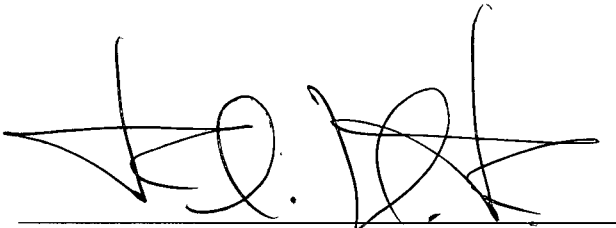
ZOS: UM AMBIENTE PARA MIGRAÇÃO E COMPARTILHAMENTO DE CONTEXTOS DE EXECUÇÃO

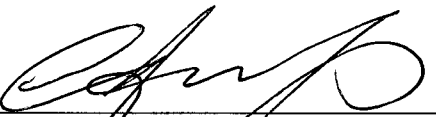
Roberto Francisco Ligeiro Marques

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:



Prof. Felipe Maia Galvão França, Ph.D.



Prof. Claudio Luis de Amorim, Ph.D.

Prof. Eugene Francis Vinod Rebello, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
FEVEREIRO DE 2005

MARQUES, ROBERTO FRANCISCO LIGEIRO

ZOS: Um Ambiente para Migração e Compartilhamento de Contextos de Execução [Rio de Janeiro] 2005

X, 66 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2005)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Sistemas Operacionais

2 - Computação Móvel

3 - Alocação de recursos

I. COPPE/UFRJ II. Título (série)

Como eu queria poder ver seu rosto de orgulho e satisfação agora...
Como eu gostaria que estivesse presente...
Em memória de meu grande incentivador, Divino Ferreira Marques,
meu pai.

Agradecimentos

Agradeço a Deus por ter me dado condições de concluir mais essa etapa de minha vida. A minha grande fonte de vontade de vencer e dedicação, meu grande exemplo de vida, Maria José, minha Mãe. A meu pai, Divino, um grande homem, que infelizmente nos deixou antes da conclusão deste trabalho, mas que sempre me dedicou todo seu esforço, suporte e incentivo. As minhas grandes companheiras de todas as horas, melhores amigas e exemplos de vida, minhas irmãs Ana Martha e Maria José. A minha segunda mãe, minha grande tia, Alzira. Aos meus tios Evaristo e Carlos Henrique e suas esposas por todo o apoio em todos os momentos de minha vida. As minhas tias sempre presentes, Iracema e Conceição. As minhas primas Analzira e Mamá. A minha namorada, Flaviana. A meu grande professor e amigo de todos os momentos, Paulo Leite. Aos professores Carlos Eduardo e José Tavares por toda a juda dispensada. A todos meus grandes e importantes amigos de Caratinga, de faculdade e mestrado que direta ou indiretamente contribuíram neste trabalho, os quais eu não poderia deixar de citar: Getulinho, Leo Delmano, Bruno Klenes, Emanel, André, Thiago Chicão, Flávio, João Camilo, Luiz Felipe, Humberto, Mirayr, Fabrício, Kiko, Nathaniel, Clodoaldo, Daniel, Edter, Leandro, Marcus, Fábio, Marcelo, Fabiano, Jamyr e Henrique. A todos os meus grandes professores que passaram em minha vida acadêmica e deixaram sua contribuição. Ao professor Edil Fernandes, por toda a ajuda dispensada em diversas etapas deste curso de mestrado. E, aos meus grandes orientadores (e amigos) Vítor e Felipe, por toda ajuda, paciência e dedicação dispensadas, sem os quais esse trabalho não teria sido realizado.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc)

ZOS: UM AMBIENTE PARA MIGRAÇÃO E COMPARTILHAMENTO DE CONTEXTOS DE EXECUÇÃO

Roberto Francisco Ligeiro Marques

Fevereiro/2005

Orientadores: Vítor Manuel de Morais Santos Costa

Felipe Maia Galvão França

Programa: Engenharia de Sistemas e Computação

O uso de diferentes tipos dispositivos computacionais tem se tornado cada vez mais rotineiro na vida das pessoas. Nos dias de hoje não é difícil de se encontrar usuários utilizando diferentes equipamentos computacionais em diversas situações (lazer, trabalho, viagens) ao longo do dia. Este tipo de situação geralmente introduz não apenas a necessidade de sincronização de dados entre os diferentes equipamentos, mas também, a constante inicialização de diferentes ambientes de trabalho (a cada novo dispositivo utilizado). E ainda, esse tipo de cenário, não raramente, gera situações de dados inconsistentes em diversos níveis e equipamentos. ZOS (Zombie Operating System) propõe que um usuário deva possuir um contexto de execução (*anima*), que carregue não apenas informações estáticas, como dados de usuários, mas também informações sobre os processos em execução que constituem o ambiente de trabalho de um dado usuário. Idealmente, um *anima* deve estar centralizado em um pequeno servidor pessoal (possivelmente um dispositivo portátil), ou *master*. Em ZOS, dispositivos *master* são capazes de dominar outros dispositivos e assim tomar para si as vantagens dos recursos disponíveis no equipamento em questão, como CPU e interfaces mais poderosas, mais espaço em disco, ou conectividade extra. Este trabalho apresenta o desenvolvimento de um protótipo de ZOS, o qual acreditamos, servirá de base para demonstração de suas idéias centrais e sua viabilidade prática.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

ZOS: AN ENVIRONMENT FOR THE MIGRATION AND SHARING OF EXECUTION CONTEXTS

Roberto Francisco Ligeiro Marques

February/2005

Advisors: Vítor Manuel de Morais Santos Costa

Felipe Maia Galvão França

Department: Computing and Systems Engineering

It is nowadays common to find users that have to use different machines at work, home, and travel. Such users often spend significant amounts of time sychronising and restarting their work environments. Often, they eventually have to cope with inconsistent data at different locations. ZOS (Zombie Operating System) proposes that users should have a main execution context (or *anima*) containing not only the user's data, but also application images. Ideally, the *anima* should reside in a small server, the *master*. In ZOS, masters take over other machines, the *zombies*, and then take advantage of their resources, such as better CPU, better interfaces, more disk, or extra connectivity. We describe our first implementation of ZOS, which, we believe, demonstrates that the idea is practical and worthwhile.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Pode-se considerar que um sistema operacional é a parte de *software* de um dispositivo computacional responsável por gerenciar os recursos de *hardware* deste dispositivo e oferecer mecanismos de acesso a estes recursos.

Observando a evolução dos primeiros dispositivos computacionais, a partir de meados da década de quarenta até os dias de hoje, constatamos que o desenvolvimento dos sistemas operacionais está diretamente ligado ao desenvolvimento das arquiteturas computacionais desenvolvidas em cada período.

Segundo [1] a evolução dos equipamentos computacionais/sistemas operacionais pode ser dividida em fases. Assim, segundo o autor, considera-se que a fase inicial do desenvolvimento de máquinas processadas ocorreu no inicio da década de quarenta até meados da década seguinte. Iniciava-se a pesquisa e o desenvolvimento das primeiras grandes máquinas de calcular valvuladas, impulsionadas principalmente pela II Guerra Mundial. Sistemas operacionais nesta época eram inexistentes, toda programação e acesso aos recursos da máquina eram realizados diretamente pelo operador.

O próximo período, que se inicia em meados da década de cinqüenta e vai até meados da próxima década, é marcado pelo surgimento do transistor, fato que impulsionou o desenvolvimento dos equipamentos computacionais. Neste período surgem os sistemas de processamento em lote, primeiros embriões do que futuramente seria chamado sistema operacional. Estes sistemas se encarregavam de ler e executar comandos contidos em um conjunto de fitas magnéticas. Este conjunto de comandos representavam o modo de operação da máquina e os processos a serem executados pelo dispositivo.

A fase seguinte foi marcada pelo surgimento dos circuitos integrados, o período atravessa a década de sessenta até o final da próxima década. Estes componentes impulsionaram consideravelmente o desenvolvimento dos computadores e conseqüentemente dos sistemas operacionais. Neste período surge o conceito de multiprogramação, o grande projeto nessa área o sistema operacional Multics [2], que mais tarde daria origem ao sistema operacional Unix [3].

A fase subsequente é marcada pelo surgimento dos primeiros computadores pessoais ao final da década de setenta, principalmente com a família de processadores lançada pela Intel a partir da CPU 8088. Os sistemas operacionais passaram então a se tornar fator importante no desenvolvimento dos computadores pessoais. Surgem as primeiras versões do sistema operacional Microsoft DOS e logo após o Microsoft Windows. Alguns sistemas do tipo Unix também tiveram certo desenvolvimento neste período, porém, apenas no inicio da década de noventa (já com o domínio quase absoluto do sistema operacional Windows para computadores pessoais) surge um sistema operacional do tipo Unix que consegue algum espaço dentro do seguimento dos computadores pessoais, o *GNU/Linux*.

Ainda durante a fase dos computadores pessoais, em meados da década de oitenta e inicio da década de noventa, iniciam-se alguns estudos sobre o desenvolvimento de *Clusters* de computadores pessoais. O grande motor desta área de estudo foi tentar obter um ambiente com grande poder computacional a partir de um investimento pequeno se comparado aos preços de supercomputadores. Estes estudos levaram ao desenvolvimento de outra gama de sistemas operacionais; os sistemas operacionais para ambientes distribuídos, os quais podemos relacionar Amoeba [4] e Sprite [5]. Estudos nessa área [6, 7] continuam gerando novas alternativas e busca de novas soluções.

Entretanto, observando o estágio atual de desenvolvimento dos dispositivos computacionais constatamos que novamente estamos próximos à necessidade do surgimento de um novo conceito de sistema operacional, ou, no mínimo, da atualização dos conceitos atuais.

Atualmente vivemos o período do desenvolvimento dos dispositivos portáteis e embutidos. Computadores de mão, celulares, *tablet PCs* são exemplos de dispositivos que possuem um considerável poder de processamento e armazenamento tornando-se, cada vez mais, dispositivos habitualmente usados na realização de trabalhos do dia-a-dia, e.g., consultar/enviar e-mails, escrever textos, apresentações, etc.

E ainda, o grande desenvolvimento dos dispositivos embutidos, espalhados nos mais diversos ambientes como: casas inteligentes e automóveis, e mesmo celulares e *PDAs*, cada vez mais necessitam de acesso a informações de usuários que devem ser compartilhadas entre estes diversos dispositivos, e.g., agenda de telefones que deve estar presente no celular, no *laptop* para realização de uma chamada utilizando voz sobre IP, ou mesmo no telefone fixo da casa que também pode possuir uma conexão IP.

Porém, o que se percebe é que a interação entre esses diversos dispositivos, que deveriam interagir de forma transparente sobre as informações e ambiente de trabalho (computacional) de usuários, ainda se dá praticamente da mesma maneira há alguns anos. Toda a comunicação entre os dispositivos está restrita a sincronização de dados entre dispositivos (e.g., entre um *PDA* e um *Desktop*) ou de forma centralizada através de um servidor (estrutura cliente/servidor), e.g., servidor de arquivos.

Com este projeto apresentamos um conceito novo para gerenciamento de usuários e interação entre dispositivos computacionais. ZOS, *Zombie Operating System*, é um sistema operacional conceitual através do qual buscaremos apresentar novos conceitos relacionados a gerência de recursos computacionais distribuidos entre diversos dispositivos e a interação de usuários sobre este ambiente.

Uma implementação prática destes conceitos também será apresentada. Entretanto, devido a complexidade para o desenvolvimento de um novo sistema operacional e a disponibilidade de tempo e recursos, buscou-se o desenvolvimento de camadas de aplicativos e alterações em um sistema operacional existente para exposição das idéias.

As próximas sessões apresentam a motivação para o desenvolvimento deste sistema, seus conceitos e contribuições dentro da área.

1.1 Motivação

Atualmente é comum encontrar usuários de computador com um PC em casa, uma estação de trabalho no escritório, um laptop para viagens e um *PDA* para trabalhos leves. Este cenário, na maioria das vezes, exige a necessidade de sincronização de dados entre os dispositivos, o que, em alguns casos, demanda tempo e pode ocasionar erros. E ainda, o deslocamento de um dispositivo para outro, geralmente demanda

um certo tempo simplesmente para iniciar o ambiente de trabalho desejado para aquele dispositivo. Notoriamente, o maior problema nesses casos é a sincronização de dados. Alguns sistemas, como [8, 9, 10, 11, 12, 13], apresentam alternativas para sanar este tipo de problema, entretanto, tais ferramentas apresentam algumas deficiências.

A primeira questão está relacionada com a latência do processo em relação o meio utilizado (rede disponível), usuários mais precavidos podem desejar sincronizar todo o disco, tornando o processo demorado. Outro problema diz respeito à proteção e segurança dos dados, tais ferramentas podem levar a quebra de limites de proteção do sistema de arquivos (geralmente de forma não intencional por parte do usuário).

Uma opção às ferramentas de sincronização que vêem ganhando força nos últimos anos é o uso de grandes servidores distribuídos pela internet [14, 15, 16, 17] que armazenariam todos os dados de usuário tornando-os disponíveis de qualquer dispositivo que possua conexão *web*. Entretanto, essa alternativa parte do pressuposto da existência de uma conexão persistente e de alta velocidade para todo o tipo de dispositivo e, além disso, pode gerar problemas relacionados a segurança e privacidade no uso dos dados.

Progressos no desenvolvimento de dispositivos portáteis (nominalmente, quantidade de memória volátil e persistente disponíveis) sugerem uma terceira alternativa. Ao invés de sincronização de dados com grandes servidores ou entre dispositivos, por que não tornar o usuário responsável por seus dados em pequenos servidores móveis? Usuários possuiriam pequenos servidores, idealmente pequenos dispositivos [18, 19], e estes dispositivos seriam capazes de transportar todo o contexto de trabalho do usuário em questão. Note que, o contexto de trabalho de um usuário não está limitado a seus dados pessoais, mas também, o contexto das aplicações que este estiver executando. Assim, um usuário seria capaz de rapidamente transportar seu ambiente de trabalho sem ter que passar pelo lento processo de inicialização do equipamento que deseja utilizar.

Este trabalho está focado neste cenário. ZOS baseia-se na idéia de que um usuário utilizando um dispositivo possui um contexto de trabalho. Nós chamamos este dispositivo de *master*. Um dispositivo *master* é capaz de dominar outros dispositivos, os *zombies*. A execução em um dispositivo *zombie* é controlada pelo *master*: interface gráfica deve ser sincronizada, processos em segundo plano podem ser migrados entre os dispositivos (geralmente de um dispositivo com menor poder

computacional para um de maior poder computacional).

O processo de transformação de um dispositivo em *zombie* deve ser transparente. Para atingir este objetivo devemos desenvolver um protocolo de comunicação, portar aplicativos e/ou um sistema operacional e ainda, considerar questões referentes à segurança e desempenho do sistema.

Desta forma cada usuário teria uma espécie de *Desktop* móvel, um pequeno servidor pessoal que contém todo seu ambiente de trabalho. Este dispositivo seria capaz de se integrar a outros dispositivos computaionais que estiverem a sua disposição de forma que estes dispositivos possam compartilhar e trabalhar sobre o mesmo contexto (e não apenas sincronizar dados). A Figura 1.1 representa graficamente a idéia.

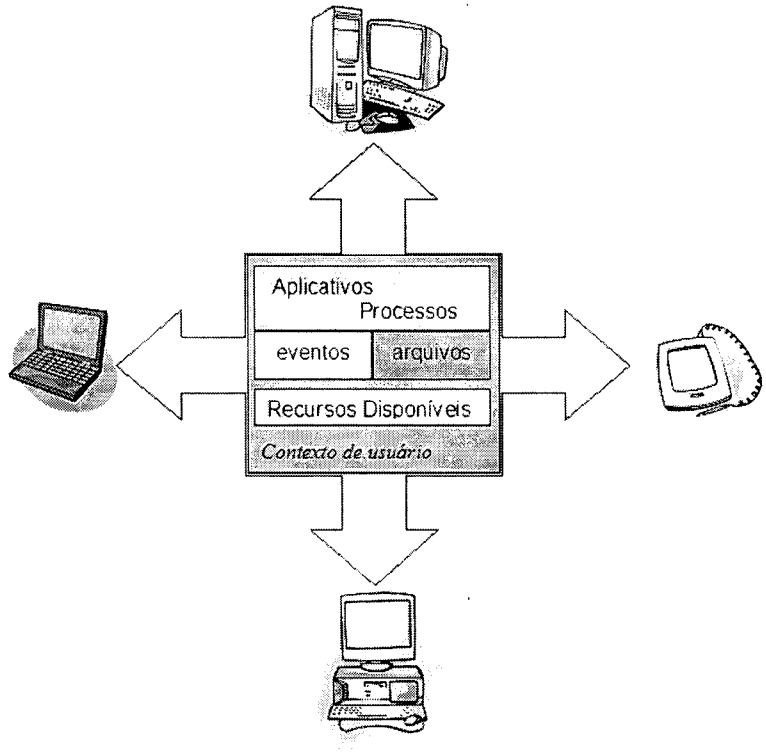


Figura 1.1: Contexto de usuário.

Dentro deste cenário, um usuário trabalhando em seu *PDA* enquanto se desloca de um ambiente a outro (e.g., de casa para o escritório), seria capaz de, ao chegar ao escritório, migrar seu ambiente de trabalho (a partir de seu *PDA*) para seu *Desktop* (ou, dominar o *Desktop*), sem precisar de passar por todo o processo de inicialização da máquina e aplicativos. Os dois dispositivos passariam então a trabalhar de forma integrada e, ao final do domínio, todo o trabalho realizado (dados e estado de

aplicativos) já estariam sincronizados com o dispositivo portátil podendo ser migrado novamente para outro dispositivo.

Nossos objetivos não se restringem simplesmente à migração de processos, como em Condor [7], Zap [20], Nomad [21], entretanto, migração de processos é parte do problema tratado; conforme será demonstrado brevemente, não apenas processos devem ser migrados, mas todo o ambiente de trabalho de usuário deve ser replicado entre diferentes dispositivos. Da mesma forma, ZOS não está restrito simplesmente ao controle de dispositivos remotos, como VNC [22]. O sistema não se restringe a simplesmente controlar a interface de uma sessão remota, mas sim que os dois dispositivos possam interagir sobre um contexto de usuário de maneira sincronizada. Por outro lado, controle de sessões remotas e migração de processos fazem parte do problema a ser tratado e, por opção de projeto, toda a tecnologia disponível será utilizada na medida do possível.

