MÓDULO ??
SISTEMAS DISTRIBUÍDOS

Vinicius Ponte Machado

APRESENTAÇÃO

Desde o advento do micro-computador e o surgimento das redes locais(LANs) de alta velocidade, o homem tenta aproveitar melhor o poder computacional das máquinas dividindo as tarefas em diversos computadores para realizar processamento paralelo afim de agilizar a computação dos dados.

Esta idéia, chamada de sistemas distribuídos (SDs), faz com que os sistemas se apresentem aos usuários como se fossem um único computador. Ou seja, o que vamos estudar aqui são coleções de computadores autônomos ligados por uma rede de comunicação e equipado com software de sistemas distribuídos. Nesta apostila veremos aspectos que permite o pleno funcionamento dos SDs, suas peculiaridades, aplicações e caracterizações que inclui: o compartilhamento de recursos, escalabilidade, tolerância a falhas e transparência.

Este material é baseado em textos de diversos autores. É extremamente recomendável ao aluno que, ao final da leitura de cada capítulo, acesse os materiais indicados na Weblografia e nas Referências Bibliográficas para complementar a leitura desses textos.

A apostila é divida em quatro unidades. Na primeira mostraremos os principais conceitos e definições da arquitetura dos sistemas distribuídos e como são realizados os processos computacionais em um SD. Na unidade dois falaremos sobre os modelos para a comunicação entre os sistemas. Mostraremos ainda algumas práticas de nomeação para designar computadores,serviços, utilizadores, objetos remotos, arquivos e recursos em geral. Na unidade três veremos como resolver os problemas de sincronização entre os sistemas distribuídos bem como aspectos que envolvem consistência e replicação dos dados. E por fim, na unidade 4, falaremos sobre segurança como devemos tratar as falhas que por ventura ocorrem em SDs.

Boa Leitura !!

Vinícius Machado

SUMÁRIO GERAL

UNIDADE I - ARQUITETURA & PROCESSOS DE SDS

1. Arquitetura ... 8
   1.1. Definição ... 8
      1.1.1. Transparência ... 8
      1.1.2. Aspectos de hardware ... 10
      1.1.3. Aspectos de software ... 12
   1.2. Arquitetura Cliente-Servidor ... 13
2. Processos ... 17
   2.1. Thread ... 17
      Exclusão Mútua ... 18
      Soluções por Hardware ... 19
      Soluções por Software ... 19
   2.2. Cliente ... 20
   2.3. Servidor ... 20
   2.4. Virtualização ... 22
   2.5. Migração de código ... 25

Exercícios ... 27

UNIDADE II - COMUNICAÇÃO

3. Comunicação ... 32
   3.1. Modelo de comunicações ... 32
      3.1.1. Modelo OSI ... 32
      3.1.2. Comunicação Orientada a mensagem ... 33
      3.1.3. Comunicação via Middleware ... 34
   3.2. Chamada de procedimento (RPC) ... 35
      3.2.1. Passagem de parâmetros ... 37
      3.2.2. RPC Assíncrono ... 38
      3.2.3. Comunicação Socket ... 39
      3.2.4. Distributed Computing Environment (DCE) ... 39
   3.3. Message-Passing Interface (MPI) ... 40
   3.3.1. Conceito e definições ... 41
4. Nomeação ... 42

4.1. Espaço de nomes 43
4.2. Implementação de resolução de nomes 44
Exercícios 46

UNIDADE III - SINCRONIZAÇÃO E CONSISTÊNCIA

5. Sincronização 50
5.1. Relógio físico 51
5.1.1. Tempo Universal Coordenado (UTC) 51
5.1.2. Tempo Atômico Internacional (TAI) 51
5.2. Algoritmos de sincronização 52
5.2.1. Algoritmo de Cristian 52
5.2.2. Algoritmo de Berkeley 53
5.2.3. Estampa de tempo de Lamport 54
5.3. Posicionamento Global de Nós 55
5.4. Algoritmo de Eleição 56
5.4.1. Algoritmo Valentão (Bully) 56
5.5. Exclusão Mútua 57
5.5.1. Algoritmo Centralizado 57
5.5.2. Algoritmo Distribuído 58
5.5.3. Algoritmo Token Ring 58
5.6. Transação 59
5.6.1. Primitivas de transações 59
5.6.2. Implementação de transação 61
5.6.3. Controle de concorrência 62
6. Consistência e replicação 63
6.1. Replicação otimista x pessimista 64
6.2. Modelos de Replicação 65
6.3. Algoritmos de Replicação Imediata 67
Técnica Write-all 67
Técnica Write-all-Available 67
Um Algoritmo de Cópias Disponíveis 68
Algoritmo de Cópias Disponíveis Orientado a Diretório 68
Algoritmo Usando Consenso do Quorum 68
Algoritmo de Partição Virtual 68

6.4. Algoritmos de Replicação Atrasada 69
Protocolos de Acesso Multivisão para Replicação em Larga Escala..........69
Replicação Two-Tier 69
6.5. Modelos de consistência 70
Consistência Forte 70
Consistência (Fraca) 71
6.6. Gerenciamento de Réplicas 71
Posicionamento do Servidor de Réplicas 71
Replicação e Posicionamento de Conteúdo 72
Exercícios 73

UNIDADE IV - SEGURANÇA

7. Tolerância e falha 78
7.1. Tipos de Falhas 78
Falhas por queda 79
Falha por omissão 79
Falha temporal 79
Falha Arbitrária 80
7.2. Tolerância a Falhas 80
Fases da tolerância a falhas 80
7.3. Redundância 81
7.3.1. Redundância de Hardware 81
Redundância estática 82
Redundância dinâmica 83
7.3.2. Redundância de Software 83
Diversidade 84
Blocos de recuperação 84
7.4. Recuperação de falhas 85
7.4.1. Pontos de controle 85
8. Segurança 86
8.1. Política de Segurança e Mecanismos de Segurança 86
8.2. Conceitos de segurança 87
8.2.1. Segurança em um SD 87
8.2.2. Ataques de segurança 88

Infiltração 88
8.3. Criptografia 89
8.3.1. DES (Data Encryption Standand) 89
8.3.2. RSA (Rivest, Shamir e Adleman) 90
8.3.3. MD5 (Message Digest 5) 91
8.3.4. Autenticação 91
8.4. Assinatura digital 92
8.5. Circuitos de Criptografia 93
Físico 94
Virtual 94
8.6. Proteção de Dados e Serviços 94
8.7. Gerenciamento de Chaves 95
Exercícios 98

UNIDADE I

Arquitetura e Processos de Sistemas Distribuídos

Objetivos

1. Entender os fundamentos de Sistemas Distribuídos
2. Distinguir os aspectos de Hardware e Software em SDs
3. Entender como funciona os processos computacionais de SDs
4. Conhecer os papeis de cliente e servidor
5. Entender como é realizada a virtualização e migração de código.

1. Arquitetura

Um Sistema Distribuído é uma coleção de computadores independentes que aparecem para o usuário como um único sistema coerente [Tanenbaum 2008]. Pode ser visto como um sistema em que os componentes se localizam em uma rede de computadores e coordenam suas ações através de passagem de mensagens [Coulouris 2007]. Os SDs deve proporcionar para as pessoas que trabalham juntas e compartilhamento de informações sem se preocupar com distribuição física dos dados, máquinas e outros usuários.

Dentre as principais vantagens de sistemas distribuídos sobre micros independentes podemos citar o compartilhamento de dados. (neste caso ressalta-se o aspecto de colaboração e a redução de custos), compartilhamento de dispositivos e o aproveitamento da estrutura de comunicação. Além disso, a mistura de computadores pessoais e compartilhados pode permitir uma distribuição de tarefas mais eficiente. (flexibilidade).

Contudo algumas desvantagens inerentes aos sistemas distribuídos devem ser consideradas. Geralmente, o software utilizado nos SDs – sistemas operacionais, linguagens de programação e aplicações, são mais complexos. A comunicação, no que diz respeito tratamento e recuperação de mensagens pode acarretar em custos altos. Outro fator relevante nos SDs é a segurança. Compartilhamento de dados implica em esquemas especiais para proteção de dados sigilosos.

1.1. Definições

1.1.1. Transparência

Já que o sistema distribuído composto de vários processadores precisa existir um mecanismo de comunicação entre eles, o ideal é que esse mecanismo seja o mais transparente possível. Atualmente é comum usarmos alguma camada intermediária para cuidar dos detalhes de conexão, geralmente criando uma rede overlay, ou seja, uma rede lógica sobre a rede física. Quanto mais transparente a forma de desenvolvimento maior a chance de um número maior de programadores adotarem a ferramenta ou biblioteca, por outro lado, devem existir mecanismos para permitir que o programador lide com os detalhes de implementação manualmente se assim quiser[1].

Ocultar o fato de que seus processos e recursos estão fisicamente distribuídos por vários computadores. Quando nós temos a possibilidade de acessar e/ou interagir com um dado, recurso ou até mesmo um dispositivo, nós precisamos de um meio para acessar. A transparência nesse aspecto significa que a forma pela qual eu interajo com a possibilidade de acessar e também a transformação desse acesso em utilização, faça parte de um único sistema.

Nos SDs deve-se ter um compromisso entre um alto grau de transparência e o desempenho do sistema. Exemplo: Aplicações de Internet tentam contatar um servidor repetidas vezes antes de desistir. Talvez seja melhor desistir mais cedo ou permitir que o usuário cancele as tentativas. Na Tabela 1 podemos ver os tipos de transparências e logo a seguir uma breve explanação sobre cada um deles [SIMOURA 2010].

---

[1] http://www.paulomotta.pro.br/2009/09/02/transparencia-flexibilidade-confiabilidade-e-desempenho-em-sistemas-distribuidos/

Tabela 1: Tipos de Transparência

Transparência	Descrição
Acesso	Oculta diferenças na apresentação de dados e no modo de acesso a um recurso
Localização	Oculta o lugar em que um recurso está localizado
Migração	Oculta que um recurso pode ser movido para outra localização
Realocação	Oculta que um recurso pode ser movido para outra localização enquanto em uso
Replicação	Oculta que um recurso é replicado
Concorrência	Oculta que um recurso possa ser compartilhado por diversos usuário concorrentes
Falha	Oculta a falha e a recuperação de um recurso

Transparência de acesso: O principal objetivo da transparência de acesso é ocultar diferenças entre arquiteturas de máquinas. Para isso, a arquitetura implementada precisa definir como os dados devem ser representados. Neste momento entra as estratégias de nomeação que veremos na Unidade II.

Transparência de localização: Os recursos, dados e dispositivos que serão compartilhados e acessados, não precisam e não devem conter nenhuma informação relevante que forneça ao usuário a localização | região que esse recurso está contido. Por exemplo, se eu quiser acessar um arquivo, um documento que fale sobre a utilização da água, para o usuário, não deve ser relevante o fato de esse documento estar localizado no Brasil ou na Austrália, pois o importante é que o usuário tenha acesso ao documento, sem se importar com a localização física do recurso.

Transparência de migração: Eu sei que na internet, nós temos muitos recursos compartilhados, é bastante comum que pela própria turbulência de acessos, seja necessário que esse recurso seja alocado para outro servidor, por exemplo, mas isso não deve interferir com a maneira que eu tenho acesso ao arquivo. Os softwares gerenciadores devem ser aptos o suficiente para referenciar corretamente os arquivos e recursos em caso de migração.

Transparência de relocação: Oculta que um recurso possa ser movido para outra localização durante o uso. Como por exemplo um celular se movimentando dentro da mesma área de cobertura ou um automóvel passando por várias redes de acesso sem fio, com conexão ininterrupta.

Transparência de replicação: É bastante comum na internet, os recursos e dados serem tratados como objetos, isso significa que a instanciação deles é também muito usual na própria rede. Por isso é bastante interessante e necessário que o sistema distribuído seja capaz de gerenciar a replicação de informações de maneira transparente ao usuário.

Transparência em relação à falhas: Esse é um fator de extrema importância, uma vez que temos a possibilidade de compartilhar recursos, não é nada agradável e necessário que os problemas também sejam compartilhados, por isso, em caso de falhas é essencial que o

sistema seja capaz o suficiente para gerenciar sem que isso passe a espalhar para todo o sistema de compartilhamento, ou seja, os outros usuários podem continuar utilizando a aplicação sem compartilhar com essa falha também, sendo assim um problema considerado isolado.

Transparência de concorrência: Alterações realizadas por um cliente não podem interferir nas operações realizadas por outros clientes fazendo um compartilhamento competitivo de recursos. Deve garantir consistência nos dados. Isso é realizada pelas técnicas de transações (Unidade II).

1.1.2. Aspectos de hardware

Tanenbaum (2008) propõe a divisão de máquinas MIMD (múltiplas instruções e múltiplos dados) em multiprocessadores, que usam memória compartilhada e multicomputadores que possuem somente memória própria. Exemplos de multiprocessadores e multicomputadores. Outra subdivisão dessa classificação é em sistemas com barramento ou *switches* (ligações ponto-a-ponto). Outro ponto da taxonomia de Tanenbaum (2008) é a caracterização dos sistemas pelo grau de ligação entre as máquinas, que podem ser fortemente acopladas ou fracamente acopladas. Máquinas fortemente acopladas possuem um baixo retardo no envio de mensagens e uma alta taxa de transmissão, o oposto de máquinas fracamente acopladas. Geralmente, sistemas muito ligados são usados como sistemas paralelos, trabalhando em um único problema enquanto que sistemas pouco ligados são utilizados como um sistema distribuído, trabalhando em diversos problemas[2]. Geralmente multiprocessadores são sistemas muito ligados enquanto que multicomputadores são sistemas pouco ligados.

Multiprocessadores baseados em barramento.

Consiste em um número de CPUs (que pode ter alguma memória local - cache) ligadas através de um barramento (Figura 1). Sem caches locais, o barramento tende a ser sobrecarregado rapidamente. Solução: adicionar caches locais. Isso acarreta um novo problema: a coerência dos dados que estão armazenados em cada cache é fundamental.

No Cache *write-trough*, toda escrita na cache acarreta em escrita na memória. Escritas sempre geram tráfego no barramento, enquanto que leituras só geram tráfego quando a palavra não está na cache (cache *miss*). Para manter a consistência, as outras caches escutam o barramento e invalidam as posições que são escritas por outras caches na memória (*snoopy* caches). Um design deste tipo é coerente e invisível ao programador. É um esquema difícil de funcionar com um grande número de processadores.

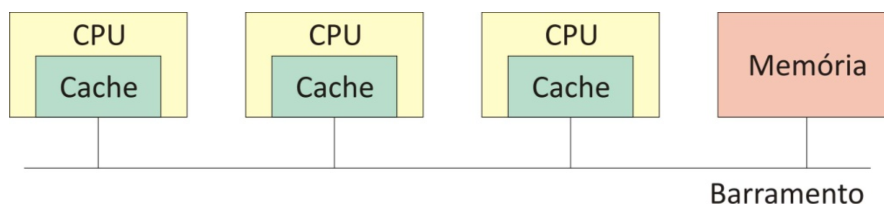


Figura 1: Multiprocessadores baseados em barramento (Tanenbaum 2008)

Multiprocessadores com switch.

[2] www.dimap.ufrn.br/~motta/dim070/Aula01.doc

Podem ser usados com barras cruzadas (Figura 2a) ou com pontos de cruzamento (Figura 2b). Memórias são localizadas de um lado e os processadores do outro. Caminho em uma comunicação é *switched* e a memória em questão tem o acesso bloqueado para os outros processadores.

Número de switches pode tornar custo do sistema proibitivo. Rede ômega diminui número de switches necessários de n^2 para n $\log_2 n$. O retardo em redes ômegas com muitos níveis pode se tornar considerável, ou o seu custo caso switches ultra-rápidos sejam utilizados.

A solução é construir um sistema que use uma hierarquia de memórias. Um design deste tipo é o *NonUniform Memory Access* (NUMA) onde cada CPU tem uma memória local além de memórias que servem a várias CPUs. O retardo diminui, mas a localização de software se torna crucial para o bom funcionamento do sistema. Neste caso, podemos concluir que construir um grande sistema de multiprocessadores ligados com memória compartilhada é possível, mas é muito caro e complicado.

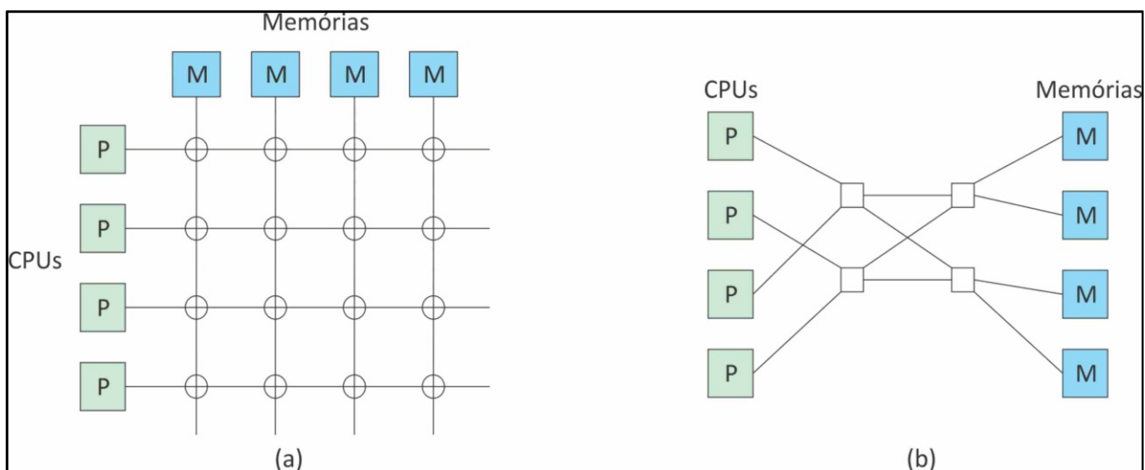


Figura 2: Multiprocessadores com switch (Tanenbaum 2008)

Multicomputadores baseados em barramento.

Conjunto de CPUs com memória local trocando mensagens através de um barramento. Rede local ou CPUs conectadas com um barramento rápido.

CPUs tem um certo número de conexões para outras CPUs e mensagens são trocadas através de CPUs que intermediam a comunicação quando necessário. Na Figura 3 estão as topologias de grid (Figura 3a) e hipercubo (Figura 3b). No grid, número de conexões e número máximo de passos em uma comunicação cresce com a raiz quadrada do número de CPUs enquanto que no hipercubo esse número cresce com o logaritmo do tamanho. Atualmente já são usados grids com dezenas de milhares CPUs. A Teragrid conecta milhares de computadores usando uma rede de 40 Gb/s.

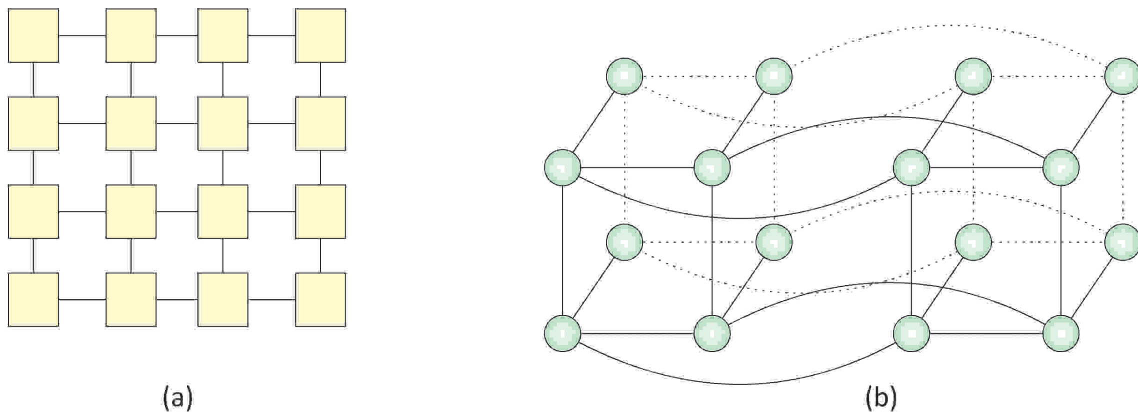


Figura 3: Multicomputadores baseados em barramento (Tanenbaum 2008)

1.1.3. Aspectos de software

A imagem que o sistema apresenta aos usuários é quase que completamente determinada pelo sistema operacional.

Software fracamente acoplado (*loosely-coupled*) permite que máquinas e usuários sejam fundamentalmente independentes dos outros, como por exemplo máquina interligadas por uma LAN. Já o software fortemente acoplado (*tightly-coupled*) funciona como uma única máquina como, por exemplo, um computador multiprocessador que executa um programa de xadrez em paralelo.

Existem 8 possíveis combinações de hardware e software, entretanto somente quatro são distinguíveis pelos usuários já que os mesmos não têm a noção de qual tecnologia de interconexão está sendo utilizada (com barramento ou *switches*). Das 4 opções restantes, a combinação hardware fortemente acoplado (*tightly-coupled*) e software fracamente acoplado (*loosely-coupled*) não faz sentido, já que se estaria indo de encontro ao objetivo principal da construção de um SD.

Sistemas operacionais de rede

Em um sistema fracamente acoplado a comunicação é explicitamente solicitada pelo usuário e se dá através de troca de mensagens. A distribuição do sistema é clara para o usuário (através de serviços como rlogin (ssh), rcp (scp), etc). Usuários diferentes tem visões diferentes do sistema, devido a diferenças em como os sistemas de arquivo locais estão organizados, quais dispositivos locais as máquinas possuem, etc.

Máquinas podem rodar diferentes versões de um mesmo SO ou até diferentes SOs desde que os protocolos de comunicação e serviços sejam usados por todas as máquinas. A "coordenação" (mínima) de serviços entre as máquinas é feita através a aderência aos protocolos.

Sistemas distribuídos verdadeiros

As vezes são chamados de sistemas operacionais distribuídos. Software fortemente acoplado (*tightly-coupled*) em um hardware fracamente acoplado (*loosely-coupled*). O objetivo é a criação de uma ilusão para os usuários que o sistema de multicomputadores funciona como uma grande máquina de tempo compartilhado. As vezes são referenciados como sistemas de imagem única ou uniprocessador virtual.

Características de SDs:

- Sistema de comunicação interprocessos único, independente se os dois processos estão localizados na mesma máquina ou não.
- Esquema de proteção global.
- Gerenciamento de processos único, isto é, como eles são criados, destruídos, iniciados ou parados. A ideia usada nos sistemas operacionais de rede de se estabelecer protocolos para a comunicação cliente-servidor não é suficiente. Deve haver um único conjunto de chamadas de sistema em todas as máquinas e as chamadas devem ser projetadas levando em conta o ambiente distribuído.
- Sistema de arquivos deve ter a mesma visão e regras em todas as máquinas. Segurança deve ser implementada em relação aos usuários e não as máquinas.

Geralmente, o mesmo núcleo (*kernel*) roda em todas as máquinas, já que os mesmos implementam as mesmas funções e chamadas. Entretanto cada núcleo pode ter um controle razoável sobre os recursos locais, por exemplo, no gerenciamento de memória, swapping, ou até escalonamento de processos, no caso de uma CPU possuir mais de um processo sendo executado.

Sistemas multiprocessadores de tempo compartilhado

Software fortemente acoplado *(tightly-coupled*) em um sistema fortemente acoplado (*tightly-coupled*). Exemplo: multiprocessadores rodando Unix. Implementação mais simples que um verdadeiro sistema distribuído devido ao projeto do SO poder ser centralizado. Isso permite um único sistema de arquivos e fila de execução, que é mantida em um espaço de memória compartilhada. Somente uma cópia do SO é executada, ao contrário dos outros dois tipos de sistema. Nenhuma CPU possui memória local e todos os programas são armazenados em memória global compartilhada.

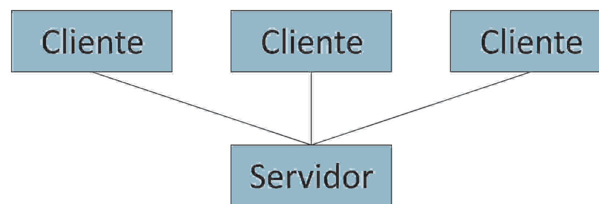
Teoricamente um processo que seja executado por um longo período passaria aproximadamente o mesmo tempo sendo executado em cada CPU do sistema, entretanto é preferível a execução contínua na mesma CPU (caso esteja disponível) devido a pater do espaço de endereçamento do processo permanecer na cache da CPU. Processos bloqueados esperando por E/S podem ser suspensos ou marcados como esperando (*busy waiting*), dependendo da carga total do sistema.

1.2. Arquitetura Cliente-Servidor

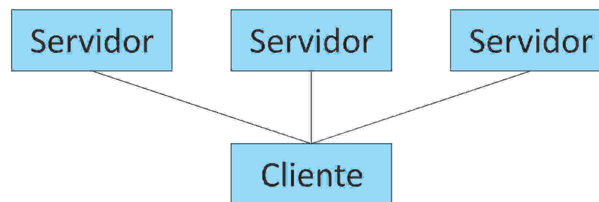
Existem cinco tipos de modelos para a implantação da arquitetura Cliente/Servidor em processamentos distribuídos [Salemi 1993]:

A primeira abordagem para um sistema distribuído é a arquitetura Cliente/Servidor Simples. Nesta arquitetura, o Servidor não pode iniciar nada. O Servidor somente executa as requisições do Cliente. Existe uma clara função de diferenciação: pode-se estabelecer que o Cliente é o mestre e o Servidor é o escravo.

