviadas pelo cliente e corresponde à execução dos métodos `SV_ReadPackets()` e `SV_SendClientMessages()`. A natureza do modelo de paralelização proposto para o servidor do Quake permitiu que o resultado da implementação desta etapa fosse simples e praticamente transparente ao mecanismo de tratamento de eventos original do servidor do Quake.

Inicialmente, o processo coordenador cria os processos auxiliares através do comando `fork()`. Sendo n o número de processadores disponíveis e m o número de grupos de entidades próximas criados na etapa de distribuição dinâmica de tarefas, são criados $\min\{n-1, m\}$ processos auxiliares. Ou seja, são criados no máximo um processo auxiliar por processador disponível, mas não são criados mais processos do que grupos para serem processados. Além disso, o próprio processo coordenador executa o processamento de alguns grupos e, portanto, não há necessidade de criarmos mais do que $n-1$ processos para utilizar os n processadores disponíveis.

Cada processo auxiliar prossegue com sua execução independentemente. Nesse trecho, há poucas modificações no código-fonte do servidor. É necessário, entretanto, ressaltar duas das modificações efetuadas.

A primeira modificação na execução é resultado imediato da divisão de tarefas realizada na etapa anterior. Um processo auxiliar só iniciará o tratamento de uma mensagem caso seu remetente esteja em algum grupo de entidades atribuído a si.

A segunda modificação importante foi a aplicação do padrão *Decorator* [22] em todos os métodos de envio de mensagens do servidor para que uma mensagem enviada por uma entidade só fosse recebida por outra entidade do mesmo grupo. À primeira vista, a necessidade da alteração parece contradizer a hipótese do modelo de paralelismo que diz que o processamento

de uma entidade só interfere em outra caso ambas estejam em um mesmo grupo de entidades próximas. Entretanto, no momento da execução do código em QuakeC, essa premissa pode ser invalidada.

É possível que um *script* em QuakeC acesse qualquer entidade do jogo e a modifique a qualquer momento. Como o servidor do Quake foi concebido para ser um simulador genérico, a semântica das ações das entidades são definidas em QuakeC, não no servidor. Decidimos nos ater à semântica das entidades originalmente presentes na versão comercial do Quake, sem nos preocupar com entidades que por ventura sejam adicionadas posteriormente pelo usuário.

Ainda assim, há dois tipos de mensagens enviadas por entidades do jogo original que podem afetar as estruturas de dados dos usuários. Mensagens com notificações sobre eventos de áudio – efeitos sonoros do jogo, por exemplo – e mensagens de texto enviadas por um jogador para outros jogadores. Alteramos a semântica de tais mensagens para que estas fossem enviadas somente para as entidades processadas pelo mesmo processo auxiliar; ou seja, deixaram de ser do tipo *broadcast* para serem do tipo *multicast*, onde o grupo *multicast* é o grupo de jogadores processados no mesmo processo auxiliar. Tais alterações não afetaram significativamente a execução da simulação.

4.3.4. Sincronização inter-processos

A sincronização inter-processos ocorre após a fase de tratamento paralelo de eventos e marca o fim do super-passo do modelo BSP de paralelização. Todos os processos auxiliares devem enviar o conjunto de modificações realizadas em suas entidades para o processo coordenador que, após receber as modificações de todos os auxiliares, prossegue para o próximo super-passo.

Como dito anteriormente, o mecanismo utilizado para a realização de comunicação inter-processos nesta implementação é o mecanismo de memória compartilhada definido pelo *System V*. Para máquinas do tipo SMP, memória compartilhada é o método mais eficiente de comunicação inter-processos disponível no sistema operacional Linux [53].

O servidor do Quake foi modificado para que os dados dos clientes fossem compartilhados entre os processos auxiliares. O código-fonte do Quake define que toda entidade simulada pelo servidor possui uma estrutura de dados definida pelo tipo `edict_t`.

Essa estrutura tem como função principal definir quais as propriedades necessárias para a realização da simulação física desta entidade (propriedades estas armazenadas em uma variável do tipo `entvars_t`), além de definir seu posicionamento no labirinto virtual do jogo. É nesta estrutura que são definidos os ponteiros que formam a árvore *Areanode*.

A estrutura `edict_t` armazena também o que foi denominado *baseline* da mensagem. O *baseline*, representado pelo tipo `entity_state_t`, contém os dados sobre essa entidade que foram transmitidos na última mensagem enviada para o cliente e é utilizado pelo servidor para a redução do tamanho das mensagens enviadas para os clientes. No momento do envio, o servidor compara o novo estado com o estado armazenado no *baseline* e só envia as modificações.

Um jogador é um tipo especial de entidade. Além dos atributos definidos por `edict_t` para sua simulação, a estrutura de dados que define o tipo `client_t` contém predominantemente informações relevantes à comunicação entre cliente e servidor. Portanto, é nessa estrutura que são armazenados dados como identificadores do usuário (nome do jogador, endereço IP, porta utilizada pelo cliente para a conexão), *buffers* para o envio e recebimento de mensagens, data da última mensagem recebida pelo servidor (para que o servidor possa desconectar o cliente, caso o mesmo esteja inativo há muito tempo), etc.

A sincronização destes dados entre os processos auxiliares se deu de duas formas distintas: acesso direto, via memória compartilhada, e sincronização indireta.

Os dados do jogador que não estavam armazenados em `edict_t` puderam ser referenciados diretamente da área de memória compartilhada. Modificou-se o servidor para que este acessasse tais dados diretamente da memória compartilhada. Novamente, o uso de *locks* não foi necessário, dada a natureza do modelo de paralelização.

Os dados de `edict_t` necessitaram um pouco mais de cuidado durante a sincronização. Apesar de, em um primeiro momento, parecer que os tipos `entvars_t` e `entity_state_t`

poderiam ser acessados diretamente da memória compartilhada, ambos mantêm parte dos dados utilizados na árvore *Areanode* e, portanto, devem ter tratamento especial.

A árvore *Areanode* é uma estrutura de dados global que é modificada por todos os processos auxiliares. Todo objeto que tem sua posição modificada deve ser re-inserido na árvore para que ela reflita a nova posição. Para evitar os problemas de sincronização apresentados no trabalho de Abdelkhalek [3], cada processo auxiliar modifica uma cópia local da árvore durante o tratamento dos eventos.

Ao fim do super-passo, cada processo auxiliar deve separar os dados referentes à simulação dos dados referentes à árvore *Areanode* e copiá-los para uma área de memória compartilhada. O processo coordenador, após esperar todos os processos auxiliares terminarem (com o uso de `wait()`), copia esses dados de volta às estruturas de dados locais. Note que no momento do `fork()` não é necessário tomar nenhum cuidado adicional para copiar essas informações para os processos auxiliares, já que estas são herdadas diretamente das páginas de memória copiadas do processo pai.

Por fim, os elementos modificados devem ser re-inseridos na árvore *Areanode* para que a árvore passe a refletir o novo posicionamento das entidades modificadas pelos processos auxiliares. Essa atualização ocorre naturalmente na etapa de atualização do modelo físico do servidor, já que os atributos da estrutura `edict_t` são atualizados pela simulação e a re-inserção é realizada pelo código original do Quake.

A Figura 4.1 exibe o diagrama de atividades para a execução de um quadro de simulação em um computador com dois processadores disponíveis.

4.4. Efeitos do escalonador do SO na implementação

4.4.1. Descrição dos efeitos

Os primeiros experimentos com a implementação proposta na Seção 4.3 não produziram bons resultados. Os experimentos – apresentados em 5.4.4 – mostravam que a paralelização

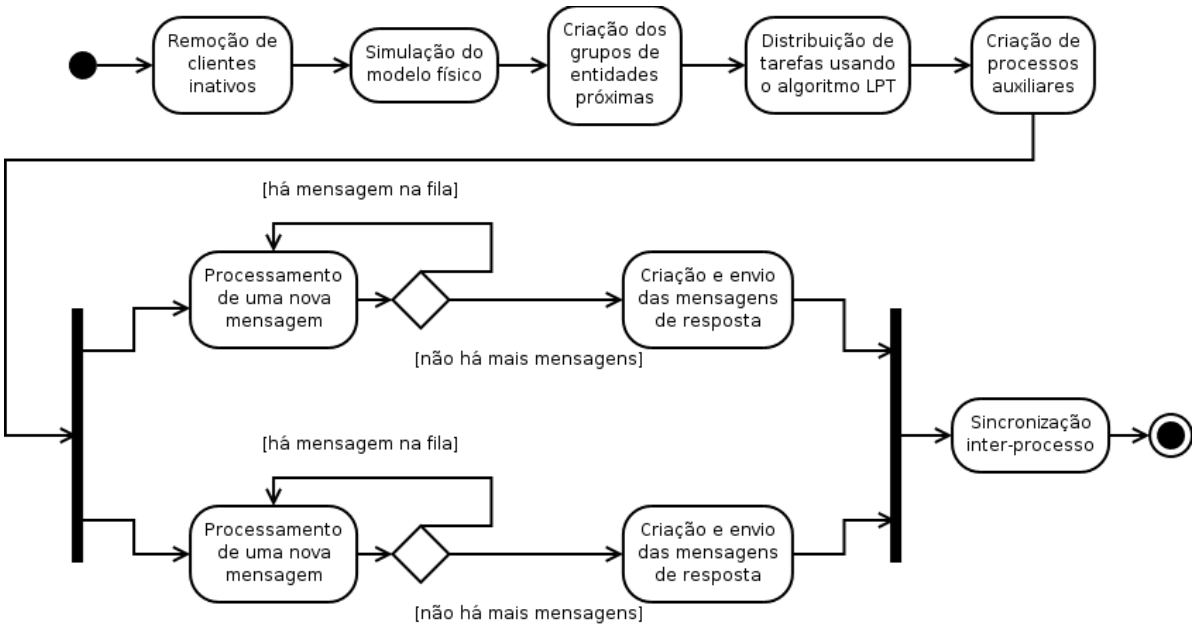


Figura 4.1.: Diagrama de atividades da execução de um quadro de simulação da versão paralela do servidor do Quake

praticamente não havia produzido efeito na taxa de quadros de simulação executados no servidor por segundo.