1.2 Contribuições

O desenvolvimento de ZOS e os conceitos apresentados pelo projeto abrangem diversas áreas relacionadas a sistemas operacionais. Acreditamos que tanto a implementação do protótipo do sistema quanto as questões levantadas a partir do desenvolvimento inicial apresentam um conjunto relevante de informações que podem permear o desenvolvimento de novos sistemas operacionais e/ou sistemas voltados para migração e compartilhamento de contextos, dos quais pode-se citar os seguintes tópicos:

- Migração de processos em um ambiente distribuído e compartilhado por usuários.
- Categorização de processos para compartilhamento e migração de contextos.
- Sincronização de interfaces gráficas em um ambiente compartilhado.
- Gerência de recursos e usuários.

1.3 Roteiro

As próximas sessões apresentam ZOS em mais detalhe. Inicialmente serão apresentados os componentes principais que constituem o sistema. A seguir

apresentamos o primeiro protótipo desenvolvido. Após será apresentado alguns resultados de testes de performance. Em seguida faremos uma comparação com trabalhos correlatos e apresentamos nossas conclusões e propostas para trabalhos futuros.

Capítulo 2

Princípios Gerais do Sistema

Neste capítulo apresentamos alguns conceitos importantes para o desenvolvimento do protótipo inicial sistema. Estes conceitos se relacionam à categorização dos dispositivos dentro do sistema, à definição do que constitui um contexto de usuário e dos passos necessários para sua migração e compartilhamento.

2.1 Visão Geral do Sistema

O conceito principal do sistema parte da divisão dos dispositivos computacionais em duas classes distintas:

- Dispositivos Master - Equipamentos que carregam o ambiente do usuário e que são capazes de dominar outros dispositivos. Dispositivos Master possuem o *anima* - contexto de execução de um ambiente de trabalho - de um usuário.
- Dispositivos *Zombie* (ou escravos) - Equipamentos habilitados a serem controlados por dispositivos *master*, por possuírem maior poder computacional, ou melhor conexão, ou por terem acesso a outros sistemas de arquivos (e.g., dados de trabalho, em um escritório); ou simplesmente por possuírem interfaces melhores, como monitor e teclado. Convencionalmente, iremos tratar como dispositivos *escravos* sempre que o equipamento esteja condicionado a ser um *zombie* porém ainda não tenha passado por um processo de domínio, e como *zombie* sempre que o dispositivo já tenha sido dominado por um *master*.

Em resumo, o sistema deve atuar da seguinte forma: dispositivos *master* devem ser capazes de buscar em uma rede por dispositivos escravos os quais eles estejam habilitados a exportar seu *anima*. Este processo é chamado *discovery*.

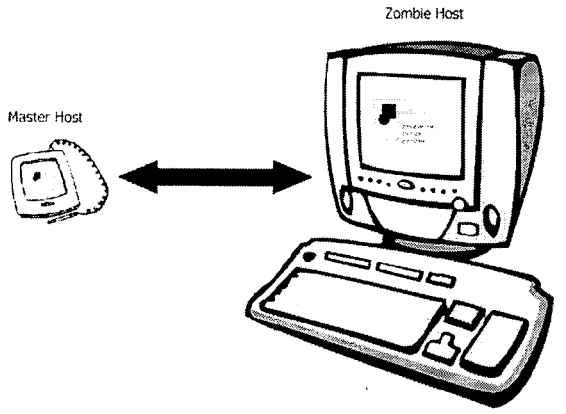


Figura 2.1: Zombie Operating System.

Durante o processo de *discovery*, dispositivos *master* extraem informações a respeito dos *escravos* encontrados e decidem sobre do tipo de domínio possível. A princípio dois tipos de domínio são sugeridos: completo e parcial. Para o domínio completo, dispositivos *master* seriam capazes de tomar o controle total do *escravo*, exportando seu sistema operacional e reiniciando o *hardware*. No controle parcial, a execução continuaria no contexto operacional do dispositivo *escravo*, apenas o contexto do usuário *master* seria exportado.

Assim, ao encontrar um *escravo*, ZOS inicia o seguinte processo:

1. O dispositivo *master* envia algumas informações relevantes sobre seu contexto: partes do sistema de arquivos e aplicativos (incluindo sua interface gráfica).

2. Os dispositivos (*master/zombie*) passam a atuar de forma sincronizada. O que significa que, alterações no contexto de um dispositivo afetam o outro. Desta forma, o usuário passa a ter seu *anima* replicado e sincronizado em outro(s) dispositivo.

Claramente, segurança torna-se um item importante nesse sistema. Inicialmente, autenticação segura de usuários deve ser provida. E ainda, técnicas de criptografia são utilizadas considerando a possibilidade de execução em ambientes não seguros. Confiança também constitui um fator importante: um *escravo* encontrado é realmente um escravo?

Por último, vale ressaltar que um processo de domínio não é persistente, ou seja, pode ser interrompido a qualquer instante pelo dispositivo *master*.

Desta forma, segundo o que foi apresentado, as principais questões levantadas por ZOS são:

1. O que é e como compartilhar um *anima*.
2. Como manter os dados de usuário protegidos.
3. Como garantir uma performance aceitável, pois, de outra forma o sistema não terá utilidade prática.

Inicialmente iremos focar na definição de *anima*. Abstratamente, um *anima* é um conjunto de dados estáticos, ou seja, um conjunto de descritores de arquivos, mais um conjunto de dados dinâmicos, ou, um conjunto de processos e aplicativos em execução. Cada processo ou aplicativo encontra-se em um estado específico o que inclui estados de manipuladores de arquivos e um contexto de interação, geralmente representado por um conjunto de janelas executando sobre um gerenciador de interface gráfica.

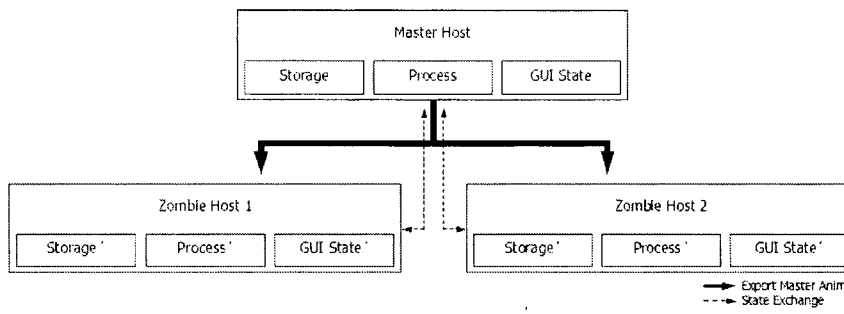


Figura 2.2: ZOS Anima.

Como ponto de partida para discussão a respeito da definição de um *anima* discutiremos algumas questões referentes a processos. Diferentes tipos de processos representam diferentes interações com o sistema, e.g., para processos de segundo plano performance é um ponto chave, por outro lado, para aplicativos de usuário, interatividade e consistência constituem pontos importantes. Assim, acreditamos que uma categorização generalista de processos não se aplicaria bem a ZOS. Em ZOS, processos serão classificados segundo sua interação com o usuário e funcionalidade, confome apresentado a seguir:

- Compartilhados - Processos compartilhados são aqueles que executam simultaneamente no dispositivos *master* e *zombie* de forma sincronizada, desta

forma, alterações no estado de um aplicativo compartilhado alteram o estado do outro e vice-versa. Exemplos incluem aplicativos interativos como x-terminais, navegadores *web* e aplicativos de escritório (*Office-like*). Com esta classe de processos um usuário é capaz de compartilhar seus aplicativos com outros dispositivos trabalhando de outras máquinas de forma distribuída.

- Migrantes - Processos que inicialmente foram disparados em um dispositivo *master* e que, após iniciado o domínio são migrados para o dispositivo *zombie*. Um exemplo típico seria processos de segundo plano que requerem grande poder computacional. Ao iniciar um domínio, processo migrantes deixam os dispositivos *master* em direção aos *zombies* (geralmente máquinas com maior poder computacional), retornando ao final do domínio.

- Latentes - Processos executando isolados do *master* em algum servidor na rede de forma latente e que iniciam atividade ao estabelecerem contato com um dispositivo *master* ou *zombie*. Exemplo seriam processos de *grid* computacional, filtros de *e-mail* e robôs de busca na *web*.

- Kernel - Processos que gerenciam ZOS ou que executam processos críticos de usuário. ZOS possui dois subsistemas, um para os dispositivos *master* outro para os *escravos*, cada um deles possui um conjunto desses processos.

Uma vez estabelicida uma definição inicial de *anima* e, tendo em vista que esta entidade representa o contexto de um usuário (seus dados e aplicativos), apresentamos alguns pontos relacionados a segurança do sistema:

- Sistema de arquivos - O sistema deve controlar como o sistema de arquivos é exportado/importado por dispositivos *master* e *zombies*. Usuários devem ser capazes de definir arquivos/diretórios que desejam exportar e sobre qual contexto.

- Protocolos de comunicação - Informações transientes pela rede devem ser protegidas.

- Confiança - Existe alguém escutando a rede? Estamos deixando alguma informação disponível ao término de um domínio? Um dispositivo *escravo* é realmente o que ele diz ser? Quanto é possível confiar em um *zombie* e, se não podemos confiar, como tratar essa possibilidade?

Em geral podemos assumir um cenário confiável de execução para o sistema. Entretanto, para que o sistema tenha viabilidade prática, deve-se considerar todas as situações, nominalmente, meio de comunicação e dispositivos *escravos* não confiáveis.

Por último, não podemos deixar de considerar questões referentes ao desempenho do sistema, uma vez que o propósito do projeto também contemple a demonstração da viabilidade prática dos conceitos apresentados. Conforme será apresentado, compartilhamento e migração de processos são pesos preponderantes neste quesito, visto que demandam grandes quantidades de troca de dados pelo meio. Assim, uma implementação completa de ZOS deve atacar diversas áreas relacionadas a sistemas operacionais, como: migração de processos [23, 24, 20], computação distribuída [7, 6], *trusted computing* [25, 26].

Neste trabalho iremos detalhar nosso protótipo inicial, o qual acreditamos ser útil para demonstração da viabilidade prática dos conceitos apresentados. Para o desenvolvimento desta primeira versão utilizamos outras tecnologias disponíveis sempre que possível, a fim de minimizar o tempo de desenvolvimento e realização do projeto. As próximas sessões apresentam alguns detalhes conceituais importantes para o desenvolvimento do sistema e, o próximo capítulo apresenta o desenvolvimento do protótipo.

2.2 Decisões de Projeto

Nesta sessão apresentamos algumas questões a serem consideradas para o desenvolvimento do protótipo do sistema. Questões referentes a implementação real do protótipo serão apresentadas no Capítulo 3.

2.2.1 Definições

ZOS assume que vários dispositivos computacionais estejam conectados em uma rede. Estes dispositivos podem ser classificados como:

- Livres - Dispositivos que não estejam atuando como *master* ou *zombie*.
- Conectados - Dispositivos atuando no sistema como *master* ou *zombie*.

Um dispositivo que esteja *livre* pode atuar como *master* ou *zombie* dependendo das conFigurações ativadas pelo usuário para o mesmo. E ainda, é possível que um

dispositivo atue tanto como *master* ou *zombie*, o que pode ser útil para depuração do sistema.

Domínio é o processo o qual transforma um dispositivo conFigurado como *escravo* em *zombie* de algum dispositivo *master*. Um processo de domínio assume que os dispositivos conectados pela rede estão pré-conFigurados como *master* ou *escravos*. Conforme mencionado anteriormente, um domínio pode ser completo ou parcial. Isto depende das permissões que um dispositivo *master* possui para um dado dispositivo *escravo* e, na confiança mutua entre os dispositivos.

Portanto, podemos relacionar *controle X confiança* da seguinte maneira: consideramos que, quanto maior o nível de confiança entre os *hosts master/escravo*, maior o controle disponibilizado ao dispositivo *master* e, consequentemente, maior a integração entre os *hosts*. A Figura 2.3 representa este conceito.

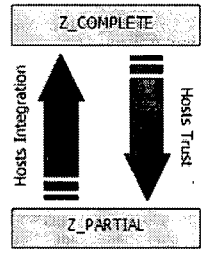


Figura 2.3: Transparência de Controle.

Uma implementação do sistema pode então ser dividida em dois subsistemas: subsistema *master* e subsistema *escravo*. Ambos os subsistemas possuem estrutura similar e trabalham de forma conjunta. Os subsistemas estão divididos em quatro partes relacionadas as suas funcionalidades:

- *Discovery* e inicialização.
- Sistema de Arquivos.
- Gerência de Processos.
- Gerência de Interface gráfica.

Como quesito mínimo de segurança podemos assumir que toda a comunicação entre os dispositivos deve ser criptografada. A Tabela 2.1 apresenta os módulos do sistema graficamente.

Tabela 2.1: Tabela ZOS Módulos.

Security Modules and Functions			
Busca Escravos	Ger. Sist. Arquivos	Escuta Master	Ger Sist. Arquivos
Ger. Processos	Ger. Interface	Ger. Processos	Ger Interface Manager
Master Subsystem		*Zombie Subsystem*	

Módulos de busca e gerenciamento de *zoinbies* (*Slave Discovery* e *Zombie Listening*) iniciam todo o processo e distribuem informações para os demais módulos. A Figura 2.4 apresenta graficamente a comunicação entre os módulos.

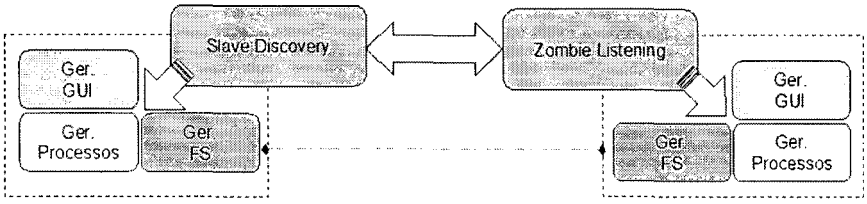


Figura 2.4: Módulos do Sistema.

2.2.2 Gerenciamento de escravos

O processo de busca (*discovery*) e inicialização de dispositivos *escravos* é realizado pelo módulo *SlaveDiscovery* no subsistema *master* e por *ZombieListen* no subsistema *escravo*.

O módulo *SlaveDiscovery* é responsável por realizar o processo de busca pela rede por dispositivos *escravos* e gerenciar o processo de domínio de um dispositivo. O módulo *ZombieListen* executa no sistema do dispositivo *escravo* aguardando por requisições vindas de um dispositivo *master*. Este módulo também é responsável por autenticar o usuário remoto no dispositivo. Estes módulos trabalham conjuntamente durante todo o domínio.

Eventualmente, um usuário solicita o término de um domínio. Este pedido é encaminhado para o módulo *SlaveDiscovery* que por sua vez notifica *ZombieListen*. Ao receber tal notificação, cada módulo *ZombieListen* (um para cada dispositivo dominado), se encarrega de terminar o domínio para o dispositivo em questão. Este processo é realizado em conjunto com os demais módulos e segue os seguintes passos:

1. Desautorizar o acesso ao sistema de arquivos.

2. Retirada de processos.

3. Fechamento do ambiente gráfico.

A limpeza do sistema deve ainda garantir que nenhuma informação do dispositivo *master* possa ser recuperada no dispositivo *escravo*. No caso mais simples, arquivos temporários e de consistência devem ser removidos. Em cenários menos confiáveis, uma varredura do sistema de arquivos e memória deve ser realizada e ainda, o reinicio do dispositivo deve ser realizado.

2.2.3 Gerenciamento do Sistema de Arquivos

O sistema utiliza um sistema de arquivos distribuído, assim, é possível para um usuário definir pontos de seu sistema de arquivos de devem ser compartilhados entre os dispositivos durante um domínio. Usuários também devem ser capazes de utilizar o sistema de arquivos dos dispositivos dominados, e assim, obterem espaço em disco extra ou conectividade a outros equipamentos.

O gerenciador do sistema de arquivos é responsável por iniciar o compartilhamento dos sistemas de arquivos entre os dispositivos, bem como desabilitar o compartilhamento ao final do domínio. E também, controlar o montante dos sistemas de arquivos que podem ser utilizados.

É desejado que o gerenciador do sistema de arquivos seja capaz de atuar de forma sensível ao desejo do usuário, e.g., um usuário pode desejar que um diretório seja exportado para um certo dispositivo *zombie* e não para outros e ainda, alguns diretórios não devem ser exportados em nenhuma circunstância. Também deseja-se que alguns dados não permaneçam nos dispositivos *zombies* após o domínio, e.g., arquivos confidencias ou relacionados ao trabalho não devem permanecer em um dispositivo móvel ao deixar o escritório.

O gerenciador do sistema de arquivos também atua no sistema armazenando informações sobre o estado do sistema.

2.2.4 Gerenciamento de Processos

Todo processo executando em ZOS deve ter um tipo, o qual caracteriza o processo segundo seu comportamento durante um domínio. Conforme mencionado anteriormente, processos em ZOS podem ser classificados como: migrantes, compartilhados, remotos ou *Kernel*.

Processos remotos não serão mencionados por não terem sido implementados na primeira versão do sistema. Processos de *Kernel* são os processos que gerenciam o sistema.

2.2.4.1 Processos Compartilhados

Os chamados processos compartilhados, conforme dito anteriormente, são aqueles que, após inicio de um domínio, permanecem em execução no dispositivo *master* e são duplicados para o dispositivo *zombie*. Estes processos (processo original e réplica) devem manter-se atualizados um em relação ao outro, ou seja, alterações no estado de um processo devem alterar o estado do outro.

Um exemplo típico seria um editor de texto; onde, um texto inserido em um dos processos deve ser replicado no processo espelho. ZOS deve então garantir que o editor esteja aberto, em todos os dispositivos participantes de um domínio, manipulando o mesmo arquivo, e ainda, que todos os eventos recebidos por algum dos processos (neste caso o editor), *mouse*, teclado, etc., sejam disseminados para os demais.

2.2.4.2 Processos Migrantes

Processos migrantes devem ser paralisados no dispositivo *master* e copiados para um dispositivo *zombie* ao inicio do domínio, retornando ao término.

Exemplos desta classe de aplicativos incluem não apenas processos computacionalmente pesados, mas também, como exemplo, processos que possam necessitar de uma conexão mais rápida disponível em um dispositivo dominado. Em nosso caso especifico, gostaríamos de aplicar a técnica em processos de *Machine Learning*, processos normalmente pesados para equipamentos como *laptops*, porém, que podem obter um desempenho melhor em máquinas maiores, mesmo que temporariamente.