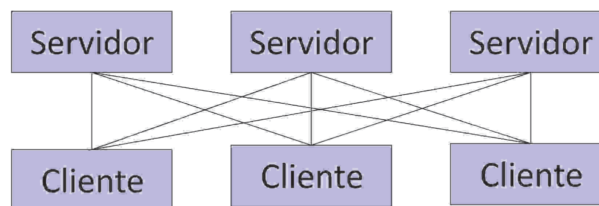
A configuração usual Cliente/Servidor encontrada na maioria das empresas, é aquela em que existem vários Clientes requisitando serviços a um único Servidor. Esta arquitetura se caracteriza como sendo centrada no Servidor (Figura 4a). Porém na visão do usuário, ele imagina que existem vários Servidores conectados a somente um Cliente, ou seja, centrado no Cliente (Figura 4b). Entretanto, com as várias ligações de comunicação possíveis, existe na realidade uma mistura de Clientes e Servidores (Figura 4c).



a. Comunicação Centrada no Servidor



b. Comunicação Centrada no Cliente



c. Comunicação Mista

Figura 4: Arquitetura Cliente/Servidor em dois níveis [Salemi 1993]

Nesta arquitetura (Figura 5), permite-se que uma aplicação possa assumir tanto o perfil do Cliente como o do Servidor, em vários graus. Em outras palavras, uma aplicação em alguma plataforma será um Servidor para alguns Clientes e, concorrentemente, um Cliente para alguns Servidores.

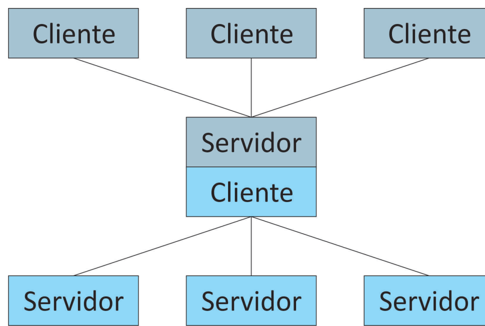


Figura 5: Arquitetura Cliente/Servidor multi-nível [Salemi 1993]

Esta arquitetura pode ser vista como o caso mais geral da arquitetura Cliente/Servidor, ilustrado na Figura 6. Cada um dos nodos desta arquitetura assume tanto o papel de Cliente quanto de Servidor. Na verdade, torna-se pouco funcional lidar com quem é o Cliente ou o Servidor. É o caso onde o processo interage com outros processos em uma base pareada, não existindo nenhum Mestre ou Escravo: qualquer estação de trabalho pode iniciar um processamento, caso possua uma interface de comunicação entre o usuário e o processo Cliente.

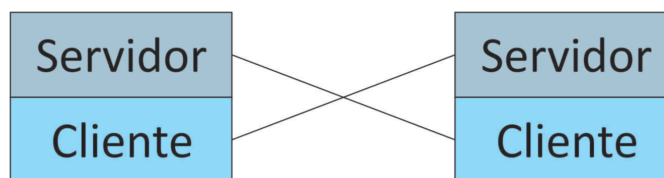


Figura 6: Arquitetura Cliente/Servidor par-par [Salemi 1993]

A arquitetura Cliente/Servidor é dividida em três camadas básicas, como ilustra a figura 7. A camada de Aplicação consiste dos processos da aplicação, entre eles, os processos Cliente e Servidor. A camada de Serviços de Sistemas compreende o Sistema Operacional (SO) e o Sistema Operacional de Rede (SOR), destinando-se ao controle do hardware. Por último a camada de hardware, onde estão localizados os periféricos ligados aos Clientes e Servidores.

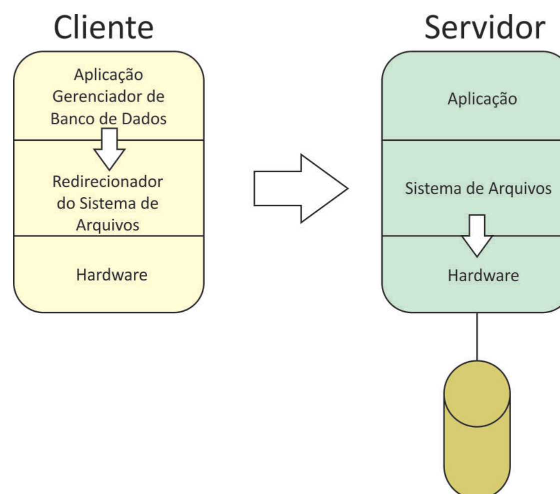


Figura 7: Servidor de Arquivos [Renaud 1994]

A tecnologia Cliente/Servidor pode existir tanto no nível da camada de Aplicação, quanto no da camada de Serviços do Sistema. A coexistência do paradigma nestas camadas surge em função da hierarquia das atuações no sistema. Caso o "usuário" seja externo ao sistema, então os processos Cliente e Servidor compõem a camada da Aplicação, enquanto que, se o "usuário" for um programa de aplicação o Cliente é um processo redirecionador, e o Servidor será um processo respondedor da camada de Serviços do Sistemas.

A utilização de sistemas Cliente/Servidor pela camada de aplicação utiliza Serviços do Sistema não Cliente/Servidor (figura 8), entretanto, sistemas não Cliente/Servidor, ao nível da aplicação utilizam Serviços do Sistema Cliente/Servidor [Renaud 1994].

Para sistemas Cliente/Servidor na camada de aplicação, a camada Serviços do Sistema oferece somente um mecanismo de IPC (*InterProcess Comunication*) para troca de mensagens. Por outro lado, a camada Serviços do Sistema configurada como Cliente/Servidor, é responsável por gerenciar o redirecionamento das solicitações de gravação/leitura, por exemplo. É importante notar que a diferença entre os sistemas Cliente/Servidor nas camadas de Aplicação e Serviços do Sistema, é o equilíbrio entre a quantidade de processamento tanto no lado do Cliente quanto no lado Servidor.

Existem vários sistemas que podem ser baseados na estrutura Cliente/Servidor. O uso mais frequente são as aplicações de Banco de Dados usando processos SQL (*Structured Query Language*) de *front-end*, para acessar remotamente, as bases de dados. A Figura 7 mostra uma estrutura baseada num Servidor de Arquivos. Esta estrutura ocasiona um maior fluxo de informações na rede, uma vez que todo o arquivo será transferido para o Cliente para então ser trabalhado.

Segundo Berson (1997), o modelo cliente servidor pode ser classificados conforme a característica de distribuição. Considere que o modelo tem três componentes básicos: dados, lógica da aplicação e apresentação. Esse três componentes podem estar no cliente e/ou no servidor e conforme a disposição desses componentes podemos classificá-los em cinco outros modelos, dentro da arquitetura cliente/servidor, são eles:

- **Apresentação Distribuída** - neste modelo o servidor fica com os dados, a lógica de aplicação e a apresentação. O cliente fica somente com a apresentação, o exemplo clássico é uma aplicação web.

- **Apresentação Remota** - este modelo difere do anterior somente na ausência de apresentação no servidor. Como exemplo temos uma aplicação clássica cliente/servidor implementada em *Java*, sem regra de negócio na interface.

- **Lógica Distribuída** - neste modelo temos a distribuição da lógica da aplicação, ou seja, ela está tanto no cliente, como no servidor. A apresentação fica no cliente e os dados no servidor. Como exemplo temos uma aplicação em *C#*, em que o lado cliente, além de ter apresentação, tem regras de negócio junto com a interface visual.

- **Gerenciamento Remoto de Dados** - nesta arquitetura a apresentação e a lógica de aplicação ficam no cliente e somente os dados ficam no servidor. Como exemplo, uma

aplicação que somente o SGBD (Sistema Gerenciador de Banco de Dados) fica em no servidor.

- **Dados Distribuídos** - neste caso, os dados é que são distribuídos, portanto, ficam tanto no cliente como no servidor. A apresentação e a lógica da aplicação ficam no cliente. Uma aplicação móvel que sincroniza os dados com o servidor somente quando possível, é um bom exemplo nesse caso.

2. Processos

2.1. Thread

O conceito de *thread* está intimamente ligado ao conceito de processo, assim é fundamental entender o que são processos, como eles são representados e colocados em execução pelo Sistema Operacional, para em seguida entender as *threads*. Dessa forma segue uma breve definição de **Processo** e posteriormente a de ***Thread***.

Definição de Processo

"Um processo é basicamente um programa em execução, sendo constituído do código executável, dos dados referentes ao código, da pilha de execução, do valor do contador de programa (registrador PC), do valor do apontador do apontador de pilha (registrador SP), dos valores dos demais registradores do hardware, além de um conjunto de outras informações necessárias à execução dos programas." [Tanenbaum 2008].

Podemos resumir a definição de processo dizendo que o mesmo é formado pelo seu espaço de endereçamento lógico e pela sua entrada na tabela de processos do Sistema Operacional. Assim, um processo pode ser visto como uma unidade de processamento passível de ser executado em um computador [Santos Júnior, 2000].

Essas informações sobre processos são necessárias para permitir que vários processos possam compartilhar o mesmo processador com o objetivo de simular paralelismo na execução de tais processos através da técnica de escalonamento, ou seja, os processos se revezam (o sistema operacional é responsável por esse revezamento) no uso do processador e para permitir esse revezamento será necessário salvar o contexto do processo que vai ser retirado do processador para que futuramente o mesmo possa continuar sua execução.

Definição de Thread

> "*Thread*, ou processo leve, é a unidade básica de utilização da CPU, consistindo de : contador de programa, conjunto de registradores e uma pilha de execução. *Thread* são estruturas de execução pertencentes a um processo e assim compartilham os segmentos de código e dados e os recursos alocados ao sistema operacional pelo processo. O conjunto de threads de um processo é chamado de *Task* e um

> processo tradicional possui uma *Task* com apenas uma *thread*." [Silberschatz 2004]

O conceito de *thread* foi criado com dois objetivos principais: facilidade de comunicação entre unidades de execução e redução do esforço para manutenção dessas unidades. Isso foi conseguido através da criação dessas unidades dentro de processos, fazendo com que todo o esforço para criação de um processo, manutenção do Espaço de endereçamento lógico e PCB, fosse aproveitado por várias unidades processáveis, conseguindo também facilidade na comunicação entre essas unidades.

Dessa forma o escalonamento de threads de um mesmo processo será facilitado pois a troca de contexto entre as *threads* exigirá um esforço bem menor. Sendo que ainda assim, ocorrerá o escalonamento de processos, pois outros processos poderão estar sendo executado paralelamente ao processo que possui as *threads*. Podemos concluir então que a real vantagem é obtida no escalonamento de threads de um mesmo processo e na facilidade de comunicação entre essas *threads*.

Threads são uma forma do programa se dividir em dois ou mais tarefas de execução simultâneas. A forma como threads são criadas e dividem seus recursos é bem diferente dos processos. Múltiplas threads podem ser executadas em paralelo em muitos sistemas. Esta *multithreading* geralmente ocorre pela divisão do tempo (*time slicing*), onde um processo alterna a execução entre diferente threads, ou seja, o processamento não é literalmente simultâneo já que o processador estará executando um único processo de cada vez. Esta troca acontece tão rápida que fica a impressão de simultaneidade para o usuário.

Nos computadores com um único processador, quando estamos realizando várias atividades no mesmo, como escutar música, utilizar o browser e demais tarefas, o que realmente acontece é que o processador alterna a execução destes processos, gastando um tempo em cada um deles. Nos computadores de *multi-core*, o que acontece é a real execução simultânea de diferentes thread nos diferentes processadores. Os sistemas operacionais modernos suportam ambos, *time-sliced e threading* com *multi-processador*, por meio de um organizador de processos[3].

Exclusão Mútua

Algoritmos de exclusão mútua são usados em programação concorrente para evitar que um recurso em comum seja utilizado simultaneamente, como variáveis globais, por pedaços de código chamados seções críticas. Exemplos destes recursos são *flags*, contadores e *queues*, usados para comunicação entre códigos que são executados concorrentemente, como uma aplicação e seus tratadores de interrupção. Estes problemas são realmente sérios porque as threads podem ser paradas e recomeçadas a qualquer hora.

Para ilustrar, suponha uma parte de código que está mudando um dado por vários passos do programa, quando uma outra ***thread*** ativada por um evento imprevisível, inicia sua execução. Se esta segunda *thread* ler este dado, durante o processo de escrita, isto leva a um inconsistente e imprevisível estado. Se a segunda *thread* tentar sobrescrever o dado, o estado

[3] http://www.pucsp.br/~jarakaki/lp4/Threads-Aracaju.pdf

possivelmente não poderá ser recuperado. Estas seções críticas de código acessando dados compartilhados devem ser protegidas, para que outros processos que tentem acessar estes dados não possam ser executados até o final de sua utilização pelo processo "dono".

Existem diversas formas de "forçar", algumas delas apresentamos abaixo.

Soluções por Hardware
Em sistemas com um único processador, uma forma comum de alcançar a exclusão mútua é desabilitar as interrupções para o menor número possível de instruções, para prevenir a corrupção dos dados compartilhados, a seção crítica. Isto irá prevenir a interrupção de um código durante sua execução na seção crítica.

Nos computadores que os processadores compartilham a memória, um indivisível *test-and-set* de uma *flag* é usado em um *loop* para esperar até que outro processador libere a *flag*. O *test-and-set* efetua as duas operações sem entregar o barramento de memória para outro processador. Quando vai sair da seção crítica, a *flag* é liberada. Isto é chamado de *busy-wait*.

Soluções por Software
Trabalhando junto com as soluções de hardware suportadas, algumas soluções via software existentes utilizam *busy-wait* para alcançar seus objetivos. Por exemplo:

- Algoritmo de Dekker
- Semáforos
- Monitores

Os métodos mais clássicos de exclusão mútua tentam reduzir a latência e as *busy-waits* utilizando-se de esperas e trocas de contextos. Alguns declaram que as medidas de desempenho indicam que estes algoritmos especiais gastam mais tempo do que ganham. Muitas formas de exclusão mútua possuem seus efeitos colaterais. Por exemplo, semáforos clássicos permitem *deadlocks*, no qual um processo recebe um semáforo, outro processo recebe o outro e ambos permanecem no estado de bloqueado esperando que o outro semáforo seja liberado. Outros efeitos colaterais conhecidos incluem, *starvation* no qual um processo nunca consegue obter recursos necessários para sua execução e inversão de prioridade no qual uma *thread* de maior prioridade espera por uma de menor prioridade. Apesar de muita pesquisa para a eliminação deste efeitos, ainda não foi descoberto um esquema perfeito.

Threads (Figura 8) podem executar suas funções de forma paralela ou concorrente, onde as *threads* são ditas paralelas quando elas desempenham o seus papeis independente uma das outras. Já na execução concorrente, as threads atuam sobre objetos compartilhados necessitando de sincronismo no acesso a esses objetos, assim deve ser garantido o direito de atomicidade e exclusão mútua nas operações das *threads* sobre objetos compartilhados.

Deadlock (interbloqueio, blocagem, impasse), no contexto do sistemas operacionais (SO), caracteriza uma situação em que ocorre um impasse e dois ou mais processos ficam impedidos de continuar suas execuções, ou seja, ficam bloqueados. Trata-se de um problema bastante estudado no contexto dos Sistemas Operacionais, assim como em outras disciplinas, como banco de dados, pois é inerente à própria natureza desses sistemas.

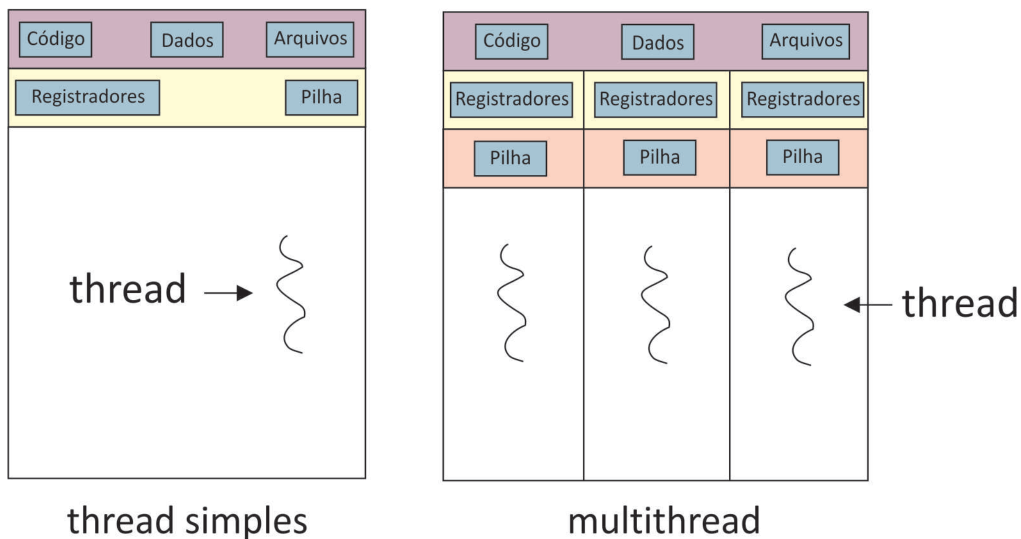


Figura 8: Processos com um e múltiplos threads

2.2. Cliente

O termo cliente em um sistema distribuído pode ser visto como um processo que necessita de um serviço que, por si só, não pode ou não quer executar; pede, então, ajuda a outros processos (servidores). Proporcionam meios para interagir com servidores, podendo ser das seguintes formas:

- Sincronizados por protocolo de aplicação;
- Através de serviços remotos, oferecendo apenas uma interface de usuário conveniente.

Proporciona também aos usuários meios de interagir com servidores remotos das seguintes formas:

- Cliente possui uma aplicação sendo executada para cada serviço remoto
- Cliente possui acesso direto a serviços remotos usando apenas uma interface de usuário
- Cliente possui uma aplicação sendo executada para cada serviço remoto como por exemplo, um aplicativos em *smartfone* de um usuário e que precisa entrar em sincronia com dados compartilhados.
- Cliente possui acesso direto a serviços remotos usando apenas uma interface de usuário. Exemplo: Máquina cliente não possui nenhuma informação para processamento, tornando-a independente da aplicação → clientes minimizados → facilidade de gerenciamento do sistema.

2.3. Servidor

Iterativo x Concorrente

O termo servidor iterativo é usado para descrever implementações que processam um único pedido de cada vez, e o termo servidor concorrente para descrever implementações que manipulam múltiplos pedidos ao mesmo tempo. Neste contexto, o termo servidor concorrente está relacionado à capacidade do servidor de manipular várias requisições concorrentemente e não à sua implementação usar vários processos ou linhas de execução (*threads*). O importante é que, sob a perspectiva do cliente, o servidor parece se comunicar com vários clientes de forma concorrente. O ´processo servidor´ não manipula a requisição propriamente dita, mas a passa para uma *thread* separada ou um outro processo.

Tempo de processamento de uma requisição

Em geral, servidores iterativos são adequados apenas a protocolos de aplicação triviais. O teste para saber se uma implementação iterativa será suficiente foca o tempo de resposta necessário, que pode ser medida local e globalmente.

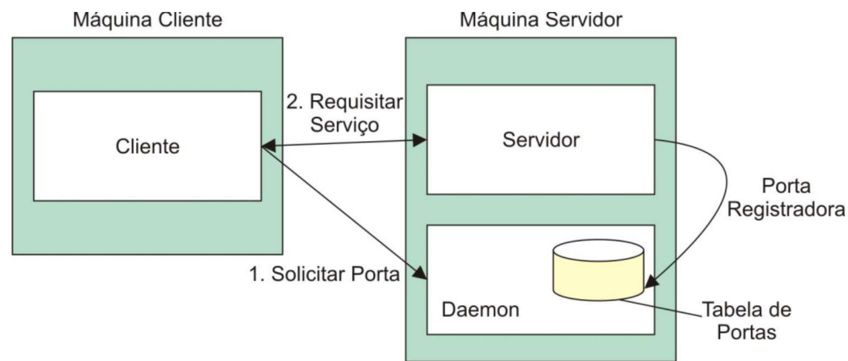
Define-se o tempo de processamento de uma requisição pelo servidor como sendo o tempo total que o servidor leva para tratar um pedido isolado, e o tempo de resposta observado pelo cliente como sendo o atraso total entre o instante em que o pedido é enviado e o instante em que a resposta do servidor chega. Evidentemente, o tempo de resposta observado nunca poderá ser menor que o tempo de processamento de uma requisição, entretanto, se o servidor mantêm uma fila de pedidos esperando atendimento, o tempo de resposta observado pode ser muito maior que o tempo de processamento de uma requisição.

Servidores iterativos manipulam um único pedido por vez. Se um pedido chega enquanto o servidor estiver ocupado manipulando um pedido anterior, o novo pedido é colocado na fila. Quando o servidor termina o processamento de um pedido, ele consulta a fila. Se N denota o comprimento médio da fila de pedidos, o tempo de resposta para um pedido de chegada será aproximadamente igual a N/2 + 1 vezes o tempo que o servidor leva para processar um pedido. Como o tempo de resposta observado é proporcional a N, a maioria das implementações restringe N a um valor pequeno (por exemplo 5) e usa-se servidores concorrentes quanto uma fila pequena não for suficiente.

A carga total tratada pelo servidor também deve ser considerada na decisão de implementação de servidores iterativos ou concorrentes. Um servidor projetado para manipular K clientes, cada qual enviando R pedidos por segundo, deve ter um tempo de processamento menor que 1/KR segundos por pedido. Se o servidor não puder tratar os pedidos a essa taxa, ocorrerá o transbordamento na fila de espera. Nesse tipo de situação, o projetista deverá considerar a implementação de um servidor concorrente.

Geralmente, requisições são enviadas a um processo servidor através de uma porta. Para alguns serviços, são atribuídas portas padrão (HTTP 80, FTP 21, SSH 22, por exemplo). Em alguns casos, pode-se implementar um *daemon* (Figura 9), que escutará uma porta conhecida e redireciona a requisição para a porta do serviço.

Daemon: acrônimo de Disk And Execution MONitor (Monitor de Execução e de Disco), é um programa de computador que roda em background, ao invés de ser controlado diretamente por um usuário.



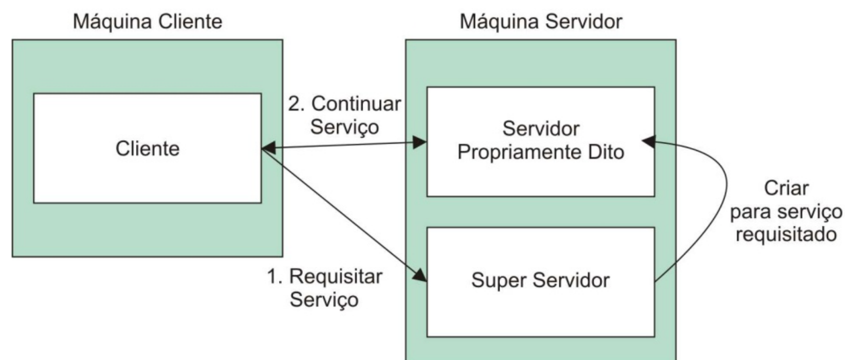


Figura 9: Vinculação cliente-servidor [Tanenbaum 2008]

Existem dois tipos de servidores. São eles:

- Servidores com estado

  Servidores com estado armazenam informação sobre cada cliente. Operações podem ser implementadas de forma mais eficiente. Mensagens com pedidos podem ser menores. Exemplo: servidor de arquivos que permite um cliente manter cópia local de um arquivo. Servidor guarda os clientes que têm permissão de mudanças no arquivo antes de atualizá-lo localmente.

- Servidor sem estado

  Em um servidor sem estado cada pedido deve conter toda a informação necessária para seu processamento. Servidores sem estado apresentam maior escalabilidade.

Escalabilidade: comportamento quando o número de clientes cresce.

A implementação de um servidor sem estado é muito mais simples que a de um servidor com estado. Neste caso, não se mantém informações sobre os estados de seus clientes e pode-se mudar o seu estado sem ter que informar a nenhum cliente como por exemplo, um Servidor Web ou Servidor de arquivos.

2.4. Virtualização

Virtualização é uma técnica que permite compartilhar e utilizar recursos de um único sistema computacional em vários outros denominados de máquinas virtuais. Cada máquina virtual oferece um sistema computacional completo muito similar a uma máquina física. Com isso, cada máquina virtual pode ter seu próprio sistema operacional, aplicativos e oferecer serviços de rede. É possível ainda interconectar (virtualmente) cada uma dessas máquinas através de interfaces de rede, *switches*, roteadores e *firewalls* virtuais.

O elemento principal para a virtualização é o sistema operacional, que basicamente pode ser definido como uma camada de software entre o hardware e as aplicações de usuários, que tem como objetivo funcionar como interface entre usuário e computador, tornando sua utilização mais simples, rápida e segura.

Para garantir e controlar a execução dos programas, o sistema operacional associa o programa a um processo que fará uso dos recursos computacionais disponíveis. Em resumo, poderíamos dizer que este processo é um programa em execução.

Tipos de Virtualização

Existem várias formas de implementar uma solução de virtualização. Na verdade, existem várias maneiras que permitem atingir o mesmo resultado através de diferentes níveis de abstração [Vieira 2008]. A implementação de máquinas virtuais pode ser feita de dois modos: Máquina Virtual de Processo ou Monitor de Máquina Virtual, também conhecido como *hypervisor*.