Os testes com uma máquina SMP com 2 processadores e sistema operacional Linux mostravam que o tempo total de um quadro era o mesmo se utilizássemos ou não um processo auxiliar, ou seja, era o mesmo se usássemos apenas um ou os dois processadores disponíveis. Experimentos em uma máquina SMP com 8 processadores e sistema operacional Solaris também exibiam pouca variação da taxa de quadros por segundo com o aumento do número de processos auxiliares.

Iniciamos a investigação do problema através de instrumentação do código-fonte do servidor do Quake. A análise dos dados produzidos pela versão instrumentada foi inconclusiva para a determinação da causa do problema.

Prosseguimos, então, para uma análise da execução do programa no nível do sistema operacional através do uso do *Linux Trace Toolkit* [68, 30]. O traço de execução do servidor do

Quake na máquina SMP com dois processadores mostrou que mesmo com a utilização de dois processos, um coordenador e um auxiliar, apenas um dos processadores era utilizado.

Pelo traço da execução ficou claro o que estava tornando os resultados insatisfatórios. Durante toda a etapa seqüencial a execução do servidor segue como previsto originalmente. Entretanto, imediatamente após a criação do primeiro processo auxiliar através da chamada de sistema `fork()`, o escalonador do núcleo do sistema operacional elege o novo processo para execução no *mesmo processador utilizado pelo processo pai*. O processo pai é retirado da CPU para que o processo filho rode em seu lugar, mas o escalonador não migra o processo pai para a CPU ociosa.

O processo pai não é escalonado para utilização de CPU durante toda a execução do processo filho. Ao término do processo filho, o escalonador do SO re-escalona o processo pai para que este execute no mesmo processador em que se encontrava antes. O processo pai, então, pode seguir com o tratamento dos eventos atribuídos a ele ou, caso haja mais do que dois processadores, pode seguir com a criação de outros processos auxiliares, situação em que o problema se repete.

O problema aqui relatado demandou uma investigação sobre o mecanismo de escalonamento de tarefas do sistema operacional Linux.

4.4.2. Escalonamento de tarefas criadas via `fork()`

Um estudo sobre o escalonador do Linux apontou uma característica presente em praticamente todos os sistemas operacionais modernos e que foi responsável pelo comportamento irregular apresentado na implementação inicial.

O artigo de Vaswani e Zahorjan [60] alertou os desenvolvedores de sistemas operacionais para os efeitos da realocação de um processo para outro processador em sistemas multiprocessados com acesso uniforme à memória – como é o caso dos computadores que seguem a arquitetura SMP.

Segundo os autores, em ambientes multiprocessados, onde cada processador tem seu próprio *cache*, a execução de um processo faz com que o preenchimento do *cache* do processador com dados e instruções do processo crie uma certa relação de afinidade entre o processo e o processador. Uma política de escalonamento de processos que ignora esta afinidade pode desperdiçar recursos computacionais com o preenchimento repetitivo do *cache*.

O escalonador da série 2.4 do Linux não levava a afinidade em consideração e os processos apresentavam um efeito que é conhecido pelos desenvolvedores do Linux como efeito *ping-pong* [46], onde hora o processo roda em um processador, hora em outro. Com a adição de uma fila de execução (`runqueue`) por processador, o escalonador do Linux 2.6 prioriza a re-execução de um processo em um mesmo processador e, com isso, aproveita melhor a afinidade entre processos e processadores.

A afinidade de processos justifica a decisão de escalonar o processo filho no mesmo processador que o pai, mas não é suficiente para explicar o desbalanceamento de carga entre os processadores. Manter o processo pai em seu processador atual seria suficiente para alcançar uma boa utilização do *cache* daquela CPU.

De fato, o comportamento do escalonador para com os processos auxiliares também sofre efeito de uma otimização do sistema operacional para processos recém criados.

O escalonador do sistema operacional Linux escalona de forma diferenciada processos recém-criados. Há duas ocasiões em que o escalonador pode aplicar algum tipo de otimização: no momento da execução de um novo programa, através de uma chamada à `execve()` e no momento de criação de um processo, via `fork()`.

Um processo que terá seu programa substituído, via chamada à `execve()`, é visto pelo escalonador como um candidato ideal a ser migrado para outro processador. No momento da chamada à `execve()` o processo ainda não possui nenhuma afinidade com o processador e sua migração acarraterá o menor custo possível de troca de contexto para um processo [2].

O contrário ocorre com um processo que acaba de ser criado com uma chamada à `fork()`. No momento de sua criação, o novo processo possui grande afinidade com o processador que

executa o processo que realizou o `fork()`.

Além disso, impedir a execução do pai por algum tempo pode ser benéfico para processos criados com `fork()`. Durante a execução do sistema operacional com uma carga de trabalho genérica, a maior parte dos processos que são criados com `fork()` realizam algumas poucas operações e logo fazem uma chamada à `execve()`. Ao impedir que o pai execute por algum tempo, o sistema operacional permite que o processo filho execute essas operações sem penalizar o pai com o provável sobrecusto das cópias das páginas de memória através do mecanismo de *copy-on-write*.

O processo pai é re-escalonado no próximo *scheduler tick* e, se necessário, um novo balanceamento de carga é realizado e um dos processos é migrado para algum processador disponível.

No caso do servidor do Quake, o processo filho termina suas tarefas antes do próximo *scheduler tick* e, por isso, a execução ocorre sem paralelismo entre os processos auxiliares e o coordenador. Como veremos na Seção 5.3.2, a carga nos processos auxiliares na implementação original é pequena e cada processo auxiliar executa um quadro de simulação pequeno, em média com apenas 7 grupos por processador.

A seção seguinte apresenta as adaptações na implementação que permitiram que o servidor apresentasse um paralelismo real na execução dos processos.

4.5. Adaptações

Implementamos uma nova metodologia de criação e destruição de processos que permitiu que a execução do servidor paralelo do Quake ocorresse de forma mais adequada. Com as modificações impostas por essa nova metodologia, foi possível obter um comportamento do escalonador mais adequado ao tipo de carga de trabalho exercido pela versão paralela do servidor.

Tais modificações, entretanto, necessitam que o sistema operacional utilizado disponibilize um mecanismo que forneça um controle fino, configurável em nível de usuário, para questões

como criação, escalonamento e inter-dependência entre processos pai e filho. O sistema operacional Linux possui mecanismos desse tipo e, por isso, a versão adaptada tornou-se dependente desse sistema operacional.

4.5.1. Criação dos processos auxiliares

A metodologia de criação de processos foi modificada para que as decisões de escalonamento baseados em afinidade entre processos e processadores fossem realizadas pelo próprio servidor, e não automaticamente pelo escalonador do sistema operacional.

A partir da versão 2.5 do Linux é possível definir manualmente a política de afinidade dos processos em execução [47]. O sistema operacional disponibiliza a chamada de sistema `sched_setaffinity()` que permite que o programa defina manualmente a afinidade entre seus processos e os processadores disponíveis. A chamada de sistema `sched_getaffinity()` recupera a informação da afinidade entre processos e processadores das estruturas de dados do escalonador. Por padrão, todo processo possui, inicialmente, afinidade com todos os processadores.

Usando-se `sched_setaffinity()`, associamos cada processo a um processador disponível com uma distribuição *round-robin*. O processo coordenador é associado sempre ao processador zero.

Para definir a afinidade de um processo é necessário que esse processo já esteja criado e que, portanto, possua um identificador de processo. Entretanto, como vimos na seção anterior, o escalonador elege para execução o processo filho imediatamente após a criação do processo pelo `fork()`.

Foi necessário, então, modificar o código de criação de processos do servidor do Quake para que o processo filho fosse criado em um estado suspenso, inelegível para utilização de CPU. Ao invés de utilizarmos a chamada de sistema `fork()` para criação do processo, utilizamos a chamada de sistema `clone()` com alguns parâmetros de configuração especiais para que fosse

possível manter um controle mais fino da criação do novo processo. Como visto em 4.3.1, `clone()` é o ponto comum de criação de todos os processos e *threads* no sistema operacional Linux.

Normalmente, o método `fork()` é apenas uma casca para o `clone()`, que é chamado com os parâmetros `CLONE_CHILD_CLEARTID`, `CLONE_CHILD_SETTID` e `SIGCHLD`. O primeiro parâmetro limpa o campo com o identificador do *thread group* e notifica, se necessário, a biblioteca de *threads* que criou o processo pai, o segundo define um novo identificador de *thread group*, para diferenciá-lo do grupo antigo e, por último, o indicador (*flag*) `SIGCHLD` define a semântica do tratamento de sinais do núcleo do sistema operacional como sendo a semântica de processos criados via `fork()`.

Substituímos a chamada à `fork()` por uma chamada à `clone()` com os mesmos parâmetros passados pelo `fork()` – exceto pelo parâmetro `SIGCHLD`, como será explicado na Seção 4.5.2 – acrescido do parâmetro `CLONE_STOPPED`, disponível desde a versão 2.6.0 do núcleo do Linux, que faz com que o processo criado seja suspenso, como se este tivesse recebido um sinal do tipo `SIGSTOP`.