2.2.5 Gerenciamento de GUI

Possivelmente um dos mais difíceis processos de se implementar no sistema. Define como as interfaces dos dispositivos devem ser sincronizadas: teclado, mouse, etc. Alterações causadas em um aplicativo compartilhado devem ser retratadas em todos os dispositivos.

Capítulo 3

Implementaçao do Sistema

De forma a demonstrar a viabilidade prática dos conceitos apresentados, foi desenvolvido um protótipo inicial do sistema que bucou permear todos os aspectos descritos no capítulo anterior. Esta sessão apresenta os detalhes desta implementação.

3.1 Protótipo do sistema

O primeiro protótipo do sistema é baseado no sistema operacional GNU/Linux RedHat9, kernel 2.4.20. Um conjunto de *daemons*/processos representam os módulos apresentados anteriormente.

As Figuras 3.1 e 3.2 ilustram a estrutura do sistema. Busca de *escravos*, conFiguração e inicialização de controle são funcionalidades implementadas pela aplicação `zdiscovery` nos dispositivos *master*. O *daemon* `Zombielisten` é responsável por identificação de dispositivos *master* na rede, autenticação de usuários e gerência do sistema de arquivos nos dispositivos *escravos*.

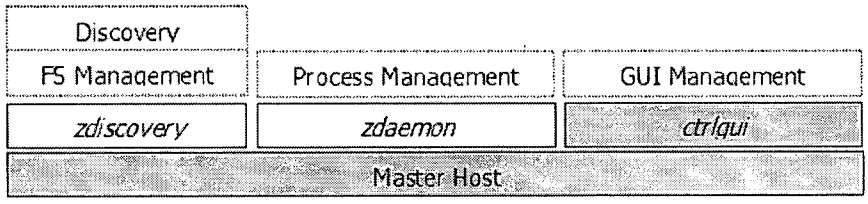


Figura 3.1: Módulos Master.

A Figura 3.3 apresenta um mapeamento entre os módulos apresentados no capítulo anterior e os processos implementados na versão inicial do protótipo.

A gerência do sistema de arquivos é realizada em conjunto pelos módulos `zdiscovery/zombielisten` utilizando primitivas oferecidas pelo sistema de arquivos dis-

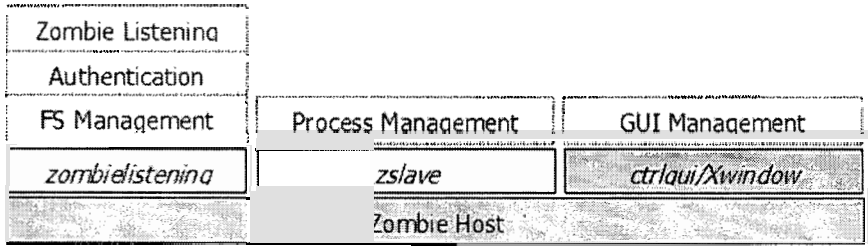


Figura 3.2: Módulos Escravo.

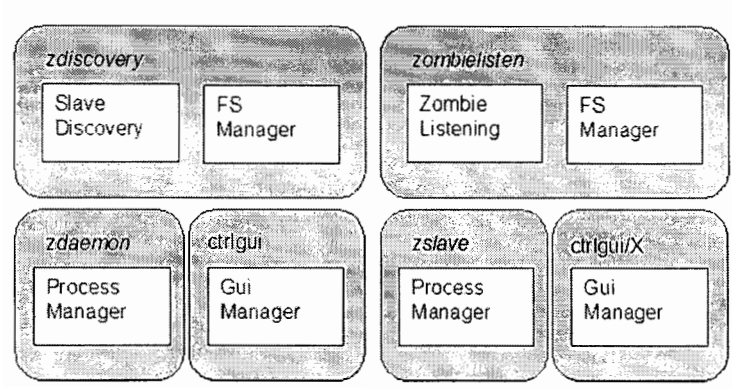


Figura 3.3: Módulos/Implementação.

tribuídos NFS [27].

Gerência de processos é realizada pelo *daemon* `zdaemon` em dispositivos *master* e por `zslave` nos *escravos*. Dispositivos *master* também possuem um processo `ctrlgui` que, em conjunto com o gerenciador de janelas X, executa o sincronismo dos ambientes gráficos entre os dispositivos [28].

3.1.1 Sessões Remotas

O objetivo principal de ZOS é possibilitar que uma sessão de usuário seja aberta sobre uma sessão pré-estabelecida em um *host* remoto. Segundo nossas avaliações, o gerenciador de ambientes gráficos *Gnome* é a melhor opção para se atingir o objetivo desejado. Mais especificamente, para o desenvolvimento do sistema, foi utilizada uma versão alterada do aplicativo `gdmflexiserver` [29]. Este aplicativo permite um usuário abrir uma nova sessão como uma janela, sobre uma sessão previamente estabelecida. Este aplicativo, por sua vez, utiliza um serviço disponibilizado pelo gerenciador X, o serviço `xnest` [30].

O aplicativo `gdmflexiserver` foi alterado de forma a se integrar com o serviço de autenticação de usuários realizado por ZOS. Desta forma, uma vez que um usuário tenha sido autenticado por ZOS (detalhes serão apresentados a seguir), o aplicativo

gdmflexiserver recebe uma notificação do *daemon* zombielisten respectivo e automaticamente abre uma nova sessão para o usuário previamente autenticado. E ainda, uma vez aberta a nova sessão, gdmflexiserver notifica o *daemon* zslave informando que o contexto do usuário pode ser migrado.

3.1.2 Protocolo de domínio - ZAP e ZIP

A migração de um contexto de usuário ocorre em duas fases distintas, chamadas: ZAP - *zombie attraction protocol* - e ZIP - *zombie install protocol*. O primeiro estágio é na verdade o comportamento padrão de um dispositivo *master* que deseje iniciar um domínio. Nesta fase, dispositivos *master* enviam pacotes pela rede a procura de *hosts escravos*. A segunda fase consiste em abrir a sessão remota no dispositivo remoto e transferir o *anima* do usuário.

Protocolo ZAP - Iniciar um domínio requer que dispositivos disponíveis possam ser encontrados em uma rede. Para realização desta funcionalidade é utilizado o aplicativo zdiscovery.

O aplicativo zdiscovery, quando solicitado por um usuário, envia pacotes de detecção de *escravos* (*zdetect packs*) em *broadcast* pela rede. Cada pacote contém informações sobre o usuário do dispositivo, sobre o próprio dispositivo (*hostname*) e o tipo de controle que se deseja obter (completo e parcial). Cada usuário possui um arquivo de conFiguração contendo todas as informações mencionadas. Este arquivo pode ser editado utilizando a interface gráfica disponibilizada por zdiscovery ou qualquer outro editor disponível. A Figura 3.4 apresenta a interface gráfica disponibilizada por zdiscovery

Dispositivos conFigurados como possíveis *escravos* escutam a rede a espera de pacotes *zdetects*. Esta funcionalidade é implementada pelo *daemon* zombielisten. Ao receber um *zdetect pack*, zombielisten (executando em um dispositivo escravo) inicia um processo de autenticação baseado na tupla <user, host, controle>. O protótipo atual utiliza o mesmo mecanismo utilizado por outros sistemas *UNIX* para autenticação (*passwd* e *shadow*). Assim, uma vez que o usuário tenha sido autenticado, o sistema verifica se este usuário/dispositivo tem permissão para realizar o domínio segundo o tipo de controle desejado. O arquivo *zpasswd* armazena essas informações organizadas como tuplas: <user:host:controle>. Nesta versão do sistema, controles parciais são disponibilizados para todos usuários como padrão.

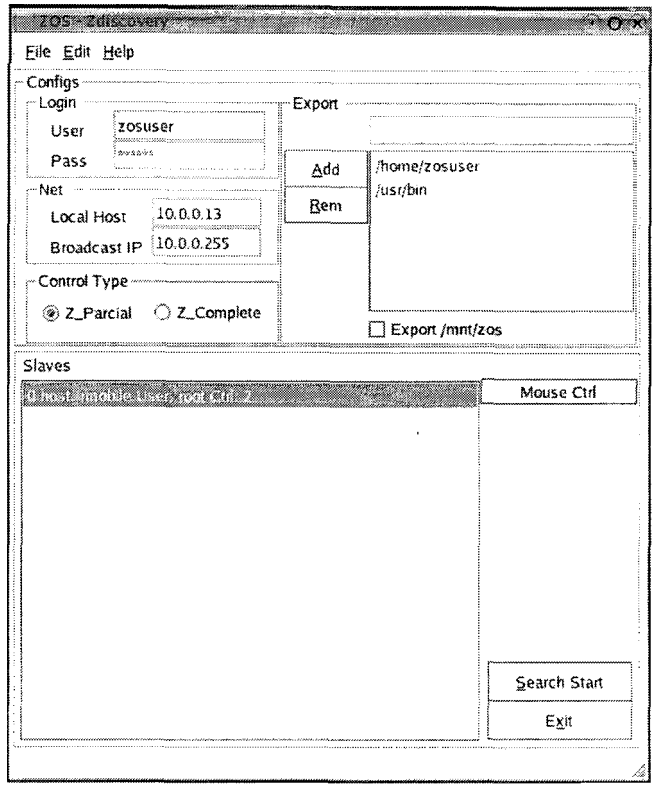


Figura 3.4: Interface Zdiscovery.

De forma a poupar tempo devido a natureza preemptiva dos mecanismos de autenticação, `zombielisten` mantém uma *cache* com as tuplas mais recentes recebidas. Assim, tuplas conhecidas não precisam repetir o processo de autenticação. Entretanto, de forma a permitir que alterações nos arquivos de autenticação de usuários (*passwd* e *zpasswd*) sejam consideradas, `zombielisten` periodicamente efetua um *refresh* desta *cache.*

Após ter passado pelo processo de autenticação, `zombielisten` envia a `zdiscovery` (dispositivo *master]) um pacote (*slave_detected*) contendo informações sobre o* host *escravo. Estas informações incluem: usuário do dispositivo, descrição do* hardware *e nome do* host *na rede. O aplicativo* `zdiscovery` *apresenta então uma lista contendo os* escravos *disponíveis encontrados na rede. Neste momento, cabe ao usuário definir um (ou mais) dispositivo os quais deseje dominar.*

Protocolo ZIP - Uma vez que um dispositivo tenha sido escolhido, `zdiscovery` envia um pacote de notificação de inicio de domínio *znotify* ao *host* correspondente. No caso do *host* desejado ainda estar ativo/disponível na rede, `zombielisten` se encarregará de retornar um pacote de notificação recebida (*znotify_ok*) e o processo

de domínio será iniciado.

A montagem do sistema de arquivos é o primeiro passo para o inicio de um domínio. Para realização deste processo, `zdiscovery` envia um conjunto de pacotes *nfs_mount* ao *daemon* `zombielisten`. Estes pacotes informam os pontos do sistema de arquivos do dispositivo *master* que devem ser montados no dispositivo *zombie*. O usuário ainda pode criptografar os dados trafegados por NFS utilizando *SSH tunnels* (implementações futuras podem disponibilizar *IPSec*).

Para qualquer domínio requisitado, ao menos um ponto do sistema de arquivos do dispositivo *master* deve ser exportado - `/mnt/zos` - neste ponto serão armazenados informações a respeito do estado do sistema e arquivos temporários. Usuários podem definir outros pontos a serem exportados utilizando a interface disponibilizada por `zdiscovery` .

Uma vez que o sistema de arquivos tenha sido devidamente exportado e montado no dispositivo *zombie*, `zombielisten` abre uma sessão *Xnest* para o usuário enviado por `zdiscovery`.

A seguir, `zdiscovery` envia uma notificação a `zdaemon` informando o nome do *host* dominado. Então, `zdaemon` e `zslave` iniciam o processo de copia do contexto do usuário e manutenção da consistência do mesmo.* A Figura 3.5 ilustra graficamente a dinâmica do protocolo.

3.1.3 Segurança em ZAP/ZIP

Uma vez que informações de usuários trafegam pela rede durante o processo de instalação e domínio do sistema (ZAP/ZIP) faz-se necessário à utilização de algum dispositivo que garanta ao menos a segurança do tráfego destas informações pela rede. A Figura 3.6 apresenta o cenário a ser tratado.

Neste cenário algumas questões podem ser levantadas como: a segurança do meio de comunicação (e.g., presença de *sniffers*[31]) e a autenticidade de um *host slave* ou mesmo um *master* (até que ponto é possível se confiar que um *host* é verdadeiramente quem ele diz ser).

Para o tratamento do segundo caso uma solução possível seria o uso de certificados utilizando alguma estrutura *PKI - Public Key Infrastructure*[31], neste caso, um terceiro *host* seria utilizado para garantir a autenticidade dos *hosts* envolvidos

*Vale lembrar que `zslave` recebe as informações de `zdaemon`, porém, apenas inicia o processo de inicialização do contexto remoto ao ser notificado por `gdmflexiserver`.

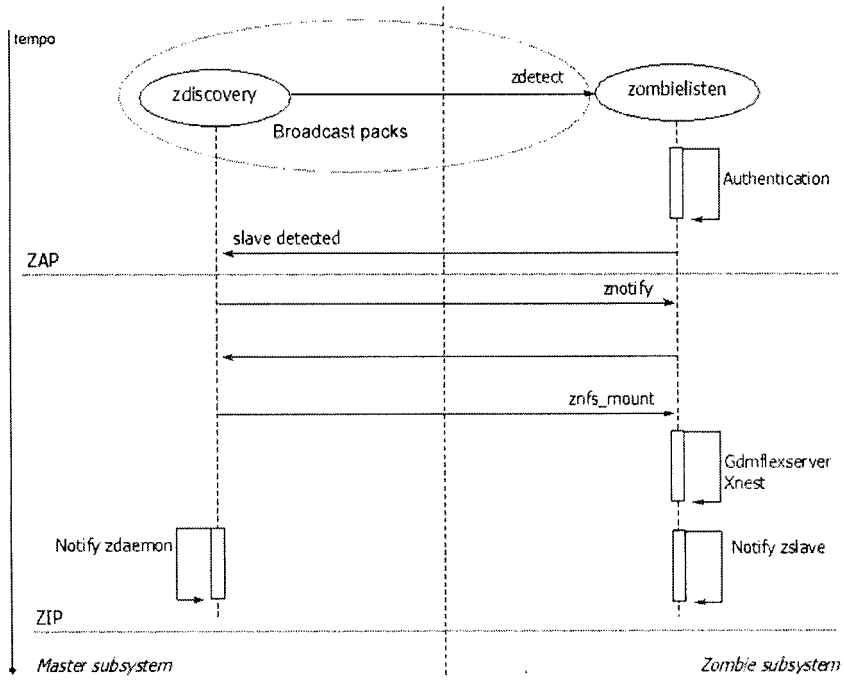


Figura 3.5: Protocolos ZAP/ZIP.

no protocolo.

Figura 3.6: Meio inseguro.

O uso de certificados requer que um usuário adquira por um meio seguro uma chave pública de uma organização emissora de certificados, e.g., *VeriSign*[31]. Ao receber a chave, o usuário deve registra-la na organização emissora e enviar algumas informações a respeito do *host*. A organização emissora de certificados deve ser capaz de reconhecer que a chave pertence realmente à pessoa que deseja registra-la e garantir a autenticidade das informações enviadas. Caso o registro seja confirmado, o usuário receberá um certificado constituído de sua chave pública e partes das informações enviadas anteriormente. Este certificado poderá então ser distribuído para outras partes que desejem trocar informações com o *host*. Clientes que tenham recebido um certificado podem então autentica-lo junto a organização emissora correspondente a fim de confirmar a identidade do *host* emissor do certificado.

Esta solução, apesar de completa - certificaria os *hosts* envolvidos e disponibilizaria uma estrutura para criptografia dos dados trafegados - não foi empregada no

protótipo inicial de ZOS. Para este protótipo buscou-se uma solução mais simples que não exigisse a aquisição de um certificado e o uso de um terceiro servidor para autenticação dos *hosts*. Ou seja, o esquema desenvolvido busca garantir apenas a segurança dos dados trafegados pela rede, não garantindo a autenticidade dos *hosts* envolvidos.

3.1.3.1 Implementação do Módulo de Segurança

A solução desenvolvida para o primeiro protótipo do sistema está focada na segurança das informações trafegadas pela rede, sem considerar a hipótese de certificação dos *hosts* envolvidos. Para esta solução procurou-se desenvolver um esquema que garantisse uma forma segura para troca de chaves públicas entre os *hosts* envolvidos de forma a garantir que, na medida do possível, toda informação trocada durante o processo seja criptografada antes de trafegar pela rede.

Para o desenvolvimento da solução duas hipóteses foram levantadas:

1. Os próprios processos envolvidos `zdiscovery/zombielisten` fossem responsáveis por criptografar/descriptografar os dados.

2. Desenvolver uma espécie de túnel de comunicação por onde os dados trafegassem e onde os processos envolvidos para manutenção deste túnel fossem responsáveis pela criptografia dos dados e gerenciamento das chaves.

A primeira opção considera que os processos `zdiscovery/zombielisten` seriam responsáveis por toda a parte de segurança do trafego dos dados; desde a criação e manutenção de chaves até questões de criptografar/descriptografar os dados. Esta solução possui como característica favorável o fato de que a comunicação entre os processos poderia ser otimizada em relação à segunda alternativa (onde mais processos são envolvidos no esquema o que gera um *overhead* na comunicação). Por outro lado, como desvantagem esta solução pode prender o protocolo de instalação e controle (ZAP/ZIP) aos mecanismos de segurança utilizados, ou seja, a necessidade de se alterar algum mecanismo de segurança pode levar a alteração do protocolo como um todo.

Desta forma, foi escolhida a segunda alternativa para integrar o sistema. Esta alternativa possui como desvantagem a inserção de um *overhead* no sistema, visto que outros processos interceptam as mensagens trocadas por `zdiscovery/zombielisten` e se encarregam de fazer a criptografia dos dados bem como a geração/manutenção

das chaves. Porém, como mencionado, este método permite que alterações nos mecanismos de segurança sejam realizados sem a necessidade de alteração nos protocolos ZAP/ZIP. A Figura abaixo demonstra graficamente o esquema utilizado.