Máquina Virtual de Processo (Figura 10) cria-se um ambiente de execução para outras aplicações, e pode gerar um conjunto de instruções que são interpretadas para criação de instruções não-privilegiadas, chamadas de sistemas e APIs de bibliotecas que correspondem à ação abstrata desejada.

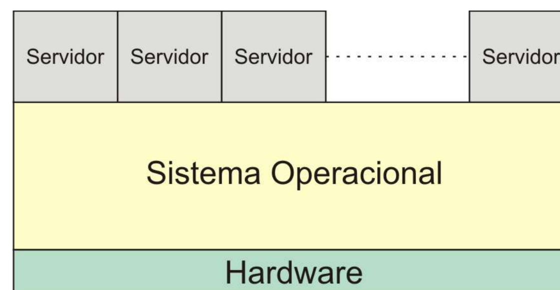


Figura 10: Máquina Virtual de Processos [Vieira 2008]

Monitor de Máquinas Virtuais (Figura 11), ou *hypervisor*, é uma camada de software entre o hardware e o sistema operacional que oferece uma cópia virtual do hardware, incluindo modos de acesso, interrupções, dispositivos de E/S, entre outros.

API, de *Application Programming Interface* (ou Interface de Programação de Aplicações) é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por programas aplicativos que não querem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços.

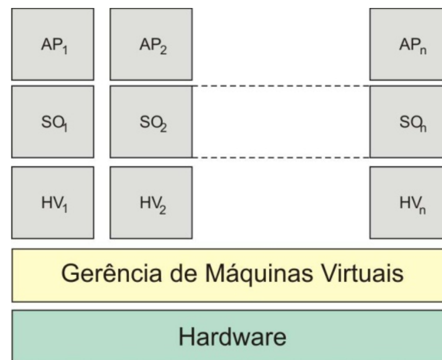


Figura 11: Monitor de Máquinas Virtuais [Machado e Maia 2007]

Uma das diferenças entre os modos de implementação está na forma em que são executados. O *hypervisor* estará sempre presente enquanto o computador estiver ligado, enquanto, a Máquina Virtual de Processo só existirá enquanto o processo correspondente a ela estiver em execução.

O *hypervisor*, como também é chamado o Monitor de Máquinas Virtuais, pode ser implementado através de dois métodos: Virtualização Total e Para-virtualização.

O modelo de virtualização total provê uma cópia (virtual) do hardware do *host*, de forma que o sistema operacional convidado trabalhe como se estivesse executando diretamente sobre o hardware do *host*, e utiliza o (*Virtual Machine Manager*) (VMM) como mediador entre o sistema operacional convidado e o hardware do *host*, porque determinadas instruções devem ser tratadas e protegidas, pois o hardware do *host* não é de propriedade do sistema operacional da máquina virtual. Uma característica que também é uma vantagem é de que o sistema operacional hóspede não precisa ser modificado.

Uma das desvantagens dessa solução é que ela provê suporte a um conjunto de dispositivos genéricos, o que pode causar subutilização dos recursos disponíveis. Outro problema é que como o sistema operacional hóspede não necessita ser modificado, logo cada instrução gerada por ele deve ser testado pelo VMM, o que gera *overhead*.

Já o método de Para-Virtualização propõe que o sistema operacional hóspede saiba que ele está sendo executado sobre um VMM e que possa interagir com ele. Dessa forma, são necessárias mudanças no sistema operacional hóspede para que ele possa chamar o VMM sempre que executar uma instrução considerada sensível. Isso garante uma cooperação entre eles levando ao aumento do desempenho do método, visto que os recursos disponíveis são utilizados de maneira mais apropriada.

Suporte de Hardware para Virtualização

Devido a proteções existentes nos modos de operação dos processadores da arquitetura x86, os fabricantes AMD e Intel, começaram a desenvolver chips com extensões que dão suporte a virtualização e trata instruções consideradas como sensíveis de forma apropriada.

As soluções criadas por esses fabricantes são incompatíveis, porém servem ao mesmo propósito. *Pacifica* é o codinome para as extensões dos processadores AMD, sua tecnologia é denominada de AMD-V, aplica-se às arquiteturas x86 e de 64 bits presentes nos processadores

mais recentes de sua linha. A Intel por sua vez apresenta suas extensões para processadores das arquiteturas x86 e 64 bits, conhecida como IVT (*Intel Virtualization Technology*).

É importante atentar para a ocasião de se utilizar uma ferramenta de virtualização desenvolvido para uma dessas tecnologias, pois AMD-V e IVT não são compatíveis, podendo gerar problemas de compatibilidade ou instabilidade, quando executados em processadores para o qual o software não foi escrito.

Exemplo de Utilização

Fica evidente que o objetivo principal da virtualização é a consolidação de servidores, mas é possível aplicar seu uso de várias formas e em diversas áreas, principalmente em alguns cenários práticos, alguns deles são [Vieira 2008]:

- Ensino – ambientes virtuais podem ser criados e testados sem comprometer a estrutura computacional existente.
- *Honeypots* – são máquinas colocadas intencionalmente na internet sem nenhum tipo de proteção, como antivírus e firewall, para que elas possam ser atacadas por *rackers*. Com o objetivo de monitorar o ataque e suas formas. Quando um *honeypot* é comprometido, pode ser substituído facilmente por outro, em alguns casos apenas restaurando a VM a um momento anterior. A grande vantagem de sua utilização é a de não comprometer a rede real.
- Consolidação de servidores – consiste em centralizar e/ou diminuir o número de equipamentos e de aplicações instaladas em cada um dos servidores da organização, com o objetivo de aumentar a produtividade da infra-estrutura, melhorar o gerenciamento do ambiente, aumentar a segurança, diminuir a manutenção e economizar em recursos humanos, físicos e financeiros.
- Consolidação de aplicações – possibilidade de virtualizar o hardware para o qual essas aplicações foram projetadas, podendo reunir em um único computador todas as aplicações legadas.
- *Sandboxing* – provimento de ambiente seguro e isolado para execução de aplicações não confiáveis.
- Disponibilidade – é possível ter ambientes com múltiplas execuções simultâneas aumentando a disponibilidade dos recursos oferecidos pela rede, como sistemas operacionais e outros serviços.
- Migração de software – facilita o processo de testes antes de colocar o software em produção, dessa forma, reduz-se o impacto sobre as migrações de software.
- Testes e medições – é possível criar cenários virtuais que, são difíceis de reproduzir em maquinas reais, facilitam testes sobre determinadas soluções.

2.5. Migração de código

Migração de código (Figura 12) é a transferência de processos entre as máquinas que constituem uma rede de computadores. A tarefa de mover um processo sendo executado para outra máquina é custosa e desafiadora [Tanenbaun 2008].

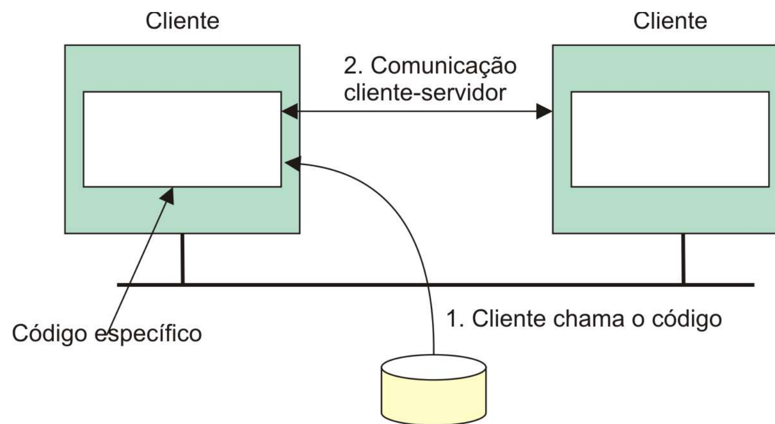


Figura 12: Modelo de Migração de Código [Tanenbaum 2002]

Existem dois modelos para migração de código, o *weak mobility* e *strong mobility*. Para descrever estes modelos é preciso primeiro dizer que um processo é dividido em três segmentos:

- Código: é a parte que contém o conjunto de instruções que faz o programa ser executado;
- Recurso: contém a relação de recursos externos necessários para a execução do programa;
- Execução: utilizado para guardar o estado corrente de um processo, consistindo de dados privados, a pilha e o contador de programa.

Tendo entendido estes conceitos é possível agora entender os modelos de migração weak e strong mobility.

No modelo *Weak Mobility* é possível apenas a transferência do segmento de código, podendo também enviar alguns dados de inicialização. Uma característica deste modelo é que o programa é transferido em seu estado inicial. Um exemplo de aplicação que utiliza este modelo é o *applet* do Java [Tanenbaum 2002].

Já no modelo Strong Mobility o segmento de execução também pode ser enviado. A característica deste modelo é que um processo que esteja sendo executado pode ser parado e enviado para outra máquina onde reinicia sua execução do ponto em que havia parado. Este modelo é mais poderoso que o *weak mobility*, mas é mais complexo de implementar [Tanenbaum 2002]. O segmento de recursos em muitos casos não é transferido, pois ele torna os modelos de transmissão muito mais complexos, uma vez que para a transmissão deste segmento ele deve sofrer alterações.

Exercícios

As questões abaixo devem ser respondidas em forma dissertativa e argumentativa com pelo menos uma lauda. Devem também refletir a interpretação da leitura do texto juntamente com pesquisas sobre o tema arguido.

1. Explique a importância da transparência em Sistemas Distribuídos.
2. Qual a diferença entre Multiprocessadores baseados em barramentos e multiprocessadores com *switch*?
3. Qual a diferença entre sistemas fracamente e fortemente acoplados?
4. Como funciona a arquitetura cliente/servidor e como eles podem ser organizados?
5. O que são *Threads*?
6. O que é exclusão mútua?
7. Existem dois tipos de servidores: com e sem estado. Explique como funciona cada um.
8. Quais são os tipos de virualização?
9. O que é o *hipervisor*?
10. Como se dá a migração de código em um sistema distribuído?

Resumo

Nesta Unidade foram abordados temas introdutórios doa Sistemas Distribuídos. Foi introduzido o conceito de transparência bem como a arquitetura cliente/servidor, comumente utilizada nos SDs, diferenciando também aspectos de software e hardware presente neste tipo de abordagem.

Além disso, foi mostrado como são implementados os processos e a definição de *Thread,* que são estrutura de execução onde estes processos revezam seu funcionamento. Por fim foi visto como é realizada a virtualização e a migração de código, que irão permitir os processos de uma máquina para outra em um sistema distribuído.

Bibliografia

AMARAL, W. H. **Arquitetura Cliente/Servidor Orientada a Objeto**. Tese de Mestrado, IME, 1993.

BERSON, A. **Client/Server Architecture**, McGraw-Hill 1992, ISBN: 0070050767

COMER, Douglas E. & STEVENS, David L. **Internetworking With TCP/IP Vol.III: Client/Server Programming and Application (Socket Version)**. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1993.

COULOURIS, G; et al. **Sistemas Distribuídos: Conceitos e Projeto**. Bookman, 2007

MACHADO, Francis Berenger & MAIA, Paulo Luiz. **Arquitetura de Sistemas Operacionais**. 4ª edição. LTC, 2007.

RENAUD, P. E. **Introdução aos Sistemas Cliente/Servidor.** IBPI Press, RJ 1993.

SALEMI, Joe. **Banco de Dados Cliente/Servidor**. IBPI Press, 1993

SANTOS JÚNIOR, J. M. Rodrigues. **Threads em Java.** Monografia. Mestrado em Informática Fora de Sede - Universidade Tiradentes Aracaju – Sergipe, 2000

SILBERSCHATZ, Abraham, GALVIN, Peter B, GAGNE, Greg. **Operating system concepts**. 7.ed. John Wiley & sons, inc. 2004.]

STALLINGS, W. **Operating Systems Internals and Design Principles**. 3.ed. New Jersey: Prentice -Hall, 1998.

TANENBAUM & VAN STEEN. **Sistemas Distribuídos: Princípios e Paradigmas**. 2. ed., Prentice-Hall, 2008.

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems principles and paradigms**. 1. ed. Prentice Hall, 2002.

VIDAL, Paulo César Salgado. **Modelagem para Arquitetura Cliente/Servidor Orientada a Objeto**. Tese de Mestrado, IME, 1994.

Weblografia

Application Program Interface - Dictionary of Computing
http://foldoc.org/Application+Program+Interface
Acessado em: 12/10/2010

CARVALHO, Bruno Motta de – Notas de aula de Sistemas distribuídos
http://www.dimap.ufrn.br/~motta/dim070/Introducao.pdf
Acessado em: 10/12/2010

Modelo OSI e TCP/IP
http://www.abusar.org.br/ftp/pitanga/Aulas/a01_modelos.pdf
Acessado em: 13/11/2010

SIMOMURA, Bruno Celio – Sistemas Distribuídos
http://www.artigonal.com/ti-artigos/sistemas-distribuidos-991878.html
Acessado em: 09/08/2010

SOUTO, Gustavo Henrique - Princípios e Finalidades do Middleware
http://gustavosouto.wdfiles.com/local--files/p-e-i-middleware/princ_final_middleware.pdf
Acessado em: 23/11/2010

VIEIRA, W. Viana. – Virtualização (2008)
http://www.slideshare.net/wancleber/virtualizacao-3258343
Acessado em: 18/09/2010

UNIDADE II

Comunicação

Objetivos

1. Entender os fundamentos da comunicação em Sistemas Distribuídos
2. Entender a importância do modelo OSI na comunicação em SDs
3. Conhecer como funciona a chamada de procedimento remoto
4. Aprender com é feita a nomeação em sistemas distribuídos
5. Aprender como é realizada a implementação de resolução de nomes.

3. Comunicação

A diferença mais importante entre os Sistemas Distribuídos e os sistemas uniprocessados é a comunicação inter-processo. Nos uniprocessados esta comunicação é feita por meio de memória partilhada. Nos sistemas distribuídos não existe memória partilhada e toda comunicação inter-processo tem que ser formulada com base em troca de mensagens.

Sistemas distribuídos modernos consistem em milhares ou até milhões de processos espalhados por uma rede cuja comunicação não é confiável, como a Internet. A menos que os recursos de comunicação oferecidos pelas redes de computadores sejam substituídos por algum outro meio, o desenvolvimento de aplicações em grande escala é extremamente difícil. [Tanebaum 2008]

Neste capítulo discutiremos as regras que às quais os processos tem que obedecer, conhecidas como protocolos e veremos alguns modelos de comunicação de ampla utilização.

3.1. Modelo de comunicações

3.1.1. Modelo OSI

Para facilitar o processo de padronização e obter interconectividade entre máquinas de diferentes fabricantes, a Organização Internacional de Padronização (ISO – *International Standards Organization*) aprovou, no início dos anos 80, um modelo de referência para permitir a comunicação entre máquinas heterogêneas, denominado OSI (*Open Systems Interconnection*). Esse modelo serve de base para qualquer tipo de rede, seja de curta, média ou longa distância, inclusive para os sistemas distribuídos.

Na área das comunicações, um protocolo é um conjunto de regras ou convenções que governam a operação e o intercâmbio de informações entre dois sistemas computadorizados. Tanto o modelo OSI como o TCP/IP (e também o SNA) funcionam através de pilhas de protocolos, formando assim diversos níveis, um utilizando os serviços do nível inferior, possuindo as seguintes vantagens:

- Sistema estruturado;
- Facilidade de entendimento e visualização;
- Permite a interconexão entre sistemas de diferentes fabricantes, desde que o padrão de cada nível seja aberto.

Devido a essas vantagens, os sistemas surgiram estruturados em níveis, e cada nível foi criado com os seguintes objetivos:

- Um nível deve ser criado sempre que uma nova forma de abstração é necessária;
- Cada nível deve executar uma tarefa bem definida;
- A tarefa de cada nível deve procurar se adaptar a protocolos já existentes;
- Os limites entre os níveis devem ser escolhidos de modo a minimizar o fluxo de informação entre eles.

Tanto o modelo OSI como o modelo TCP/IP são estruturados em pilhas de protocolos.

O modelo OSI é dividido em sete níveis, sendo que cada um deles possui uma função distinta no processo de comunicação entre dois sistemas abertos. A Figura 13 mostra os sete níveis do modelo OSI, que serão analisados a seguir, iniciando pelo nível mais próximo ao meio físico e terminando no nível mais próximo do usuário. Pode-se ver através da figura que cada nível possui um ou mais protocolos que realizam as funções específicas daquele nível, e esses protocolos são compatíveis entre as máquinas que estão se comunicando (*host* **A** e *host* **B**).

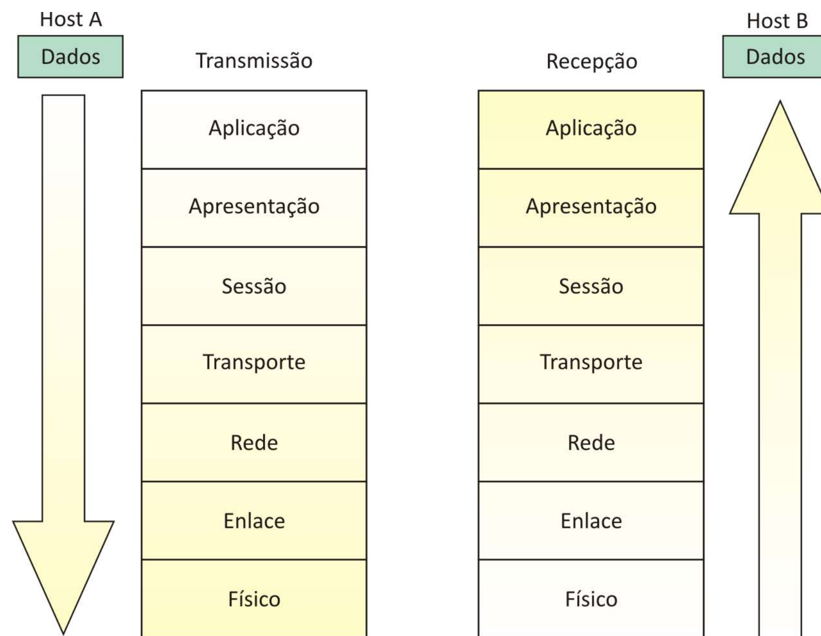


Figura 13: As 7 camadas do modelo OSI

Entre cada nível existe uma interface. Essa interface permite que dois níveis quaisquer troquem informações. A interface também define quais primitivas, operações e serviços o nível inferior oferece ao imediatamente superior. Cada nível é independente entre si e executa somente suas funções, sem se preocupar com as funções dos outros níveis. Assim, por exemplo, o nível 2 preocupa-se em fazer uma transmissão livre de erros, não importando se o nível físico esteja utilizando par trançado, cabo coaxial ou fibra ótica.

3.1.2. Comunicação Orientada a mensagem

Na comunicação orientada a mensagem a troca de dados entre Cliente e Servidor procede de forma implícita. Quando o Cliente espera a resposta da mensagem enviada para continuar o seu processamento, diz-se que o protocolo utilizado é um protocolo com bloqueio, onde o sincronismo entre Cliente e Servidor está implícito no mecanismo de passagem de mensagem [Tanenbaum 2002].

Caso o Cliente possa continuar suas tarefas, enquanto espera a resposta da mensagem, o protocolo de comunicação é um protocolo sem bloqueio. Isto ocorre quando o sistema operacional do Cliente é multitarefa ou multiprocessamento, possibilitando ao Cliente executar outras tarefas enquanto aguarda a resposta do Servidor. A teoria de programação concorrente é baseada na noção de processos de comunicação sendo executados em paralelo a outros processos.

Esses processos se comunicam compartilhando memória ou passando mensagens por meio de um canal de comunicação compartilhado [Amaral 1993] [Vidal 1994]. O termo IPC (*Interprocess Comunication*) se refere às técnicas utilizadas na passagem de mensagem.

No compartilhamento de memória, os processos concorrentes compartilham uma ou mais variáveis, e utilizam a mudança de estados dessas variáveis para se comunicarem. Essa técnica inclui espera ocupada, semáforos, regiões críticas condicionais e monitores [Tanenbaum 2002]. Como esta técnica exige que os processos estejam na mesma máquina, não são considerados base para a programação Cliente/Servidor.

Em técnicas baseadas na passagem de mensagem, os processos enviam e recebem mensagens explicitamente, em vez de examinar o estado de uma variável compartilhada [Comer 1993]. O benefício principal da passagem de mensagem é que existe pouca diferença entre o envio de mensagens a processos remotos ou locais. Portanto, a passagem de mensagem é poderosa para criação de aplicações em rede. Outra vantagem é que mais informações podem ser trocadas numa mensagem do que por mudança no estado de uma variável compartilhada.

3.1.3. Comunicação via Middleware

Middleware é uma camada que está localizada sobre a plataforma, Hardware e S.O., e sob a camada de aplicação. Fornece um conjunto de API para desenvolver em especificas áreas, por exemplo, rede, sistemas distribuídos, segurança e etc.

> API, de *Application Programming Interface* (ou Interface de Programação de Aplicações) é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por programas aplicativos que não querem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços [http://foldoc.org/Application+Program+Interface].

O *Middleware*, assim como o modelo OSI, é composto por camadas, em sua infraestrutura é provida de mecanismos para reutilizar eventos de demultiplexação, comunicação interprocessos, concorrências e sincronização de objetos.

A reutilização de API's e objetos do modelo automático e mecanismos nativos do S.O., são encapsulados pela infraestrutura do *Middleware*, assim então, as aplicações podem invocar operações de objetos, sem a questão de dependências, como: Interconexão, protocolos de comunicação, plataforma de SO ou até mesmo hardware.

Um bom *Middleware* faz aplicações confiáveis sem a necessidade de torná-las complexas, podendo ser mais escalonáveis. Existem dois modos de comunicação distribuída interprocessos baseadas no *Middlewere*, estas são: *Remote Procedure Call* (RPC) e *Messaging*.

O RPC reflete uma mudança no contexto computacional, além dos limites de um único CPU permitindo que uma aplicação, denominada cliente, invoque serviços providos por outra aplicação, chamada servidor. Esse último pode residir na mesma máquina que o cliente, mas em processos distintos, ou até mesmo, em máquinas distintas.

Existem alguns detalhes que, intencionalmente, o RPC esconde do cliente, os detalhes são:

- Transferência de Dados;
- Programação de Rede;
- Falhas;

Na transferência de dados um ou mais parâmetros retornam um valor, esse parâmetro é passado numa rede entre o cliente e o servidor onde este é convertido para outra forma de transmissão na rede.

Na programação de rede para se ter uma requisição remota, uma ou mais mensagens devem ser enviadas da aplicação do cliente para o servidor, essas mensagens contem informações, também. Contém um "Conversor" de linguagem de programação representativa para outra forma de transmissão de rede, está é chamada de *marshaling*, e a sua forma contrário é denominada de *demarshaling*.

O protocolo RPC pode utilizar tanto o protocolo orientado a conexão quanto orientado sem conexão, mas por razões de "modernidade", é utilizado o protocolo de conexão orientada, também usado para esconder os detalhes do estabelecimento da conexão, da aplicação, faz uso do protocolo de controle de transmissão (TCP).

O *Middleware* trata problemas como, perda e duplicação de pacotes, ordem inversa de pacotes e várias outras dificuldades, para proteger a aplicação [Souto 2010]. Já nas falhas, há a chance de falhas parciais na aplicação de rede, mesmo assim se essa executar em um único espaço de endereço, poderá vim à ocorrer falhas.

O *Middleware* suporta comunicação via UDP mas é essencialmente recomendado a implementação via TCP. O *Middleware* é capaz de prever o estouro de tempo das aplicações, também, às vezes, pode proteger de erros inerentes de computação distribuída.

A *Messaging* age diferente do RPC, o sistema de mensagem propicia diretamente e explicitamente a sua exposição e das API´s para a aplicação, essa não imita as chamadas de procedimento, linguagens de programação ou métodos de invocação. Ela não provê suporte a *marshaling*, significando que ela executa a sua formatação e interpretação.

Nas concorrências muitas aplicações baseadas em *Middleware's* tem um alto nível de escalabilidade, basicamente em temos de números de requisições ou processamento de mensagens por segundo. Esses requerimentos maximizam por total a concorrência no sistema, podendo realizar várias operações ao mesmo tempo, um sistema de *Middleware* incorpora diversas técnicas e padrões para aumentar a concorrência, também, adicionando *multithread's*, a concorrência pode envolver muitos processos.

3.2. Chamada de procedimento (RPC)

RPC (*Remote Procedure Calls*) é um método comum e largamente aceito para o encapsulamento de mensagens em um sistema distribuído [Stallings 1998]. A essência desta técnica é permitir que programas em diferentes máquinas possam interagir usando simples semântica de procedimentos *call/return*, como se os dois programas estivessem na mesma máquina. Isto é a chamada de um procedimento é usada para acessar serviços remotos. A popularidade deste enfoque é devido as seguintes vantagens:

- A chamada de um procedimento é amplamente aceita, usada e de abstração entendida.
- O uso de chamadas remotas de procedimentos permite que interfaces remotas sejam designadas como um conjunto de operações nomeadas com tipos designados. Assim esta interface pode ser claramente documentada e os programas distribuídos podem ser conferidos estaticamente, em tempo de compilação, para validar os erros de tipos (permite conferir os tipos de dados que estão sendo usados nas chamadas, em tempo de compilação).
- Devido à especificação de uma interface padronizada e precisa o código de comunicação para uma aplicação pode ser gerado automaticamente.
- Devido à especificação de uma interface padronizada e precisa o programador pode escrever módulos clientes e módulos servidores que podem ser usados em diferentes computadores e sistemas operacionais com pequenas modificações e pouca recodificação.