Com o processo criado com estado suspenso, podemos definir a afinidade do novo processo através do método `sched_setaffinity()`. Definida a afinidade, o processo coordenador envia um sinal do tipo `SIGCONT` para que a execução do processo auxiliar seja retomada no processador apropriado.

Dessa forma, duas adaptações ao escalonamento dos processos foram realizadas. Os processos filhos não executam imediatamente após a chamada à `fork()`/`clone()` e não executam necessariamente no mesmo processador que o processo pai, viabilizando, portanto, a execução paralela do servidor.

4.5.2. Destruição dos processos auxiliares

Como efeito colateral do fato dos processos auxiliares terem ficado com pouca carga de execução, o custo de criação e de destruição dos processos auxiliares pode chegar a até 30% do tempo total de execução de um processo auxiliar.

Na etapa de sincronização inter-processos, o processo coordenador utilizava-se do comando `wait()` para verificar se seus processos auxiliares haviam terminado. Além de funcionar como uma barreira de sincronização, o `wait()` garante que o processo filho será removido da tabela de processos após o seu término. Caso o processo filho termine antes que o pai execute o comando `wait()`, este é mantido na tabela de processos pelo sistema operacional em um estado denominado *zombie*.

Uma chamada à `wait()` só retorna quando o processo filho é completamente destruído, isto é, quando todas as estruturas de dados alocadas pelo sistema operacional para representar o processo são liberadas. Porém, apenas a destruição do processo é responsável por cerca de 25% do tempo de execução do processo auxiliar.

A destruição do processo é uma boa candidata à paralelização nesta implementação. Imediatamente após o término de cada processo filho, isto é, após a etapa de sincronização global, o servidor executa a etapa de simulação do modelo físico, que é seqüencial.

Para implementar a destruição paralela dos processos auxiliares, é necessário resolver dois problemas: implementar outra forma de barreira de sincronização e resolver o problema dos processos ficarem em estado *zombie* após seu término.

O primeiro problema foi resolvido com o uso de um semáforo. Antes de realizar a chamada à `exit()`, cada processo auxiliar notifica o processador sobre término incrementando o valor do semáforo em um. O processo coordenador, por sua vez, decrementa o semáforo em um número igual ao número de processos auxiliares criados neste quadro do servidor. Dessa forma, o processo coordenador é suspenso até que todos os processos auxiliares terminem suas tarefas. Neste ponto, a liberação dos recursos utilizados por cada processo auxiliar no sistema

operacional ocorre em paralelo.

Para resolver o problema dos processos ficarem em estado *zombie*, duas modificações foram feitas. A primeira foi excluir o parâmetro `SIGCHLD` da chamada à `clone()` no momento de criação do processo auxiliar. A exclusão desse parâmetro faz com que o processo filho não notifique o pai sobre alterações no estado de sua execução.

Além disso, utilizamos a chamada de sistema `sigaction()` para configurar o tratamento dos sinais recebidos pelo processo coordenador. Ao passar o parâmetro `SA_NOCLDWAIT` – disponível a partir do Linux versão 2.6 – para a chamada `sigaction()`, o coordenador instrui o sistema operacional a não manter os processos filhos no estado *zombie* caso terminem sua execução.

Dessa forma, o processo coordenador não precisa realizar nenhuma tarefa adicional para impedir que seus processos auxiliares fiquem em estado *zombie* e a liberação dos recursos do sistema operacional utilizados para manter as informações sobre os processos auxiliares é realizada em paralelo à execução da etapa seqüencial do servidor.

A seguir, veremos dados relativos a execuções reais do servidor implementado em diversos ambientes.

5. Parte experimental

Neste capítulo serão apresentados dados que permitem caracterizar o desempenho do protótipo inicial e da versão adaptada da implementação do modelo paralelo para o servidor do Quake.

A Seção 5.1 descreve o ambiente onde os experimentos foram realizados. Na Seção 5.2 descreveremos o impacto que o mapa utilizado durante uma sessão do jogo exerce sobre o desempenho da versão paralela do servidor. A Seção 5.3, caracterizamos a formação dos grupos de entidades próximas durante a execução do servidor e, por fim, a Seção 5.4 descreve as métricas de desempenho analisadas e os resultados obtidos.

5.1. Ambiente de testes

O ambiente de execução dos clientes utilizados nos experimentos foi composto por 5 computadores equipados com processador AMD Athlon^{TM} XP 2800+ (2,25 GHz), 1,0 GB de memória RAM e inter-conectados através de uma rede Ethernet de 100 Mbit/s. Utilizamos o sistema operacional Linux versão 2.6.16.2.

Para a execução dos experimentos, criamos uma versão modificada do cliente do Quake que permite a execução automatizada de uma sessão do cliente. O funcionamento dessa versão modificada foi descrito na Seção 3.3.4.

Os experimentos com a implementação da versão paralela do servidor foram realizados em dois ambientes distintos, um com sistema operacional Solaris e outro com sistema operacional Linux.

Adaptamos o código-fonte do Quake para executá-lo no sistema operacional Solaris. O ambiente de testes Solaris é composto de um computador Sun Fire V880, equipado com 8 processadores UltraSPARC 900 MHz e 32,0 GB de memória RAM. O sistema operacional utilizado foi o Solaris versão 9.

No ambiente Linux, utilizamos um computador equipado com dois processadores Intel Xeon de 3.0 GHz e 2,0 GB de memória RAM. O sistema operacional utilizado foi o Linux versão 2.6.17.7, com os *patches* do *Linux Trace Toolkit* versão 0.5.76 aplicados.

Não foi possível utilizar o `gprof` na análise da versão paralela. A versão atual do `gprof`, 2.19, não possui suporte para aplicações que utilizem múltiplas *threads* ou processos. Experimentamos alternativas como o `qprof` [41], mas o custo de desempenho com a sobrecarga imposta por essas aplicações mostrou-se alto demais para sua utilização com o Quake. Além disso, é importante notar que não há versões do LTT para o sistema operacional Solaris.

5.2. Influência dos cenários

Desde os primeiros testes com a versão seqüencial do Quake, a influência dos cenários mostrou-se determinante na execução da simulação do jogo.

Três fatores principais explicam o impacto do mapa virtual na execução da simulação:

- número de entidades associadas ao mapa;
- tamanho das mensagens de resposta;
- concentração de usuários em áreas específicas.

5.2.1. Entidades associadas ao mapa

Associado a cada mapa virtual do Quake, existem entidades que devem ser simuladas ao longo da execução do servidor. Podemos classificar tais entidades em dois grupos: entidades

que definem os objetivos do cenário e entidades que atribuem funções especiais a objetos do cenário.

Entidades do primeiro tipo definem metas a serem alcançadas pelo jogador. Em geral, assim que todas as metas são alcançadas, o código dessas entidades, escrito em QuakeC, determina que um novo mapa seja carregado pelo servidor.

As entidades do segundo tipo permitem que sejam definidos modos de interação entre um jogador e os elementos do cenário. Elevadores, por exemplo, são definidos como entidades sólidas que disparam a simulação de movimento vertical – simulação esta definida em QuakeC – no momento de sua colisão com um jogador.

Quanto mais complexo for o mapa virtual, maior o número de entidades associadas a ele. Como conseqüência, maior se torna o tempo total de execução da etapa de simulação física do jogo. Além disso, um mapa com muitas entidades faz com que o tamanho dos grupos de entidades próximas aumente, uma vez que, para cada mensagem executada, o servidor deve verificar se algumas dessas entidades foi afetada pelo movimento.

Em nossos experimentos, foram utilizados mapas que possuem poucas entidades associadas. Cenários idealizados para um grande número de jogadores possuem poucas entidades associadas, uma vez que para esse tipo de jogo a interação entre os usuários é priorizada.

5.2.2. Tamanho das mensagens de resposta

Conforme visto na Seção 3.1, o servidor do Quake utiliza-se de métodos de computação mais agressivos para obter o menor tamanho possível nas mensagens que envia para seus clientes. Na construção de uma mensagem, são considerados apenas os elementos que estão na possível área de alcance do jogador (*Possible Visible Set* ou PVN).

O cálculo da PVN de cada jogador é feito de forma eficiente, porém computacionalmente intensa, graças ao modo como o mapa virtual é representado no servidor. Para permitir o cálculo eficiente da PVN de um jogador, o servidor utiliza a estrutura de dados *Binary Space*

Tree [56] para representar o mapa virtual do jogo.

O uso de uma árvore BSP faz com que apenas os elementos visíveis pelo jogador sejam considerados na construção da resposta. Elementos separados do jogador por obstáculos como, por exemplo, paredes e portas são desconsiderados na construção de uma mensagem de resposta.

Isso determina a importância que o desenho do labirinto que define o cenário utilizado tem para o desempenho da execução da simulação.