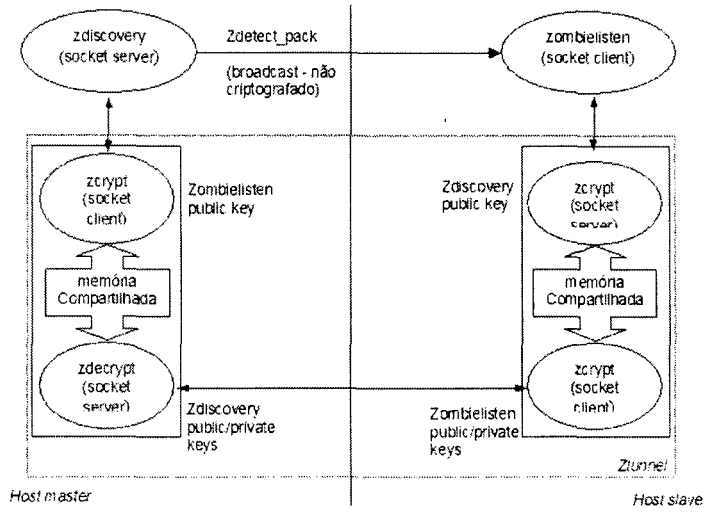


Figura 3.7: Módulo de Segurança: Tunnel.

O processo ocorre da seguinte forma:

- `Zdiscovery` inicia um processo responsável por criar suas chaves e realizar a comunicação segura com os *hosts slaves*, em seguida, envia por *broadcast* o pacote *zdetect* que possui dentre outras coisas, o nome do arquivo onde se encontra sua chave pública e o nome da chave a ser criada por *zombielisten*. A chaves criadas pelo processo responsável pela comunicação são armazenadas no diretório `/mnt/zos/tmp`, a chave privada possui permissão de acesso apenas para o dono do arquivo, a chave pública possui permissão de leitura para o dono e para outros[†].

- `Zombielisten` ao receber o pacote *zdetect* monta (caso tenha permissão) o diretório `/mnt/zos` do *host master* e inicia o processo que será responsável por criar as chaves e realizar a criptografia dos dados (repare que para que `zombielisten` possa ter acesso a chave pública de `zdiscovery` é necessário que o administrador do sistema tenha exportado (permissão para montagem remota) o diretório `/mnt/zos` para o *host* em questão). Este processo (`zombielisten`)

[†]Maiores informações sobre permissões de arquivos em Unix podem ser encontradas em [32].

recebe como parâmetros (pacote *zdetect*) o nome das chaves a serem utilizadas (*zdiscovery public key*) e criadas (*zombielisten private/public keys*) e o *host*/porta para comunicação. Ao ser iniciado o processo cria as chaves referentes a `zombielisten` e recebe a chave pública de `zdiscovery` (via diretório montado `/mnt/zos`). As chaves criadas possuem o mesmo esquema de permissões do processo referente a `Zdiscovery`.

- Deste ponto em diante toda a comunicação entre `zdiscovery` e `zombielisten` é realizada por meio dos processos intermediários responsáveis por criptografar/descriptografar os dados trafegados.

Os processos responsáveis por criar as chaves e realizar a comunicação entre `zdiscovery/zombielisten` são compostos por dois *threads* de execução, um *thread* implementa um cliente *socket* e outro um servidor. Estes *threads* possuem uma área de memória compartilhada protegida por um *mutex*.

Assim, um *thread* ao receber dados de sua conexão *socket* executa os seguintes passos: (a)libera o *mutex* para seu concorrente, (b) envia um sinal a ele, (c) aguarda em uma variável de sincronização *cond* por um sinal. O outro *thread* ao ser acordado, envia os dados para o *host* remoto (tendo feito antes a criptografia/descriptografia dos dados - dependendo do sentido do tráfego) e aguarda em *read*, ao chegar um dado o procedimento se repete na direção contrária.

Toda a estrutura de chaves e criptografia de dados foi baseada no *toolkit openssl - Security Socket Layer*[31]. Esta biblioteca oferece um conjunto de APIs para o desenvolvimento de sistemas que requeiram o uso de mecanismos de segurança embutidos.

O algoritmo utilizado para geração das chaves públicas/privadas escolhido foi o RSA[31]. O tamanho da chave gerada é de 2048 *bits*.

Normalmente, a escolha de um algoritmo de criptografia em detrimento de outros está relacionada diretamente com o grau de segurança oferecido pelo algoritmo. Outra questão importante diz respeito a restrições de uso relacionadas a propriedade intelectual de certos algoritmos.

A escolha pela chave RSA se deu por esses dois motivos. Primeiramente por ser um algoritmo livre, ou seja, não necessita de licensa de uso e por possuir diversas implementações largamente utilizadas. E ainda, este algoritmo possui um melhor desempenho para operações sobre chaves privadas em relação a outros algoritmos

semelhantes, e.g., DSA[31]. E desempenho semelhante a DSA para operações sobre chaves públicas.

O *bit size* de um algorimo de criptografia está diretamente relacionado com o nível de segurança oferecido pela chave gerada pelo algoritmo. Para RSA, considera-se uma chave de tamanho 2048 como conservadora, sendo 1024 um tamanho mínimo necessário se considerada a capacidade computacional das máquinas atuais.

3.1.4 Gerência de Processos

O gerenciamento de processos é a área mais critica do sistema. Conforme mencionado anteriormente, neste protótipo inicial, foram focadas apenas as classes de processos migrantes e compartilhados. O gerenciamento de processos é realizado por `zdaemon` nos dispositivos *master* e por `zslave` para os dispositivos *zombie* .

Ambos os tipos de processos, migrantes e compartilhados, encontram-se em execução normal em dispositivos *master* até que um domínio seja estabelecido, quando devem então ser migrados/replicados no dispositivo *zombie* em questão.

Todo processo que deseje participar de um domínio segundo algum estereotipo (migrante ou compartilhado), deve ser registrado por `zdaemon`. Para realização desta premissa, todo processo deve ser carregado por um *loader* especial, chamado `preloader`. Para carregar um processo, `preloader` deve receber como parâmetros o nome (e caminho) do processo a ser iniciado e seu estereotipo (migrante ou compartilhado). `Preloader` então cadastra o processo em `zdaemon`, informando o nome do processo e o estereotipo o qual o processo será iniciado.

`Zdaemon` por sua vez, armazena uma lista contendo todos os processos cadastrados (nome e estereotipo) e seu estado - exportado ou não exportado. Ao receber uma notificação de inicio de domínio, `zdaemon` estabelece comunicação com `zslave` no dispositivo *zombie*.

O primeiro passo realizado por `zdaemon` é enviar uma lista de todos os processos cadastrados, `ZDAEMONLIST` a `zslave`. Todos os processos compartilhados são exportados para todo domínio realizado. Processo migrantes são exportados apenas uma vez, atualmente para o primeiro *escravo* dominado[‡].

`Zslave` ao receber a lista mencionada, se responsabiliza por iniciar cada processo segundo a regra definida por seu estereotipo (processos compartilhados ou migrantes).

[‡]Implementações futuras podem escolher o melhor escravo a receber um processo migrante.

Processo compartilhados iniciados após inicio de um domínio são imediatamente notificados a `zslave`, `ZDAEMONAPPENDLIST`, logo após terem sido devidamente cadastrados e iniciados.

Ao final de um domínio `zslave` se encarrega de direcionar todos os processo migrantes que tenham sido exportados (e que ainda estejam em execução) de volta ao dispositivo *master*.

3.1.4.1 Processos Compartilhados - Implementação

Processos compartilhados são aqueles que permanecem em execução no dispositivo *master* e *slave* simultaneamente (de forma sincronizada) após início de um domínio.

Com esta classe de aplicativos, acreditamos ser possível que um usuário compartilhe seus aplicativos com outros dispositivos, a fim de, por exemplo, permitir que mais de um usuário (um em cada máquina) seja capaz de executar um serviço sobre um aplicativo simultaneamente (e.g., usuário desenvolvendo um texto em um editor pode desejar compartilhar este editor com outros usuários para que ambos possam trabalhar simultaneamente sobre o mesmo texto).

A seguir relataremos as abordagens utilizadas para o desenvolvimento dos processos compartilhados. Duas abordagens foram utilizadas:

1. Conversão de aplicação para a plataforma ZOS - Alterar o comportamento original de uma aplicação de forma que ela conheça a necessidade do sistema ZOS e possa assim sincronizar suas informações com outros processos que estejam em execução remota.

2. Executar a aplicação sobre um mesmo gerente de GUI, porém, replicado em outros *hosts* remotos - processos virtuais.

A primeira abordagem é mais próxima de nossos objetivos com relação a ZOS, pois, a conversão de um aplicativo pode proporcionar um melhor uso dos recursos e funcionalidades oferecidas pelo sistema. Por outro lado, essa abordagem exige que muito esforço seja despendido para se fazer a conversão da aplicação. A maior preocupação nesse caso é a inicialização correta da aplicação dentro do sistema ZOS e o desenvolvimento de um mecanismo eficiente para sincronização de informação.

Inicialização, geralmente, implica em um processo difícil, por necessitar de um rígido mecanismo que permita que um processo iniciado em um dispositivo *zombie* seja capaz de estabelecer comunicação com o processo original em um dispositivo

master. Realizada a inicialização correta dos processos, a sincronização torna-se um pouco mais simples, restringindo-se a manipulação de *streams* de *input/output* dos processos de forma ordenada.

A segunda abordagem é bem mais simples, visto que se pode utilizar a aplicação em sua conFiguração original. Essencialmente, executa-se a aplicação no dispositivo *master* e habilita-se que a interface deste aplicativo possa ser exporta para outros ambientes gráficos (neste caso, outros dispositvos *zombie*). Nossa abordagem utiliza o sistema VNC [22] para realização desta tarefa. Neste caso, ainda utiliza-se algumas vantagens disponibilizadas por ZOS, como manipulação automática do ambiente e exportação transparente da aplicação.

Para a primeira abordagem, foi realizada a conversão do aplicativo *gnome- terminal*, que possui alguns recursos gráficos e exige grande interatividade com o sistema/usuário. Para a segunda abordagem, foi utilizado o sistema VNC.

3.1.4.2 Processos Compartilhados - Conversão de Aplicativos: *gnome-terminal*

Terminais constituem uma classe de aplicações importantes do ponto de vista dos processos compartilhados. Estes aplicativos recebem linhas de comando como entrada de usuários e geram, na maioria dos casos, textos de saída refletindo o estado do sistema. Desta forma, terminais devem refletir o estado da maquina onde estiverem executando. Assim, faz sentido que comandos gerados em dispositivos *zombies* sejam executados no dispositivo *master*. Portando, um terminal compartilhado em um domínio refletirá o estado do dispositivo *master* e não do dispositivo que esteja executando (evidentemente, usuários continuam com a opção de abrirem terminais locais - não compartilhados - nos dispositivos *zombie*).

O aplicativo *gnome-terminal* é uma ferramenta amplamente utilizadas dentro do ambiente de janelas *Gnome*. Apesar de ser basicamente um aplicativo de texto, ele suporta um conjunto de eventos de *mouse* bem como outros recursos gráficos, porém limitados.

Para nosso *gnome-terminal* alterado foram definidas três regras de execução: *master, slave, inslave*. A primeira regra, *master*, caracteriza um terminal inicializado em um dispositivo *master* que deseje participar de um domínio. Para ativar esta funcionalidade, deve-se passar como parâmetro de inicialização do terminal o *flag* - `szos master`. Desta forma, antes de iniciar, o aplicativo irá estabelecer comunicação

com o *daemon* `zdaemon` para que este possa o cadastrar em sua lista de processos (conforme explicado anteriormente).

Vale ressaltar que, este aplicativo não necessita ser iniciado com o auxilio de `preloader`, visto que o próprio aplicativo conhece as necessidades de ZOS e possui um protocolo próprio para comunicação com `zdaemon`.

Após o cadastro do novo aplicativo, `zdaemon` envia um conjunto de parâmetros para o terminal. Estes parâmetros são: um *stream* onde o terminal possa armazenar um histórico de sua execução a fim de ser exportada ao iniciar um domínio. Um *stream* onde serão armazenados *diffs*, ou seja, após o inicio de um domínio (e conseqüentemente o inicio de um terminal replicado), este *stream* armazenará todas as alterações que ocorram na saída do terminal *master* para que o *slave* correspondente possa ser atualizado. E ainda, um *stream pipe*, por onde o terminal *slave* enviará seus comandos a serem executados pelo *master*.

Conforme já mencionado, durante a inicialização de um domínio, `zdaemon` envia a `zslave` uma lista contendo todos os processo a serem exportados ao dispositivo *zombie*. Esta lista contém as informações necessárias para a correta inicialização de todos os aplicativos (migrantes ou compartilhados). No caso particular da classe de aplicativos *gnome-terminal*, esta lista informa a `zslave` os *streams* corretos para cada um dos terminais a serem iniciados/compartilhados. Com estas informações disponíveis, cabe ao *daemon* `zslave`, iniciar o aplicativo passando como parâmetro o *flag* `-szos slave`.

Assim, o terminal será capaz de se conectar aos *streams* a ele endereçados (nosso protótipo utiliza como base para compartilhamento destes *streams* o sistema de arquivos distribuídos NFS).

A interação entre os aplicativos compartilhados durante um domínio segue o seguinte algoritmo:

- Ao ser iniciado, um terminal *slave* abre o *stream* de histórico e atualiza seu *display*.
- Um *thread* executando no terminal *slave* periodicamente lê o *stream* de *diff* e o esvazia ao final. Este *stream* recebe todas as saídas geradas pelo terminal *master*. Os terminais utilizam um *lock* sobre o *stream*.
- Um segundo *thread* no aplicativo *slave* periodicamente lê as entradas do terminal (comandos) e envia pelo *stream* de *pipe* para o aplicativo *master*, este por

sua vez, possui um *tread* que periodicamente lê (e esvazia) o *pipe*, executando os comandos lidos no mesmo. A Figura 3.8 representa o esquema de *streams* apresentados.

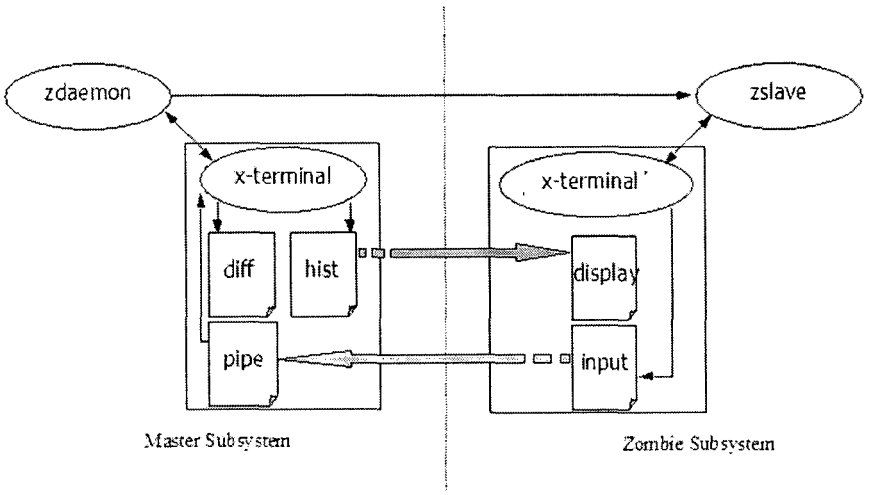


Figura 3.8: Processos Compartilhados: Gnome-terminal.

Novos terminais podem ser iniciados a qualquer instante, mesmo durante um domínio, neste caso, o terminal será imediatamente exportado.

A última opção de *flag* de incialização de um terminal e a `-szos inslave`. Esta opção contempla a necessidade de se iniciar um novo terminal em um dispositivo *zombie* após o inicio de um domínio. Neste caso, o aplicativo comunica a `zslave` da necessidade de inicialização de um terminal no dispositivo *master* e termina. `Zslave` então requisita a `zdaemon` um novo terminal, este por sua vez, inicia normalmente um novo terminal *master* (utilizando a primeira opção de *flag* mencionada - *-szos master*) que, conforme referenciado anteriormente, será imediatamente exportado.

3.1.4.3 Processos Compartilhados - Compartilhamento virtual

A segunda abordagem requer muito menos esforço. Neste caso, nossa maior preocupação era garantir que todos os eventos encaminhados a uma aplicação fossem devidamente direcionados ao aplicativo correspondente.

Para o caso de aplicativos que não possuem grande complexidade ou uma grande quantidade de eventos relacionados, como os terminais, o trabalho de se portar a aplicação não é uma barreira. Entretanto, para outras classes de aplicativos com maior complexidade e grande quantidade de eventos a serem capturados, este trabalho passa ter uma dimensão bem maior. Porém, um grande esforço já foi despendido

para o desenvolvimento de ferramentas que permitam o controle remoto de sessões gráficas. Em nosso sistema, utilizamos a ferramenta VNC [22]. Inicialmente daremos uma visão geral do funcionamento do VNC e em seguida demonstraremos como a ferramenta foi integrada ao nosso sistema.

A idéia central da ferramenta VNC é o uso de um *display* virtual. Este display, para plataformas UNIX, é criado por um processo chamado, *VNCserver* e fica oculto ao usuário. Nele pode-se abrir diversos aplicativos. Tal *display* torna-se disponível por meio de um cliente, *VNCviewer*. Este cliente pode ser disponibilizado tanto localmente (no próprio *host* onde se encontra o *display* virtual) , como remotamente.

Em ZOS, aplicativos compartilhados virtualmente devem ser carregados utilizando a ferramenta `preloader`. Esta ferramenta (`preloader`) irá conFigurar um arquivo utilizado por VNC, indicando que um novo *display* virtual deve ser criado e possuir o aplicativo em questão em execução. E ainda, que tal *display* poderá ser compartilhado por diversos clientes.

Ao terminar a fase de conFiguração, `preloader` inicia o servidor VNC e comunica a `zdaemon` da necessidade de se iniciar um novo aplicativo virtual e o endereço do servidor correspondente a este aplicativo. Assim, cada aplicativo irá possuir seu próprio servidor VNC e conseqüentemente, seu próprio *display* virtual.

Zdaemon ao receber a notificação a respeito do novo display/aplicativo virtual, se encarrega de iniciar um cliente (*VNCviewer*) tornando assim o aplicativo disponível no dispositivo *master*. O mesmo processo é repetido para cada *escravo* dominado (utilizando conexão remota ao servidor VNC).

No inicio de um domínio, `zdaemon` envia uma lista de servidores VNC disponível a `zslave` (na realidade a já referida *ZDAEMONLIST*, contendo o nome do aplicativo e o servidor a ser conectado - no caso de aplicativos virtuais) . `Zslave` se encarrega de abrir um novo cliente para cada um dos aplicativos virtuais na lista. Assim, cada aplicativo virtual terá um servidor VNC executando no dispositivo *master*, mais um cliente (*VNCviewer*) local e n cliente remotos, onde n é dado pelo número de dispositivos dominados. A Figura 3.9 apresenta o conceito.