Na Figura 14 tem-se uma visão do mecanismo de RPC [Goulard 2010]. O programa que faz uma requisição faz uma chamada comum de procedimentos com parâmetros em sua própria máquina. Por exemplo a chamada CALL P(X,Y) onde P = Nome do procedimento, X = Parâmetro enviado e Y = Parâmetro recebido.

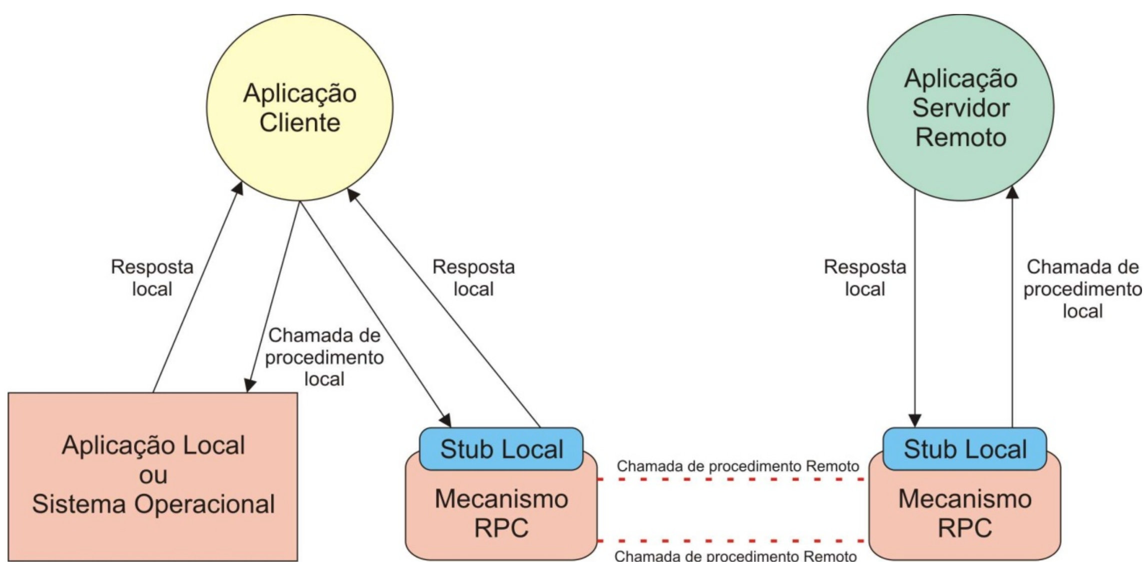


Figura 14: Chamada de Procedimento Remoto [Goulart 2010]

A intenção é chamar um procedimento remoto em outra máquina que pode ou não ser transparente ao usuário. Um pequeno procedimento, chamado de *STUB*, P deverá ser incluído no procedimento que chama ou ser ligado dinamicamente na hora da chamada. Este *STUB* cria uma mensagem que identifica o procedimento sendo chamado e inclui os parâmetros. Envia a mensagem ao sistema remoto e aguarda a resposta. Quando uma resposta é recebida este STUB P retorna ao programa chamador passando os valores que retornaram.

Na máquina remota, um outro *STUB* é associado com o procedimento chamado. Quando uma mensagem chega, ela é examinada e uma chamada local CALL P(X,Y) é gerada. Este

procedimento remoto é assim chamado localmente e tratado como uma chamada local. Em resumo os passos para um RPC são:

1. O procedimento cliente chama localmente o *stub* cliente;
2. *Stub* cliente constrói a mensagem e chama o SO local;
3. O SO local envia a mensagem ao SO remoto;
4. O SO Remoto entrega a mensagem ao *stub* servidor;
5. O *stub* servidor desempacota os parâmetros e chama o procedimento servidor;
6. O servidor realiza o trabalho retornando o resultado ao *stub*;
7. O *stub* servidor empacota o resultado em uma mensagem e chama o SO local;
8. O SO do servidor envia a mensagem ao SO do cliente;
9. O SO do cliente entrega a mensagem ao *stub* cliente;
10. O *stub* cliente desempacota o resultado e o retorna ao cliente.

São características básicas que se pode observar no mecanismo RPC:

- Passagem de parâmetros – Geralmente por valor e não por referência pois os dados dos parâmetros são copiados dentro da própria mensagem enviada.
- Representação dos parâmetros – Quando se tem ambientes heterogêneos pode-se ter diferença na representação de números e até de texto (tipos inteiros, tipos *float*, ASCII/EBCDIC).
- Forma de ligação – A forma de ligação entre o cliente e o servidor pode ser de forma persistente e não persistente. Diz-se persistente quando é estabelecida a comunicação na primeira chamada e esta ligação permanece após a primeira resposta. De outra forma quando a cada mensagem é estabelecida uma comunicação, enviada a mensagem, recebida a resposta e desfeita a comunicação, chamamos de ligação não persistente.
- Sincronismo – O tradicional é a chamada de procedimentos de forma síncrona, ou seja o procedimento que chamou, passa a mensagem e fica aguardando a resposta do procedimento chamado. Para prover flexibilidade, várias facilidades de chamadas de procedimentos assíncronas foram implementadas visando um maior grau de paralelismo enquanto mantêm a simplicidade das chamadas de procedimentos remotos.

Segundo Tanenbaum (2002), o objetivo da chamada remota de procedimentos é manter escondido do usuário todos os procedimentos relativos à comunicação remota.

3.2.1. Passagem de parâmetros

A passagem dos parâmetros deve ser muito bem organizada porque se as máquinas cliente e servidor são diferentes. Neste caso é necessário que elas façam uma conversão dos dados para a sua própria representação dos bytes. Isto será feito a partir do tipo de cada parâmetro recebido como por exemplo: inteiro, caractere, cadeia de caractere, ponto flutuante, etc. Uma informação é colocada na mensagem indicando qual o formato dos caracteres. Quando esta mensagem chega ao destino. o receptor verifica se o formato é o mesmo que é utilizado por ele. Se sim, a conversão não é necessária, se não, a conversão será realizada. Uma solução adotada no caso de vetores é que o emissor faz uma cópia do mesmo colocando-o na mensagem, desta forma a máquina destino pode manipulá-lo e devolvê-lo à origem.

A passagem de parâmetros pode ser dada de três formas: por valor, por referência ou por copy/restore.

Por valor (fd e nbytes): O parâmetro é copiado para a pilha. Mudanças em seu valor não afetam quem chamou o procedimento.

Por referência (buf): A referência ao parâmetro é copiada para a pilha, isto é, um apontador (endereço de memória) da variável. Portanto, alterações em seu valor no procedimento chamado afeta quem o chamou. Sem sentido, em princípio, por não haver espaço de endereçamento comum. Contudo algumas soluções contornam o problema tais como:

- Empregar uma passagem de parâmetros do tipo *call-by-copy*/restore.
- Diferenciar o ponteiro a cada referência no servidor (mensagens adicionais).
- Usar mecanismos de DSM (*Distributed Shared Memory*) ou orientação a objetos.

Call-by-copy/restore: Chamada por cópia e restauração - a variável é copiada para a pilha (como se fosse em uma chamada por valor) e no retorno da chamada o valor alterado sobrescreve o valor original (como se fosse uma chamada por referência)

Copy/restore: O parâmetro é copiado para a pilha e depois da chamada copiado de volta, sobrescrevendo o valor original. Normalmente possui a mesma semântica da cópia por referência exceto em situações como o mesmo parâmetro sendo enviado mais de uma vez

Geração de Stubs

Em vários sistemas baseados em RPC os *stubs* são gerados automaticamente. Tendo a especificação do procedimento do servidor e as regras de codificação o formato da mensagem pode ser bem definido. O compilador lê a especificação do servidor e gera um *stub* cliente que empacota os parâmetros em um formato pré-determinado. É gerado um *stub* servidor que desempacota os parâmetros e chama o servidor. Isso reduz as chances de erros e torna o sistema transparente em relação a diferentes representações de dados.

3.2.2. RPC Assíncrono

Uma variação em torno do tema RPC são as interconexões assíncronas ou RPC assíncrono, que são um tipo de RPC que não exige que o cliente se bloqueie aguardando a resposta do servidor e, às vezes, não requer nem mesmo que uma resposta seja enviada.

A rigor, não se poderia chamar uma interação cliente/servidor deste tipo de RPC, uma vez que o bloqueio do módulo que chamou a rotina (pré-requisito básico de uma chamada a procedimentos) não ocorre, porém este termo já se firmou na literatura sobre o assunto.

O benefício do RPC assíncrono é que ele vem justamente eliminar algumas “desvantagens” do RPC convencional, como, por exemplo, a necessidade do cliente bloquear-se esperando pela resposta do servidor, ao invés de ficar executando outras operações em paralelo, o que melhoraria a resposta do sistema. Por outro lado, a programação utilizando primitivas assíncronas é mais complexa e, no caso de requisições que não pedem uma resposta, fica difícil saber se estas foram executadas no servidor.

De qualquer forma, há casos em que é vantajoso utilizar o RPC assíncrono. Por exemplo, sistemas de janelas geralmente usam alguma forma de comunicação assíncrona [Coulouris 2007]. O sistema, nesse caso, é programado como um servidor e os programas que quiserem mostrar textos ou gráficos em uma janela devem enviar solicitações a ele.

Fica clara a melhoria de desempenho proporcionada pelo RPC assíncrono, uma vez que a mensagem do cliente, muitas vezes, é um caractere ou um movimento do apontador do mouse e a resposta vem a ser a própria alteração na tela, não sendo necessário e até indesejável o bloqueio do cliente a cada tecla pressionada. Uma melhoria, inclusive, seria armazenar diversas solicitações do cliente e enviá-las ao servidor em intervalos de tempo pré-determinados (de milissegundos), de modo a fazer melhor uso capacidade da rede, uma vez que uma RPC já possui um *overhead* inevitável. O RPC assíncrono também possibilita que um cliente envie requisições em paralelo a diversos servidores.

3.2.3. Comunicação Socket

A comunicação *Socket* Interface de acesso às facilidades locais para comunicação remota, Fornecendo "pontos finais" (*endpoints*) para a comunicação entre processos (*end-to-end*) e geralmente é ligado a uma porta local.

É um serviço de transporte de sequência de bytes permitindo a comunicação entre os processos segundo protocolo estabelecido para a forma dos bytes enviados. Possui ainda baixo nível de abstração suportado pela arquitetura e maior organização das aplicações. Contudo, exige um maior esforço de desenvolvimento/depuração.

Um *socket* pode ser usado em ligações de redes de computadores para um fim de um elo bidirecional de comunicação entre dois programas. A interface padronizada de soquetes surgiu originalmente no sistema operacional Unix BSD (*Berkeley Software Distribution*). Portanto, eles são muitas vezes chamados de Berkeley Sockets. É também uma abstração computacional que mapeia diretamente a uma porta de transporte (TCP ou UDP) e mais um endereço de rede. Com esse conceito, é possível identificar unicamente um aplicativo ou servidor na rede de comunicação IP.

O *socket* possui uma clara distinção entre quem é o servidor e quem é o cliente, podendo vários clientes se comunicar com um único servidor. Para um melhor entendimento sobre sockets, é interessante pensar nele como se fosse uma porta de um canal de comunicação que permite a um processo em execução enviar/receber dados para/de outro processo que pode estar sendo executado no mesmo computador ou num outro remotamente [Cerqueira 2005]. A Figura 15 mostra ações realizadas numa comunicação com socket em uma comunicação comum cliente/servidor.

3.2.4. Distributed Computing Environment (DCE)

Distributed Computing Environment (DCE) é um sistema de software por um consórcio que inclue HP, IBM, *Digital Equipment Corporation*, e outros. O DCE fornece uma estrutura e um toolkit para implementar aplicações cliente/servidor. A estrutura inclui o mecanismo de *Remote Procedure Call* (RPC) conhecido como DCE/RPC, um serviço de nomes (do diretório), um serviço do tempo, serviço de autenticação e a sistema de arquivo distribuído (DFS) sabido como DCE/DFS.

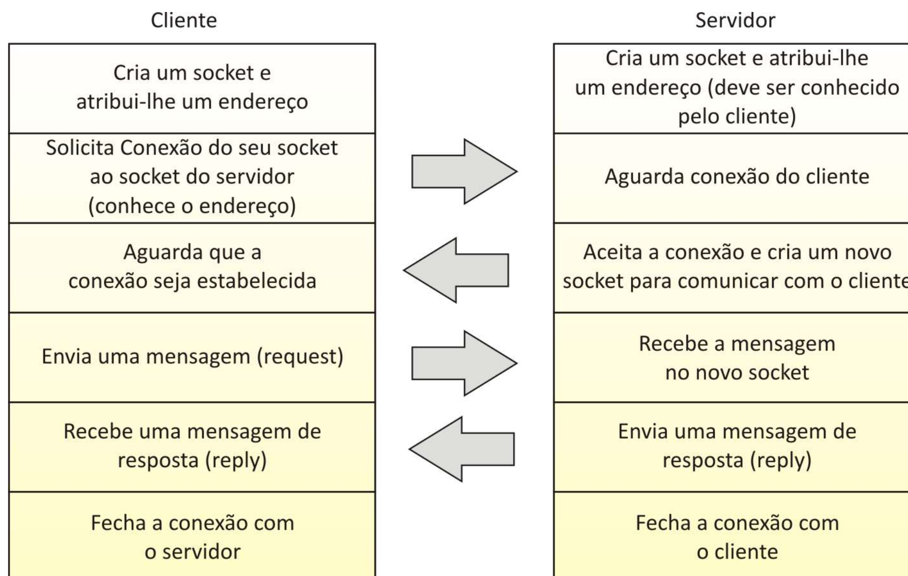


Figura 15: Modelo de comunicação socket [Cerqueira 2005].

O RPC não implementa a transparência necessária, pois é preciso saber onde executar o procedimento. O DCE resolve isso criando um serviço de diretório que endereça a localização do procedimento desejado a partir do nome do serviço (Figura 16).

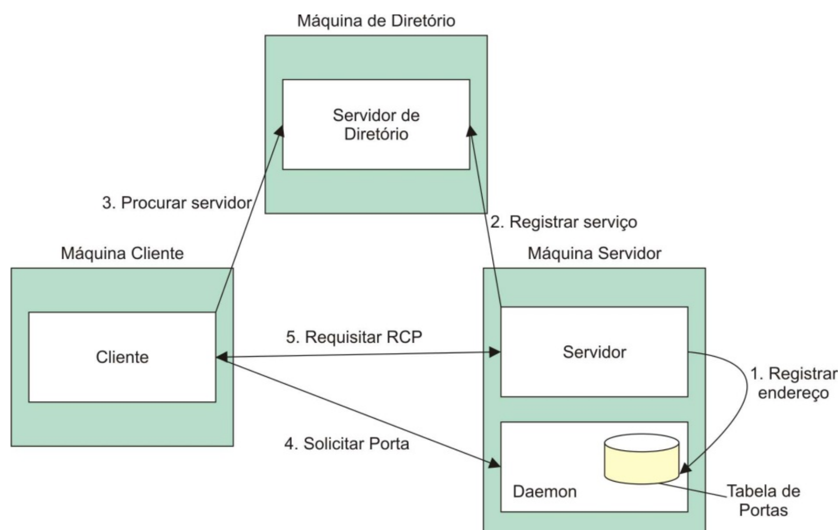


Figura 16: Serviço de diretório que endereça a localização do procedimento [Tanenbaum 2008]

3.3. *Message-Passing Interface* (MPI)

Message Passing Interface (MPI) é um padrão para comunicação de dados em computação paralela. Existem várias modalidades de computação paralela e, dependendo do problema que se está tentando resolver, pode ser necessário passar informações entre os vários processadores ou nodos de um cluster. O MPI oferece uma infraestrutura para essa tarefa.

No padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado. Porém,

esses processos podem executar diferentes programas [Zampieri 2010]. Por isso, o padrão MPI é algumas vezes referido como MPMD (*multiple program multiple data*). Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. Dado o fato do número de processos no MPI ser normalmente fixo, neste texto é enfocado o mecanismo usado para comunicação de dados entre processos. Os processos podem usar mecanismos de comunicação ponto a ponto (operações para enviar mensagens de um determinado processo a outro). Um grupo de processos pode invocar operações coletivas (*collective*) de comunicação para executar operações globais. O MPI é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores (*communicator*) que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna.

O objetivo de MPI é prover um amplo padrão para escrever programas com passagem de mensagens de forma prática, portátil, eficiente e flexível. MPI não é um IEEE[4] ou um padrão ISO, mas chega a ser um padrão industrial para o desenvolvimento de programas com troca de mensagens.

A plataforma alvo para o MPI, são os ambientes de memória distribuída, máquinas paralelas massivas, "*clusters*" de estações de trabalho. Todo paralelismo é **explícito:** o programador é responsável em identificar o paralelismo e implementar um algoritmo utilizando construções com o MPI.

3.3.1. Conceito e definições

Rank

Todo processo tem uma única identificação, atribuída pelo sistema quando o processo é iniciado. Essa identificação é continua e começa no zero até n-1 processos.

Group

Grupo é um conjunto ordenado de N processos. Todo e qualquer grupo é associado a um "*communicator*" e, inicialmente, todos os processos são membros de um grupo com um "*communicator*" já pré-estabelecido (MPI_COMM_WORLD).

Communicator

O "*communicator*" define uma coleção de processos (grupo), que poderão se comunicar entre si (contexto). O MPI utiliza essa combinação de grupo e contexto para garantir uma comunicação segura e evitar problemas no envio de mensagens entre os processos. É possível que uma aplicação de usuário utilize uma biblioteca de rotinas, que por sua vez, utilize "*message-passing*". Essa rotina pode usar uma mensagem idêntica a mensagem do usuário. As rotinas do MPI exigem que seja especificado um "communicator" como argumento. MPI_COMM_WORLD é o comunicador pré-definido que inclui todos os processos definidos pelo usuário, numa aplicação MPI.

---

[4] O Instituto de Engenheiros Eletricistas e Eletrônicos ou IEEE é uma organização profissional sem fins lucrativos, fundada nos Estados Unidos. Um de seus papéis mais importantes é o estabelecimento de padrões para formatos de computadores e dispositivos.

Aplication Buffer

É um endereço normal de memória (Ex: variável) onde se armazena um dado que o processo necessita enviar ou receber.

System Buffer

É um endereço de memória reservado pelo sistema para armazenar mensagens. Dependendo do tipo de operação de ***send/receive***, o dado no "aplication buffer" pode necessitar ser copiado **de/para** o "system buffer" (**"*Send Buffer" e "Receive Buffer*"**). Neste caso teremos comunicação assíncrona.

Blocking Comunication

Uma rotina de comunicação é dita **"*bloking*"**, se a finalização da chamada depender de certos eventos. Por exemplo, em uma rotina de envio, o dado tem que ter sido enviado com sucesso, ou, ter sido salvo no "system buffer", indicando que o endereço do "*aplication buffer*" pode ser reutilizado. Numa rotina de recebimento, o dado tem que ser armazenado no "*system buffer*", indicando que o dado pode ser utilizado.

Buffered Send

O programador cria um "*buffer*" para o dado antes dele ser enviado. Necessidade de se garantir um espaço disponível para um "*buffer*", na incerteza do espaço do "System Buffer".

Ready Send

Tipo de "*send*" que pode ser usado se o programador tiver certeza de que exista um "*receive*" correspondente, já ativo.

Standard Receive

Operação básica de recebimento de mensagens usado para aceitar os dados enviados por qualquer outro processo. Pode ser "blocking" e "non-blocking".

Return Code

Valor inteiro retornado pelo sistema para indicar a finalização da sub-rotina.

4. Nomeação

Num sistema distribuído os nomes são imprescindíveis para designar computadores, serviços, utilizadores, objetos remotos, arquivos e recursos em geral. Os diferentes componentes do sistema, assim como os utilizadores, só podem partilhar recursos se os poderem designar.

Os nomes permitem designar e identificar entidades ou objetos. Por vezes, os nomes incluem informação relativa a propriedades/atributos dos objetos como por exemplo:

“xpto@foo.com”, http://asc.di.fct.unl.pt/sd1

Um serviço de nomes permite obter dados (atributos) sobre uma entidade dado o seu nome. Já um serviço de diretório ou descoberta permite obter dados sobre as entidades que satisfazem uma dada descrição.

Nomes são sequências de símbolos (geralmente codificados como sequências de bytes, bits) que designam entidades. Um nome tem de ser interpretado para se chegar (aos dados da) entidade. Este processo de interpretação diz-se resolver ou interpretar o nome (*to resolve*). A associação entre um nome e uma entidade designa-se por ligação (*binding*).

Em geral, os nomes estão ligados a atributos da entidade e não diretamente a entidade. A interpretação de um nome deve ser feita num contexto pois o mesmo nome em contextos diferentes pode designar objetos diferentes. Os nomes tomam varias formas conforme o nível do sistema em que são interpretados.

- Designar - ação de apontar, indicar, mostrar, escolher
- Identificar - tornar idêntico, o que faz com que uma coisa seja idêntica a outra, o que permite saber se duas coisas são distintas ou não

Um nome permite designar uma entidade e em um dado contexto, uma entidade pode ser designada por mais do que um nome. Já um identificador permite identificar uma entidade. Geralmente, uma entidade tem somente um identificador. Dois identificadores distintos identificam duas entidades diferentes. Se duas entidades tiverem o mesmo identificador, então são a mesma entidade e um identificador único permite identificar uma entidade de forma permanente.

Utiliza-se o termo nome simbólico, textual, orientado aos utilizadores, ou externo para designar ou identificar entidades através de nomes mnemônicos e legíveis para os utilizadores como por exemplo: *www.die.ufpi.br*

Utiliza-se o termo identificador único sistema (UID) para designar sequências de símbolos, geralmente sem significado mnemônico, que permitem identificar entidades de forma (quase) permanente, ao nível interno do sistema distribuído. Exemplos: AF.65.8F.89.1B.23.FF.45.A5.89.8B (UID de um objeto remoto)

Utiliza-se o termo endereço para designar formas especiais de designação transitória, volátil ou temporária das entidades, geralmente associadas à localização das mesmas como por exemplo: 10.0.0.12. Um endereço é um nome que dá acesso imediato a uma entidade. Para se manipularem os objetos assim designados, os nomes são traduzidos de um espaço de designação noutro como a abertura de um arquivo em tempo de execução, determinação do endereço IP associado a um nome DNS, acesso a um objeto remoto, etc.

4.1. Espaço de nomes

Espaço de nomes é assim chamado o grupamento dos nomes (ou atributos) de entidades, conjunto de regras de especificação de nomes (ou atributos) que permitem a resolução de nomes (ou consulta de atributos) e podem ser representado por um grafo com dois tipos de nós:

- nó-folha: informações sobre entidade (atributos) que permite aceder ao conteúdo da entidade;
- nó-diretório que contém informações sobre outras entidades, a ele associadas permitindo aceder aos nós-folha dessas entidades podendo ser representado por tabela(s) e conter os atributos das entidades.

4.2. Implementação de resolução de nomes

O serviço de nomes cuida de indicar a localização de um determinado arquivo dado o seu nome ou caminho. Se a localização do arquivo estiver armazenada no nome dele, como por exemplo *ufpi:/tmp/teste*, então esse serviço de nomes não provê transparência de localização. Para prover essa transparência, o nome ou caminho de um arquivo não deve ter indícios de sua localização física, e caso esse arquivo mude de lugar, ou tenha várias cópias, o seu nome ou caminho não precisará ser alterado. Para isso, o serviço precisa oferecer ou resolução por nomes, ou resolução por localização, ou ambos.

Resolução por nomes mapeia nomes de arquivos legíveis por nós humanos, normalmente *strings*, para nomes de arquivos legíveis por computadores, que normalmente são números, facilmente manipuláveis pelas máquinas [Carvalho 2001].

Resolução por localização mapeia nomes globais para uma determinada localização. Se transparência por nome e por localização estiverem sendo utilizadas, pode ser muito difícil realizar um roteamento para determinar a localização de um determinado nome. Pode-se pensar em soluções com servidores centralizados ou distribuídos, porém os centralizados podem se tornar um gargalo, enquanto os distribuídos precisam usar alguma técnica de descentralização, como por exemplo cada servidor seria responsável por um determinado subconjunto de arquivos, ou cada um cuidaria de resolver a localização de determinados tipos de arquivos, etc.