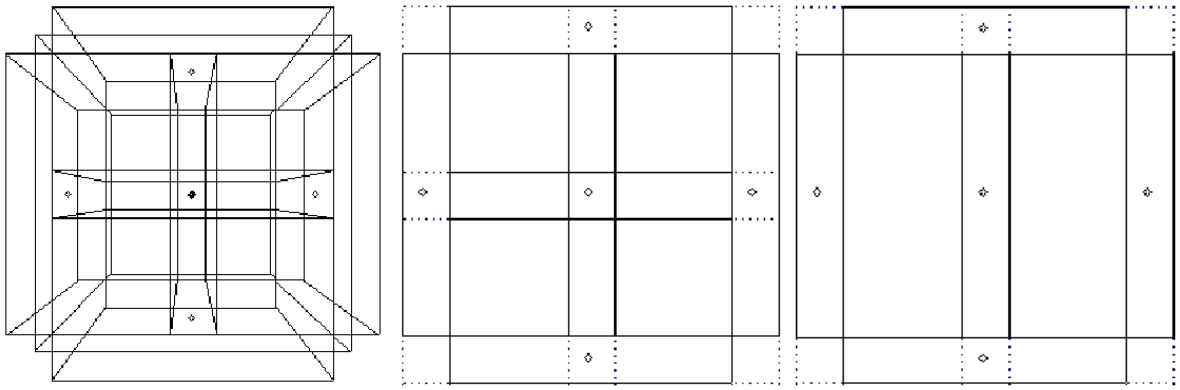
Se o mapa for composto por grandes áreas abertas, ou seja, se definir poucos obstáculos como paredes ou portas, a estrutura da árvore BSP gerada por esse mapa será mais simples e, por isso, a computação desta etapa será menos intensa. Entretanto, o tamanho das mensagens será maior, uma vez que mais entidades serão consideradas para inclusão na mensagem de resposta.

Por outro lado, mapas formados por áreas menores, compostos por pequenas salas ou corredores, tornam o processo de criação da possível área de alcance do jogador computacionalmente mais difícil. Porém, o conjunto de entidades contidos na área será muito menor e, por isso, o tamanho das mensagens será menor. Os dois primeiros trabalhos de Abdelkhalek [4, 5] trazem um estudo aprofundado sobre o impacto da utilização de mapas de diferentes tamanhos e desenhos no desempenho do servidor.

5.2.3. Concentração de usuários em áreas específicas

A configuração do labirinto virtual e os objetivos de cada cenário podem induzir a existência de áreas com diferentes graus de importância no mapa virtual do jogo. Áreas de maior importância tendem a concentrar um maior conjunto de jogadores. Essa concentração pode ser explicada pelo desenho do labirinto ou pelas entidades definidas pelo cenário.

O desenho do labirinto pode propiciar o aparecimento de uma área de maior importância. Por exemplo, suponha um mapa virtual composto de dois andares, onde não há forma do jogador voltar para o andar superior. Com o decorrer da simulação, o andar inferior se tornará



(a) Visão 3D (b) Visão Frontal (c) Visão Superior

Figura 5.1.: Representação estrutural do mapa criado

uma área de maior importância.

Do mesmo modo, o objetivo do cenário pode criar áreas mais importantes. Por exemplo, há uma modalidade do jogo Quake denominada *capture-the-flag* (capture a bandeira) [26]. Nessa modalidade, os jogadores são divididos em dois times. Cada time possui uma área que deve proteger e uma bandeira. O objetivo do jogo é capturar a bandeira do time adversário e trazê-la para a sua área. Nesse caso, tanto as áreas que devem ser protegidas quanto a localização atual de cada uma das bandeiras tornam-se áreas que concentram a maior parte dos jogadores.

O modelo de paralelização proposto por esse trabalho não trata áreas de concentração como casos especiais. Por isso, áreas de concentração podem levar à criação de grupos grandes de entidades próximas e, possivelmente, a algum desbalanceamento de carga.

Para analisar o desempenho da implementação sem a questão das áreas de concentração, criamos um mapa para o jogo Quake que não contém áreas especiais. O mapa consiste de um cubo, dividido por duas paredes que o dividem transversalmente e longitudinalmente. Não há entidades que definam metas no mapa. A Figura 5.1 mostra a representação estrutural do mapa criado.

Os resultados obtidos com a utilização dos diferentes mapas será discutido na Seção 5.3.

5.3. Distribuição dinâmica de carga

Para analisar os resultados obtidos com a metodologia de distribuição de carga é necessário entender qual o efeito que o mapa utilizado pelo servidor exerce sobre a criação dos grupos de entidades próximas e como estes se caracterizam.

5.3.1. Áreas de concentração de mapas

Realizamos experimentos com um dos mapas disponibilizados pela id Software (*death32c* [27]) para jogos do tipo *deathmatch*. No ambiente de testes Solaris, a execução mostrou um grande desbalanceamento de carga em um dos processadores, como mostra a Figura 5.2(a).

Investigações através de instrumentação do código-fonte mostraram que em todo quadro de simulação havia um grupo com um número de elementos muito maior do que o número de elementos dos demais grupos. A existência de tal grupo apontou a possível existência de uma área de concentração de usuários no mapa utilizado para os experimentos, conforme explicado na Seção 5.2.3.

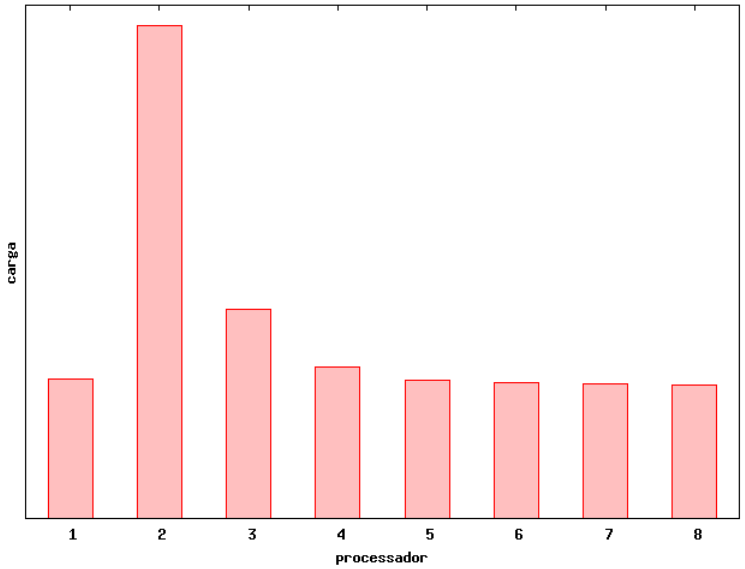
Para validar a hipótese de existência de uma área de concentração, repetimos os experimentos no mapa que denominamos *cubo*. Esse mapa, por construção, não possui áreas de concentração. Os resultados obtidos são apresentados de forma normalizada na Figura 5.2(b), mostram que, de fato, o desbalanceamento de carga era causado pelo mapa utilizado.

5.3.2. Caracterização dos grupos

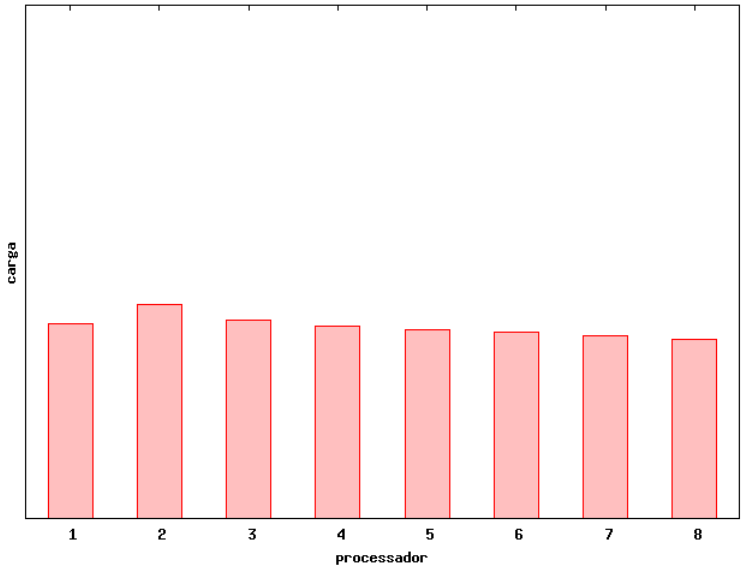
Modificamos o código do Quake a fim de colher estatísticas sobre a quantidade e tamanho dos grupos gerados durante a execução do servidor.

Utilizando o mapa *death32c*, realizamos uma análise da criação dos grupos na versão seqüencial do servidor. Os resultados mostram que apenas 4 grupos são criados em cada quadro de simulação. Cada grupo, por sua vez, possui uma média de 7 entidades por grupo.

A execução da versão paralela do servidor mostra um resultado diferente. A execução



(a) Distribuição de tarefas no mapa *death32c*



(b) Distribuição de tarefas no mapa *cubo*

Figura 5.2.: Distribuição das tarefas entre os processadores

no ambiente Linux, com a utilização de um processo coordenador e um processo auxiliar, resultou na criação de 9 grupos por quadro de simulação onde cada grupo possuía, em média, 8 entidades.

Tal diferença ocorre por causa da diferença no tamanho dos quadros de simulação entre as versões paralela e seqüencial, como é explicado na Seção 5.4.

Utilizando o mapa *cubo* no servidor paralelo, no mesmo ambiente de testes, verificamos que em média são criados 20 grupos de entidades próximas que possuem uma média de 4 entidades por grupo.

A diferença entre as execuções com mapas diferentes são explicadas pelo desenho do mapa. Como não há áreas de concentração no mapa *cubo*, os usuários estão melhor distribuídos no espaço e, portanto, existe a tendência da criação de grupos menores.

5.4. Métricas de desempenho

5.4.1. Motivação

Durante o decorrer da pesquisa percebemos a necessidade de definir métricas para a análise de desempenho do servidor. Anteriormente utilizávamos apenas nossas medições da taxa de utilização da CPU para determinar se o servidor estava utilizando toda a capacidade de processamento disponível. Entretanto, essa métrica despreza os requisitos funcionais do servidor, como, por exemplo, a qualidade de serviço.

Tradicionalmente [52, 12, 44], servidores web são avaliados de acordo com sua vazão (*throughput*), isto é, o número de requisições atendidas por segundo. Optamos por analisar o servidor do Quake através de duas métricas: quantidade de quadros processados por segundo e quantidade de mensagens enviadas por quadro de simulação.