3.1.4.4 Processos Migrantes

Processos migrantes são aqueles que, ao inicio de um domínio, terão suas execuções interrompidas no dispositivo de origem (*master*) e a seguir, novamente iniciados em um dispositivo *zombie* dominado, retornando ao final do domínio.

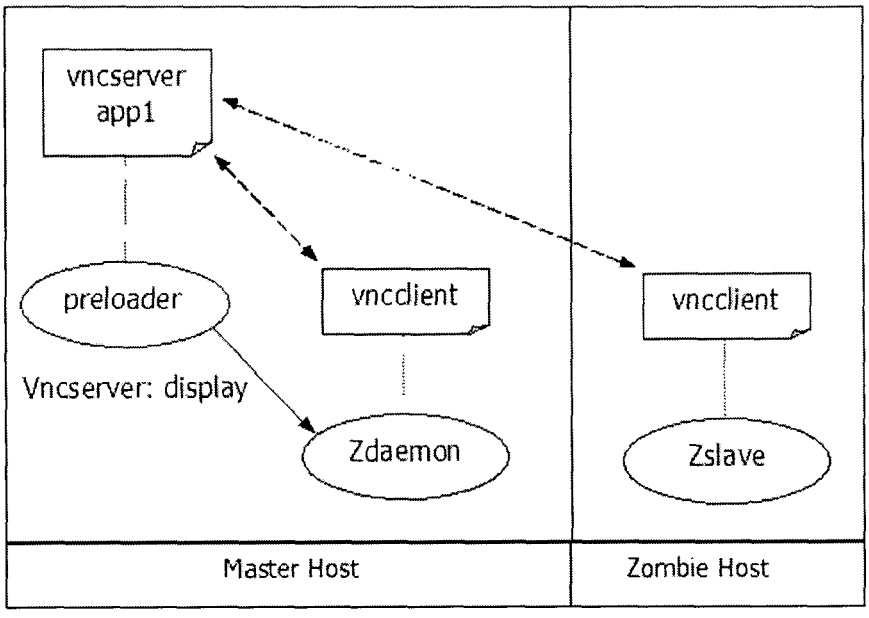


Figura 3.9: Processos Compartilhados: VNC.

Várias ferramentas para migração de processos encontram-se disponíveis na literatura. EPCKPT [23], por exemplo, não requer alterações nos binários a serem utilizados, entretanto, requer que o *kernel* do sistema operacional seja alterado. Por outro lado, Condor [7], Libckpt [24] e *Dynamite checkpointer* [33] não requerem alterações no sistema operacional, porém, requerem que os binários a serem utilizados sejam recompilados e ligados a uma biblioteca de funções. Outro exemplo, ZAP [20], é uma ferramenta particularmente interessante visto que não requer alterações no *kernel* do sistema operacional ou dos binários a serem utilizados.

Em nosso primeiro protótipo foi utilizada a ferramenta *Dynamite checkpointer*. A escolha por tal ferramenta ocorreu basicamente pela facilidade de uso de suas funcionalidades e, principalmente, por ser compatível com a versão do *kernel* utilizada em nosso sistema. *Dynamite* utiliza o sinal `USR1` para informar aos processos que estes devem ser interrompidos (*checkpointed*), e então, uma imagem do processo é gerada. Para a continuar a execução do processo, basta iniciar a imagem gerada.

ZOS utiliza o mecanismo da seguinte maneira:

1. Ao iniciar um domínio, `zdaemon` envia um sinal `USR1` para cada um dos processos migrantes cadastrados (processos migrantes requerem o uso de `preloader`).

2. Após, uma imagem de cada um dos processos é gerada em um diretório temporário (que será compartilhado entre os dispositivos via NFS).

3. `Zslave`, ao receber a lista de processos a serem exportados, se encarrega de reiniciar todos os processos classificados como migrantes.

O mecanismo é invertido ao final do domínio, `zslave` envia um sinal `USR1` para todos os processo migrantes que ainda estejam em execução, uma imagem é gerada no mesmo diretório temporário, e então, `zdaemon` recebe uma lista de processos retirantes e se encarrega de reinicia-los.

3.1.5 Controle de GUI

O gerenciamento da interface gráfica é fundamental em ZOS. Para o gerenciamente inter-processos compartilhados, este serviço é realizado de forma transparente, seja pela própria aplicação (no caso do aplicativo *gnome-terminal* alterado), seja por via do sincronismo disponibilizado por VNC.

Porém, em diversas outras situações usuários podem desejar manipular diretamente a interface de um dispositivo dominado. Para estes casos, ZOS utiliza o *daemon* `ctrlgui`. Esta ferramenta é acionada via a interface disponibilizada por `zdiscovery` e funciona da seguinte forma: ao ser iniciado o `daemon`, todos os eventos (*mouse*, teclado) relativos ao dispositivo *master* serão interceptados e direcionados ao dispositivo *zombie* desejado (deve-se selecionar o dispositivo desejado na própria interface de `zdiscovery`).

Vale ressaltar que, os eventos interceptados não serão atendidos pelo dispositivo *master*, o processo continua ativo até que o usuário pressione a tecla *escape*. E ainda, os eventos direcionados apenas serão atendidos dentro da sessão *Xnest* aberta por `zombielisten`, ou seja, outras sessões abertas no dispositivo não serão manipuladas.

Este *daemon* utiliza algumas funções disponibilizadas pela API do sistema de gerenciamento de janelas X (a Figura 3.10 representa graficamente o funcionamento do sistema *X-Window*), estas funções se encontram nas bibliotecas Xlib e Xtest [34, 35, 36, 37]. Tais bibliotecas utilizam o artefato de que toda a comunicação com o gerenciador X ser realizado via *streams* o que facilita o serviço de se direcionar/receber eventos a outros gerenciadores X remotos.

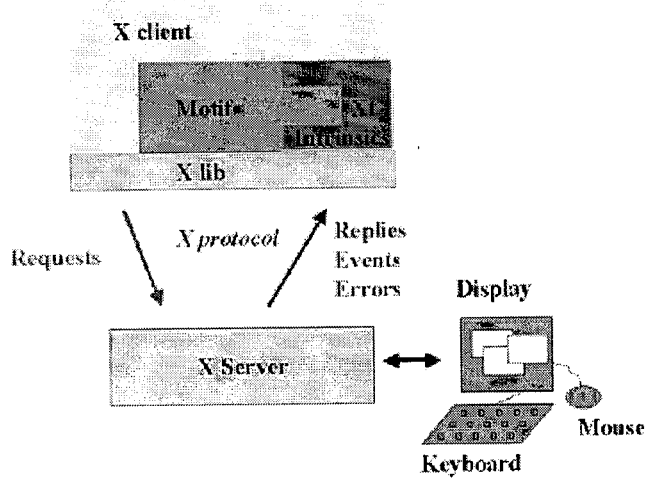


Figura 3.10: X Window System.

Capítulo 4

Resultados

Para confirmar a viabilidade prática do trabalho, algumas medidas foram realizadas no sistema sobre diferentes cargas computacionais. Os resultados foram obtidos no Laboratório de Micro Arquiteturas LAM. Como dispositivo *master* foi utilizado um *laptop* AMD Athlon XP 2.6 GHz com 512 Mb de memória. Para o dispositivo *escravo* foi utilizado um *desktop* Pentium 4 2.4 GHz e 512 Mb de memória. Para comunicação entre os dispositivos foi utilizada uma rede relativamente lenta, uma *ethernet* 10 Mbs sobre um *switch.*

4.1 Medição do tempo de migração de contextos

Inicialmente foi medido o tempo gasto para migração de um ambiente de trabalho do dispositivo *master* para um *escravo.* O tempo gasto pode ser dividido em três componentes básicos: (i) tempo de zip/zap, zt consiste do tempo para autenticação de um usuário e montagem do sistema de arquivos; (ii) tempo para inicialização da sessão remota, xt é o tempo para se iniciar uma *Xnest* no dispositivo remoto; (iii) tempo para migração dos processos, pt. O tempo total gasto para uma migração, tm, é então dado por:

$$tm = zt + xt + \sum_{k=0}^{n} pt$$

Onde n é o número de processo envolvidos no domínio. Este número inclui todos os processos, migrantes e compartilhados. É esperado que zt seja independente de n, sendo relativo as condições de carga da rede e a quantidade de pontos do sistema de arquivos a serem montados. E ainda, a migração de processos apenas deve ocorrer após a inicialização do ambiente gráfico *Xnest.* O tempo xt, tempo gasto para inicialização de *Xnest* remota foi estimado como uma constante. Para isso,

alguns mecanismos foram adicionados a ferramenta `gdmflexiserver` de forma que este notificasse o *daemon* `zslave` a respeito da inicialização da sessão. Com isto, o *daemon* é capaz de iniciar a migração dos processos apenas após decorrido um tempo constante (xt) após a notificação realizada por `gdmflexiserver`. Os experimentos demonstraram que este tempo deveria ser um valor entre 7 segundos, entretanto, em um ambiente real, xt deve variar de acordo com a capacidade computacional de cada dispositivo a ser dominado. Os primeiros resultados foram então obtidos para migração de processos compartilhados, variando n de 1 a 16 processos. Figura 4.1 apresenta os resultados.

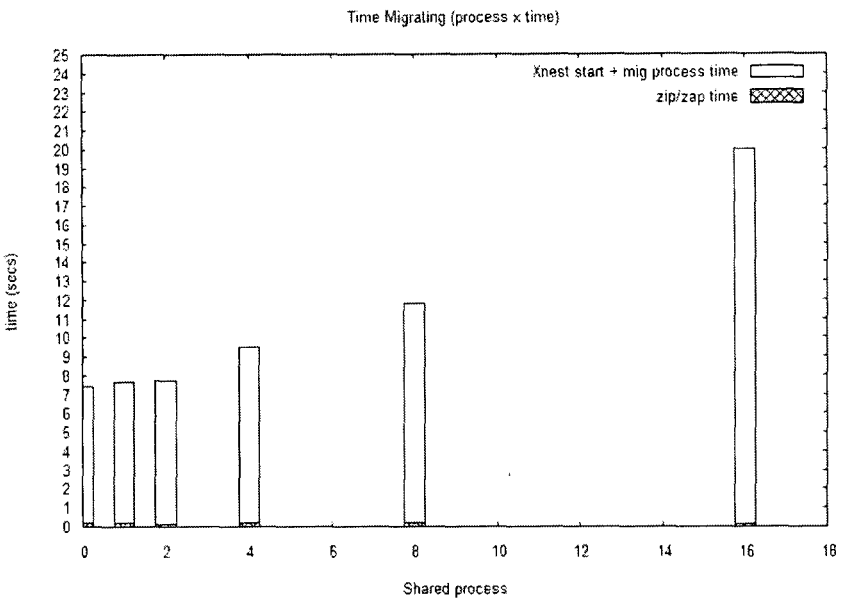


Figura 4.1: Tempo de Migração (tempo x no. processos).

O gráfico da Figura 4.1 demonstra que o tempo para migração de processos compartilhados com a carga máxima utilizada, está próximo a 20 segundos, o que representa um valor considerável, entretanto, pequeno se comparado ao tempo gasto para si iniciar manualmente todo um ambiente de trabalho. Conforme imaginado, xt é o fator preponderante para o tempo total tm. Por outro lado, o tempo zt é praticamente constante e pequeno. Pode-se perceber que o sistema consegue lidar bem com o acréscimo na quantidade de processos a participarem do domínio. Variando de 1 a 16 pode-se perceber algo próximo a 2 segundos de acréscimo por processo. Vale ressaltar que os processos migrados eram aplicativos gráficos e que `zdaemon` inicia cada um dos processos *master* no dispositivo *zombie* aguardando a inicialização de cada um deles para disparar o seguinte.

O segundo experimento contemplou a migração de processos migrantes. Novamente a carga do sistema variou de 1 a 16 processos. Figura 4.2 apresenta os resultados obtidos.

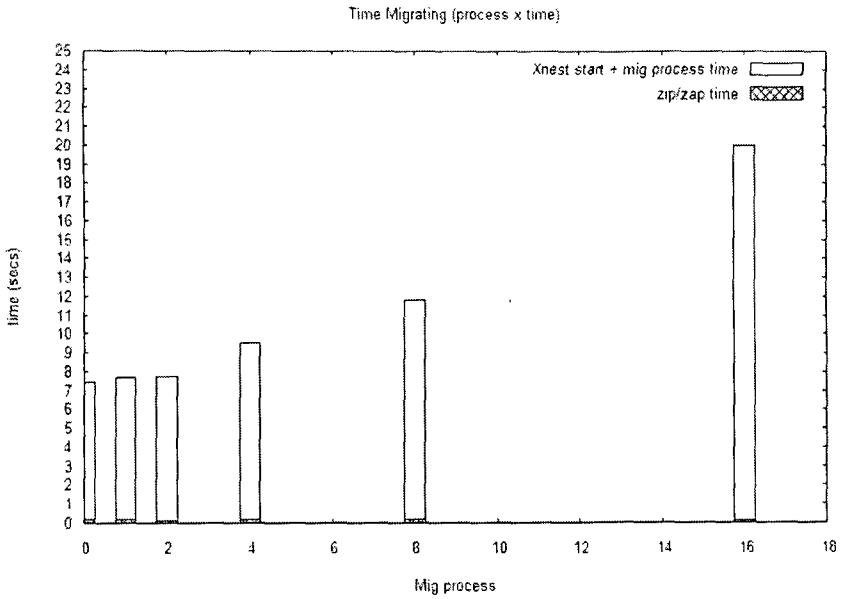


Figura 4.2: Tempo de Migração (tempo x no. processos).

Os resultados obtidos são semelhantes aos apresentados anteriormente. Novamente o fator preponderante é representado por xt. O tempo para migração de processos cresce de forma mais acentuada em relação ao anterior, tendo em vista que uma imagem completa do processo deve ser carregada de um dispositivo para outro. O último experimento foi realizado utilizando uma carga mista de processos migrantes e compartilhados, variando de 2 a 16 processos, sempre com a mesma quantidade de processos migrantes e compartilhados. Com o gráfico da Figura 4.3 podemos novamente notar que o fator preponderante é definido por xt e o aumento de tempo com a variação de processos dominada principalmente pelos processos migrantes.

Os resultados apresentados demonstram que o sistema é viável para a migração de contextos de execução e pode auxiliar e poupar tempo para inicialização de ambientes de trabalho. Tendo em vista que o tempo para migração de um contexto contendo dezesseis processos (aplicativos e processos em *background*) ficou em torno de vinte segundos, acreditamos que, caso o processo fosse realizado manualmente, ou seja, cada um dos aplicativos/processos fossem disparados manualmente pelo usuário, e os respectivos arquivos manipulados pelos aplicativos fossem abertos, etc, o tempo

gasto seria superior aos vinte segundos obtidos.

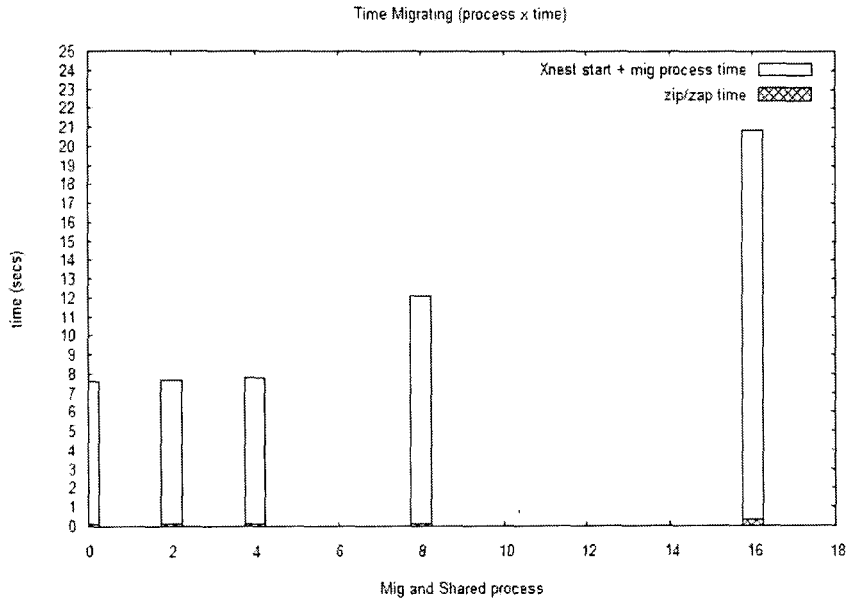


Figura 4.3: Tempo de Migração (tempo x no. processos).

Alguns resultados obtidos demonstraram que o tempo para criar/terminar um domínio são similares. Conforme esperado, o tempo para migração de processos é similar em ambas as direções. Entretanto, neste caso, uma pequena diferença se faz devido à maneira como a migração entre dispositivos é feita. Por se tratar de uma cópia de imagens de processos em disco, uma pequena diferença ocorre favorecendo a migração entre um dispositivo *master* para um *zombie*, devido a maneira como o sistema de arquivos é compartilhado entre os dispositivos.

A cópia das imagens dos processos deve ser realizada em um ponto do sistema de arquivos do dispositivo *master* exportado para o dispositivo *zombie* (para ambas as direções e migração). Assim, para migração de um processo saindo de um dispositivo *master* para um *zombie*, uma cópia da imagem do processo deve ser gerada sobre o sistema de arquivos local do dispositivo para ao final ser inicializada a partir de um sistema de arquivos remoto (no dispositivo *zombie*), no caso reverso, uma cópia da imagem do processo migrante deve ser gerada em um sistema de arquivos remoto e a seguir iniciada a partir de um sistema de arquivos local (no dispositivo *master*).

Certamente, o fator preponderante nesta operação é definido pela cópia da imagem do processo no sistema de arquivos, o que leva ao melhor desempenho da migração no sentido *master/zombie* tendo em vista que a cópia da imagem deve ser feita no sistema de arquivos local, o que, sem dúvida, possui um melhor desempenho

se comparado à cópia de imagem em um sistema de arquivos remoto.

Capítulo 5

Trabalhos Relacionados

Neste capitulo serão apresentados alguns trabalhos de maior relevância que estão relacionados ao tema desta dissertação. Os trabalhos apresentados estão divididos em três partes: migração de processos, sincronização de dados e armazenamento remoto, e controle remoto de sessões. Conforme já mencionado, os três tópicos relacionados nesta sessão constituem partes isoladas de ZOS e são temas largamente estudados na área de sistemas operacionais. Para cada um dos tópicos, será apresentado um conjunto de trabalhos que consideramos mais relevantes dentro da área.