Serviço que permite que usuários e processos adicionem, removam e consultem nomes é chamado de Serviço de Nomeação e é implementado por servidores de nomes. Costumam ser organizados em hierarquia. Segundo Cheriton e Mann (1989) é conveniente dividir os espaços de nomes em três camadas (Figura 17):

- Camada global
    - Raiz e seus filhos
    - Principal característica: Estabilidade
    - Podem representar organizações
- Camada Administrativa
    - Nós de diretórios
    - Gerenciados por uma única organização
    - Relativamente estáveis
- Camada Gerencial
    - Nós cujo comportamento típico é a mudança periódica
    - Mantidos por administradores de sistemas e usuários finais

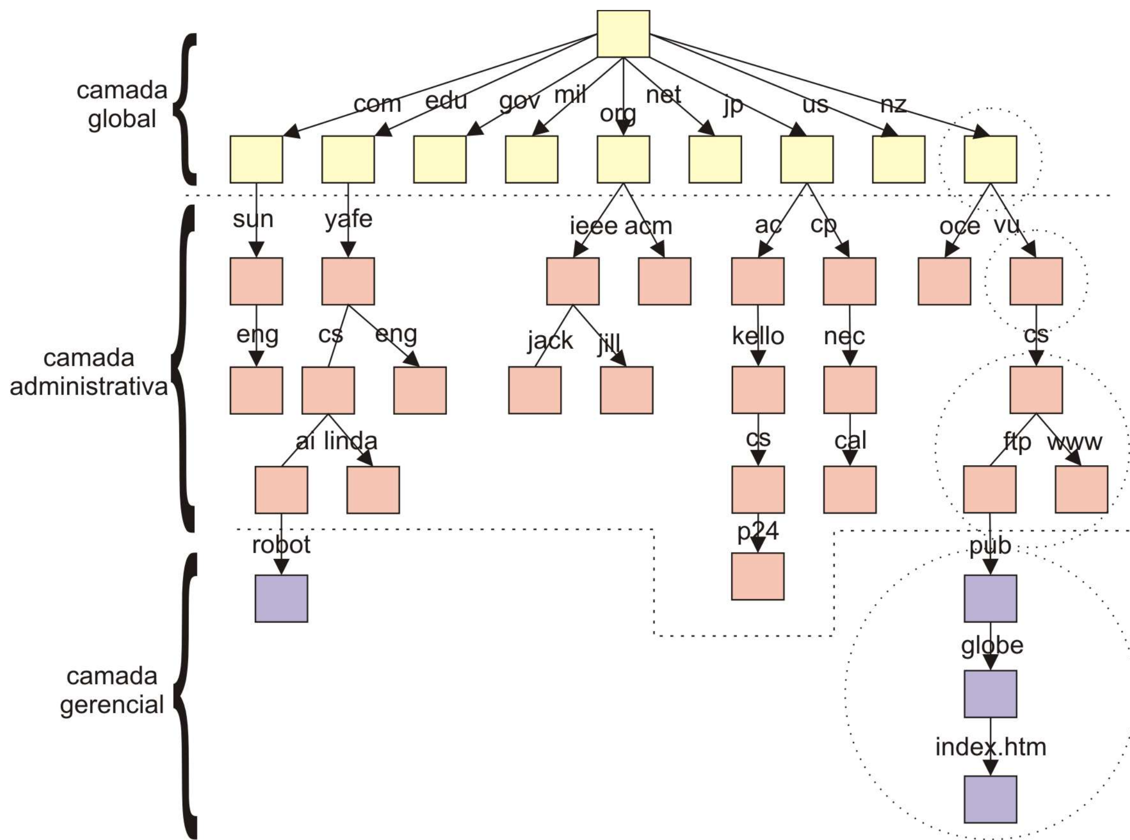


Figura 17: Exemplo de repartição de nomes DNS [Carvalho 2001]

Exercícios

As questões abaixo devem ser respondidas em forma dissertativa e argumentativa com pelo menos uma lauda. Devem também refletir a interpretação da leitura do texto juntamente com pesquisas sobre o tema arguido.

1. Explique a importância do modelo OSI para a comunicação de computadores.
2. O que é comunicação orientada a mensagem?
3. Explique como se dá a comunicação via middleware?
4. Como funciona a chamada remota a procedimento (RPC)?
5. Como se dá a passagem de parâmetros em um RPC?
6. O que é *Distributed Computing Environment* (DCE)?
7. Para que serve o sistema de nomeação em sistemas distribuídos?
8. O que é espaço de nomes?

Resumo

A Unidade II apresentou os principais tópicos de comunicação em sistemas distribuídos. Foi mostrado o modelo OSI, a base dos principais protocolos de comunicação de rede. Em seguida apresentado o procedimento de Chamada remota de Procedimento que permite que os SDs possam executar ou requisitar remotamente tarefas ou serviços de outras máquinas que compõem o sistema.

Foi visto ainda aspectos de nomeação de recursos (espaço de nomes e resolução de nomes), o que possibilita que os diversos dispositivos de um sistema distribuído sejam identificados e acessados.

Bibliografia

AMARAL, W. H. **Arquitetura Cliente/Servidor Orientada a Objeto**. Tese de Mestrado, IME, 1993.

CARVALHO, Roberto P. **Sistema de Arquivos Distribuídos**. Monografia. IME/USP, 2001

CERQUEIRA, A.Gomes. **Implementação de Módulos PAM e NSS para Autenticação Segura e Distribuída.** Monografia, Lavras, 2005

CHERITON, David R.; MANN, Timothy P. **Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance**. ACM Trans. Comput. Syst. 7(2): 147-183. 1989.

COULOURIS, G; et al. **Sistemas Distribuídos: Conceitos e Projeto**. Bookman, 2007

TANENBAUM & VAN STEEN. **Sistemas Distribuídos: Princípios e Paradigmas**. 2. ed., Prentice-Hall, 2008.

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems principles and paradigms**. 1. ed. Prentice Hall, 2002.

VIDAL, Paulo César Salgado. **Modelagem para Arquitetura Cliente/Servidor Orientada a Objeto**. Tese de Mestrado, IME, 1994.

Weblografia

GOULART, Ademir – Sistemas Distribuídos e Comunicação em Grupo
http://www.goulart.pro.br/sd/arquivos/APOSTILA_AdemirGoulart_SD_CGv10.PDF
Acessado em 10/12/2010

ZAMPIERI, Franciele; ET al. - Princípios de Programação Paralela Utilizando MPI e Análise de Algoritmo de Multiplicação de Matrizes Utilizando o Padrão MPI
http://www.uri.com.br/~adario/disciplinas/AC2/files/Artigo%20MPI.pdf
Acessado em: 14/09/2010

UNIDADE III

Sincronização e Consistência

Objetivos

1. Entender como funciona o processo de sincronização.
2. Entender o conceito de relógios físicos
3. Conhecer como funciona uma transação em sistemas distribuídos
4. Aprender com é feita a replicação de dados em sistemas distribuídos
5. Conhecer os modelos de consistência.

5. Sincronização

Problemas de sincronização em um sistema de uma única CPU são geralmente resolvidos usando métodos como semáforos e monitores. Já em sistemas distribuídos onde não haja sincronização entre as diversas máquinas que o integram, podem surgir muitos problemas. Processos podem ser executados em ordem inadequada, causando resultados inadequados, erros e travamentos. Em sistemas distribuídos os problemas de sincronização exigem soluções específicas. Algoritmos distribuídos têm, em geral, as seguintes propriedades:

1. As informações relevantes estão dispersas em várias máquinas.
2. Processos tomam decisões baseados apenas em informação local.
3. Um único ponto de falha no sistema (que possa derrubar o sistema) deve ser evitado.
4. Não existem relógios ou outras fontes precisas de tempo global.

Normalmente, em sistemas distribuídos não é possível ou desejável obter as informações necessárias à sincronização em um único local (centralizador). E mesmo que isso seja feito, diversos problemas ainda podem persistir. Vários fatores colaboram para dificultar a sincronização dos relógios em um sistema distribuído, como por exemplo:

- Impossibilidade de manter todos os relógios trabalhando na mesma frequência. Mesmo que os cristais sejam feitos do mesmo material físico, a própria estrutura do computador onde ele se encontra faz com que o seu comportamento possa ser diferente do relógio de outro computador na mesma rede.
- Dificuldade de definir o tempo de propagação de uma mensagem por uma rede. Uma mensagem ao ser enviar por um processo em um computador mantendo a hora do relógio local para outro processo em outro computador, o tempo gasto para o recebimento da mensagem poderá apresentar variações.
- Possibilidade de ocorrer falhas nos processadores ou no meio de comunicação.

Todos esses fatores possuem mecanismos diferentes de tratamento. O dispositivo físico que permite medir o tempo com maior precisão é o relógio atômico com base no Césio 133 (Cs133). A frequência desse dispositivo possui uma precisão de aproximadamente 10-13. Essa precisão é grande se comparada com os cristais de quartzo com precisão 10-6, dando uma diferença de 1.000.000 de segundos ou 11,6 dias.

Portanto, cada máquina na rede tem seu próprio valor para a hora local, a qual tende ser diferente das outras máquinas. Já foi provado que os métodos de medição do tempo baseados no zênit solar permitem e acumulam desvios. Essa situação foi em parte contornada com a adoção de métodos mais precisos, baseados na regularidade do número de transições que o átomo de Césio 133 realiza em uma unidade de tempo (relógios atômicos). Inclusive existem, em diversos países, serviços de acesso público que informam a hora precisa, a partir de relógios atômicos (esses relógios, dos quais existem poucas dezenas no mundo, também devem estar sincronizados entre si).

Um problema adicional é que mesmo os relógios atômicos têm que ser ajustados de tempos em tempos, para corrigir distorções mínimas que acumulam.

5.1. Relógio físico

Em alguns sistemas (tempo-real por exemplo) um *clock* real é importante. Nestes sistemas necessitamos de *clocks* físicos externos. Por motivos de eficiência e de redundância, é desejável que se tenha mais de um *clock* externo, o que causa os seguintes problemas: como sincronizá-los com o *clock* de tempo real e como sincronizar estes *clocks* entre si.

5.1.1. Tempo Universal Coordenado (UTC)

O Tempo Universal Coordenado (em inglês: *Coordinated Universal Time*), ou UTC (acrônimo de *Universal Time Coordinated*), também conhecido como tempo civil, é o fuso horário de referência a partir do qual se calculam todas as outras zonas horárias do mundo. É o sucessor do Tempo Médio de Greenwich (*Greenwich Mean Time*), cuja sigla é GMT. A nova denominação foi cunhada para eliminar a inclusão de uma localização específica num padrão internacional, assim como para basear a medida do tempo nos padrões atômicos, mais do que nos celestes.

Ao contrário do GMT, o UTC não se define pelo sol ou as estrelas, mas é sim uma medida derivada do Tempo Atômico Internacional (TAI). Devido ao fato do tempo de rotação da Terra oscilar em relação ao tempo atômico, o UTC sincroniza-se com o dia e a noite de UT1, ao que se soma ou subtrai segundos de salto (*leap seconds*) quando necessário. Os segundos de salto são definidos, por acordos internacionais, para o final de julho ou de dezembro como primeira opção e para os finais de março ou setembro como segunda opção. Até hoje somente julho e dezembro foram escolhidos como meses para ocorrer um segundo de salto. A entrada em circulação dos segundos de salto é determinada pelo Serviço Internacional de Sistemas de Referência e Rotação da Terra (IERS), com base nas suas medições da rotação da terra.

No uso informal, quando frações de segundo não são importantes, o GMT pode ser considerado equivalente ao UTC. Em contextos mais técnicos é geralmente evitado o uso de "GMT". UTC é uma variante do tempo universal (universal time, UT) e o seu modificador C (para coordenado) foi incluído para enfatizar que é uma variante de UT. Pode-se considerar como uma solução conciliatória entre a abreviatura inglesa CUT e a francesa TUC.

Os tempos UTC de alta precisão só podem ser determinados uma vez, sendo conhecido o tempo atômico, que se estabelece mediante a reconciliação das diferenças observadas entre um conjunto de relógios atômicos mantidos por um determinado número de oficinas do tempo nacionais. Isto é feito sob coordenação do Escritório Internacional de Pesos e Medidas (*Bureau International des Poids et Mesures* - BIPM). Não obstante, os relógios atômicos são tão exatos que só os mais precisos computadores necessitam usar estas correções; e a maioria dos utilizadores de serviços de tempo utilizam os relógios atômicos que tenham sido previamente configurados como UTC, para estimar a hora UTC.

5.1.2. Tempo Atômico Internacional (TAI)

O Tempo Atômico Internacional (TAI), é a escala de tempo calculada pelo Escritório Internacional de Pesos e Medidas (BIPM), na França, usando informações de cerca de duzentos relógios atômicos em mais de 50 laboratórios nacionais ao redor do mundo.

O Tempo Atómico Internacional (TAI) é o valor de referência internacional para medir o passar do tempo e foi criado em 1967 baseado na média de vários relógios atómicos distribuídos por várias instituições mundiais

Em longo prazo, a estabilidade do TAI é assegurada por um balanceamento cuidadoso dos relógios participantes. A unidade de medida do TAI é mantida o mais próximo possível do segundo do Sistema Internacional de Unidades, usando informações dos laboratórios que mantém os melhores padrões primários de césio (1 segundo = 9.192.631.770 oscilações do átomo de Césio 133).

O TAI é uma escala uniforme e estável que, por consequência, não se mantém em sincronia com a rotação ligeiramente irregular da Terra. Por razões práticas e de uso público é necessário que se tenha uma escala que mantenha tal sincronia. Esta escala é o Tempo Universal Coordenado (UTC), que é idêntico ao TAI, exceto que de tempos em tempos um segundo de salto é definido para garantir que, no decorrer de um ano, o Sol cruze o meridiano de Greenwich ao meio-dia com um desvio máximo de 0,9s. As datas para efetivação dos segundos de salto são definidas pelo Serviço Internacional de Sistemas de Referência e Rotação da Terra (IERS).

5.2. Algoritmos de sincronização

A maioria dos algoritmos de sincronismo de relógio tendem a ser implementados em forma de cliente servidor. Esta solução é composta por um cliente que requisita a hora local, a qual é implementada em forma de uma mensagem *send*. E um servidor que atende às requisições de hora atual com a primitiva *receive* [Pereira Júnior 2007]. Quando várias requisições chegam ao servidor a solução é armazenar estas requisições em um buffer e atendê-las de acordo a ordem.

5.2.1. Algoritmo de Cristian

Algoritmo de Cristian ou Algoritmo de Christian é um algoritmo que pressupõe que uma das máquinas do sistema distribuído acessa um serviço de informações da hora atômica e ajustando-se, passa a ser um “servidor de tempo” para as demais máquinas do sistema, que periodicamente a consultam para ajustar seus relógios.

Uma máquina pergunta a um servidor com a hora de referência e ajusta o seu relógio. Considera que o tempo de viagem das mensagens são iguais para corrigir a diferença entre a saída e chegada da mensagem. Provoca erro se o tempo de viagem for diferente [Fernandez 2005].

O algoritmo de Cristian trabalha utilizando sistemas que possuem uma máquina com receptor WWV. A máquina com o receptor WWV será denominada Time Server. Periodicamente, cada máquina envia uma mensagem para o *Time Server* pedindo pelo valor da hora corrente. O *Time Server* responde, o mais rápido que puder, com uma mensagem contendo o seu tempo corrente, *Cutc*.

Como uma primeira aproximação, quando o emissor recebe a mensagem *reply*, ele pode atualizar o seu *clock* para *Cutc*. Este algoritmo possui dois problemas: um maior e outro menor. O problema maior é que o tempo UTC nunca deve estar menor do que a hora local da

máquina emissora. Isso pode ocorrer se o *clock* do emissor for mais rápido. O problema menor diz respeito ao tempo que a mensagem *reply* leva para chegar. Uma forma de resolver esse problema é calculando a média entre T0 e T1 como pode ser observado na Figura 18.

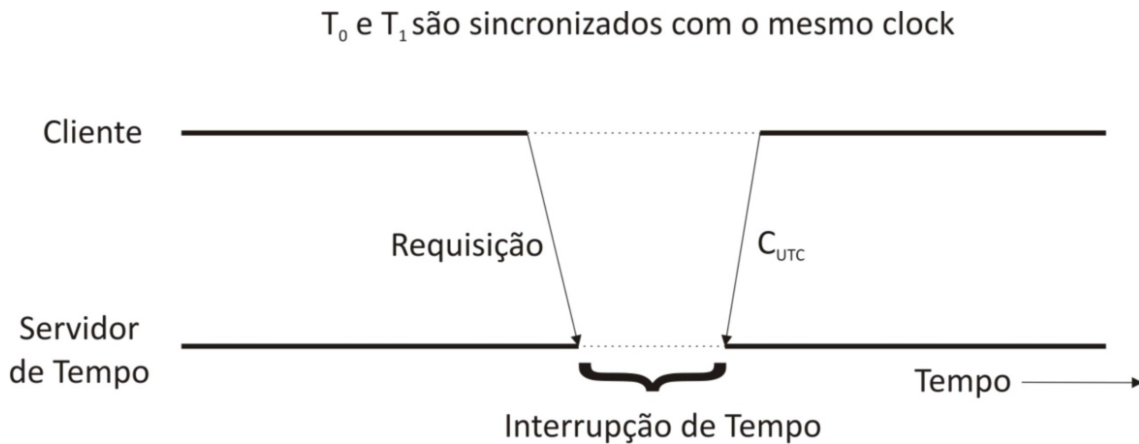


Figura 18: Solução de Cristian [Fernandez 2005]

Para se aferir o tempo atual é necessário calcular o tempo de propagação que pode ser obtido da média da diferença de T1 e T0 e adicionado ao tempo UTC retornado pelo servidor. Pode-se considerar o tempo de manipulação da interrupção para no intuito de minimizar o erro, para tal deve ser feito a média da diferença de T1, T0 e I (tempo da manipulação da interrupção).

Entretanto, há dois problemas. O principal é a distorção relativa de cada máquina em relação ao “servidor de tempo”, já que os mecanismos de avanço de tempo, tanto de uma como de outra máquina não são absolutamente precisos.

Outro problema é que a comunicação entre essas máquinas também exige certo tempo, decorrente de fatores das próprias máquinas e dos meios que as interligam. Assim, o algoritmo propõe a adoção de certos ajustes, que levam em consideração a diferença de tempo verificada entre as máquinas, depois de decorrido certo período do último ajuste entre elas.

No processo periódico de sincronização também é utilizada a média ajustada (devido aos reflexos de possíveis momentos de congestionamento da rede) dos tempos necessários à comunicação entre as máquinas.

5.2.2. Algoritmo de Berkeley

Nesse algoritmo, o “servidor de tempo” é ativo e consulta periodicamente cada uma das máquinas sobre os valores de seus relógios. Então calcula-se uma média das leituras realizadas e informa cada máquina para que se ajuste, adiantando ou atrasando seu relógio. Essa média pode ser simples ou ajustada, desprezando-se valores extremos, o que permite contornar eventuais falhas em alguns relógios. Também é possível considerar o tempo de comunicação entre as máquinas. Nesse algoritmo, não há necessidade de que o “servidor de tempo” consulte um serviço de hora atômica.

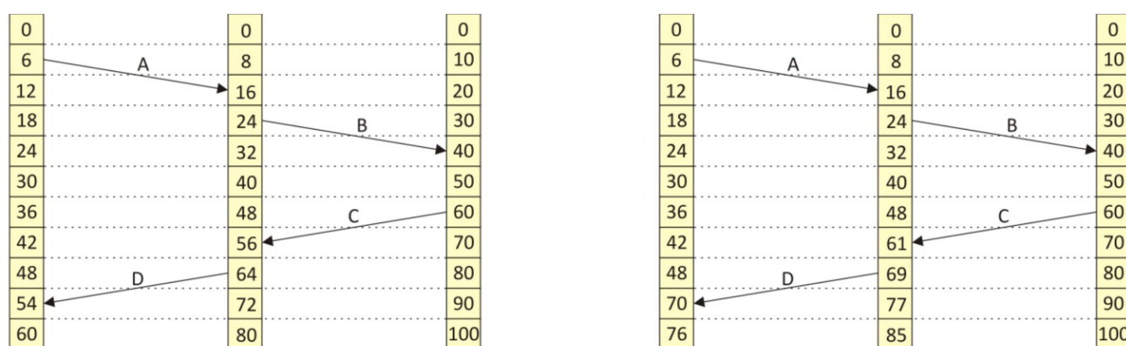
Ambos os métodos citados acima dependem de alguma centralização ("servidor de tempo"). Só muito recentemente passou a existir o hardware e software necessário para permitir sincronização em larga escala (por exemplo toda a Internet). E novos algoritmos que utilizam relógios sincronizados estão começando a aparecer, como os exemplos discutidos por Liskov (1993) ("*At-Most-Once-Message Delivery*" e "*Clock-Based Cache Consistency*"). Esses algoritmos são descentralizados, onde cada máquina se auto-ajusta em função das mensagens ("*broadcast*") que recebe de outras máquinas. Trabalham diminuindo o tempo para um intervalo de tamanho fixo de resincronização. Começa T0 + I e avança até T0 + (I+1)R (T é o momento e R parâmetro do sistema). Começando cada intervalo, cada máquina envia seu horário. Essas mensagens não acontecerão ao mesmo tempo pois cada máquina tem sua velocidade. Cada máquina coleta as mensagens das outras e calcula um novo horário descartando o maior e o menor.

Outra variação é tentar adicionar uma estimativa de propagação de acordo com a topologia de Rede. Sistemas que sincronizam com UTC requerem receptores WWV, GEOS e outros. Existem flutuações de sinal, e o sistema operacional pode estabelecer um intervalo de tempo cujo UTC falha. Para alcançar um sincronismo, nenhum processador obterá o tempo instantaneamente. Há o tempo de demora entre a transmissão e a recepção, distância, gateway, colisões e outros.

5.2.3. Estampa de tempo de Lamport

Sincronização de relógio não necessita ser absoluto, se 2 processos não interagem, não existe a necessidade que seus relógios estejam sincronizados. Para muitos propósitos é suficiente que todas as máquinas concordem com o mesmo tempo. Não é essencial que o tempo coincida com o tempo real, por isto podemos definir este relógio como relógio lógico e relógio físicos como relógios que acompanham o tempo real.

Lamport apresentou o seguinte algoritmo para sincronizar os relógios lógicos. A Figura 19 mostra o exemplo de três processos que enviam mensagens entre si, cada um com seu próprio relógio.



a. Três processos com seu relógio individual

b. Correção segundo algoritmo de Lamport

Figura 19: Sincronismo de Relógio de Lamport [Pereira Júnior 2007]

A Figura 19 (a), as mensagens 'C' e 'D' enviam para outros processos que tem relógio com tempos inferiores ao processo de origem. Esta situação deve ser evitada. A Figura 19 (b), apresenta como deveria ser o resultado com os devidos incrementos após o recebimento da mensagem.

O algoritmo de Lamport sincroniza estes relógios resolvendo o problema através da alteração do relógio destino com o valor do relógio origem mais tempo gasto para enviar a mensagem. Os relógios lógicos foram sincronizados pelo algoritmo do Lamport através da alteração dos relógios destinos com o valor dos processos origem mais tempo para envio da mensagem.

Em 1978 Lamport desenvolveu um algoritmo baseado nas observações:

- Se dois processos não interagem não é necessário sincronizar seus relógios já que a ausência de sincronismo não será observada
- O importante não é que os processos estejam de acordo com uma hora global e sim na ordem de ocorrência dos eventos

Em outras palavras, o objetivo da estampa de tempo de Lamport é a sincronização de *clocks* lógicos onde os tempos associados aos eventos não são necessariamente próximos ao tempo real e os processos não precisam estar de acordo sobre o valor exato do tempo, mas sobre a ordem em que os eventos ocorrem. Eles se baseiam na relação acontecimento-anterioridade:

- Se a e b são eventos dentro do mesmo processo e se o evento “a” acontece antes do “b” então: $C(a) < C(b)$
- Se a é o envio de uma mensagem para um processo e se b é a recepção desta mensagem por outro processo, então devem ser atribuídos valores a C(a) e C(b) de maneira que $C(a) < C(b)$.

Os processos executam em máquinas diferentes, cada uma com seu clock e cada mensagem leva o valor do clock do transmissor. Se a mensagem traz um tempo superior ao do receptor, este adianta seu clock em uma unidade maior que o tempo recebido.

A regra 1 pode ser facilmente implementada já que ocorre dentro do mesmo processo. A regra 2 pode ser implementada se o processo emissor enviar na mensagem seu tempo lógico e o processo receptor atualizar seu relógio lógico tal que $Cj (b) = max(TSa + d,Cj (b))$, onde TSa é o tempo do evento de envio e de um número positivo.

5.3. Posicionamento Global de Nós

É difícil que um nó monitore outros em um sistema distribuído quando há um crescimento do número de nós. Para solucionar esse problema utiliza-se o posicionamento global dos nós, que informa onde cada nó está. Em redes de sobreposição geométrica é designada uma posição em um espaço geométrico n-dimensional a cada nó, de modo que a distância entre cada nó represente uma métrica de desempenho no mundo real.

O posicionamento Global de nós é útil para redirecionar para réplica de um servidor com menor tempo de resposta para o cliente e auxilia identificar melhor posicionamento para réplicas. É um roteamento baseado em posição.

GPS: é um sistema de navegação por satélite que fornece a um aparelho receptor móvel a posição do mesmo, assim como informação horária, sob todas quaisquer condições atmosféricas, a qualquer momento e em qualquer lugar na Terra, desde que o receptor se encontre no campo de visão de quatro satélites GPS.

O posicionamento global de nós requer m+1 medições de distância. Como no GPS, o cálculo pode ser feito por:

$$\sqrt{(x_i - x_p)^2 + (y_i - y_p)^2} \quad (i = 1,2,3)$$

Na Figura 20 Di equivale a metade do RTT (Round-Trip Time), sendo diferente ao longo do tempo.