O número de quadros de processamento realizados por segundo pelo servidor nos dá informações sobre a velocidade com que a simulação é realizada e sobre a latência da comunicação entre clientes e servidor. Um número pequeno de quadros por segundo pode indicar a

saturação do servidor.

O número de mensagens enviadas pelo servidor por quadro de simulação indica qual a vazão (*throughput*) do servidor. Nos permite avaliar, também, qual o número médio de clientes que são processados em cada quadro do servidor.

Todos os testes apresentados nesta seção foram realizados utilizando-se 160 clientes simultâneos.

5.4.2. Análise da versão seqüencial

Avaliamos, primeiramente, o desempenho da versão original do Quake. A execução no ambiente de testes Linux mostrou que são executados 2.250 quadros por segundo, mas que são enviadas apenas 0,35 respostas por quadro de simulação do servidor.

O baixo número de respostas é conseqüência do mecanismo de controle de utilização de banda de rede empregado pelo servidor. Esse mecanismo evita que os clientes recebam muitas mensagens em um período muito curto de tempo. Com a utilização de redes mais rápidas, a velocidade com que as mensagens são enviadas e recebidas pelo cliente é maior e, por isso, há um aumento no número de mensagens trocadas entre cliente e servidor. Isso leva o mecanismo de controle de banda a impedir o envio de muitas das mensagens e faz com que alguns quadros de simulação sejam executados sem que nenhuma mensagem seja enviada ao seu final.

Modificamos o servidor original para que não fosse utilizado esse controle. Nesse caso, o número de quadros por segundo cai para cerca de 2.050, porém o número de mensagens por quadro sobe para 2. A queda no número de quadros por segundo se deve ao esforço extra do servidor para o envio das mensagens.

5.4.3. Custo do cálculo de grupos

A fim de avaliar o impacto do esforço de criação dos grupos no desempenho do servidor, executamos a versão paralela de forma que os grupos fossem calculados, mas que não fosse

criado nenhum processo auxiliar. Dessa maneira, toda a simulação ocorre em um mesmo processador. O número de quadros de simulação por segundo caiu para 420. O número de mensagens por quadro, porém, aumentou para 10.

Para entender o motivo da queda de 2.050 para 420 quadros por segundo, é necessário entender a execução de uma sessão do jogo Quake como sendo a execução de uma simulação distribuída.

Ao mesmo tempo em que o servidor está executando a simulação de um quadro, os clientes estão processando as mensagens recebidas, capturando a entrada dos usuários e enviando novas mensagens para o servidor.

Com o custo extra de processamento ocasionado pela criação de quadros, há mais tempo para o servidor acumular mensagens durante o processamento de um quadro. Portanto, o aumento do tempo de simulação de um quadro acarreta em um acúmulo de mensagens para o próximo quadro, o que implica no aumento da quantidade de tarefas que um quadro de simulação precisa realizar.

O aumento na quantidade de tarefas realizadas por cada quadro de simulação explica tanto a queda do número de quadros por segundo, como o aumento no número de mensagens por quadro. Com o aumento do número de mensagens tratadas em um mesmo quadro, ocorre o aumento da quantidade de entidades ativas no quadro de simulação e, conseqüentemente, um aumento no tamanho de cada grupo de entidades próximas, como verificamos na Seção 5.3.2.

O impacto do acúmulo de mensagens antes do processamento delas foi estudado por [12] e comentado na Seção 2.3.1.

5.4.4. Resultados iniciais

Os primeiros experimentos com a versão paralela em ambientes multiprocessados não produziram os resultados esperados. Como explicado na Seção 4.4, técnicas empregadas pelos escalonadores de processos dos sistemas operacionais modernos privilegiam a execução de pro-

cessos criados através da chamada de sistema `fork()` em um mesmo processador a fim de aproveitar melhor a afinidade de *cache* entre processos e processadores.

Os resultados experimentais com a primeira implementação da versão paralela refletem essa característica dos sistemas operacionais testados.

A Figura 5.3 mostra os dados das métricas de desempenho coletadas em simulações com 160 clientes, utilizando-se 2, 4, 6 e 8 processadores. Os resultados deixam claro que a criação de processos auxiliares apenas aumenta o tamanho de cada quadro, devido ao custo de criação dos processos auxiliares. Devido ao aumento no tamanho de cada quadro, o número de mensagens processadas também aumenta.

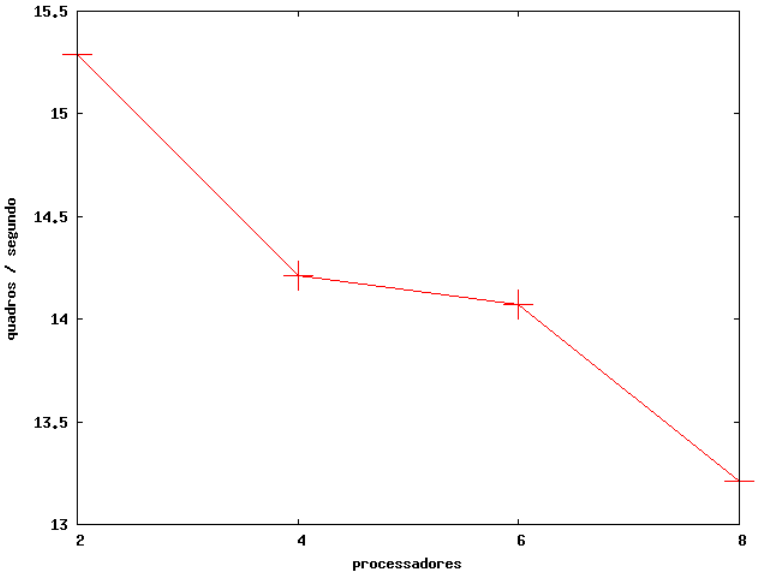
No ambiente Linux, os testes com dois processadores mostraram que o servidor executa 95 quadros por segundo e envia, em média, 33 pacotes por quadro.

A prova cabal de que o escalonador de processos estava afetando o nível de paralelismo do servidor se deu com a análise dos traços de execução através do *Linux Trace Toolkit*. O traço gerado pelo LTT deixou claro que apenas uma das duas CPUs disponíveis estava sendo utilizada durante a execução do experimento.

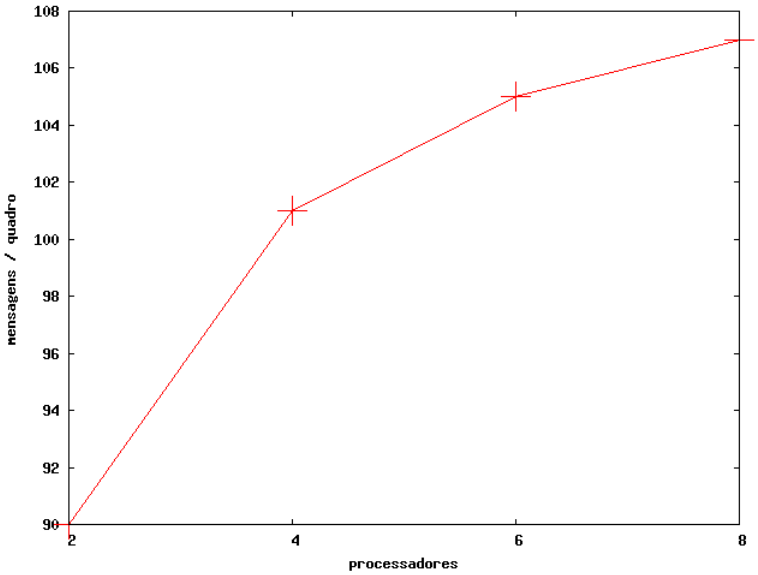
5.4.5. Resultados da versão final

As adaptações descritas na Seção 4.5 foram realizadas utilizando recursos específicos para o sistema operacional Linux e, por esse motivo, os testes não puderam ser repetidos no ambiente Solaris.

Para garantir a precisão dos resultados, repetimos cada experimento com a versão final pelo menos 30 vezes. O Teorema Central do Limite [48], conhecido teorema da estatística básica, garante que a função de distribuição de probabilidade dos valores amostrais médios é, aproximadamente, uma função de distribuição de probabilidade Normal. Ou seja, para um conjunto de n amostras com média μ e desvio padrão σ, então: $\overline{X} \sim N(\mu, \frac{\sigma}{\sqrt{n}})$. Dessa forma, podemos garantir com 95% de certeza que o valor real da média dos dos resultados está no



(a) Quadros executados por segundo



(b) Pacotes enviados por quadro

Figura 5.3.: Métricas de desempenho coletadas no ambiente Solaris

intervalo $[\mu - 1,96\frac{\sigma}{\sqrt{n}}, \mu + 1,96\frac{\sigma}{\sqrt{n}}]$. Todos os valores apresentados nesta seção correspondem à média dos valores obtidos.

A análise da versão com a nova implementação de criação de processos auxiliares – descrita na Seção 4.5.1 – mostrou bons resultados. A quantidade de quadros executados por segundo subiu de 95 para 198 quadros por segundo. A taxa de mensagens tratadas caiu de 33 para 18 pacotes por segundo.

Os traços da execução, porém, demonstraram que a utilização das CPUs disponíveis ainda era ineficiente. A destruição do processo filho correspondia a cerca de 30% do tempo total de execução do processo auxiliar. Além disso, durante toda a destruição do processo, o sistema rodava seqüencialmente, uma vez que ao executar o método `wait()`, o processo pai fica suspenso até que todos os recursos utilizados pelo sistema operacional sejam liberados.