5.1 Migração de Processos

A migração de processos entre dispositivos computacionais é uma área de estudo já explorada há vários anos na área de sistemas operacionais e redes de computadores. Os primeiros estudos buscavam simplesmente o balanceamento de carga de *background jobs* de usuários que demandavam grande poder computacional. Entretanto, mais recentemente, alguns estudos foram desenvolvidos que consideram o balanceamento de processos em *clusters* de computadores, e ainda, questões referentes a tolerância a falhas e escalonamento de servidores/máquinas.

Dentro da área de migração de processos para balanceamento de carga em *clusters* vários sistemas já foram desenvolvidos, como Mosix[6], Amoeba[4] e Sprite[5]. Estes sistemas provêm uma abstração do sistema operacional que visa permitir o melhor uso de recursos distribuídos em um *cluster* de computadores. Desta forma, o *kernel* do sistema deve ser cuidadosamente desenvolvido de forma a prover um mecanismo de identificação global para processos e acesso transparente a recursos distribuídos pelos nós da rede, de forma a possibilitar a migração dos processos. Para

tanto, conforme pode ser observado, o sistema operacional deve ter características específicas (não observadas em sistemas operacionais de uso comum) o que demanda a necessidade de desenvolvimento de novos sistemas operacionais ou, no mínimo, um grande esforço de modificação de sistemas existentes.

Como alternativa aos sistemas citados anteriormente, alguns mecanismos para migração de processos ao nível de usuário foram desenvolvidos de forma a possibilitar o uso de sistemas operacionais não modificados. Estas soluções procuram atender a demanda crescente do uso de *clusters* de computadores independentes, ou seja, cada máquina da rede possui seu próprio sistema operacional.

Como exemplo desta classe de mecanismos para migração de processos podemos citar: Condor[7] e libckpt[24]. Estes sistemas estão primordialmente voltados para migração de processos em *background* entre computadores independentes conectados em uma rede. Por não possuírem recursos do *kernel* (o gerenciamento dos processos a serem migrados é realizado por processos executando em modo usuário e/ou bibliotecas ligadas ao processo que se deseja migrar), estes sistemas ficam limitados a migração de processos "bem comportados", o que pode ser traduzido como processos que não devem utilizar alguns recursos do *kernel*, como IPC.

Uma terceira alternativa para migração de processos é chamada *virtualização* do sistema operacional. Este é o mecanismo empregado por Zap[20]. Este sistema adiciona uma camada ao *kernel* do sistema operacional que possibilita a migração transparente de processos entre dispositivos que possuam este módulo carregado ao *kernel*. Desta forma, a *virtualização* pretende possibilitar que recursos do *kernel* possam ser usados pelos processos a serem migrados (em contrapartida aos mecanismos em execução em modo usuário), porém, que não necessite da alteração do sistema operacional (apenas a adição dinâmica de módulos).

5.1.1 Mosix

O foco deste projeto, difere de ZOS no sentido que objetiva o gerenciamento de *clusters* de computadores, enquanto o primeiro, busca atacar o gerenciamento de máquinas de usuários independentes, dando menos foco ao balanceamento de carga e mais ao gerenciamento de recursos de usuários.

Pode-se considerar que o principal objetivo de sistemas como Mosix[38] é aproximar o universo dos *clusters* de computadores (conjunto de dispositivos *shared nothing* interconectados em uma rede) às máquinas *SMP - Symetric Multiprocessing*.

Em resumo, máquinas *SMP* são constituídas por um conjunto de processadores que compartilham recursos como: memória, disco, E/S. Estes equipamentos são gerenciados por um tipo especial de sistema operacional que distribui os recursos disponíveis entre diversos usuários/processos na máquina. A grande vantagem de sistemas como estes é aumentar o volume de processamento do dispositivo compartilhando de forma eficiente os recursos disponíveis na máquina.

Diferentemente dos ambientes *SMP*, os *clusters* de computadores são constituídos de máquinas independes (máquinas com capacidades distintas e sem compartilhamento de recursos) conectadas por uma rede. Na maioria dos casos o usuário é responsável por determinar explicitamente como se deseja que processos sejam executados sob os nós da rede (executar o balanceamento da rede). Normalmente cada dispositivo da rede carrega uma imagem independente do sistema operacional utilizado. A cooperação entre os nós torna-se limitada visto que os recursos disponibilizados pelo sistema operacional normalmente estão associados ao nó local. Alguns pacotes de bibliotecas são largamente utilizados para alocação e comunicação entre processos em *clusters*, e.g., PVM[39] e MPI[40].

Assim, sistemas como Mosix tem por objetivo permitir uma maior integração aos nós de um determinado *cluster* de computadores. Mosix implementa uma série de algoritmos responsáveis por efetuar automaticamente o balanceamento de carga de uma rede de computadores. Tal balanceamento é realizado utilizando a migração de processos entre nós da rede de acordo com a necessidade de recursos necessária para cada processo.

Algumas versões desta ferramenta já foram desenvolvidas utilizando diferentes sistemas operacionais como base, a versão atual utiliza o *kernel* do sistema operacional *Linux*. As alterações neste *kernel* visam então permitir uma maior cooperação entre os diversos dispositivos de um *cluster* de computadores.

Os algoritmos utilizados por Mosix visam responder automaticamente a variações de disponibilidade de recursos que ocorram na rede, executando quando necessário à migração de processos entre os nós. Para tanto o sistema é dividido em duas partes: um mecanismo para migração de processos *PPM - Preemptive Process Migration* e algoritmos para disponibilizar o compartilhamento de recursos.

A migração de processos ocorre geralmente em resposta a algum evento gerado por um dos algoritmos de compartilhamento de recursos (tais algoritmos são responsáveis por indicar a necessidade e a disponibilidade de recursos para um de-

terminado nó), entretanto, um usuário pode manualmente determinar a migração de um processo em detrimento à forma automática citada anteriormente.

Mosix não possui nenhum tipo de controlador central ou relação master/escravo entre os nós, o que permite reconFigurações dinâmicas da rede e ainda facilita a inserção de novos nós no *cluster*.

5.1.1.1 Migração de Processos em Mosix - PPM

O mecanismos para migração de processos - *PPM* - é a mais importante ferramenta do sistema. Sempre que um nó da rede chega a um limite mínimo de recursos pré-estabelecido (e.g., CPU, memória), a ferramenta PPM é acionada de forma a efetuar um balanceamento de carga na rede.

Em Mosix processos migrantes são encapsulados em dois contextos: contexto de usuário (parte que pode ser migrada) e o contexto de sistema (não pode ser migrado).

O contexto de usuário contém: código, pilha, área de dados e mapas de memória e registradores do processo. Este contexto encapsula o processo enquanto este executa em modo usuário.

O contexto de sistema armazena informações a respeito de recursos que o processo esteja utilizando e a pilha de *kernel* do processo*. Ou seja, esta estrutura armazena o código dependente ao nó local do processo e por isso não é migrado.

Toda interação entre os dois contextos apresentados anteriormente é interceptada e direcionada pela rede para o nó local de origem do processo. Para isto, um camada de comunicação especial é estabelecida entre os contextos executando em diferentes nós.

5.1.2 Condor

Assim como o trabalho anterior, este projeto difere de ZOS no sentido de estar focado basicamente no balanceamento de carga entre computadores ociosos em interconectados em uma rede.

Este sistema foi originalmente desenvolvido para operar sob redes de *workstations* executando o sistema operacional BSD4.3. Com o objetivo de maximizar o uso das máquinas na rede, Condor procura identificar máquinas que não estejam sendo utilizadas por seus usuários e assim escalonar *jobs* para esse dispositivo. Quando

*Pilha que permite que o código de usuário execute em modo *kernel*.

o usuário de uma das *workstations* retorna ao dispositivo, o sistema interrompe os *jobs* a ela transferidos e busca por outra máquina onde estes possam ser enviados.

Todo conceito do sistema baseia-se na idéia de que para alguns ambientes computacionais, principalmente aqueles onde cada usuário possui uma máquina privada e controla todos os recursos disponíveis nesta, a maioria dos equipamentos passam parte do tempo com recursos ociosos, principalmente aqueles em que os usuários das respectivas máquinas não estão em trabalho e não possuem aplicativos que requeiram grande poder computacional. Entretanto, para outros tipos de usuários, o poder computacional de sua estação de trabalho pode ser insuficiente para certos *jobs* que ele deseja executar, esses casos geralmente incluem aplicativos que necessitam de pouca ou nenhuma interação com o usuário e podem ser executados em *background*. Processos que podem ser adicionados a essa classe incluem: estudos de modelos neurais de aprendizado de máquinas, problemas de matemática combinatória dentre outros.

Assim, Condor procura uma forma de migrar tarefas de máquinas que estejam sobrecarregadas para máquinas que estejam com tempo ocioso disponível (sem usuário *logado*). Condor deve então ser capaz de:

- Garantir a migração de jobs em background de forma transparente para os usuários.
- O sistema deve ser capaz de identificar quais estações estão ociosas e transportar os *jobs* sem que o usuário se preocupe com essa tarefa.
- O sistema em si não deve ser oneroso, de outra forma, se tornaria inviável.

5.1.2.1 Estrutura do escalonador

O sistema está estruturado da seguinte forma: uma estação central coordena o escalonamento dos jobs pelas diversas estações ociosas da rede. Cada estação possui um escalonador próprio e uma fila de *jobs* local. A Figura 5.1 representa a estrutura do escalonador.

A cada dois minutos a estação central consulta as diversas estações da rede a procura de máquinas ociosas (sem usuários *logados*) para as quais possa enviar *jobs* de outras estações sobrecarregadas.

Caso alguma estação sem usuário *logado* possua um ou mais *jobs* em execução (vindos de outras estações), a estação central efetua uma consulta a cada meio

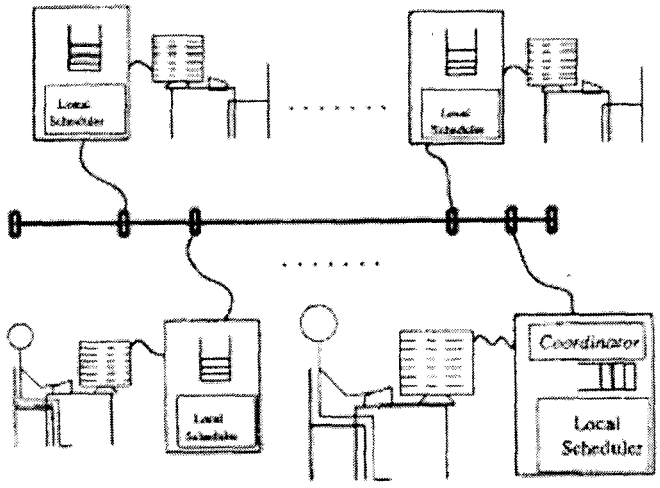


Figura 5.1: Escalonador - Condor

segundo para verificar se estes *jobs* necessitam ser paralisados, na eventualidade do usuário da estação ter iniciado uma sessão no equipamento em questão. Neste caso, o escalonador local imediatamente interrompe a execução do *job* enviando-o para a lista de *jobs* em espera por *workstations* ociosas. O coordenador central é responsável por encontrar *workstations* ociosas para *jobs* aguardando em filas de escalonadores locais. A prioridade de execução dos *jobs* a serem transferidos é responsabilidade do escalonador local.

O objetivo do sistema consiste então em fazer a distribuição de *jobs* de usuários que que estejam executando em máquinas sobrecarregadas para máquinas ociosas de forma eficiente pela rede. Desta forma, o escalonador central deve ser capaz de identificar estações ociosas e *jobs* prioritários e, de uma forma ordenada distribuir esses *jobs* para as estações disponíveis.

Uma questão de grande importância para o sistema é a forma como o *checkpointing* dos processos será realizado. No sistema original todas as estações possuiam o sistema operacional BSD4.3 Unix e faziam uso do recurso Remonte Unix (RU) para a execução remota de jobs. Este recurso também é responsável por efetuar o *checkpointing* dos *jobs*.

5.1.2.2 Estado atual de desenvolvimento

As evoluções no sistema durante os anos estiveram também relacionadas ao sistema operacional utilizado como base. Atualmente o foco principal de desenvolviemnto do sistema é baseado no sistema operacional *GNU/Linux* [41].

O conceito proposto pelo sistema continua o mesmo, entretanto alguns mecanimos foram alterados, sendo o mais significativo deles a parte relacionada ao *checkpointing*/migração de processos. Para a versão atual do sistema, processos a serem migrados devem ser *ligados* a uma biblioteca disponibilizada por Condor.

Outra evolução do sistema, Condor-G [42], está relacionada a computação em *grid*. A computação em *grid* procura utilizar recursos computacionais de máquinas ociosas espalhadas em diversos domínios.

Condor-G representa o casamento de Condor com o sistema Globus [43]. De Globus foram incorporadas as estruturas para comunicação segura entre domínios e acesso remoto a serviços de execução em *background* (vale lembrar que Condor originalmente foi desenvolvido para redes locais). De Condor utilizou-se os conceitos de escalonamento remoto de *jobs* e recuperação de erros.

5.1.3 ZAP

Segundo a solução apresentada por Zap[20] a migração de processos, apesar de se apresentar como uma alternativa valiosa dentro do contexto atual dos ambientes computacionais - dispositivos heterogêneos interconectados por algum meio de comunicação - deve buscar solucionar algumas questões para que se torne uma realidade, tais como:

- Aplicativos legados não devem necessitar de nenhum tipo de alteração para que estejam habilitados a serem migrados e não devem sofrer nenhum tipo de restricao quanto aos serviços do sistema operacional que podem utilizar (e.g., rede, IPC, etc).

- Sistemas operacionais devem sofrer pouca ou nenhuma alteração para que permitam a troca de processos entre eles. E ainda, sistemas operacionais diferentes devem ser capazes de permitir a permutação de processos

- A migração deve ser independente das máquinas de origem dos processos. Ou seja, deve-se evitar manter algum tipo de dependência entre um processo migrado e seu *host* de origem.

Zap foi desenvolvido sobre o sistema operacional GNU/Linux e procurou abranger os pontos apresentados anteriormente. O sistema foi desenvolvido como um módulo dinâmico do *kernel* o que permitiu que nenhuma alteração no sistema

operacional original fosse necessária. Para a migração de processos Zap apresenta o conceito de *Pod - PrOcess Domain*, esta estrutura busca permitir o encapsulamento de processos de forma a torna-los independentes do *host* de origem.

Assim como em ZOS, este sistema permite que todo o contexto de execução de um usuário seja migrado entre dispositivos, entretanto, diferentemente de ZOS, o compartilhamento de contextos, ou seja, a interoperabilidade entre os dispositivos sobre um mesmo ambiente de trabalho, não está prevista como foco de estudos em ZAP.

O sistema operacional Linux, assim como outros sistemas *Unix*, possuem kernel monolítico, o que significa dizer que todo o sistema é *ligado* como um único objeto. A Figura 5.2 [44] apresenta a estrutura do *kernel* de um sistema operacional monolítico.

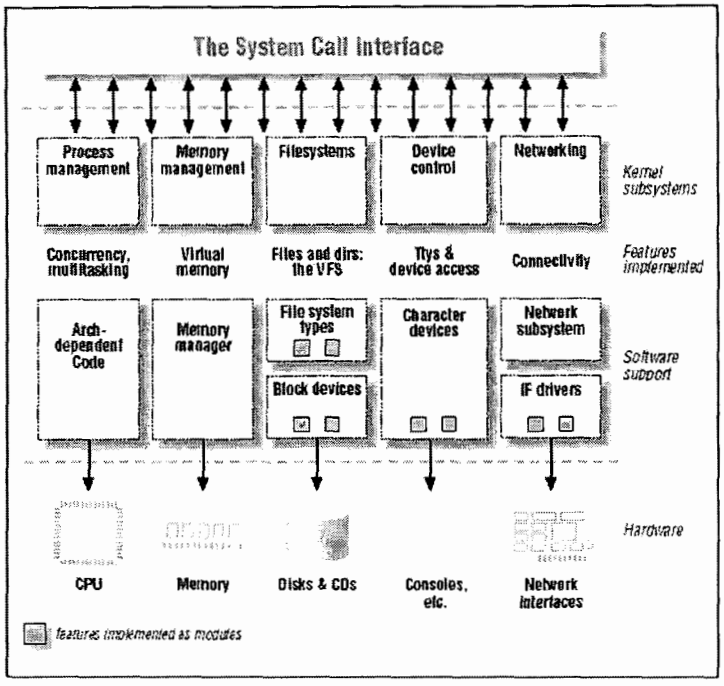


Figura 5.2: Kernel Monolítico

No caso do sistema operacional Linux partes de código podem ser *ligados* dinamicamente ao kernel, adicionando a ele novas funcionalidades e características, esta tarefa pode ser realizada com o desenvolvimento de módulos dinâmicos do kernel. Entretanto, em alguns casos, algumas características devem ser alteradas estaticamente, e recompiladas em conjunto com o kernel, estes casos ocorrem geralmente quando se deseja alterar alguma estrutura do kernel (e.g., uma alteração na estru-

tura que define processos,*task_struct*). A Figura 5.3 [44] representa graficamente a estrutura de um módulo e sua conexão com o *kernel* do sistema[†].

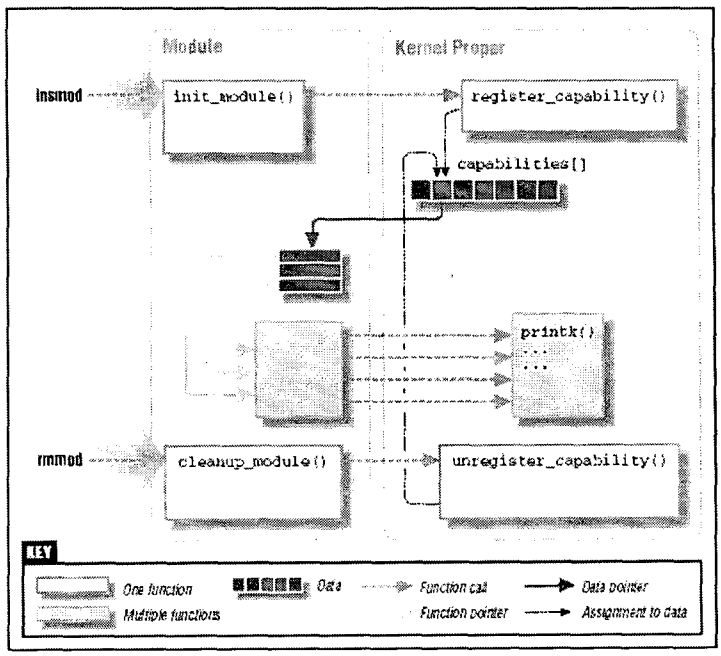


Figura 5.3: Módulos do Kernel

Utilizando-se do fato de ser um módulo do *kernel*, Zap é capaz de interceptar chamadas de sistema de forma a possibilitar que recursos que estejam sendo utilizados por processos tornem-se independente do *host* local e ainda, o sistema é capaz de salvar e restaurar estados do *kernel* de forma a possibilitar a migração/*checkpoint* de processos.