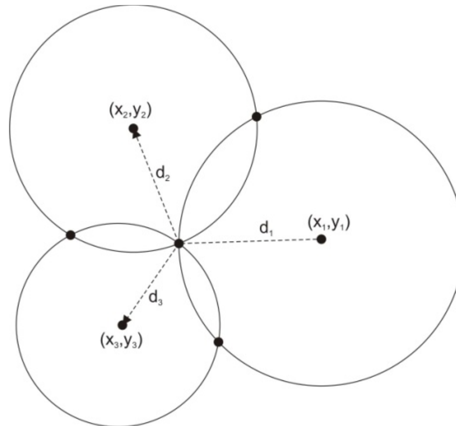


Figura 20: Cálculo da Posição de um nó em um espaço bidimensional [Tanenbaum 2008]

5.4. Algoritmo de Eleição

Muitos algoritmos distribuídos requerem um processo como coordenador. Em geral não importa qual seja o processo coordenador, mas um deles inequivocamente tem que exercer esta função.

Em um algoritmo de eleição cada processo ativo tem um número associado e o coordenador sempre residirá no processo com número de identificação mais alto. O objetivo do algoritmo é eleger um processo como novo coordenador. Quando um coordenador falha, o algoritmo precisa eleger outro processo ativo que tem o numero de identificação mais alto. Esse número precisa ser enviado para cada processo ativo do sistema. Além disso, o algoritmo precisa proporcionar um mecanismo por meio do qual um processo recuperado possa identificar o coordenador corrente.

5.4.1. Algoritmo Valentão (Bully)

Quando um processo nota a falta do coordenador atual, envia uma solicitação de eleição para todos os processo de maior prioridade que a sua. Após algum tempo se ninguém responder, significa que todos os processos que teriam direito a vencer a eleição estão mortos e portanto, o processo que iniciou a eleição envia uma mensagem a cada processo de prioridade menor que a sua indicando a vitória. Caso algum processo de prioridade maior responder ao que iniciou a eleição, então este se retira e o de maior prioridade assume o pleito mandando novamente mensagens aos processos de prioridade maior, repetindo novamente todo o esquema. Isto eventualmente se repete até que seja descoberto o processo de maior prioridade ainda ativo, que então se tornará o novo coordenador.

Round-Trip Time: estimativa do tempo total de transmissão de ida e volta.

A qualquer momento que o processo de maior prioridade retornar a atividade, enviará uma mensagem a todos os outros reabilitando-se como coordenador, uma vez que sabe que é o processo com a melhor característica para vencer as eleições dentro do sistema.

5.5. Exclusão Mútua

É uma técnica usada em programação concorrente para evitar que dois processos ou *threads* tenham acesso simultaneamente a um recurso compartilhado, acesso esse denominado por seção crítica.

Um meio simples para exclusão mútua é a utilização de um semáforo binário, isto é, que só pode assumir dois valores distintos, 0 e 1. O travamento por semáforo deve ser feito antes de utilizar o recurso, e após o uso o recurso deve ser liberado. Enquanto o recurso estiver em uso, qualquer outro processo que o utilize deve esperar a liberação.

Porém, essa técnica pode causar vários efeitos colaterais, como *deadlocks*, em que dois processos obtêm o mesmo semáforo e ficam esperando indefinidamente um outro processo liberar o semáforo; e inanição, que é quando o processo nunca dispõe de recursos suficientes para executar plenamente.

5.5.1. Algoritmo Centralizado

A maneira mais simples de se conseguir exclusão mútua em um sistema distribuído é imitar o que é feito no sistema centralizado (com um único processador na Figura 21):

1. Um processo é eleito como Coordenador. Quando um processo quer entrar na região crítica, ele envia uma mensagem requisitando ao Coordenador permissão para isso.
2. Se nenhum outro processo está na região crítica, o coordenador envia uma resposta dando permissão. Ao receber a mensagem o processo requisitante entra na região crítica.

Supondo que outro processo peça permissão para entrar na região crítica, e o Coordenador sabendo que outro processo está na região, não envia resposta bloqueando este processo até que ele possa entrar na região [Tanenbaum 2008]. Uma outra opção é enviar uma mensagem negando a solicitação. Por fim, quando o processo deixa a região, ele envia para o Coordenador uma mensagem liberando a região crítica.

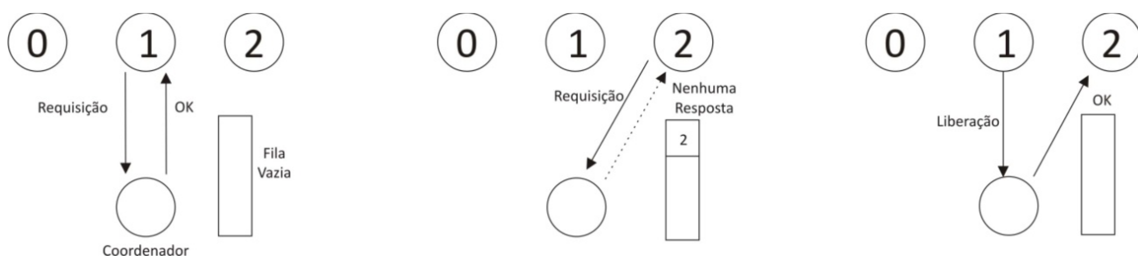


Figura 21: Exclusão Mútual (algoritmo centralizado) [Tanenbaum 2008]

5.5.2. *Algoritmo Distribuído*

Algoritmo centralizado tem o problema de uma falha no Coordenador inviabilizar o mecanismo. Como alternativa usa-se o Algoritmo distribuído (Figura 22). Quando um processo quer entrar na região crítica, ele constrói uma mensagem contendo o nome da região, número do processo e o tempo corrente.

Ele envia a mensagem para todos os outros processos. Quando um processo recebe uma mensagem de requisição de outro processo sua ação vai depender de sua situação relativa a região crítica:

1. Se o receptor não está na região crítica e não quer entrar, ele envia de volta uma mensagem OK;
2. Se o receptor já está na região, ele não responde e coloca a requisição na fila;
3. Se o receptor quer entrar na região crítica mas ainda não o fez, ele compara o tempo da mensagem que chegou com o tempo da mensagem que ele enviou para os outros processos. O menor tempo vence. Se a mensagem que chegou é menor ele envia de volta uma mensagem com OK. Se a sua mensagem tem o menor tempo ele coloca na fila a requisição que chegou e não envia nada.

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

5.5.3. *Algoritmo Token Ring*

É construído um anel lógico por software no qual a cada processo é atribuído uma posição no anel (Figura 23). Quando o anel é inicializado, o processo 0 ganha o "*token*". O "*token*" circula no anel (passa do processo k para o k+1). Quando o processo ganha o "*token*" ele verifica se ele quer entrar na região crítica. Caso positivo, ele entra na região, realiza o seu trabalho e ao deixar a região passa o "*token*" para o elemento seguinte do anel. Não é permitido entrar em uma segunda região crítica com o mesmo "*token*".

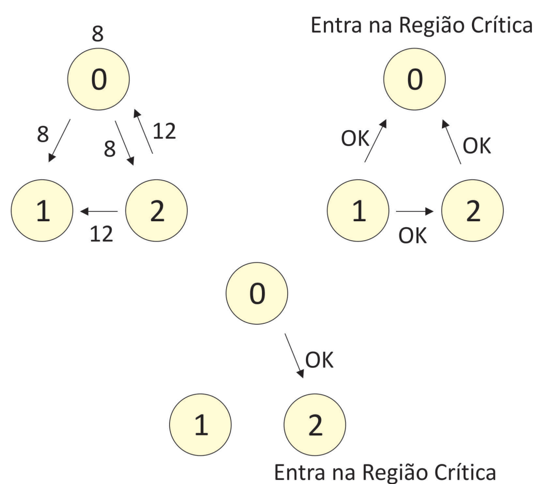


Figura 22: Exclusão Mútua (Algortimo Distribuído) [Tanenbaum 2008]

Se o processo não quer entrar na região crítica ele simplesmente passa o “*token*”. Como consequência quando nenhum processo quer entrar na região crítica o “*token*” fica circulando pelo anel.

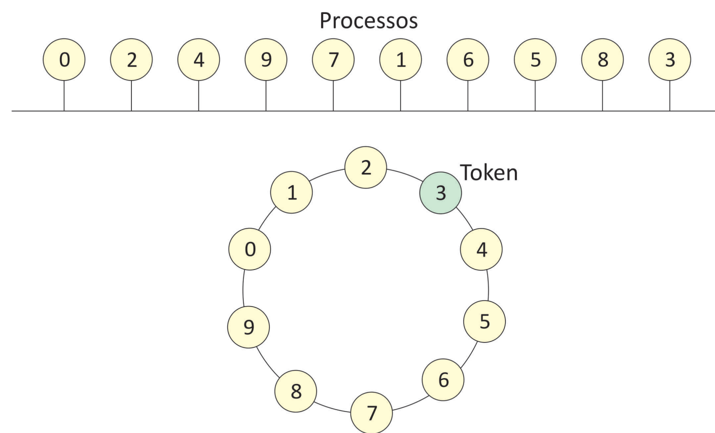


Figura 23: Algortimo *Token Ring* [Tanenbaum 2008]

Problemas:

1. Se o ‘*token*” é perdido ele precisa ser regenerado. A detecção de um “*token*” perdido é dificil.
2. Se um processo falha também ocorrem problemas. A solução é fazer o processo que recebe o “*token*” confirmar o recebimento. O processo que falhou pode ser retirado do anel, e o “*token*” enviado para o processo seguinte. Essa solução requer que todos os processos conheçam a configuração do anel.

5.6. Transação

5.6.1. Primitivas de transações

São termos (abstração) em alto nível para ocultar uso de semáforos para controle de concorrência, prevenir *deadlocks* e auxiliar na recuperação de falhas.

Para melhor compreensão, vamos nos concentrar em aplicações de banco de dados. Na prática, operações em um banco de dados costumam ser realizada sob a forma de transações. Programar a utilização de transações requer primitivas especiais que devem ser fornecidas pelo sistema distribuído subjacente ou pelo sistema de linguagem em tempo de execução.

Exemplos típicos de primitivas de transação são mostrados na tabela abaixo:

Tabela 2: Exemplos de primitivas de transações [Tanenbaum 2008]

Primitiva	Descrição
BEGIN_TRANSACTION	Marca o início de uma transação
END_TRANSACTION	Termina a transação
ABORT_TRANSACTION	Elimina a transação e restaura os valores antigos
READ	Leia dados de um arquivo, tabela ou de outra fonte
WRITE	Escreve dados em um arquivo, tabela ou de outra fonte
COMMIT	Valida a transação dando como válidas todas as escritas

BEGIN_TRANSACTION e END_TRANSACTION são usadas para delimitar o escopo de uma transação. As operações entre elas formam o corpo da transação. O aspecto característico de uma transação é que todas essas operações são executadas ou nenhuma é executada. Elas podem ser procedimentos de biblioteca, chamadas de sistema ou declarações entre parênteses em uma linguagem, dependendo da implementação.

Essa propriedade tudo-ou-nada das transações é uma das quatro propriedades características que elas têm [Bendini 2010]. Mais especificamente, transações são:

1. Atômicas: para o mundo exterior, a transação acontece como se fosse indivisível.
2. Consistentes: a transação não viola invariantes de sistema.
3. Isoladas: transações concorrentes não interferem umas das outras.
4. Duráveis: uma vez comprometida uma transação, as alterações são permanentes

Essas propriedades costumam ser citadas por suas letras iniciais: ACID. A primeira propriedade fundamental exibida por todas as transações é que elas são atômicas. Essa propriedade garante que cada transação aconteça completamente ou não aconteça. Se acontecer, será como uma única ação indivisível e instantânea. Enquanto uma transação está em progresso, outros processos, estejam ou não envolvidos nas transações, não podem ver nenhum dos estados intermediários.

A segunda propriedade afirma que elas são consistentes. Isso quer dizer que se o sistema tiver certas variantes que devem valer para sempre, se eles forem válidos antes da transação, também o serão após a transação. Por exemplo, em um sistema bancário, uma invariante fundamental é a lei da conservação do dinheiro. Após toda transferência interna, a quantidade de dinheiro no banco tem de ser a mesma que era antes da transferência; contudo, por um breve instante durante a transação, esse invariante pode ser violado. Todavia, a violação não é visível fora da transação.

A terceira propriedade diz que as transações são isoladas e serializáveis. Isso significa que se as duas ou mais transações são executadas ao mesmo tempo, o resultado final para cada uma delas e para outros processos se apresentará como se todas as transações fossem executadas em sequência em certa ordem, dependente do sistema.

A quarta propriedade diz que as transações são duráveis. Refere-se ao fato de que não importa o que aconteça, uma vez comprometida uma transação, ela continua e os resultados tornam-se permanentes. Nenhuma falha após o comprometimento pode desfazer os resultados ou provocar sua perda.

Até aqui, transações foram definidas em um único banco de dados. Uma transação aninhada é construída com base em uma quantidade de subtransações, como mostra a Figura 24. A transação do nível mais alto pode se ramificar e gerar filhos que executam em paralelo uns aos outros em máquinas diferentes para obter ganho de desempenho ou simplificar a programação [Bendini 2010]. Cada um desses filhos também pode executar uma ou mais subtransações ou se ramificar e gerar seus próprios filhos.

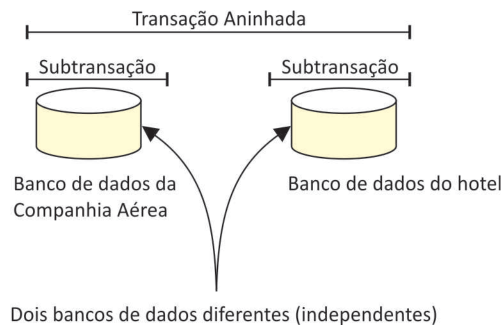


Figura 24: Transação Aninhada (Tanembaum 2008)

Exemplo de uma transação: um cliente, em um PC ligado por modem, faz transferência de fundos de uma conta bancária para outra, em dois passos:

(1) Saque(quantia, conta1)
(2) Deposite(quantia, conta2)

Se a ligação telefônica cair entre os passos (1) e (2) o dinheiro desaparece. A solução é que os passos (1) e (2) devem ocorrer como uma transação atômica (como se fosse um único passo). Se a ligação telefônica cair entre os passos (1) e (2), os efeitos do passo (1) devem ser cancelados.

5.6.2. Implementação de transação

Quando várias transações estão sendo executadas simultaneamente, é necessário criar um mecanismo de controle para que uma transação não atrapalhe a outra. Serão verificados três tipos de controle: *Locking*, controle otimista e *timestamps.*

Locking

Em sua forma mais simples, provê exclusão mútua de arquivos. Para executar operações de leitura e escrita, a transação primeiro faz um "*lock*" dos arquivos envolvidos. Depois de conseguir, executa as operações indicadas e em seguida libera todos os arquivos.

Uma forma de implementar este mecanismo é com um sistema centralizado para dar acesso (ou não) às transações, porém é muito restritivo. Alternativas incluem a separação de operações de *read* (que não precisam ser bloqueadas e de *write* (que precisam). Algoritmo para exclusão "*readers and writers*".

Também é possível diminuir a granularidade e não bloquear o arquivo inteiro, mas sim os registros, campos, etc. Isto permite maior vazão de transações, porém com custo de maior complexidade de implementação e possibilidades de *deadlocks*.

Exemplo: *two-phase locking* (bloqueio em duas fases):

1. Cada transação bloqueia todos os arquivos que precisa.
2. Se conseguir todos, segue a transação.
3. Se não conseguir, libera todos os *locks* obtidos até o momento e volta ao passo 1.

Soluções mais eficientes podem levar a *deadlocks*. Quando tais soluções são adotadas, é necessário implementar algoritmos para prevenção e detecção de *deadlocks*. Como forma de prevenção todas as transações bloqueiam os recursos desejados em uma ordem pré-determinada.

Na detecção todas as requisições são modeladas como um grafo. Se o grafo tiver ciclos, há um deadlock. Se a transação estiver parada há mais de T segundos, assume que há um deadlock.

Controle Otimista

Assume que a maioria das transações são "bem comportadas", e que conflitos são raros. Ao chegar ao final da transação, antes de executar o "*commit*", verifica se há alguma outra transação que usou os mesmos registros.

Se usou, cancela tudo e recomeça. Se não usou, executa a transação. Apropriado para sistemas que usam "*private workspaces*" e são livres de *deadlock*, permitindo paralelismo em larga escala (pois não há bloqueios). A desvantagem ocorre quando existe um erro. Neste caso a trasação é reiniciada. (o que é desagradável em aplicações de alto uso computacionalO.

Timestamps

Timestamp é uma sequência de caracteres, que denota a hora e data (ou alguma delas) na qual ocorreu determinado evento. Assinala um *timestamp* a cada transação no "*begin_trans*". Cada arquivo (registro, campo) é associado a um *timestamp* de leitura e de escrita, que permitem dizer o "tempo lógico" em que foram lidos ou escritos pela última vez.

As operações *read*/*write* da transação seguem normalmente se o *timestamp* da transação for mais "nova" que os arquivos. Se não for, a trasação é cancelada. Também é otimista, e é adequado ao modelo de "*private workspaces*".

Implementação

Existem várias formas de se implementar uma transação como atômica, dentre as quais se destacam a *private workspaces, writeahead logs e two-phase commit protocol.*

No *private workspaces,* ao executar uma transação, cada processo ganha um espaço de trabalho isolado dos demais. Este espaço de trabalho é composto por todos os arquivos e registros que ele utiliza, e são criadas cópias próprias para o caso de escrita. Ao final da transação, as cópias de escritas substituem as originais. Já os *writeahead logs* são registros contendo a ordem com que as operações foram efetuadas e os valores contidos nos registros antes e depois delas serem efetuadas.

Se a transação for bem sucedida, os dados já estão escritos. Se não for bem sucedida (por exemplo com a queda do computador), o log será "desfeito" até todos os valores retornarem ao estado inicial.

O *two-phase commit protocol* sera visto com mais detalhes na subsessão seguinte.

5.6.3. Controle de concorrência

Para garantir que a transação possa sempre realizar *commit* sem violar a propriedade de isolamento deve ser implementado um algoritmo de controle de concorrência, normalmente, implementado como um algoritmo de duas fases. O algoritmo de bloqueio de duas fases (*two-phase locking*) é realizado pela aquisição de *lock* sobre dados de modo que operações de diferentes transações ativas possam se intercalar.

Garante que a ordem das operações no momento da confirmação da transação (commit) é uma ordem serial (*serialization order = commit order*) [Galante 2010]. Assim, um algoritmo *two-phase locking* possui as seguintes as fases de obtenção de *locks* e liberação de *locks*.

Uma vez que uma transação tiver liberado um *lock* qualquer, a transação não poderá adquirir nenhum novo *lock* pois a primeira liberação indica o término da primeira fase. Na prática, a maior parte dos sistemas utiliza um algoritmo *strict two-phase locking* que garante que os *locks* adquiridos são mantidos até que a transação seja cancelada ou confirmada. Isso equivale a dizer que durante toda a execução de uma transação é possível adquirir novos *locks*, ou então, pode-se dizer que a segunda fase apenas se inicia após a execução da última operação da transação. O uso dessa forma mais restritiva de *two-phase locking* deve principalmente a dois motivos:

- *locks* são dependentes dos dados, podendo ser impossível de saber quando exatamente que a primeira fase terminou;
- se um *lock* for liberado antes do final da transação, dados modificados podem ser vistos por outras transações. Se essa transação precisar ser abortada (*abort*), as demais transações que "viram" os resultados intermediários deverão também ser canceladas, gerando uma sucessão de cancelamentos (*cascading aborts*). O principal problema desse tipo de controle de concorrência é a possibilidade de *deadlocks*.

6. Consistência e replicação

Para garantir desempenho e disponibilidade, replicação é a técnica utilizada em sistemas distribuídos. É o uso de múltiplas cópias de dados ou serviços (e estado associado). É essencialmente utilizada por razões de 3 ordens:

1. disponibilidade (*availability*): se um serviço não estiver acessível, pode-se tentar aceder a uma sua réplica;
2. fiabilidade (*reliability*): mantendo réplicas em locais distintos, a capacidade de sobrevivência dos dados a acidentes sérios, p.ex. tremores de terra, é maior;
3. desempenho (performance): a "carga" pode ser distribuída pelas diferentes réplicas, possivelmente mais próximas dos clientes.

Contudo um problema inerente a replicação como garantir a consistência entre as réplicas. Para isso é utilizados os *caches* (Figura 25). A *cache* do lado do cliente pode ocorrer em dois lugares. Um deles é no browser, quando falamos em sistemas na Web. Neste caso sempre que um documento for buscado, ele é armazenado na *cache* do browser, de onde será carregado da próxima vez. Os clientes podem configurar a *cache* para verificação de consistência. Outro lugar que pode abrigar o *cache* no lado do cliente é no *Proxy Web* que aceita requisições e repassa para servidores neste caso a *cache* compartilhada.

Proxy é um servidor que atende a requisições repassando os dados do cliente a frente. Um usuário (cliente) conecta-se a um servidor proxy, requisitando algum serviço, como um arquivo, conexão, website, ou outro recurso disponível em outro servidor.

Uma maneira de organização de *cache* é chamada de Cache Cooperativa ou distribuída. Neste caso, sempre que ocorrer uma ausência da cache em um *Proxy Web*, é feita uma verificação em alguns proxies vizinhos para ver se um deles contém o recurso requisitado. Geralmente este tipo de organização ocorre primordialmente com caches Web que pertencem à mesma organização e que estão na mesma LAN.

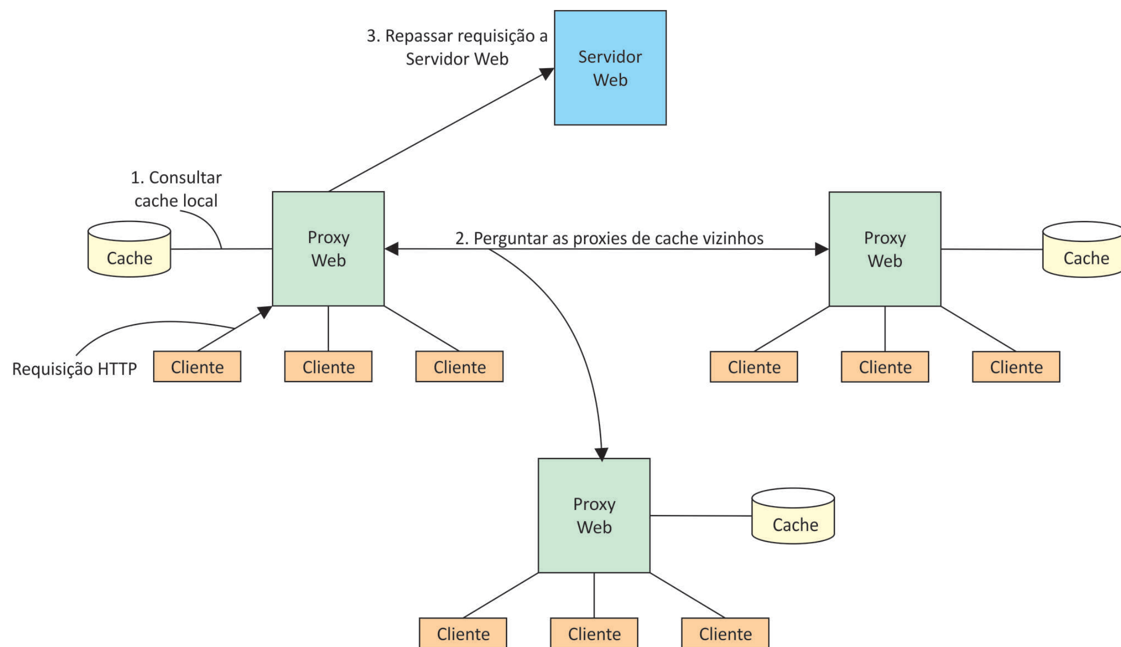


Figura 25: Princípio de cache cooperativa [Tanembaum 2008]

Outra maneira de organização de *cache* é através da Cache Hierárquica. Nesse método, *caches* são colocadas em uma região ou até mesmo em um país. Assim, *caches* são pesquisadas de uma maneira hierárquica, o que pode aumentar a latência de resposta. No entanto, é alta a probabilidade de encontrar uma cópia de documentos ou recursos populares em uma cache mais próxima.

6.1. Replicação otimista x pessimista

A principal diferença entre a replicação otimista e a pessimista está no controle de concorrência. A abordagem pessimista faz o controle de concorrência enquanto a otimista não faz. Os algoritmos pessimistas, utilizando quaisquer protocolos para controle de concorrência, coordenam de forma síncrona as atualizações das bases de dados replicadas e bloqueiam os usuários durante o processo de atualização [Oliveira 2007]. Dessa forma matem o sincronismo entre as réplicas. Já os algoritmos otimistas, coordenam essas atualizações em *background*, sem bloquear o acesso aos dados pelos usuários, fazendo "reconciliação" quando ocorre uma situação de conflito. Essa abordagem pressupões que problemas raramente acontecerão e, caso aconteçam, poderão ser resolvidos em sua maioria com um esforço maior de processamento.

É fácil perceber as consequências de cada abordagem na arquitetura de sistemas distribuídos. Se existe controle de concorrência síncrono, é porque as conexões estarão sempre disponíveis e provavelmente serão rápidas e confiáveis (como em uma LAN). No caso de controle assíncrono, processado em *background*, provavelmente as conexões não tem garantias de rapidez (como nas WANs). O bloqueio a usuários fica restrito às operações locais e entra em cena os paradigmas da reconciliação.