A versão adaptada para destruição paralela dos processos, descrita na Seção 4.5.2, apresentou um melhor desempenho. O número médio de quadros por segundo aumentou para 350 (com desvio padrão $\sigma = 6,12$) e a taxa de mensagens simuladas foi de 11 mensagens por quadro de simulação ($\sigma = 0,64$).

A Tabela 5.1 mostra um sumário com os resultados.

Implementação Avaliada	quadros/s	mensagens/quadro	mensagens/s
versão com `fork()` e `wait()`	95	33	3.135
versão com `clone()` e `wait()`	198	18	3.564
versão com `clone()` e sem `wait()`	350	11	3.850

Tabela 5.1.: Resultados das medições de desempenho da versão paralela do servidor do Quake no ambiente Linux

5.4.6. Análise do traço de execução

A análise do traço de execução mostra dados interessantes sobre a versão final, testada no ambiente Linux.

Todos os traços de execução apresentados nesta seção foram colhidos por 30 segundos. O

início da coleta do traço se dava assim que os 160 clientes terminassem de se conectar ao servidor.

O traço da com o mapa *death32c* mostra que o custo total de utilização de CPU com a cópia das páginas de memória, ocasionadas principalmente pelo mecanismo de *copy-on-write* do Linux, corresponde a 4,69% do total de execução. O custo com a execução da chamada de sistema `clone()` – correspondente ao custo da criação de um novo processo, sem contar o custo das cópias de página de memória – corresponde a 2,12% do tempo total da execução.

A análise do traço também fornece dados sobre a paralelização efetiva da versão final do servidor paralelo do Quake.

Em função do paralelismo, podemos dividir o processamento de um quadro de simulação em 4 fases.

Na primeira fase, o quadro de simulação acaba de iniciar e executa em paralelo com a destruição do processo auxiliar utilizado pelo quadro anterior. Esta etapa corresponde a 17,93% ($\sigma = 0,51\%$) do tempo total de execução do quadro de simulação.

Durante a segunda fase, o processo pai termina de executar a etapa seqüencial do servidor do Quake. Nesta etapa não há paralelismo e são consumidos 21,66% ($\sigma = 0,67\%$) do tempo total de execução do quadro.

A terceira fase corresponde ao tratamento paralelo da fila de eventos, até que o primeiro processo auxiliar termine de processar todas as suas tarefas. Esta fase é paralela e corresponde a 37,69% ($\sigma = 0,42\%$) do tempo de execução do quadro.

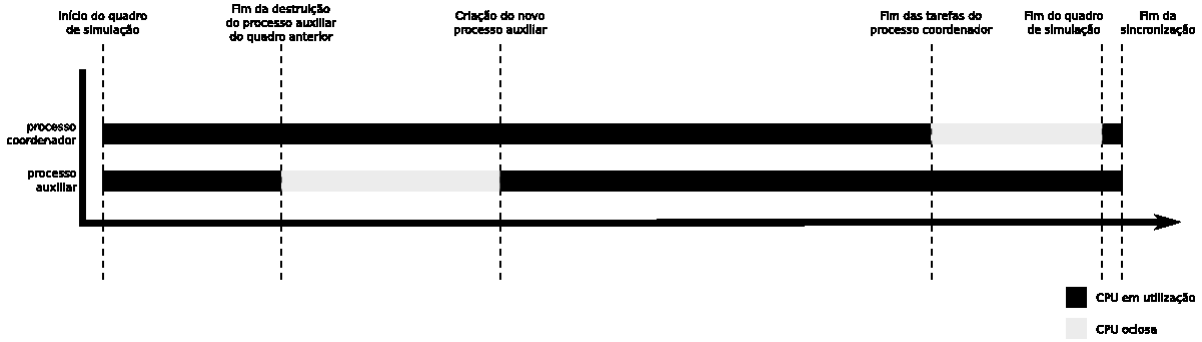
Por fim, a última fase compreende o término das tarefas do último processo auxiliar, somada a etapa de sincronização global entre os processadores. Esta fase é seqüencial e consome 22,70% ($\sigma = 0,67\%$) do tempo total de execução do quadro de simulação.

Temos, portanto, a utilização das duas CPUs disponíveis em 55,65% do tempo.

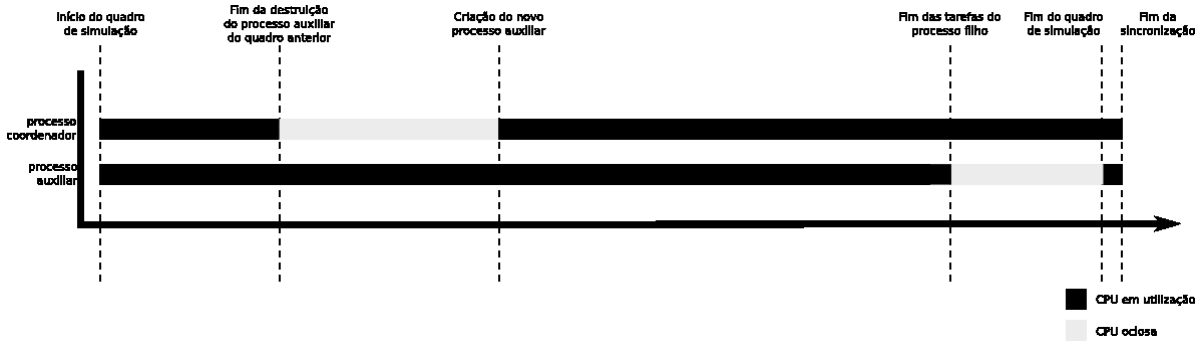
A análise de uma sessão utilizando o mapa *cubo* mostrou que 5,05% do tempo total de utilização de CPU foram gastos com o tratamento de falhas de páginas. As chamadas à `clone()` corresponderam a 1,49% do tempo total da execução.

Quanto às taxas de utilização de CPU, 19,13% ($\sigma = 0,83\%$) do tempo foi gasto na primeira fase, 22,77% ($\sigma = 0,97\%$) na segunda fase, 34,43% ($\sigma = 0,63\%$) na terceira fase e 23,65% ($\sigma = 0,97\%$) na última fase. No total, há utilização simultânea das duas CPUs disponíveis em 53,56% do tempo.

A Figura 5.4 mostra o diagrama de Gantt que representa dois dos quadros de simulação observados durante a execução da versão paralela do servidor do Quake. Estão indicadas na figura, também, as fases de paralelização que caracterizam a versão final do servidor.



(a) Exemplo de quadro onde o processo coordenador termina primeiro de processar seus eventos



(b) Exemplo de quadro onde o processo filho termina primeiro de executar seus eventos

Figura 5.4.: Distribuição do processamento entre as CPUs disponíveis

6. Conclusão

6.1. Comentários finais

Neste trabalho, estudamos arquiteturas de sistemas de computação que almejam a criação de serviços que consigam atender um grande número de usuários simultâneos sem que apresentem degradação em seu desempenho. Particularmente, estudamos técnicas e questões relacionadas ao sistema operacional que permitem a utilização eficiente dos recursos computacionais disponíveis.

Das técnicas estudadas, apresentamos aquelas que permitem que sistemas baseados em eventos executem de forma eficiente em ambientes multiprocessados. Tais técnicas abordam arquiteturas de *software* que permitem utilização eficiente de todos os processadores disponíveis e métodos que avaliam o custo de algumas operações com o sistema operacional e que propõem maneiras de amenizá-las.

Baseados nessas técnicas, desenvolvemos um modelo para a paralelização do jogo interativo, multi-usuário, Quake. A análise do funcionamento do Quake mostrou que o jogo nada mais é do que um sistema de simulação distribuída, onde os eventos de entrada são gerados pelos jogadores conectados a uma sessão da simulação e enviados para serem processados pelo servidor. O servidor realiza a simulação do modelo físico dos objetos e do modelo do jogo e consolida o novo estado da simulação. O novo estado é, então, enviado novamente para os clientes. A simulação dos modelos físico e lógico e o tratamento de mensagens e envio de respostas é realizado utilizando-se uma arquitetura baseada em eventos.

Abdelkhalek propôs um modelo de paralelização de granularidade fina com atribuição estática de tarefas. Apesar do modelo de proteção contra modificações concorrentes levar em consideração o posicionamento das entidades, a distribuição de tarefas entre os processadores disponíveis não considera as modificações no posicionamento das entidades ao longo da simulação. Isso aliado ao controle fino de concorrência, levou a graves problemas no desempenho. Contenção de *locks* e desbalanceamento de carga fizeram com que a versão inicial de sua implementação tivesse alguma CPU ociosa em até 70% do tempo total da simulação.

Neste trabalho propusemos um novo modelo de paralelização para o servidor do Quake que utiliza a semântica da simulação para realizar a distribuição dinâmica de carga entre os processadores disponíveis. Utilizamos o conceito de grupos de entidades próximas para agrupar as entidades que estão próximas entre si. Qualquer ação realizada por um elemento contido em um grupo só poderá causar algum efeito colateral em um elemento contido no mesmo grupo.

Dessa forma, foi possível simular cada um dos grupos de entidades próximas em processadores diferentes, sem que fosse necessário nenhum tipo de sincronização de acesso concorrente durante o processamento dos eventos de um quadro de simulação. Nossa implementação cria um processo por processador disponível, distribui os grupos de entidades próximas que possuem entidades ativas entre os processos criados e, a partir daí, cada processo interpreta as mensagens recebidas pelo servidor e constrói as mensagens de resposta. Cada processo funciona como uma instância independente do servidor.