5.1.3.1 Process Domains - Pods

Um *Pod* é uma abstração definida por Zap para tornar um processo (ou grupo de processos) independente do *host* de origem. Esta estrutura define um conjunto virtual de identificadores de recursos do sistema (e.g., identificadores de processos, endereços de rede) de forma a tornar tais recursos independentes da máquina onde estejam situadas.

O sistema operacional atribui diversos identificadores aos recursos que disponibiliza e processos que estejam em execução, e.g., PID, descritores de arquivos, portas para comunicação via *sockets*. Como os sistemas operacionais a principio não foram designados para suportar migração de processos, estes assumem que identificadores

[†]Mais informações sobre detalhes de implementação do *kernel* do linux em [44].

irão permanecer constantes durante o ciclo de vida de um processo. Desta forma o primeiro fator a se considerar ao se migrar processo seria como manter a consistência destes identificadores de forma a garantir que o processo continuará executando de forma correta, uma vez que todos os recursos utilizados pelo processo e sua identificação por parte do sistema operacional pode ser alterada ao se transferir o processo para algum *host* diferente de seu host de origem.

Outra questão importante decorre do fato de se evitar o conflito de identificadores. Uma vez que o sistema supõe que processos não possam ser migrados é natural imaginar que identificadores localmente criados podem ser gerenciados de forma a evitar conflitos.

Da mesma forma, outro problema a ser tratado é evitar que dependências entre processos e recursos impeçam a migração. Por exemplo, um processo que deseje compartilhar uma área de memória com outro processo só poderá ser migrado se uma das situações for atendida: (a) o segundo processo também seja migrado ou (b) um canal de comunicação entre os processos seja estabelecido.

Zap ataca os pontos apresentados anteriormente com a criação dos chamados *Pods*. Esta abstração provê a uma coleção de processos uma imagem virtual dos recursos disponibilizados pelo sistema operacional. *Pods* são estruturas auto-contidas que podem ser suspensas e salvas em um meio secundário de armazenamento, para então ser copiado para um outro *host* e novamente executado.

Entretanto, a interface entre os processos e o sistema operacional (chamadas de sistema) continua a mesma o que permite que sistemas legados não necessitem ser alterados (chamadas de interesse para a realização de algum serviço que possa afetar a migração de um processo são interceptadas e tratadas por Zap).

Todas os identificadores relativos a processos em um *Pod* estão contidos dentro desta estrutura isoladamente das demais. O que permite que *pods* sejam migrados sem que seja necessário alterar seus identificadores. Entretanto, processos criados dentro de *pods* não podem sair ou se integrarem a outros *pods*.

Assim, ao se migrar um *pod* seus processos internos continuaram com os mesmos identificadores virtuais que possuíam anteriormente (porém com novos identificadores reais atribuídos pelo sistema operacional corrente). Chamadas de sistema que fazem uso de algum tipo de identificador de processos são capturadas por Zap que se responsabiliza por traduzir o identificador real para um identificador virtual.

Processos dentro de um mesmo *pod* podem se comunicar utilizando algum mecanismo de IPC livremente, entretanto, processos em *pods* diferentes (ou fora de *pods*) não podem utilizar mecanismos de comunicação a fim de não limitarem a possibilidade de migração.

Migrar um *pod* significa parar todos os processos contidos nele, salvar (em um arquivo) todos seus mapas de identificadores virtuais e o estado de cada processo (estado dos registradores da CPU, mapas de memória e descritores de arquivos abertos). Uma vez copiada a imagem de um *pod* para algum *host* o processo de reinicio do *pod* acontece segundo os seguintes passos: Zap restaura o mapa de descritores virtuais e em seguida restaura os processos (em estado *stopped*[‡]). Zap deve então mapear os recursos utilizados pelos processos restaurados em recursos disponíveis no novo *kernel* e mapear todas as chamadas relativas aos novos processos para identificadores contidos no *pod*. Em seguida, Zap habilita a execução dos processos que estavam parados.

5.2 Sincronização

A miniaturização e o grande aumento da capacidade dos dispositivos de armazenamento secundário, e ainda, a possibilidade de um meio de comunicação persistente e sem fio em dispositivos móveis, tem possibilitado cada vez mais que os usuários de dispositivos computacionais possam de alguma forma acessar seus dados particulares a qualquer momento, independente do dispositivo que esteja utilizando.

Alguns trabalhos têm sido desenvolvidos nesta área e estão concentrados principalmente em três vertentes. A primeira busca o desenvolvimento de um dispositivo móvel com grande capacidade de armazenamento, porém, com pouca ou nenhuma interface de E/S com o usuário. Este trabalho desenvolvido pela Intel[19] procura disponibilizar ao usuário uma espécie de servidor de arquivos móvel, este dispositivo carregaria toda informação necessária ao usuário e seria capaz de se integrar a outros dispositivos maiores (e.g., *Desktops*, *information kiosk*) de forma a disponibilizar ao usuário uma interface de E/S para acesso de suas informações.

A segunda alternativa apresentada busca a solução do problema através da viabilidade de comunicação persistente (*Internet*) para todo e qualquer dispositivo (inclusive dispositivos móveis). Vários trabalhos voltados para esta direção foram

[‡]Em sistemas operacionais do tipo *Unix* processos podem estar em três estados *running, suspended* ou *stopped*, mais detalhes em [32, 1].

desenvolvidos, nesta sessão será apresentado apenas o projeto *Oceanstore*[15], porém outros de igual importância podem ser citados, tais como: *CFS*[16] e *Past*[17]. Todos os trabalhos relacionados consideram a viabilidade do desenvolvimento de uma grande rede *P2P*[45] com grande poder computacional que seria responsável por armazenar e tornar disponível todas as informações de usuários que estivessem conectados a esta rede.

Uma terceira alternativa, também apresentada pela Intel, chamada: *Internet Suspend Resume* [46], propõe o desenvolvimento de um sistema que permita que ambientes de execução de usuários sejam migrados a partir de uma conexão persistente de um dado dispositivo utilizando uma estrutura de máquinas virtuais.

5.2.1 Intel Personal Server

Segundo os conceitos apresentados por este projeto[19], um *personal server* é um dispositivo portátil responsável por prover o acesso a dados privados de usuários utilizando recursos de E/S de outros dispositivos o qual seja possível estabelecer comunicação.

A idéia apresentada sugere o desenvolvimento de um dispositivo móvel com grande capacidade computacional, de armazenamento e recursos de comunicação que seja capaz de fazer uso de dispositivos de E/S (e.g., *Display*, Teclado) de outros equipamentos como e.g., *Desktops e Information kiosks.* Desta forma, segundo os autores, o trabalho procura atacar dois pontos importantes no desenvolvimento de equipamentos portáteis:

- Interfaces de E/S restritas para equipamentos de pequeno porte.
- Acesso restrito as informações/recursos de outros dispositivos os quais seja possível estabelecer algum tipo de comunicação.

Assim, seria capaz tornar disponível um equipamento realmente portátil, com grande capacidade computacional e com baixo consumo de energia.

O primeiro protótipo desenvolvido é focado em três aspectos:

- Serviços disponibilizados: servidor *web*, compartilhamento de arquivos e controle remoto de aplicativos.
- Serviços de infraestrutura: busca de servidores pessoais e protocolos de comunicação entre o dispositivo móvel e o cliente.

- Plataforma utilizada: Recursos de *hardware* disponíveis, novas técnicas para gerenciamento de energia, meios de comunicação disponíveis, etc.

O protótipo inicial foi baseado inteiramente em tecnologias disponíveis e largamente utilizadas nos atuais *PDAs*. O módulo responsável pela comunicação é baseado na tecnologia *bluetooth*. Maiores detalhes a respeito da infraestrutura de *hardware* utilizada pode ser encontrada em [18].

Os conceitos deste sistema estão bem próximos aos apresentados por ZOS, porém, acreditamos que nossos conceitos sejam mais abrangentes no sentido de procurar atribuir mais capacidades ao servidor móvel e ainda, estabelecer uma maior integração entre os dispositivos (não estando simplesmente restrito ao uso das interfaces de E/S do dispositivo dominado).

5.2.1.1 Arquitetura do Sistema

O sistema é dividido em duas partes: (a) sistema relativo ao *personal server* e (b) sistema do cliente. O sistema executando no dispositivo portátil é baseado no sistema operacional *GNU/Linux*.

O sistema cliente é baseado no sistema operacional *Windows XP*. O sistema consiste de uma pilha *bluetooth* padrão e um conjunto de softwares especiais responsáveis por disponibilizar habilitar as funcionalidades do sistema.

Parte do sistema de um *personal server* é responsável por responder a requisições *bluetooth* vindas de clientes. O restante disponibiliza alguns serviços básicos *web* (baseado no servidor *web Apache*), compartilhamento de arquivos e controle administrativo responsável por prover o gerenciamento de recursos do dispositivo.

A comunicação entre os *hosts* é baseada em protocolos *standard* como: *HTTP, SOAP* e *UPnP*. Todos eles estabelecidos sobre a estrutura *IP* estabelecida pelo módulo *bluetooth*. A Figura5.4 apresenta a arquitetura descrita.

5.2.2 OceanStore

O objetivo deste projeto [15] é disponibilizar uma estrutura computacional global que permita o acesso persistente de dados por usuários. Esta estrutura consiste de uma rede *P2P* de servidores espalhados pela *internet*.

O grande desenvolvimento de dispositivos embutidos e portáteis, tais como, celulares, *handhelds*, e até mesmo carros e casas inteligentes, faz com que cada vez mais

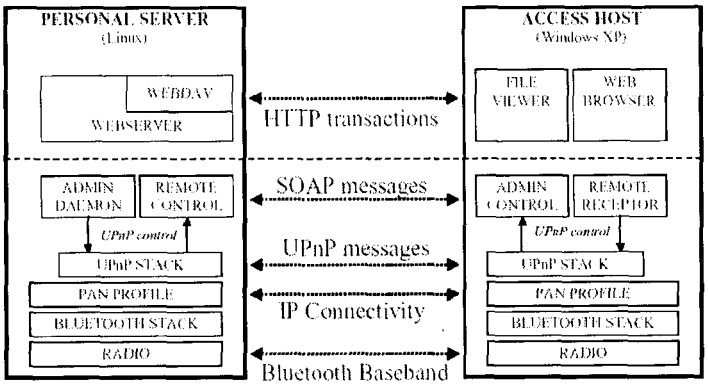


Figura 5.4: Estrutura do *Personal Server*

dados de usuários devam estar disponíveis a todos estes equipamentos de forma persistente independente de sua localização.

Acesso onipresente a dados de usuários se faz necessário de forma a permitir que o comportamento destes dispositivos seja independente do próprio dispositivo, permitindo, por exemplo, a reconFiguração, reinicialização e troca de equipamentos de forma transparente. E ainda, manter a consistência de informação entre os dispositivos nesta estrutura permitiria que a mesma informação fosse acessada simultaneamente por diversos dispositivos (e.g., agenda de contatos persistente, pode ser acessada do celular, *PDA*, *laptop*, sem que a troca do celular acarrete a perda da informação).

Para se obter o acesso persistente de informação deve-se considerar algumas questões:

- Conectividade (ainda que intermitente) garantida para todo e qualquer dispositivo.
- Dados devem ser mantidos em segurança. Questões referentes a ataques de interrupção de serviço (*DoS*) devem ser consideradas.
- Informações devem ser duráveis. Todo e qualquer tipo de arquivamento de dados deve ser automático e confiável.
- Dados devem ser independentes de localização.

Desta forma, deseja-se criar uma grande rede de servidores cooperativos capaz de prover uma grande capacidade de armazenamento a usuários cadastrados. Esta estrutura, *OceanStore*, deve estar sempre disponível a qualquer ponto de acesso pela

internet, empregar replicação automática de dados de forma a evitar perda de informação, utilizar mecanismos de segurança e criptografia das informações e garantir uma performance semelhante aos serviços de armazenamento de dados baseados em redes locais.

O sistema busca o desenvolvimento de dois objetivos básicos:

- Ser desenvolvida sobre uma base de servidores não confiáveis: *OceanStore* assume que toda estrutura do sistema é não confiável - servidores podem sofrer interrupções a qualquer instante, informações podem ser extraviadas para outras partes.

- Suportar dados nômades: Em um sistema de grandes proporções como *OceanStore* localidade de informações deve ser algo importante, principalmente para a garantia de performance. Entretanto, o objetivo do sistema é garantir que dados podem ser armazenados em *caches* localizados em qualquer parte do sistema. Ou seja, dados são livres para serem armazenados em qualquer servidor do sistema. A Figura 5.5 representa graficamente a idéia.

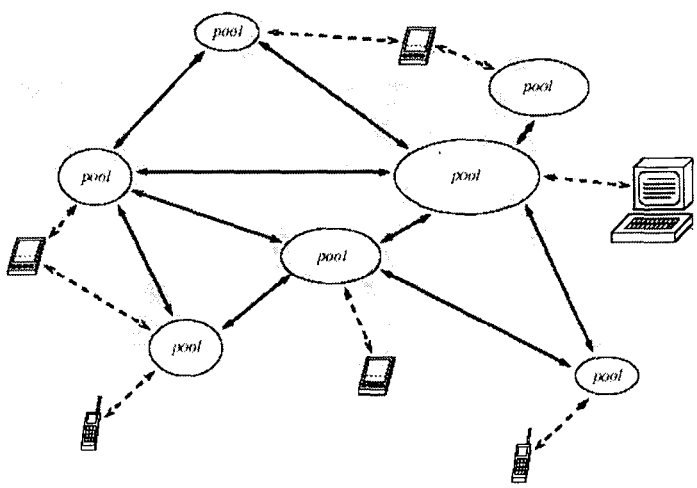


Figura 5.5: *OceanStore*: Dados Nômades.

Desta forma, todo a arquitetura do sistema está focada no desenvolvimento de uma estrutura que permita a inserção e busca eficiente de dados sobre um conjunto de servidores interconectados [47]. E ainda, o gerenciamento de replicas de dados e o arquivamento persistente [48] de informações.

A concepção de uma grande rede que armazene e disponibilize todas as informações de usuários de forma persistente de certa forma, está relacionada ao conceito de ZOS de permitir ao usuário o acesso a suas informações a partir de um servidor pessoal. Entretanto, acreditamos que no caso do armazenamento distribuído das informações pela rede, problemas relacionados principalmente a segurança, podem caracterizar uma questão importante a ser tratada.

5.2.3 Internet Suspend Resume

O projeto *Internet Suspend Resume* [46] apresenta um mecanismo voltado para a migração de contextos de execução entre dispositivos. Para realização de seus propósitos o projeto utiliza conceitos de máquinas virtuais e sistemas de arquivos distribuídos.

Com este mecanismo espera-se conseguir que um usuário utilizando um dispositivo computacional qualquer que possua conexão a internet seja capaz de salvar o estado de seu contexto de execução em um determinado *site* para que, em um momento futuro, possa reiniciar este contexto em algum outro dispositivo.

Um protótipo inicial do projeto foi desenvolvido utilizando dois recursos principais: máquinas virtuais e sistema de arquivos distribuídos. Basicamente, o sistema de arquivos distribuído é responsável por dar acesso aos estados suspensos de máquinas virtuais previamente utilizadas.

Para o primeiro protótipo utilizou-se o sistema *VMware* para prover os recursos de virtualização de contextos e o sistema de arquivos distribuído *NFS*.

A implementação inicial do sistema é baseada em um sistema virtual Linux executando sobre o sistema operacional Windows XP (utilizando o sistema *VMware*). O sistema é responsável por executar a suspensão da máquina virtual (quando solicitado pelo usuário) e salvar seu estado em um diretório remoto (*NFS*).

Salvar o estado da máquina virtual significa salvar alguns arquivos previamente definidos pelo sistema *VMware* para mapear o estado da memória, disco, etc. da máquina em questão. Uma vez salvo o estado da máquina virtual, o usuário pode reinicia-la em outro dispositivo que possua acesso ao diretório *NFS*. O sistema se encarrega de prover mecanismos para o reinicio correto da máquina virtual.

A migração de contextos é o ponto em comum entre este projeto e ZOS, entretanto, diferentemente de ZOS, este projeto nao contempla questões referentes ao compartilhamento de contextos e interoperabilidade entre dispositivos.

5.3 Controle de Sessões Remotas

O controle de dispositivos/sessões remotas é um recurso atualmente muito utilizado que tem se mostrado muito útil principalmente com o advento da *internet*. Suas aplicações estão voltadas principalmente para duas funcionalidades básicas:

- Gerenciamento de *hosts* remotos.
- Centralização de recursos computacionais.

O primeiro item corresponde aos sistemas que se propõe a executar o gerenciamento remoto de *hosts*, permitindo a execução remota de funções de *help desk* nesses dispositivos.

O segundo item procura tornar disponivel *hosts* com algum poder computacional, ou com acesso a *softwares* restritos, a outros dispositivos atuando como uma espécie de terminal, ou seja, atuando basicamente como um mero utilizador de recursos - e.g., CPU, memória, disco, *software* restritos, etc.

Alguns sistemas como [22, 49, 50] se propõe a realizar estes serviços. Nesta sessão faremos uma breve apresentação de VNC, pelo fato de ser este um sistema aberto e com maior disponibilidade de documentação.

5.3.1 VNC

A proposta de VNC [22] (*Virtual Network Computer*) é permitir que um *Desktop* qualquer que esteja conectado a *internet* possa ser simultaneamente acessado por diversas máquinas remotas. Com isto, os recursos desta máquina passam a ser compartilhados por todas as outras, e ainda , as interfaces de E/S dos dispositivos (e.g., *mouse*, teclado, *display*) passam a interagir de forma integrada.

O conceito do sistema VNC está relacionado a ZOS no sentido que propõe um mecanismo que permite o controle de um dispositivo remoto a partir de uma conexão de rede. Entretanto, o objetivo de ZOS não está restrito apenas ao controle de sessões remotas conforme apresentado por VNC. Em ZOS deseja-se o controle remoto de um dispositivo computacional executando um contexto compartilhado iniciado pelo dispositivo local (que iniciou o controle).