A replicação otimista é definida por Saito (2005) como um grupo de técnicas para um compartilhamento eficiente de dados em áreas de grande abrangência ou em ambientes móveis. O recurso chave que separa os algoritmos de replicação otimista da pessimista é sua

abordagem de controle de concorrência. Algoritmos pessimistas coordenam de forma síncrona as réplicas durante o acesso e bloqueiam os demais usuários durante a atualização. Algoritmos otimistas permitem que o dado seja acessado sem ter como condição prévia a sincronização e baseiam-se na hipótese otimista de que problemas somente ocorrerão raramente, se ocorrer.

O fato das atualizações de dados ocorrerem em segundo plano, de forma assíncrona, traz o conceito de operação. Esse termo é usado por Saito (2005) para significar em evento de atualização ocorrido em qualquer lugar e suas consequências.

6.2. Modelos de Replicação

Quando uma transação solicita uma operação de escrita, o SBD é responsável por atualizar um conjunto de cópias de X (o conteúdo do conjunto depende do algoritmo utilizado para gerenciar dados replicados). O SBD pode distribuir as operações de escrita imediatamente ou pode deferir as operações de escrita em cópias replicadas até que a transação termine. Com isso, existem dois modelos básicos de replicação: replicação imediata (*eager replication*) e replicação atrasada (*lazy replication*) (Figura 26) [Bernstein 1987], [Gray 1996]. A Figura 26 mostra as duas formas de propagar atualizações para as réplicas.

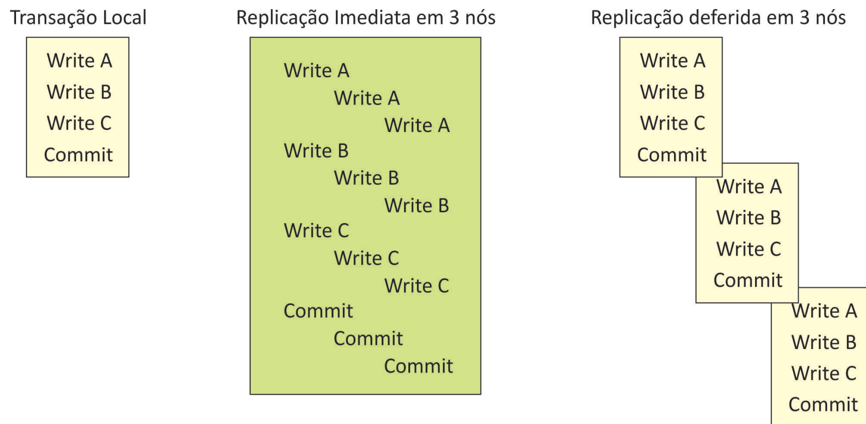


Figura 26: Replicação imediata e atrasada [Silva Neto 2005]

A replicação imediata mantém todas as réplicas exatamente sincronizadas em todos os nós atualizando todas elas como parte da transação original. Esse tipo de replicação fornece execução serializável, pois não existem anomalias de concorrência. Entretanto, a replicação imediata reduz a performance de atualização e aumenta o tempo de resposta das transações porque atualizações e mensagens extras são acrescentadas à transação.

A replicação imediata não é uma boa opção para aplicações móveis, onde a maioria dos nós está desconectada. As aplicações móveis requerem algoritmos de replicação atrasada que propagam as atualizações em réplicas de maneira assíncrona para os outros nós depois que a transação de atualização é comprometida (*commit*). Com isso, uma réplica é atualizada pela transação original e as atualizações em outras réplicas são propagadas assincronamente, normalmente com uma transação separada para cada nó. Alguns sistemas conectados permanentemente utilizam a replicação atrasada para aumentar o tempo de resposta.

Com escrita atrasada, o SBD usa uma visão não replicada do banco de dados enquanto a transação executa. Isto é, para cada item de dado que a transação lê ou escreve, somente uma

cópia do dado é acessada pelo SBD. Diferentes transações podem utilizar diferentes cópias. O SBD atrasa a distribuição das operações de escrita nas outras cópias até que a transação termine e esteja pronta para comprometer. Com isso, pode utilizar a votação da primeira fase do protocolo de comprometimento para enviar as operações para os sites correspondentes.

Uma desvantagem do método deferido é que ele pode atrasar o comprometimento de uma transação, pois a primeira fase do protocolo de comprometimento deve processar um número potencialmente grande de operações de escrita antes de responder a mensagem de votação. [Silva Neto 2005]. Com a técnica imediata os sites participantes podem processar operações de escrita enquanto a transação ainda está executando, evitando o atraso no momento do comprometimento.

A replicação imediata normalmente utiliza um esquema de bloqueios para detectar e controlar a execução concorrente. A replicação atrasada normalmente usa um esquema de concorrência com múltiplas versões para detectar comportamento não-serializável [Bernstein 1987]. Com isso, a replicação atrasada pode permitir que uma transação acesse um valor desatualizado.

Outra desvantagem do método deferido é que ele tende a atrasar a detecção de conflitos entre as operações, o que só acontece no fim da execução das transações. Por exemplo, duas transações T1 e T2 executando concorrentemente e ambas escrevendo em X. Suponha que o SBD utilize a cópia XA para executar T1 e a cópia XB para executar T2. Até que o SBD distribua a operação de escrita de T1 para as outras cópias de x ou a operação de escrita de T2, nenhum escalonador detecta o conflito.

Essa desvantagem do método deferido pode ser resolvida com o uso de uma mesma cópia do item de dados, chamada de cópia primária, para executar todas as transações. Nesse caso, tanto o método imediato quanto o deferido detectam o conflito no mesmo ponto de execução da transação.

Trabalhos mais antigos sobre replicação focavam em replicação imediata [Bernstein 1987], enquanto trabalhos mais recentes focam em replicação atrasada [Anderson 1998]. As atualizações podem ser controladas de duas formas: atualização em grupo (*group*) ou atualização feita pelo mestre (*master*). A Figura 27 ilustra ambas as formas.

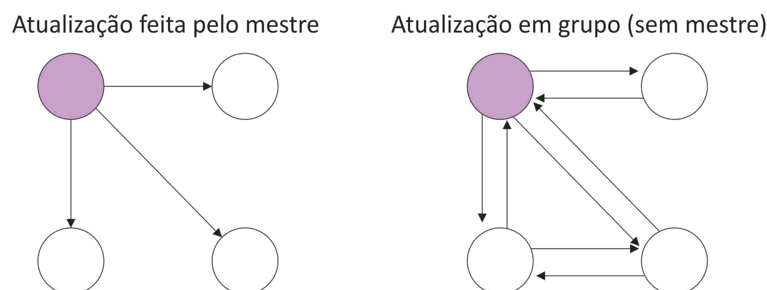


Figura 27: Atualização em grupo ou feita pelo mestre [Silva Neto 2005]

Na atualização em grupo, qualquer nó que possua uma cópia do item de dado pode atualizá-la. Isso é chamado de atualização em qualquer lugar (*update anywhere*). Esse tipo de atualização tem mais chance de ocasionar conflitos.

Na atualização feita pelo mestre, cada objeto tem um nó mestre e somente ele pode atualizar a cópia primária do objeto. Todas as outras réplicas são para leitura somente. Outros nós querendo atualizar o objeto devem requisitar a atualização ao mestre.

Na replicação atrasada com atualização em grupo (*lazy group replication*), qualquer nó pode atualizar qualquer item de dado local. Quando a transação é comprometida, uma transação é enviada para cada nó para fazer as atualizações da transação raiz em cada cópia do item de dado. É possível que dois nós atualizem o mesmo objeto e enviem suas atualizações para os outros nós. Nesse caso, o mecanismo de replicação deve detectar o problema e reconciliar as transações para que as atualizações não sejam perdidas.

Na replicação atrasada com atualização feita pelo mestre (*lazy master replication*), cada objeto possui o seu dono. As atualizações devem ser feitas pelo dono e depois propagadas para outras réplicas. Diferentes objetos têm diferentes donos. Esse tipo de replicação também não é apropriada para aplicações móveis, pois um nó querendo atualizar um item de dado deve estar conectado e participar em uma transação atômica com o dono do objeto. Da mesma forma que na replicação imediata, o esquema deferido com mestre não possui problemas de reconciliação, ao invés disso, os conflitos são resolvidos por espera ou por *deadlocks*.

6.3. Algoritmos de Replicação Imediata

Nesta seção, serão abordadas algumas técnicas de replicação imediata [Bernstein 1987], onde as operações de escrita nas réplicas são distribuídas de forma síncrona.

Técnica Write-all

Em um mundo ideal onde os sites nunca falham, um SBD pode gerenciar facilmente dados replicados. Ele traduz cada operação de leitura Read(X) em Read(XA), onde XA é qualquer cópia de X (XA denota a cópia de X no site A). E traduz cada operação de escrita Write(X) em {Write(XA1), ..., Write(XAm)}, onde { XA1, ..., XAm} são todas as cópias de X. Essa técnica para manipular dados replicados chama-se *write-all*.

Infelizmente, o mundo não é ideal, pois os sites podem falhar e se recuperar. Esse é um problema para a técnica *write-all*, porque requer que o SBD processe cada operação de escrita escrevendo em todas as cópias de um item de dado, mesmo se algumas delas estiverem em falha. Nesse caso, o SBD teria que atrasar o processamento de operações de escrita até que pudesse escrever em todas as cópias do item de dado. Esse atraso é ruim para as transações de atualização, pois quanto maior a replicação dos dados, menor é a disponibilidade do sistema para as transações de atualização. Por essa razão, a técnica de *write-all* não é satisfatória [Bernstein 1987].

Técnica Write-all-Available

Usando a técnica *write-all-available*, o SBD deve escrever em todas as cópias de um item de dados que estejam disponíveis, isto é, ele pode ignorar as cópias em sites que estão em falha. Isso resolve o problema de disponibilidade, mas pode causar problemas com relação à corretude.

Usando essa técnica, existirão situações onde algumas cópias de um item de dado não refletem o seu valor mais atualizado. Com isso, uma transação pode ler uma cópia

desatualizada, causando uma execução incorreta. Existe uma série de algoritmos para evitar essas execuções incorretas [Bernstein 1987], alguns deles serão comentados a seguir.

Um Algoritmo de Cópias Disponíveis

Os algoritmos baseados em cópias disponíveis tratam dados replicados usando variações da técnica *write-all-available* [Bernstein 1987]. Isto é, toda operação de leitura é traduzida em uma operação de leitura em qualquer cópia do item de dado e toda operação de escrita é traduzida em operações de escrita em todas as cópias disponíveis do item de dado em questão. Esses algoritmos tratam falhas nos sites, mas não tratam falhas de comunicação.

Algoritmo de Cópias Disponíveis Orientado a Diretório

Ao invés da distribuição das cópias nos sites ser estática, como no algoritmo anterior com cópias disponíveis, o algoritmo de cópias disponíveis orientado a diretório [Bernstein 1987] utiliza diretórios para definir o conjunto de sites que armazenam cópias de um item de dado em certo momento. Mais precisamente, para cada item de dado X, existe um diretório d(x) que lista o conjunto de cópias de X.

Como um item de dado, o diretório também pode estar replicado, ou seja, pode ser implementado como um conjunto de cópias de diretórios armazenadas em diferentes sites. O diretórios são atualizados por duas transações especiais, Include (IN) para criar novas cópias de itens de dados e Exclude (EX) para destruir cópias de itens de dados.

Algoritmo Usando Consenso do Quorum

No algoritmo usando consenso de *quorum* (QC – *Quorum Consensus*), atribui-se um peso não-negativo para cada cópia XA de X. Depois define-se um limiar de leitura RT (*Read Threshold*) e um limiar de escrita WT (*Write Threshold*) para X, tal que ambos 2WT e (RT+WT) são maiores que o peso total de todas as cópias de X. Um *quorum* de leitura (ou escrita) de X é qualquer conjunto de cópias de X com peso igual a pelo menos RT (ou WT) [Silva Neto 2005].

Algoritmo de Partição Virtual

O algoritmo de partição virtual (VP – *Virtual Partition*) foi projetado de tal forma que uma transação nunca precise acessar mais de uma cópia de um item de dados para leitura [Silva Neto 2005].]. Com isso, a cópia disponível que estiver mais perto pode ser usada para leitura.

Como no algoritmo usando consenso de *quorum*, cada cópia de X tem um peso não-negativo e cada item de dado tem limiares de leitura e escrita (RT e WT respectivamente). *Quoruns* de leitura e escrita são definidos como antes. Entretanto, eles servem para um propósito diferente nesse algoritmo.

A ideia básica do VP para cada site A é manter uma visão, que consiste de sites com os quais A pode se comunicar. Denota-se esse conjunto por v(A). Como sempre, cada transação T tem um site original (home(T)), onde ela foi iniciada. A visão de uma transação, v(T), é a visão do seu site original v(home(T)), no momento em que T começou sua execução. Se v(home(T)) for modificado antes de T “comitar”, T será abortada. T pode ser reiniciada e tentar executar considerando a nova visão do site.

6.4. Algoritmos de Replicação Atrasada

Neste seção, serão abordadas algumas técnicas de replicação atrasada [Silva Neto 2005] onde as operações de escrita nas réplicas são distribuídas assincronamente depois que a transação original terminar.

Protocolos de Acesso Multivisão para Replicação em Larga Escala

A replicação de dados não é só útil em aplicações de pequena escala aonde o número de sites é pequeno (por exemplo: < 10), mas também pode ser usada em muitas aplicações de larga escala. Com o advento das WAN e da Internet tem sido possível replicar dados para 100 ou mesmo 1000 localizações diferentes por exemplo.

Um importante ponto na gerência da replicação de dados é assegurar o acesso correto ao dado no que tange a concorrência, a falhas e a propagação das atualizações. O critério mais usado para a gerência da replicação de dados é o 1SR (*One-Copy Serializability*) [Bernstein 1987].

O modo mais frequente de garantir o correto acesso a dados replicados é usar protocolos de controle de concorrência para assegurar a consistência dos dados; protocolos de controle da replicação para assegurar a consistência mútua; e protocolos para estabelecer pontos de sincronismo (*commit*) para assegurar que as mudanças, ora para uma, ora para todas as cópias do dado replicado sejam feitas de forma permanente.

A maioria dos protocolos utilizados são: *Two-Phase Locking* (2PL) para controle de concorrência, *Read-One-Write-All* (ROWA) para controle da replicação e *Two-Phase Commit* (2PC) para estabelecer pontos de comprometimento. Estes três protocolos juntos asseguram o 1SR e garantem que cada transação enxergue as atualizações feitas através da serialização prévia das transações.

O critério 1SR para gerência de replicações trabalha bem em muitos sistemas replicados de pequena escala, porém não suporta eficientemente sistemas muito grandes, isto devido a natureza centralizada dos protocolos 2PL, 2PC e ROWA. Por exemplo, o protocolo 2PC requer uma mensagem trafegando entre o site *master* onde a atualização é iniciada e os demais sites envolvidos na atualização [Silva Neto 2005]. Isto não somente introduz um *delay* significativo para o sincronismo das atualizações em todos os sites, como também resulta em uma alta taxa de transações abortadas. Acontece que em muitas situações onde os dados são replicados, principalmente em sistemas distribuídos de larga escala, nem sempre a atualização precisa ser efetivada em todos os sites de forma síncrona e ao mesmo tempo.

Replicação Two-Tier

O *two-tier replication* é uma técnica de dividir o sistema replicado em duas camadas. Uma sempre conectada, apresenta topologia em anel ou totalmente conectada e congrega os chamados *sites base*. A outra, eventualmente conectada, inclui os *sites móveis* [Gray, 1996]. Os sites base trabalham com algoritmos pessimistas, o que garante a integridade de todas as réplicas. Cada site móvel comunica-se apenas sob sua responsabilidade e realiza os procedimentos de conciliação. Cada site base então retorna para todos os sites móveis sob sua tutela as efetivações definitivas ou as rejeições. Percebe-se que essa técnica é uma extensão

da topologia estrela onde os sites centrais de cada estrela formam a camada dos sites base. Essa técnica melhora a escalabilidade, mas sacrifica a flexibilidade da comunicação.

6.5. Modelos de consistência

Um modelo de consistência se baseia em um contrato entre um *data store* e processos "usuários", no qual se especifica como os resultados de operações *read* e *write* se comportam na presença de concorrência.

Os desenvolvedores de sistemas têm que escolher/conceber um modelo de consistência fácil de usar e com bom desempenho. Já os usuários ou utilizadores de sistemas têm que conhecer o modelo de consistência suportado pelo sistema para evitar surpresas desagradáveis.

Em termos gerais os modelos de consistência uniforme podem ser classificados em fortes e fracos (relaxados). Nos modelos de consistência forte, uma série de restrições é imposta na ordem das operações.

Consistência Forte

Nestes modelos, as operações são executadas considerando a ordem total, ou seja, o comportamento do sistema tenta se aproximar ao máximo ao de um sistema com um único processador [Pousa 2005] [Chagas 1999] e as operações em dados compartilhados são sincronizados.

Consistência sequencial

O modelo de consistência sequencial possui modelo de programação fácil e é muito forte em relação ao estado consistente dos dados compartilhados e todas as suas operações de escrita podem ser executadas localmente [Pousa 2005]. Este modelo de consistência define que uma execução esta sequencialmente consistente se: *"... o resultado de qualquer execução é o mesmo se as operações de todos os processos são executados em alguma ordem sequencial, e as operações de cada processo aparecem nesta sequência na ordem especificada pelo programa*" [Lamport 1979].

Então, o modelo de consistência sequencial proposto por Lamport (1979) pode ser definido formalmente como:

1. a ordem em que as operações aparecem no programa é mantida em cada processador individualmente e;
2. uma única ordem sequencial das operações de todos os processadores é vista, de modo que uma operação de acesso à memória deve ser atômica ou instantânea em relação às outras operações.

Consistência causal (apenas relações de causa)

O modelo de consistência causal é classificado como um modelo uniforme e relaxado. Neste modelo a ordem das operações é definida pela relação de dependência que existe entre duas ou mais operações. As operações de escrita em um objeto que geram alguma dependência em outras operações devem ser vistas pelos nodos na mesma ordem [Pousa 2005].

Consistência FIFO (apenas ordenação individual)

Neste tipo de abordagem todas as escritas por cada processo são vistas pelos outros, pela mesma ordem em que são emitidas. Dessa forma escritas por diferentes processos podem ser vistas por ordens diferentes pelos outros. Também são conhecidas como *Pipelined* RAM *consistency*. Como se as escritas de cada processo formassem um *pipeline* com origem no processo.

A consistência FIFO é geralmente fácil de implementar pois não há necessidade de se garantir da ordem de visão das escritas pelos diferentes processos, exceto que todos veem as escritas vindas de uma mesma fonte, por uma mesma ordem. Todas as escritas de diferentes processos são tratadas como concorrentes

Na consistência sequencial aceita-se uma ordenação qualquer, não-determinística, mas é a mesma em todos os processos.

Consistência (Fraca)

Nos modelos de consistência relaxada, as restrições impostas nos modelos de consistência são relaxadas e a ordem da execução das operações passa a ser parcial [Chagas 1999] e a sincronização ocorre apenas com bloqueio e liberação dos dados compartilhados.

Só impõe consistência em grupos de operações em vez de em escritas / leituras individuais. É útil quando há pouca frequência de acessos a dados partilhados: muitos intervalos sem acessos, depois muitos acessos num curto intervalo [Pousa 2005].

O modelo permite que o programador controle os momentos em que a consistência se verifica, em vez da sua forma e acaba por impôr consistência sequencial entre grupos de operações.

6.6. Gerenciamento de Réplicas

O problema de posicionamento de réplicas pode ser subdividido em dois subproblemas: posicionar servidores de réplicas e posicionar o conteúdo.

Posicionar servidores de réplicas refere-se a achar as melhores localizações para colocar um servidor que pode hospedar um depósito de dados (ou parte dele). Já o posicionamento de conteúdo: refere-se a achar os melhores servidores para colocar conteúdo. Antes de decidir o posicionamento do conteúdo, é preciso que os servidores de réplicas já tenham sido posicionados

Posicionamento do Servidor de Réplicas

Para definir o correto posicionamento dos servidores de réplicas algumas soluções são consideradas como, por exemplo, a distância entre clientes e localizações que podem ser medidas em termos de latência ou largura de banda. A posição é definida baseada na métrica que a distância média entre o servidor e os seu clientes é mínima. Deve-se ignorar a posição dos clientes e apenas considerar que a topologia da Internet é formada pelos sistemas autônomos (AS).

Outra solução é ignorar a posição dos clientes Neste caso considera-se o maior AS e coloca-se um servidor no roteador com maior número de enlaces e os ASs são escolhidos em ordem de tamanho

Replicação e Posicionamento de Conteúdo

Existem três tipos diferentes de réplicas organizadas logicamente (Figura 28): Réplicas permanentes, réplicas iniciadas por servidor e Replicas iniciadas por cliente.

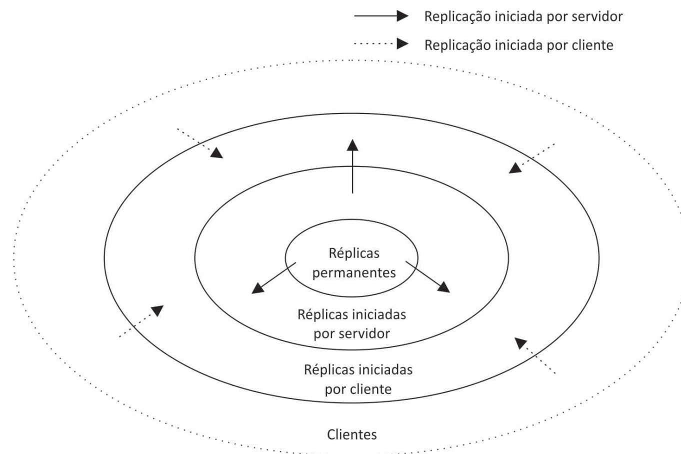


Figura 28: Organização lógica de diferentes cópias de um depósito de dados em três anéis concêntricos [Tanenbaum 2008]

Réplicas Permanentes é um conjunto inicial de réplicas que constituem um depósito de dados distribuído possuindo um número de réplicas permanentes e pequeno. Neste caso, os arquivos são replicados para um número limitado de servidores que estão em uma única localização.

Já as Réplicas Iniciadas por Servidor são as cópias de um depósito de dados que existem para aprimorar desempenho e que são criadas por iniciativa do (proprietário do) depósito de dados. Por exemplo: Considere um servidor na região do Centro, em Teresina. Imaginemos uma rajada de requisições que vêm de uma localização inesperada, por exemplo, na região de Altos. Ora, essa localidade é longe do servidor e não mapeada como região com características de exigência de alto nível. Em casos assim, vale a pena instalar uma quantidade de réplicas temporárias na região de onde se originam tais requisições, até que cesse o movimento atípico.

As Réplicas Iniciadas por Cliente são mais conhecidas como *caches* (de cliente). São um recurso de armazenamento local, usado por um cliente para armazenar temporariamente uma cópia dos dados que ele acabou de requisitar. Seu gerenciamento cabe inteiramente ao cliente e o depósito de dados de onde os dados foram trazidos nada tem a ver com a manutenção da consistência dos dados em cache. Geralmente são usadas para melhorar o tempo de acesso aos dados.

Quando um cliente quer acessar alguns dados, se conecta com a cópia do depósito de dados mais próxima. No caso de a maioria das operações envolver somente leitura de dados, o desempenho pode ser melhorado, dado que o cliente pode armazenar dados requisitados em uma cache mais próxima.

Exercícios

As questões abaixo devem ser respondidas em forma dissertativa e argumentativa com pelo menos uma lauda. Devem também refletir a interpretação da leitura do texto juntamente com pesquisas sobre o tema arguido.

1. Qual a importância da sincronização de relógios em sistemas distribuídos?
2. No contexto dos algoritmos de sincronização, explique com suas palavras como se dá a execução do algoritmo de Cristian.
3. O que é a Estampa de tempo de Lamport?
4. Para que serve um algoritmo de eleição em sistemas distribuídos?
5. O que é a Exclusão Mútua?
6. Explique as principais propriedades das transações
7. No contexto de replicação de dados nos sistemas distribuídos, para que ser o uso de múltiplas cópias de recursos?
8. Quais são as diferenças entre caches hierárquicas e cooperativas?
9. Diferencie replicação otimista e replicação pessimista.
10. Quais os principais modelos de replicação?

Resumo

Nesta unidade foram abordados aspectos de sincronização em Sistemas distribuídos. Para entender como os algoritmos de sincronização, mencionados nesta unidade, funcionam foi necessária uma prévia explanação sobre os tipos de relógios e sua importância no processo de sincronização dos diversos sistemas incluídos em um processamento distribuídos.

Ainda foram vistos temas como transações que fazem com que os SDs incorporem características como atomicidade e consistência. Foram vistos também mecanismos que reforçam essa consistência como por exemplo, a replicação. Nesse tema foram apresentados os principais modelos e técnicas de gerenciamento de réplicas

Bibliografia

ANDERSON, T.; BREITBART, Y.; KORTH, H. F.; WOOL, A; **Replication, Consistency, and Practicality: Are these Mutually Exclusive?**, Proceedings of the ACM SIGMOD'98, Seattle, USA, 1998.

BERNSTEIN, P. A.; HADSILACOS, V.; GOODMAN, N. **Concurrency Control and Recovery in Database Systems**, Addison-Wesley Publishing Company, 1987.