A comunicação inter-processos ocorre em dois momentos: durante a criação e imediatamente antes da destruição dos processos auxiliares. Nossa implementação utiliza-se da semântica de *copy-on-write* de cópia de páginas de memória que os sistemas operacionais baseados no Unix apresentam quando processos são criados através da chamada de sistema `fork()`. Após a criação dos processos auxiliares, são copiadas para os processos auxiliares apenas as páginas de memória que sofrerem alguma modificação. Conseguimos, assim, evitar a cópia de informações que são utilizadas apenas para leitura e das que não são utilizadas em nenhum momento do processamento dos eventos de um determinado quadro. Nossos experimentos mostraram que

o tempo de CPU gasto com essas cópias de páginas corresponde a, aproximadamente, 5% do tempo total de execução e o custo com a criação de processos auxiliares corresponde a cerca de 2% do tempo total de execução.

Os primeiros experimentos mostraram resultados aquém dos esperados. Dois foram os principais motivos: cenário da simulação e questões relacionadas ao escalonamento de processos em sistemas operacionais que aplicam o conceito de afinidade entre processos e processadores.

Mostramos que o modelo de distribuição dinâmica de cargas proposto por este trabalho é afetado diretamente pela existência de áreas de concentração do cenário, ou seja, áreas de maior interesse para os jogadores ou que, devido ao desenho do labirinto, acumulam maiores quantidades de jogadores. Mostramos, assim, que o desenho do labirinto e o tipo de jogo afetam diretamente o desempenho dessa classe de aplicação.

Descobrimos, também, que a simulação realizada pelo Quake determina que modificações no tempo de duração de cada quadro de simulação do jogo impliquem em variação na carga de trabalho realizada em cada quadro de simulação. Quanto maior for o quadro anterior, maior será o número de mensagens acumuladas para processamento no quadro seguinte.

Além disso, percebemos que aplicações que criam processos auxiliares de curta duração (via `fork()` ou `clone()`) têm seu desempenho afetado por sistemas operacionais que implementam políticas de escalonamento de processos que privilegiem a execução inicial no mesmo processador para aproveitar melhor a afinidade existente entre o processador e o processo pai.

Criamos uma metodologia de criação e destruição de processos que se adequa melhor a execução da implementação da metodologia de paralelização proposta ao sistema operacional. A utilização dessa metodologia permitiu que o servidor executasse 390 quadros por segundo, sendo capaz de servir cerca de 3.850 mensagens por segundo.

Por fim, a análise dos traços de execução revelaram que foi possível manter o paralelismo do servidor em cerca de 55% do tempo total de execução.

6.2. Trabalhos futuros

A análise do modelo de paralelismo foi profundamente prejudicada pela limitação de 160 clientes imposta pelo protocolo de troca de mensagens do Quake. O pequeno número de clientes e, conseqüentemente, o pequeno número de grupos de entidades simulados por quadro de simulação fez com que cada processo auxiliar se ocupasse de poucas entidades por quadro de simulação.

Uma solução seria reescrever o protocolo de troca de mensagens do servidor. Para a realização desse projeto, seria necessário remodelar o protocolo para que aceitasse campos com valores maiores ou de tamanho variável. Seria necessário, também, estudar todos os objetos escritos em QuakeC para que as entidades que enviam mensagens passem a respeitar o novo protocolo.

Outra forma de estudar os efeitos de um número maior de jogadores simultâneos seria a criação de um sistema simulador que abstraísse a carga de execução (*workload*) e simulasse os efeitos do processamento de um número maior de entidades em ambientes multiprocessados, sem que fosse necessário levar em consideração detalhes sobre o processamento do servidor. Essa abordagem é utilizada em outros trabalhos sobre escalabilidade de jogos como [57] e [58].

O método de comunicação inter-processos proposto explorou o mecanismo de *copy-on-write* dos sistemas Unix baseados no *System V* para copiar as informações computadas durante a fase seqüencial do servidor. Para utilizar esse mecanismo, o servidor cria um novo processo auxiliar por processador disponível em cada quadro do servidor. Consideramos que seria interessante comparar esse mecanismo de comunicação inter-processos com uma implementação que fizesse a cópia explícita das informações e utilizasse um *pool* de processos ou *threads*. Gostaríamos de avaliar os ganhos que o mecanismo de *copy-on-write* produz e a sobrecarga devido a criação dos processos.

A versão final adaptada explorou aspectos de gerenciamento e criação de processos específicos do sistema operacional Linux. Outros sistemas operacionais poderiam ser explorados. Parti-

cularmente, o sistema operacional K42 [1] seria adequado para a realização de experimentos futuros.

K42 é um sistema operacional para fins de pesquisa sendo desenvolvido no IBM T. J. Watson Research Center. O K42 possui uma inovadora estrutura interna que oferece ao usuário a interface do Linux [7]: binários de aplicações que executam no Linux executam no K42 sem modificações. O sistema operacional K42 foi projetado para multiprocessadores de 64 bits com memória compartilhada.

Além disso, escalabilidade é um dos pontos cruciais deste sistema operacional, que foi concebido para máquinas com centenas (ou milhares) de processadores. Para que isso seja possível, não são utilizadas estruturas de dados ou *locks* globais pois constatou-se que estes são usualmente gargalos para se atingir escalabilidade nas implementações tradicionais do Unix. Resultados experimentais e comparações entre o Linux e o K42 foram apresentados em [8].

Duas características do K42 seriam particularmente úteis para uma futura investigação: (1) a infra-estrutura para rastreamento de execução [67] que permite um monitoramento sofisticado do comportamento dos *locks* do sistema operacional, e (2) muitos dos serviços, que usualmente estão presentes no *kernel*, são implementados no K42 em espaço do usuário, tornando muito simples fazer otimizações como as propostas por Rosu em [54].

A. Glossário

A seguir listamos e definimos alguns termos técnicos que foram utilizados em inglês no decorrer da dissertação.

accept queue: é a fila mantida pelo sistema operacional com as conexões que foram estabelecidas, mas que ainda não estão associadas a um soquete (e, conseqüentemente, não possuem um descritor de arquivos). A chamada de sistema `accept()` remove a conexão da fila, cria e devolve para a aplicação um soquete que representa a conexão.

Assymmetric multi-process event-driven: nome da arquitetura de sistemas descrita na Seção 2.1.4.

broadcast: mensagens enviadas em *broadcast*, no contexto do Quake, são mensagens enviadas para todos os participantes de uma sessão do jogo.

bytecode: formato do código intermediário gerado pelo compilador da linguagem QuakeC, que é interpretado pelo servidor do Quake para simular o modelo do jogo.

callback: métodos de *callback* são os métodos executados pelo despachante de eventos na ocorrência de um determinado evento.

copy-on-write: também conhecido pela sigla COW, *copy-on-write* é uma estratégia de cópia postergada, onde o utilizador recebe um ponteiro para um determinado recurso. O ponteiro é utilizado somente durante operações de leitura, permitindo que esse recurso

seja compartilhado por diversos utilizadores. No momento em que for necessário realizar alguma operação que possa modificar o recurso, este é copiado para que seja modificado localmente.

CPU : *Central processing unit* ou unidade central de processamento.

event-driven programming: programação baseada em eventos é um paradigma de programação onde o fluxo de execução do programa é definido pelos eventos presentes em sua fila de eventos. Um despachante de eventos lê um evento de sua fila de eventos e executa o método de tratamento apropriado. Os eventos podem ser externos, como novas mensagens no soquete de rede, ou internos, criados durante o processamento de outros eventos.

game engine: componente do servidor de um jogo que é responsável pela simulação física das entidades de um jogo e que define como o modelo do jogo interferirá na simulação.

kernel threads: implementação de *threads* no sistema operacional. Também são conhecidos como processos leves em alguns sistemas operacionais.

locks: mecanismo de sincronização que delimita o acesso a recursos computacionais compartilhados, como memória ou arquivos.

Multi-process: nome da arquitetura de sistemas descrita na Seção 2.1.1.

Multi-thread: nome da arquitetura de sistemas descrita na Seção 2.1.2.

multicast: mensagens enviadas em *multicast*, no contexto do Quake, são mensagens enviadas para um grupo específico de participantes. Mensagens que notificam ocorrência de algum som, por exemplo, são enviadas através de *multicast* apenas para o grupo de entidades que estão próximas à entidade que originou o evento de som.

page view: um *page view* é uma requisição feita para a visualização de uma única página em um servidor web.

QuakeC: linguagem de programação desenvolvida pela id Software utilizada pelos desenvolvedores para definir o modelo de jogo do Quake.

ray tracing: técnica de computação gráfica que utiliza algoritmos para calcular os efeitos de luz na superfície de objetos em um cenário.

round-robin: método de escalonamento que distribui as tarefas entre os recursos em turnos, de forma igualitária e em ordem, sem nenhum tipo de prioridade.

scheduler-tick: interrupção de relógio que notifica o sistema de escalonamento de processos e indica que o escalonamento deve ser recomputado.

Single-process event-driven: nome da arquitetura de sistemas descrita na Seção 2.1.3.

threads: são fluxos de execução de um mesmo programa que podem ser executados simultaneamente e que utilizam o mesmo espaço de endereçamento de memória. Podem ser implementadas em nível de usuário ou pelo sistema operacional.

thread group: em sistemas operacionais que respeitam o padrão POSIX, trata-se de um conjunto de *threads* que compartilham um mesmo número identificador de processo.

throughput: geralmente traduzido como *vazão*, é a taxa com que um computador ou rede recebe ou envia dados.

user-level: termo que referencia eventos e operações que ocorrem fora do sistema operacional.

workload: carga de trabalho que uma aplicação exerce em um sistema computacional.

zombie process: diz-se que um processo está em estado *zombie* quando o processo já foi concluído, mas ainda se encontra na tabela de processos do sistema operacional para que o processo que o criou possa inspecionar seu valor de retorno.