O funcionamento do sistema se divide em duas partes: (a) Máquinas que serão acessadas remotamente, ou servidoras (*VNC Servers*), executam os processos responsáveis por compartilhar a sessão do *host* local, (b) máquinas que farão acessos

a *hosts* servidores, ou cliente, executam os chamados *thin clients* ou *VNC Viewers*, processos leves, responsáveis por tomar o controle de um *host* servidor.

Toda comunicação entre servidores e clientes é realizada segundo o protocolo *VNC Protocol*. Este protocolo é independente do sistema operacional base e dos aplicativos em execução no *host* servidor. Sua implementação está no nível da construção gráfica da interface, ou seja, em um nível inferior as aplicações e independente do sistema operacional. O protocolo opera sobre qualquer meio confiável de transporte sobre a rede, como *TCP/IP*. A Figura 5.6 ilustra a arquitetura do sistema.

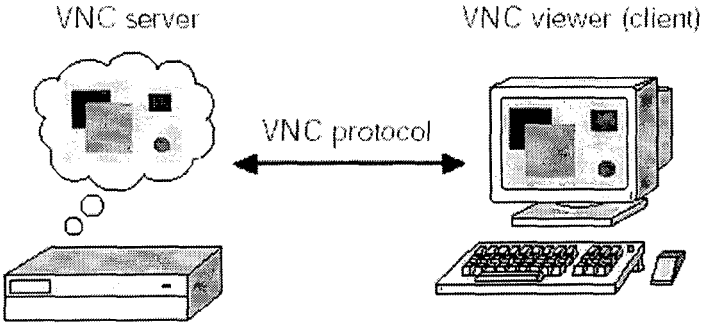


Figura 5.6: Arquitetura VNC.

A estrutura do sistema foi desenvolvida de forma que os clientes necessitem de poucos recursos computacionais, e assim seja possível seu uso nos mais diversos tipos de conFigurações de *hardware*, principalmente naqueles com menos recursos.

5.3.1.1 E/S em VNC

A saída de dados para o *display*, seja da máquina servidora ou cliente é realizada pelo protocolo utilizando a primitiva:

```
Put a rectangle of pixel data at a given x,y position
```

Vale lembrar que alterações em alguma das interfaces de saída das máquinas envolvidas irão afetar todas as outras. Ou seja, este processo pode ser muito dispendioso com relação aos recursos de comunicação. Para resolver este problema vários mecanismos para compactação/codificação da informação exibida/alterada foram desenvolvidos.

Com relação a manipulação de eventos de entrada, como teclado e *mouse*, o

protocolo é responsável por enviar cada um deles para os *hosts* envolvidos sempre que alguma das interfaces de entrada de algum dos dispositivos gerar um novo evento.

5.3.1.2 VNC *Viewers*

É o componente mais simples do sistema, possui versões para vários tipos de sistemas operacionais, como: Solaris, Linux, Windows. Este aplicativo requer apenas um meio para escrever *pixels* diretamente do vídeo do dispositivo, capturar os eventos das interfaces de entrada e um protocolo de comunicação (e.g., *TCP/IP*) .

5.3.1.3 VNC *Servers*

Este componente é o mais complexo do sistema, uma vez que se procurou deixar nele todo o esforço necessário para o funcionamento do sistema deixando os clientes livres de qualquer processamento extra. Por exemplo, o servidor é responsável por fazer qualquer tipo de tradução de formatação de *pixels* (caso os sistemas não sejam homogêneos), de forma a entregar ao cliente o comando para escrita na tela no formato correto para aquele *host* específico.

Capítulo 6

Conclusões e Trabalhos futuros

O presente trabalho propõe um sistema que permita o compartilhamento e migração de ambientes de trabalho entre diversas máquinas. O desenvolvimento desse projeto foi motivado pela experiência dos participantes com a manipulação de diversos dispositivos e plataformas nas mais diversas situações tendo de manter um ambiente consistente. A idéia principal de ZOS consiste no fato de que cada usuário possui um ambiente dinâmico que pode ser transportado entre dispositivos segundo suas necessidades. E ainda, evoluções no desenvolvimento de dispositivos portáteis demonstram que esse ambiente poderá estar disponível em um pequeno dispositivo móvel.

O projeto *Personal Server* [18, 19] desenvolvido pela Intel está diretamente relacionado com os conceitos de ZOS. Entretanto, acreditamos que nossos objetivos sejam mais abrangentes, no sentido que ZOS propõe a migração e compartilhamento de ambientes entre diversos dispositivos, onde estes possam atuar ativamente sobre um ambiente; o outro projeto, por sua vez, considera apenas que o interfaceamento do servidor portátil com outros dispositivos seja passivo, utilizando apenas recuros de I/O de dispositivos maiores.

Outros projetos como [15, 16, 17] consideram a hipótese de se armazenar toda a informação em um ambiente distribuído pela internet. Entretanto, segundo nossa avaliação, essa abordagem pode ser perigosa para o armazenamento de informações pessoais e sigilosas.

A proposta de ZOS é realmente muito ambiciosa, este trabalho procurou focar os esforços no desenvolvimento de um protótipo inicial que contemplasse as funcionalidades principais propostas no projeto, como migração de processos, utilizando sempre que possível ferramentas disponíveis. Nosso intuito foi desenvolver

um protótipo que fosse viável e motivasse trabalhos futuros. Isto nos levou ao foco em processos compartilhados ou, mais especificamente, processos interativos. Esta classe de processos, por sua natureza interativa, demonstrou ser um dos grandes desafios do projetos, entretanto, as abordagens utilizadas para seu desenvolvimento demonstraram ser um problema tratável e de utilidade para diversos usuários.

Trabalhos futuros serão focados em três pontos: segurança, evoluções no sistema de arquivos e compartilhamento de processos. A seguir discutiremos cada um dos pontos iniciando pela parte de segurança.

Em nosso protótipo inicial, um processo de domínio exige uma confiança mutua entre os dispositivos. Desta forma, dispositivos de segurança devem ser considerados para ambos os dispositivos *master* e *zombie.* Pelo lado do dispositivo *master* podemos considerar três requisitos: trafego de dados não deve ser registrado por dispositivos alheios, informações não devem ser mantidas em dispositivos *zombie* e, certificações a respeito se um dispositivo *zombie* é realmente o que ele diz ser. Criptografia de dados é um ponto bem definido e estudado onde tecnologias bem difundidas podem ser empregadas como *tunneling* e *IPSec.* Para questões sobre permanência de dados nos dispositivos *zombie* alguns trabalhos sobre criptografia de sistemas de arquivos como [51, 52] podem ser utilizados, e ainda, projetos mais recentes sobre criptografia de memória principal, como [53], também constituem boas alternativas.

A respeito do desenvolvimento do sistema de arquivos, nosso objetivo principal é desenvolver um mecanismo que permita um maior controle dos pontos a serem exportados/importados entre os dispositivos. Tais funcionalidades devem incluir o desenvolvimento de uma gramática que permita uma definição mais granular dos pontos de acesso dos sistemas de arquivo compartilhados bem como o desenvolvimento de uma nova interface gráfica de configuração. O desenvolvimento de um novo sistema de arquivos pode ser baseado em outros sistemas bem difundidos e documentados, como EXT2 [54], ou ainda, pode-se considerar a hipótese de um novo sistema de arquivos translúcido, que permita a sobreposição de arquivos em diretórios montados.

Para as questões referentes à migração de processos, não apenas questões referentes ao uso de novas ferramentas de *checkpoinitng* devem ser levantadas, mas também, questões referentes a quando e como processos devem ser migrados podem ser atacadas. Este problema foi abordado por Agile [55], o qual propõe uma

ferramenta de interface entre aplicativos e um sistema operacional que possibilite aos aplicativos se adaptarem dinamicamente ao sistema (em relação a quantidade de recursos disponíveis/utilizados), visando basicamente, uma melhor utilização de recursos para dispositivos móveis. Em ZOS, pode-se viabilizar uma interface que permita a migração de processos apenas em momentos de escassez de recursos, não apenas entre um dispositivo *master* e *zombie*, mas também entre *zombies* dominados por um mesmo *master*, quando necessário.

Para a classe de processos compartilhados, o presente trabalho restringiu-se ao desenvolvimento de uma camada virtual sobre os dispositivos de I/O, de forma que a sincronização entre os processos ocorresse sem grandes (ou nenhuma) alteração nos próprios aplicativos. Até o momento evitou-se a alteração do sistema operacional corrente, entretanto, hipóteses mais ambiciosas sugerem que tal possibilidade seja estudada de forma a permitir que um aplicativo compartilhado seja inteiramente copiado para um dispositivo *zombie* e sincronizado utilizando *diffs* da memória utilizada por este processo entre os dispositivos.

Tal abordagem pode ser desenvolvida alterando o gerenciamento de memória do sistema operacional, passando-o a ser controlado por um mecanismo constituinte do sistema ZOS. Um meio termo a essa abordagem poderia ser o uso de migração de máquinas virtuais como proposto em [56], entretanto, esta abordagem deve ser acompanhada de mecanismos de confiança no gerenciador das máquinas virtuais, como proposto em [57, 58].

Referências Bibliográficas

[1] TANENBAUM, A. *Sistemas Operacionais: Projeto e Implementação.* Second. [S.l.]: Bookman, 1997.

[2] CORBATó, F. J.; VYSSOTSKY, V. A. Introduction and overview of the multics system. In: *Proceedings of the Fall Joint Computer Conference.* [S.l.: s.n.], 1965.

[3] RITCHIE, D. M. The evolution of the unix time-sharing system. In: *Proceedings of Language Design and Programming Methodology.* [S.l.]: Springer-Verlag, 1980.

[4] TANENBAUM, A. et al. The amoeba distributed operating system-a status report. *Computing Systems*, December 1991.

[5] DOUGLIS, F. *Transparent Process Migration in the Sprite Operating System.* Dissertação (Mestrado) — University of California Berkeley, 1990.

[6] BARAK, A.; LAZADAN, O. The mosix multicomputer operating system for high performance computing. *Journal of Future Generation of Computer Systems*, March 1998.

[7] LITZKOW, M.; SOLOMON, M. Supporting checkpointing and process migration outside the unix kernel. In: USENIX. *Proceedings of USENIX Winter conference.* Computer Sciences Department University of Wisconsin - Madson, 1992.

[8] JOSEPH, A. D. et al. Rover: A toolkit for mobile information acess. In: ACM. *Proceedings of the Fifteenth ACM Sysposium on Operating Systems Principles.* Cambridge, MA 02139,USA., 1995.

[9] HONEYMAN, P.; HUSTON, L. *Comunications and Consistency in Mobile File System.* [S.l.], 1995.

[10] TAIT, C. et al. Intelligent file hoarding for mobile computers. In: *ACM Conference on Mobile Computing and Networking (Mobicom'95).* [S.l.: s.n.], 1995.

[11] DWYER, D.; BHARGHAVAN, V. A mobility-aware file system for partially connected operation. *Operating System Review*, January 1997.

[12] TRIDGELL, A.; MACKERRAS, P. *The rsync algorithm.* Department of Computer Science, jun. 1996. (http://rsync.samba.org).

[13] RASCH, D.; BURNS, R. In-place rsync: File synchronization for mobile and wireless devices. In: USENIX. *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX Track.* Hopkins University, 2003. p. 91–100.

[14] SWIERK, E. et al. The roma personal metadata service. In: IEEE. *Proceedings of the Third IEEE Workshop on Mobile Computing Systems.* Stanford, CA 94305 USA., 2000.

[15] RHEA, S. et al. Pond: the oceanstore prototype. In: USENIX. *Proceedings of USENIX Conference on File and Storage Technologies.* [S.l.], 2003.

[16] DABEK, F. et al. Wide-area cooperative storage with cfs. In: ACM. *Proceedings of the Eighteenth ACM Sysmposium on Operating Systems Principles.* [S.l.], 2001.

[17] ROWSTRON, A.; DRUSCHEL, P. Past: A large-scale, persistent peer-to-peer storage utility. In: USENIX. *Proceedings of HotOS VIII.* Microsoft Researh, 2001.

[18] WANT, R. et al. The personal server: Changing the way we think about ubiquitous computing. In: *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing.* Gotebog, Sweden: Springer-Verlag, 2003. (LNCS), p. 194–209.

[19] WANT, R. et al. *The Personal Server: The Center of Your Ubiquitous World.* [S.l.], 2003.

[20] OSMAN, S. et al. The design and implementation of zap: A system for migrating computing environments. In: USENIX. *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI'02).* [S.l.], 2002.

[21] PINHEIRO, E. S. de A. *Nomad: Um Sistema Operacional Eficiente para Clusters de Uni e Multiprocessadores.* Dissertação (Mestrado) — Federal University of Rio de Janeiro - UFRJ/COPPE, 1999.

[22] RICHARDSON, T. et al. Virtual network computing. *IEEE Internet Computing*, January/February 1998.

[23] PINHEIRO, E. *Truly-Transparent Checkpointing of Parallel Applications.* [S.l.], 1999.

[24] PLANK, J. S.; BECK, M.; KINGSLEY, G. Libckpt: Transparent chekpointing under unix. In: USENIX. *Proceedings of USENIX Winter conference.* [S.l.], 1995.

[25] PETERSON, D. S.; BISHOP, M.; PADEY, R. A flexible contaiment for execution of untrusted code. In: USENIX. *Proceedings of 11th USENIX Security Symposium.* Computer Sciences Department University of California - Davis, 2002.

[26] GARFINKEL, T. et al. Terra: A virtual machine-based platform for trusted computing. In: ACM. *Proceedings of the 19th ACM Sysmposium on Operating Systems Principles.* MIT, 2003. p. 193–206.

[27] RUSSEL et al. Design and implementation of the sun network filesystem. In: USENIX. *Proceedings of USENIX Conference.* 2550 Garcia Ave. Mountain View, CA. 94110, 1985.

[28] SHEIFLER, R. W.; GETTYS, J. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD.* [S.l.].

[29] PETERSON, M. K. *Gnome Display Manager Reference Manual.* [S.l.].

[30] LEBL, G. *Using and Managing GDM.* [S.l.].

[31] VIEGA, J.; MESSIER, M. *Secure Programming Cookbook.* First. [S.l.]: O´Reilly, 2003. ISBN 0-596-00394-3.

[32] VAHALIA, U. *Unix Internals: The New Fronteirs.* Second. [S.l.]: Prentice Hall, 1996.

[33] ISKRA, K. A.; ALBADA, G. D. van; SLOOT, P. M. A. *The implementation of Dynamite - an enviroment for migrating PVM tasks.* Informatics Institute, Universiteit van Amsterdam.

[34] SHEIFLER, R. W. *X Window System Protocol.* [S.l.].

[35] SHEIFLER, J. G. R. W. *Xlib: C Language X interface.* [S.l.].

[36] DRAKE, K. *XTest: Extention Protocol.* [S.l.].

[37] PATRICK, G. S. M. *X Input Device Extention Library.* [S.l.].

[38] BARAK, A.; LAZADAN, O.; A., S. Scalable cluster computing with mosix for linux. In: USENIX. *Proceedings of USENIX Annual Tech. Conference.* Hebrew University of Jerusalen, 2000.

[39] SUNDERAM, V. S. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, December 1990.

[40] GEIST, A.; BEGUELIN, A.; DONGARRA, J. *Using MPI.* Second. Cambrige, MA: MIT Press, 1994. ISBN 0-262-57132-3.

[41] LITZKOW, M.; LIVNY, M.; MUTKA, M. W. Condor - a hunter of idle workstations. In: USENIX. *Proceedings of 8th ICDCS.* [S.l.], 1988.

[42] FREY, J. et al. Condor-g: A computation management agent for multi-institutional grids. In: IEEE. *Proceedings of 10th IEEE Symposium on High Performance Distributed Computing (HPDC).* [S.l.], 2001.

[43] FOSTER, I.; KESSELMAN, C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, January 1997.

[44] BOVET, D. *Understanding the Linux Kerenl.* Second. [S.l.]: O'Reilly, 2003.

[45] LIBEN-NOWELL, D.; BALAKRISHNAN, H.; KARGER, D. Observations on the dynamic evolution of peer-to-peer networks. In: IEEE. *Proceedings of the First International Workshop on Peer-to-Peer Systems.* [S.l.], 2002.

[46] KOZUCH, M.; SATYANARAYANAN, M. Internet suspend/resume. In: IEEE. *Proceedings of the Fouth IEEE Workshop on Mobile Computing Systems and Aplications.* Intel, 2002.

[47] RHEA, S. et al. Oceanstore: An architecture for global-scale persistent storage. In: USENIX. *Proceedings of Ninth International Conference on Architectural Support for Programing Languages and Operating Systems (ASPLOS 2000).* [S.l.], 2000.

[48] GEELS, D. *Data Replication in OceanStore.* [S.l.], November 2002.

[49] GREENE, T. Symantec adds tools to pcanywhere. Http://www.nwfusion.com/news/2003/0602symantec.html.

[50] CALHOON, J. Mobile computing with windows xp. Http://www.microsoft.com/technet/prodtechnol/winxppro/evaluate/mblxp.mspx.

[51] WRIGHT, C. P.; MARTINO, M. C.; ZADOK, E. Ncryptfs: A secure and convenient cryptographic file system. In: USENIX. *Proceedings of USENIX Annual Technical Conference.* Stony Brook University, 2003.

[52] DOWDESWELL, R. C.; LOANNIDIS, J. The cryptographic disk driver. In: USENIX. *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX Track.* [S.l.], 2003. p. 179–186.

[53] PROVOS, N. Encrypting virtual memory. In: USENIX. *Proceedings of 9th USENIX Security Symposium.* Center for Information Technology Integration - University of Michigan, 2000.

[54] CARD, R.; TS'O, T.; TWEEDIE, S. Design and implementation of the second extended filesystem. Http://e2fsprogs.sourceforge.net/ext2intro.html.

[55] NOBLE, B. D. et al. Agile application-aware adaptation for mobility. In: ACM. *Proceedings of 16th ACM Sysmposium on Operating Systems Principles.* CMU, 1997.

[56] SAPUNTZAKIS, C. P. et al. Optimizing the migration of virtual computers. In: USENIX. *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI'02).* Stanford University, 2002.

[57] KING, S. T.; DUNLAP, G. W.; CHEN, P. M. Operating system support for virtual machines. In: USENIX. *Proceedings of USENIX Annual Technical Conference.* University of Michigan, 2003.

[58] CZAJKOWSKI, G.; DAYNèS, L.; TITZER, B. A multi-user virtual machine. In: USENIX. *Proceedings of USENIX Annual Technical Conference.* Sun/Purdue University, 2003.