Referências Bibliográficas

[1] The K42 Project. Disponível em: `http://www.research.ibm.com/K42/`. Acesso em: 02/10/2006.

[2] Josh Aas. Understanding the linux 2.6.8.1 CPU scheduler. Disponível em: `http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf`, February 2005. Acesso em: 29/08/2006.

[3] Ahmed Abdelkhalek and Angelos Bilas. Parallelization and performance of interactive multiplayer game servers. In *Proceedings of 18th International Parallel and Distributed Processing Symposium*, page 72a. IEEE Computer Society, April 2004.

[4] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos. Behavior and performance of interactive multi-player game servers. In *Proceedings of the International IEEE Symposium on the Performance Analysis of Systems and Software (ISPASS-2001)*, Arizona, USA, November 2001.

[5] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos. Behavior and performance of interactive multi-player game servers. *Cluster Computing*, 6(4):355–366, October 2003.

[6] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[7] Jonathan Appavoo, Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Freenix*, pages 323–336, San Antonio, TX, EUA, June 2003.

[8] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, Ben Gamsa, Reza Azimi, Raymond Fingas, Adrian Tam, and David Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. Technical report, IBM Research Technical Report RC22863, 2003.

[9] Martin Arlitt and Carey Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ ACM Trans. Networking*, 5(5):631–645, October 1997.

[10] International Game Developers Association. Disponível em: `http://www.igda.org/`. Acesso em: 07/09/2006.

[11] Daniel Plerre Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 3rd edition, November 2005.

[12] Tim Brecht, David Pariag, and Louay Gammo. accept()able strategies for improving web server performance. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pages 227–240, Boston, EUA, June 2004.

[13] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[14] Butterfly.net. Disponível em: `http://www.butterfly.net/`. Acesso: em 09/05/2005.

[15] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable system for consistently

caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, USA, 1999.

[16] Jim Challenger, Arun Iyengar, Paul Dantzig, Daniel M. Dias, and Nathaniel Mills. Engineering highly accessed web sites for performance. In *Web Engineering, Software Engineering and Web Application Development*, pages 247–265, London, UK, 2001. Springer-Verlag.

[17] Comitê Gestor da Internet no Brasil. Indicadores sobre utilização de internet no brasil. Disponível em: `http://www.nic.br/indicadores/`. Acesso: em 19/09/2006.

[18] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A middleware system which intelligently caches query results. In *IFIP/ACM International Conference on Distributed systems platforms*, pages 24–44, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[19] Daniel Dias, William Kish, Rajat Mukherjee, and Renu Tewari. A scalable and highly available web server. In *Proceedings of IEEE COMPCON'96*, pages 85–92, 1996.

[20] Jay Fenlason and Richard Stallman. The GNU profiler (gprof) manual. Disponível em: `http://www.gnu.org/manual/`, 1998. Acesso em: 07/09/2006.

[21] Michael Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, September 1972.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[23] Ronald Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Math*, 17:416–429, 1969.

[24] Mark D. Hill. What is Scalability? *ACM SIGARCH Computer Architecture News*, 18(4):18–21, December 1990.

[25] id Software. Disponível em: `http://www.idsoftware.com/`. Acesso em: 07/09/2006.

[26] id Software. Conjunto de mapas para jogos do tipo *capture-the-flag*. Disponível em: `ftp://ftp.idsoftware.com/idstuff/quakeworld/mods/ctf4/`. Acesso em: 01/10/2006.

[27] id Software. Conjunto de mapas para jogos do tipo *deathmatch*. Disponível em: `ftp://ftp.idsoftware.com/idstuff/quakeworld/maps/`. Acesso em: 12/09/2006.

[28] id Software. Jogo quakeworld. Disponível em: `http://www.idsoftware.com/games/quake/quake/`. Acesso em: 07/09/2006.

[29] Free Software Foundation Inc. GNU General Public License. Disponível em: `http://www.gnu.org/licenses/gpl.html`, June 1991. Acesso em: 21/09/2006.

[30] Opersys Inc. Linux Trace Toolkit. Disponível em: `http://www.opersys.com/LTT/`. Acesso em: 07/09/2006.

[31] Yahoo! Inc. Portal de serviços yahoo! Disponível em: `http://www.yahoo.com/`. Acesso em: 19/09/2006.

[32] Yahoo! Inc. Relatório fincanceiro referente ao segundo quadrimestre de 2006. Disponível em: `http://docs.yahoo.com/docs/pr/`, July 2006. Acesso em: 19/09/2006.

[33] Zona Inc. Terazona whitepaper. Disponível em: `http://www.zona.net/whitepaper`. Acesso em: 09/05/2005.

[34] Carnegie Mellon Software Engineering Institute. How do you define software architecture? Disponível em: `http://www.sei.cmu.edu/architecture/definitions.html`, November 2004. Acesso em: 07/09/2006.

[35] Alexa Internet. Disponível em: `http://www.alexa.com/`. Acesso em: 19/09/2006.

[36] Daniel James, Gordon Walton, Brian Robbins, Elonka Dunin, Greg Mills, John Welch, Jeferson Valadares, Jon Estanislao, and Steven DeBenedictis. 2004 persistent worlds whitepaper. Technical report, International Game Developers Association, 2004.

[37] Alex Jarett, Jon Estanislao, Elonka Dunin, Jennifer MacLean, Brian Robbins, David Rohrl, John Welch, and Jeferson Valadares. Igda online games whitepaper. Disponível em: `http://www.igda.org/online/`, March 2003. Acesso em: 07/09/2006.

[38] Edward Coffman Jr. and Ravi Sethi. A generalized bound on LPT sequencing. In *SIGMETRICS '76: Proceedings of the 1976 ACM SIGMETRICS conference on Computer performance modeling measurement and evaluation*, pages 306–310, New York, NY, USA, 1976. ACM Press.

[39] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *Proceedings of INFOCOM'04*, pages 96–107. IEEE Press, March 2004.

[40] Lauro Eduardo Kozovits. Arquitetura para jogos massive multiplayer. Technical report, Pontifícia Universidade Católica do Rio de Janeiro, 2003.

[41] HP labs. The qprof project. Disponível em: `http://www.hpl.hp.com/research/linux/qprof/`. Acesso em: 01/10/2006.

[42] Thomas Lane. Studying software architecture through design spaces and rules. Technical report, Carnegie Mellon Software Engineering Institute, November 1990.

[43] Jonathan Lemon. Kqueue - a generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 141–153, Berkeley, CA, USA, 2001. USENIX Association.

[44] Eric Levy-Abegnoli, Arun Iyengar, Junehwa Song, and Daniel M. Dias. Design and performance of a web server accelerator. In *INFOCOM (1)*, pages 135–143, 1999.

[45] Davide Libenzi. Linux epoll patch. Disponível em: `http://www.xmailserver.org/linux-patches/nio-improve.html`. Acesso em 16/11/2006.

[46] Robert Love. Introducing the 2.6 kernel. *Linux Journal*, 2003(109):2, 2003.

[47] Robert Love. Kernel korner: Cpu affinity. *Linux Journal*, 2003(111):8, 2003.

[48] Marcos Magalhães and Antônio Carlos Pedroso de Lima. *Noções de Probabilidade e Estatística*. Editora da Universidade de São Paulo, 6a ed. edition, 2005.

[49] David Mazières, Michael Kaminsky, Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 124–139, New York, NY, USA, 1999. ACM Press.

[50] Information Sciences Institute of University of Southern California. RFC 793 – Transmission Control Protocol. Disponível em: `http://www.ietf.org/rfc/rfc793.txt`, September 1981. Acesso em: 25/09/2006.

[51] John Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at 1996 USENIX Technical Conference. Apresentação disponível em: `http://home.pacbell.net/ouster/`, January 1996. Acesso em: 25/09/2006.

[52] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 199–212, California, EUA, June 1999.

[53] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[54] Marcel-Catalin Rosu and Daniela Rosu. Kernel support for faster web proxies. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 225–238, Texas, EUA, June 2003.

[55] Robert Sedgewick. *Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, and Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997.

[56] Carl Shimer. Binary space partition trees. Disponível em: `http://www.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html`, 1997. Acesso: em 07/09/2006.

[57] Amund Tveit. Empirical performance evaluation of the zereal massively multiplayer online game simulator. Technical report, Norges teknisk-naturvitenskapelige universitet, November 2003.

[58] Amund Tveit, Øyvind Rein, Jørgen Iversen, and Mihhail Matskin. Scalable agent-based simulation of players in massively multiplayer online games. In *Proceedings of the 8th Scandinavian Conference on Artificial Intelligence*, pages 80–89, Bergen, Norway, November 2003. IOS Press.

[59] Leslie Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[60] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the thirteenth ACM symposium on operating systems principles*, pages 26–40, New York, NY, USA, 1991. ACM Press.

[61] Shivakumar Venkataraman, Miron Livny, and Jeffrey Naughton. Memory management for scalable web data servers. In *Proceedings of International Conference on Data Engineering*, pages 510–519, 1997.

[62] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pages 189–202, 2001.

[63] Rob von Behren, Jeremy Condit, Feng Zhou, George Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 268–281, New York, NY, USA, 2003. ACM Press.

[64] Robert von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Workshop in Hot Topics in Operating Systems (HotOS)*, pages 19–24, 2003.

[65] Matt Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California at Berkley, 2002.

[66] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.

[67] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

[68] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 13–26, Berkeley, CA, June 2000. USENIX Association.

[69] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazieres, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 239–252, June 2003.