CHAGAS S., **Visualização de Modelos de Consistência de Memória**, Dissertação de Mestrado, Universidade de Brasília, 1999.

GRAY J.; HELLAND, P.; O'NEIL, P.; SHASHA, D. **The Dangers of Replication and a Solution**, Proceedings of the ACM SIGMOD'96, Montreal, Canadá, Junho 1996.

LAMPORT, L., **How to make a multiprocessor computer that correctly executes multprocess programs**, In IEEE Transactions on Computers, pp.690-691, 1979.

LISKOV, Barbara et. al., **The Language-Independent Interface of the Thor Persistent Object System.** Em: *Object-Oriented Multidatabase Systems.* Prentice-Hall, 1993.

OLIVEIRA, V. Fernandes. **Especificação e implementação de um modelo assíncrono para replicação, propagação e conciliação de bases de dados distribuídas.** Dissertação de Mestrado. PPGEEC/UFRN, 2007.

POUSA, C. Vilaça; **Proposta e Desenvolvimento de um Algoritmo Reconfigurável de Consistência de Objetos.** Dissertação de Mestrado, PUC-MG, 2005.

SAITO, Yasushi; SHAPIRO, Marc. **Optimistic Replication**. *ACM Computing Surveys*, v. 37, N. 1, Março 2005, pp 42-81

SILVA NETO, V. P. **Um estudo teórico sobre Replicação em Sistemas de Bancos de Dados.** Monografia de Graduação – Universidade Federal de Lavras. Departamento de Ciência da Computação. Minas Gerais, 2005.

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems principles and paradigms**. 1. ed. Prentice Hall, 2002.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas Distribuídos: Princípios e Paradigmas**. 2. ed., Prentice-Hall, 2008.

Weblografia

BENDINI, Silvio – **Sistemas Paralelos e Distribuídos**
http://fortium.edu.br/blog/silvio_bendini/files/2010/08/SPD_Aula1.pdf
Acessado em: 10/12/2010

FERNANDEZ, M. Porto – **Sistema Distribuído** (2005)
http://www.larces.uece.br/~marcial/cursos/sisdist/apostila_sisdist.pdf
Acessado em: 14/09/2010

GALANTE, Guilherme – **Transações Distribuídas**
http://www.inf.ufrgs.br/gppd/disc/cmp134/trabs/T2/041/ggalante/trans_distribuidas.pdf
Acessado em: 06/11/2010

PEREIRA JÚNIOR, L. Alves – **Temporização em Sistemas Distribuídos** (2007)
http://lasdpc.icmc.usp.br:10080/disciplinas/pos-graduacao/sistemas-distribuidos/2007/monografias-seminarios/ljr-tempo-20071206.pdf
Acessado em: 28/11/2010

Tempo Atômico Internacional
http://wapedia.mobi/pt/Tempo_At%C3%B4mico_Internacional
Acessado em: 06/10/2010

UTC – **Universal Time Coordinated**
http://www.aquaviarios.com/index.php?option=com_glossary&task=list&glossid=1&letter=U&Itemid=53
Acessado em: 06/10/2010

UNIDADE IV

Segurança

Objetivos:

1. Identificar e reparar falhas
2. Entender os tipos de falhas
3. Conhecer como implementar redundância em sistemas distribuídos
4. Aprender políticas e Mecanismos de Segurança
5. Conhecer os métodos de criptografia e autenticação de dados.

7. Tolerância e falha

Antes de falarmos como tratar ou reparar uma falha, precisamos entender seu conceito. Falha (ou avaria) de um sistema ocorre quando o sistema não consegue efetuar adequadamente (de forma correta e em tempo hábil) as tarefas que lhe competem. Uma falha pode ser desencadeada pela falha de um só componente (hardware/software).

A distribuição de partes de um sistema por diferentes meios de armazenamento leva a falhas intrínsecas de sistemas distribuídos que é a inconsistência e a falta de integridade dos dados. Para resolver este problema são utilizadas várias técnicas que garantem que todas as partes sejam consistentes, ou seja, que eles possuam o mesmo estado no momento em que houver uma falha e uma outra parte possa assumir o processamento a partir do ponto em que ocorreu a falha.

Outros conceitos podem levar a confusão sobre o conceito de falha como *defeito* e *erro*. Weber (2005) e Filho (2004) apontam defeito (ou avaria) como um desvio da especificação. Ou seja, o sistema não pode prover o serviço desejado. Já erro é um estado tal que, o processamento posterior a esse estado, leva a um defeito. E falha é a causa física ou algorítmica do erro.

Tanenbaum (2008) é mais específico para sistemas distribuídos e define que um sistema apresenta defeito quando não pode cumprir suas promessas. Se um sistema distribuído é projetado para oferecer a seus usuários uma série de serviços, o sistema falha quando ou mais desses serviços não podem ser fornecidos (completamente). Um erro é uma parte do estado de um sistema que pode levar a uma falha. A causa de um erro é denominada falha.

Assim, pode-se concluir que a falha pode levar ao erro, e o erro pode levar ao defeito. O defeito ocorre quando o sistema não consegue atender total ou parcialmente seus usuários. Por outro lado, quando ocorreu um defeito, então ocorreu um ou mais erros, cujas causas foram uma ou mais falhas. A Figura 29 ilustra os conceitos.

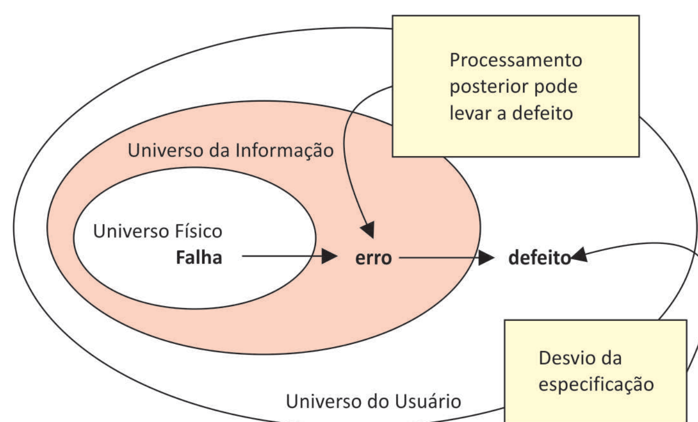


Figura 29: Falha, erro e defeito [Webber 2005]

7.1. Tipos de Falhas

As falhas podem ser transientes (mau funcionamento temporário), persistentes (nunca mais voltará a funcionar) ou intermitentes (às vezes funciona, às vezes não). Para compensar os problemas decorrentes das falhas usam-se técnicas para mascarar tais falhas. Desta forma é

possível confinar os seus efeitos sobre o sistema. De uma forma geral as falhas podem ser classificadas de acordo com sua natureza (ou causa) ou de acordo com sua duração ou persistência.

De acordo com sua natureza, as falhas podem ser classificadas em falhas físicas e falhas humanas [Laprie 1985]. Falhas físicas são aquelas causadas por algum mau funcionamento de um componente físico. Este mau funcionamento pode ser originado por diversas razões, tais como curto-circuito, perturbações eletromagnéticas, mudança de temperatura, etc.

Falhas humanas são imperfeições que podem ser oriundas de falhas de projeto, cometidas durante as fases de projeto e planejamento do sistema, ou durante a execução de procedimentos de operação ou manutenção. Outro tipo de falhas humanas são as falhas de interação, cometidas por violar inadvertidamente ou deliberadamente procedimentos de operação ou manutenção.

De acordo com sua duração e momento de ocorrência, as falhas podem ser classificadas como transientes ou permanentes. Falhas transientes são aquelas de duração limitada, causadas por mau funcionamento temporário ou por alguma interferência externa. Tais falhas podem ser também intermitentes, ocorrendo repetidamente por curtos intervalos de tempo.

Falhas permanentes são aquelas em que uma vez que o componente falha, ele nunca volta a funcionar corretamente. Muitas técnicas de tolerância a falhas assumem que os componentes falham permanentemente. Nos sistemas distribuídos, segundo Tanenabaum (2008), as falhas podem ser classificadas como a seguir:

Falhas por queda
O servidor para de funcionar, mas estava funcionando corretamente até parar.

Falha por omissão
Dá-se quando um processo ou um canal de comunicação falha a execução de uma ação que devia executar, como por exemplo: uma mensagem que devia chegar não chegou. O processo de falha (*crash*), pode se dar de duas formas:

- Omissão de recebimento: servidor não consegue receber mensagens;
- Omissão de envio: servidor não consegue enviar mensagens;

As falhas de omissão podem ser mascaradas por replicação ou repetição. Exemplo: se uma mensagem não chegou dentro de um certo período – o que se detecta por um *timeout* – então pode-se emiti-la novamente. Outra hipótese é duplicar o canal, enviar mais do que uma cópia em paralelo e filtrar as mensagens duplicadas.

Falha temporal
Uma falha temporal dá-se quando um evento que se devia produzir num determinado período de tempo ocorreu mais tarde. A resposta do servidor se encontra fora do intervalo de tempo. As falhas temporais são difíceis ou impossíveis de mascarar. Normalmente, apenas os sistemas de tempo real se preocupam com este tipo de falha.

Falha Arbitrária
Uma falha arbitrária ou bizantina dá-se quando se produziu algo não previsto, como por exemplo quando chega uma mensagem corrompida, um atacante produziu uma mensagem não esperada. Para lidar com estas falhas é necessário garantir que elas não levam a que outros componentes passem a estados incorretos.

As falhas arbitrárias podem ser difíceis de mascarar. Pode-se tentar transformá-las em falhas por omissão. Exemplo: Um CRC numa mensagem permite transformar uma falha bizantina (alguns ou todos os usuários percebem o serviço incorreto de forma diferente, alguns até podem perceber o serviço como correto) do canal numa falha por omissão. Os mesmos tipos de técnicas são utilizadas, em muitas vezes, nos próprios componentes hardware (discos, memórias, etc.), mesmo nos sistemas centralizados.

CRC (Cyclic redundancy check), ou verificação de redundância cíclica é um código detector de erros, que gera um valor expresso em poucos bits em função de um bloco maior de dados, como um pacote de dados, ou um arquivo, por forma a detectar erros de transmissão ou armazenamento.

7.2. Tolerância a Falhas

Estando um sistema sujeito aos diversos tipos de falhas citados, a reação da comunidade acadêmica, empresas e profissionais da área foi pesquisar soluções para que sistemas possam "conviver" com a possível condição de falha. Esta convivência é chamada de tolerância a falhas, Filho (2004) entende como válida também chamá-la de alta disponibilidade ou dependabilidade), palavra não existente no português, mas adaptada pelos autores do assunto por causa do termo em inglês *dependability*.

Fases da tolerância a falhas
Um sistema tolerante a falhas tenta evitar a ocorrência de defeitos no sistema mesmo na ocorrência de falhas. Ainda, a implementação de um sistema tolerante a falhas será totalmente ligada a arquitetura e desenho dos recursos e serviços que serão disponibilizados. Apesar de não haver uma técnica geral sobre como adicionar alta disponibilidade a qualquer sistema, existem algumas atividades que todo sistema tolerante a falhas deve executar a fim de atingir este objetivo [Souza & Campos 2008].

1. Detecção de erros – fase em que o sistema percebe a situação de erro e se prepara para o tratamento destes.
2. Confinamento do estrago e avaliação – Esta fase visa conter o estrago ao mínimo possível e evitar que eles se propaguem. Para isso é importante não haver intervalo entre a falha e a detecção.
3. Recuperação de erros;

Esta fase visa a correta retirada do erro do sistema, as técnicas mais comuns são FILHO (2004):

a) Recuperação para trás: voltar ao estado anterior da falha;
b) Recuperação para frente: aplica-se medidas corretivas pois nenhum estado anterior está disponível;

c) Tratamento de falhas;
d) Serve para impedir que futuros erros aconteçam. Aqui se faz:
    a. Localização da falha (localiza-se os componentes);
    b. Reparo do Sistema (isola os componentes com defeito);

7.3. Redundância

Segundo Weber (2005), toda técnica de tolerância a falhas envolve alguma forma de redundância e segundo Filho (2004), esta é um pré-requisito para atingir alta disponibilidade.

São formas de redundância, segundo Tanenbaum (2007):

- Redundância de informação: São adicionados bits extras para permitir a recuperação de bits deteriorados;
- Redundância de tempo: onde uma ação é realizada e, se necessário, ela é executada novamente – útil para falhas transientes ou intermitentes;
- Redundância Física: São adicionados processos ou equipamentos extras para possibilitar que o sistema como um todo tolere a perda ou mau funcionamento de alguns componentes.

Weber (2005) classifica desta maneira:

- Redundância de Hardware – Replicação dos componentes. Pode ser passiva (mascaramento), ativa (detecção, localização e correção) ou híbrida.
- Redundância de Software – Softwares precisam ser projetados para tolerância a falhas. São exemplos de projetos: n-versões (programação de versões diferentes), blocos de recuperação e verificação de consistência

É preciso lembrar que não só os componentes informáticos precisam ser redundantes mas todos os envolvidos no sistema, como no-breaks, geradores, ar-condicionado etc. A redundância pode ser utilizada para aumentar a confiabilidade de um sistema sem qualquer alteração na confiabilidade dos componentes individuais que formam o sistema [Pham 2003].

Segundo [Piazza 2000], a confiabilidade de um sistema é a probabilidade de que, quando em operação sob condições ambientais estabelecidas, o sistema apresentará uma performance desejada (sem falhas) para um intervalo de tempo especificado e a redundância é a existência de mais de um meio para se atingir um objetivo determinado.

Finkel et al. (2006) comenta que a redundância deve ser utilizada para fornecer um sistema em operação constante, embora um ou mais instrumentos possam falhar. Uma das formas mais utilizada para a prevenção dos efeitos de falhas é a utilização de módulos redundantes. A redundância de hardware implica inclusão de circuitos de hardware adicionais ao mínimo necessário para o funcionamento do sistema e a redundância de software implica geração de versões distintas do software do sistema ou de partes desse software, sempre se baseando em uma especificação comum.

7.3.1. Redundância de Hardware

O emprego de redundância em um Sistema Instrumentado de Segurança consiste na utilização de componentes auxiliares com a finalidade de realizar as mesmas funções desempenhadas

por outros elementos presentes no sistema. A finalidade principal da utilização de redundância nesses sistemas é a prevenção de condições ou estados inseguros.

Embora a redundância sempre implique na adição de novos componentes, deve-se fazer o possível para não aumentar a complexidade do sistema, de forma a não se ter efeito contrário, ou seja, diminuição da confiabilidade e da segurança do sistema. A redundância de hardware pode ser implementada através de três formas básicas: Redundância Estática, Redundância dinâmica e Redundância por software.

Redundância estática

A redundância estática utiliza o mascaramento de falhas como principal técnica. O projeto é feito de forma a não requerer ações específicas do sistema ou de sua operação em caso da ocorrência de falhas. Todos os elementos executam a mesma tarefa e o resultado é determinado por votação. Exemplos são TMR (*Triple Modular Redundancy*) e NMR (*N Modular Redundancy*).

A TMR, redundância modular tripla, é uma das técnicas mais conhecidas de tolerância a falhas. Ela mascara falha em um componente de hardware triplicando o componente e votando entre as saídas para determinação do resultado Figura 30. A votação pode ser por maioria (2 em 1) ou votação por seleção do valor médio. (Weber 2005).

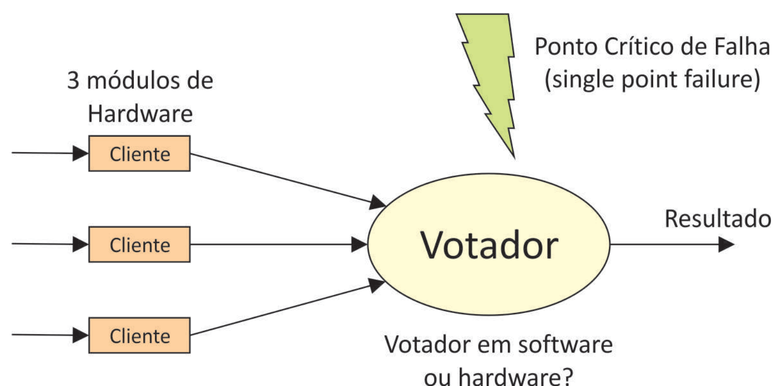


Figura 30: Redundância Mudular Tripla [Webber 2005]

Para Weber (2005), o votante realiza uma função simples, fácil de verificar a correção. É interessante observar que o votante não tem a função de determinar qual o módulo de hardware discorda da maioria. Se essa função for necessária no sistema, ela deve ser realizada por um detector de falhas.

Apesar de simples, o votante, por estar em série com os módulos de hardware e ter a responsabilidade de fornecer o resultado, é o ponto crítico de falhas no esquema TMR. Se o votante apresentar baixa confiabilidade, todo o esquema vai ser frágil, tão frágil como o votante. (WEBER, 2001b).

Soluções para contornar a fragilidade do votante são:

- Construir o votante com componentes de alta confiabilidade;
- Triplicar o votador;
- Realizar a votação por software.

Conforme Weber (2005), a TMR apresenta uma confiabilidade maior que um sistema de um único componente até a ocorrência da primeira falha permanente. Depois disso, perde a capacidade de mascarar falhas e vai apresentar uma confiabilidade menor que um sistema de um único componente. Com o tempo, quando aumenta a probabilidade de componentes falharem (ou seja, aumenta a taxa de defeitos), a TMR apresenta uma confiabilidade pior do que um sistema não redundante.

Torna-se ideal então para períodos não muito longos de missão. Suporta uma falha permanente apenas e é ideal para falhas temporárias, uma de cada vez [Fillipi 2007]. A redundância modular múltipla, NMR, é a generalização do conceito de TMR, ou seja, a TMR é um caso especial de NMR.

Redundância dinâmica
Implica na detecção de falhas, caso em que o sistema deve tomar alguma ação para anular seus efeitos, o que normalmente envolve uma reconfiguração do sistema. Costuma ser usada em aplicações que suportam permanecer em um estado errôneo durante um curto período de tempo necessário para a detecção do erro e recuperação para um estado livre de falhas. Um exemplo de implementação de redundância dinâmica é através de módulos estepes (*hot standby*).

A redundância dinâmica é usada em aplicações que suportam permanecer em um estado errôneo durante um curto período de tempo, tempo esse necessário para a detecção do erro e recuperação para um estado livre de falhas. Um exemplo de implementação de redundância dinâmica é através de módulos estepes (*standby sparing*).

Existe ainda a chamada redundância híbrida que consiste na combinação de técnicas estáticas com técnicas dinâmicas. Utiliza mascaramento de falhas para prevenir que erros se propaguem, detecção de falhas e reconfiguração para remover, do sistema, unidades com falha.

7.3.2. Redundância de Software

Simplesmente replicar softwares idênticos pode não ser uma estratégia muito eficaz. Rotinas idênticas de software vão apresentar erros idênticos. Logo, não basta copiar um programa e executá-lo em paralelo ou executar o mesmo programa duas vezes.

Um problema que surge é a falta de uma metodologia de eficiência reconhecida para a avaliação da segurança do software para os sistemas que precisam de segurança e que tem o software como componente crítico [Soppa 2009].

Pode-se dizer que a falta de experiência em especificações de software e das especificações em relação ao ambiente de aplicação representa um grave problema, fazendo com que o sistema atinja situações imprevistas, como consequência de procedimentos operacionais incorretos, de mudanças não esperadas no ambiente operacional, ou ainda de modos de falhas não previstas do sistema [Jaffe et al.,1991].

Um caminho para se definir e avaliar melhor o conceito de segurança do software é através da definição de um conjunto de fatores que consigam representar adequadamente o conceito de segurança. Em função da dificuldade da comprovação da não existência de falhas na

implementação de um software, em relação à sua especificação, são utilizadas técnicas de redundância de software, cujo objetivo é tornar o software mais robusto em relação à segurança, ou seja, tolerante a falhas porventura ainda existentes [Soppa 2009].

Diversidade

Diversidade, também chamada programação *diversitária*, é uma técnica de redundância usada para obter tolerância à falhas em software. A partir de um problema a ser solucionado são implementadas diversas soluções alternativas, sendo a resposta do sistema determinada por votação [Soppa 2009].

A aplicação da técnica não leva em conta se erros em programas alternativos apresentam a mesma causa (por exemplo, falsa interpretação de uma especificação ou uma troca de um sinal em uma fórmula). Os erros, para poderem ser detectados, devem necessariamente se manifestar de forma diferente nas diversas alternativas, ou seja, devem ser estatisticamente independentes. Experimentos comprovam que o número de manifestações idênticas (erros que assim não seriam detectados) é significativamente menor que o número total de erros, conforme Weber (2005).

A diversidade pode ser utilizada em todas as fases do desenvolvimento de um programa, desde sua especificação até o teste, dependendo do tipo de erro que se deseja detectar (erro de especificação, de projeto ou de implementação). Essa técnica é chamada de projeto diversitário, quando o desenvolvimento do sistema é realizado de forma diversitária e de programação em n-versões. Pode ser também usada como técnica de prevenção de falhas.

Nesse último caso, várias alternativas de projeto ou de implementação são desenvolvidas para que, na fase de teste, erros eventuais possam ser localizados e corrigidos de uma forma mais simples e efetiva. No final da fase de teste é escolhida a alternativa em que se detectou a menor ocorrência de erros. Apenas esta alternativa é integrada ao sistema [Weber 2005].

Blocos de recuperação

Nessa técnica programas secundários só serão necessários na detecção de um erro no programa primário. Essa estratégia envolve um teste de aceitação. Programas são executados e testados um a um até que o primeiro passe no teste de aceitação. Essa estratégia envolve um teste de aceitação (Figura 31). Programas são executados e testados um a um até que o primeiro passe no teste de aceitação. A estratégia de blocos de recuperação tolera n-1 falhas, no caso de falhas independentes nas n versões.

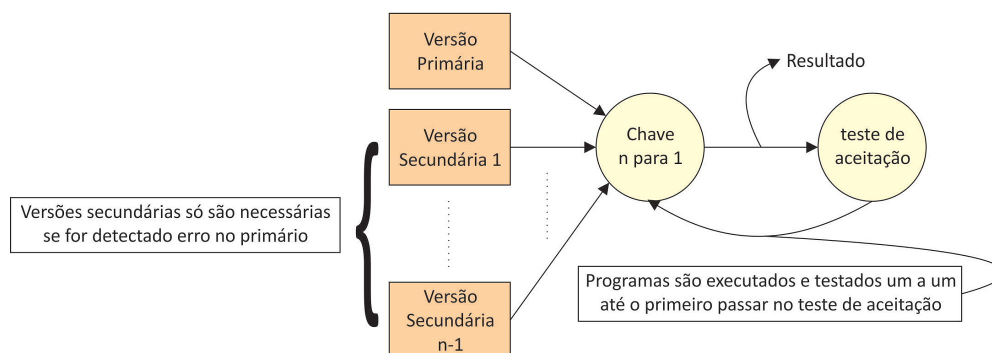


Figura 31: Blocos de Recuperação [Webber 2005]

7.4. Recuperação de falhas

Para que um sistema não sofra pane total ao ser detectada uma falha, o sistema deve ser automaticamente reconfigurado. Ou seja, um processo que estava sendo executado deve ser realocado para outros caminhos alternativos que mantenham a comunicação contínua. Mas, para que o processo de recuperação não reduza a performance do sistema e seja o mais transparente possível para o usuário, a técnica de recuperação por avanço é bastante utilizada e requer permanentemente um estado consistente entre as partes de um sistema [Mendonça 2002]. Os mecanismos utilizados na técnica de recuperação são (Figura 32): *Logging* e *checkpoint.*

Logging: O *logging* é o mecanismo pelo qual as requisições feitas pelos usuários a uma parte do sistema que está distribuído em algum meio físico são gravadas em um arquivo *log*, para que no caso de uma falha, uma outra parte do sistema que assumir o lugar do anterior possa obter o estado corrente e daí continuar o processo normalmente.

Checkpoint: O *checkpoint* é um determinado ponto no qual o estado atual de uma parte do sistema que está no arquivo log é verificado e copiado no arquivo de log das demais partes. Isto faz com que todas as partes distribuídas do sistema estejam sempre atualizadas, mantendo-se assim consistentes. Na Figura 32, podemos ver um exemplo dessas abordagens.

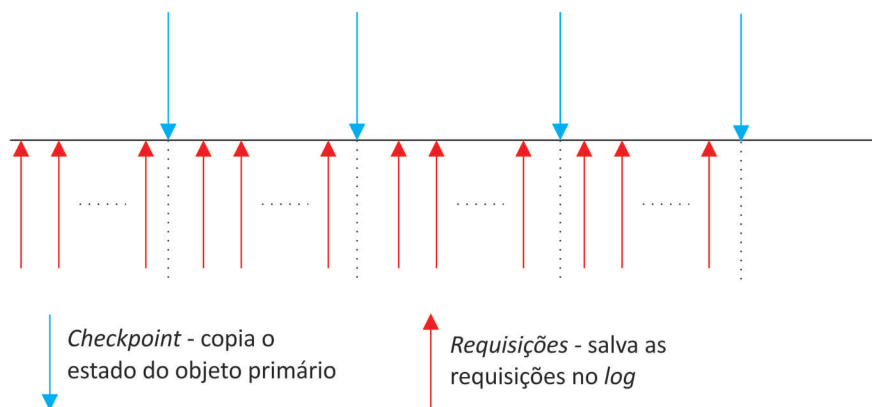


Figura 32: Checkpoint e logging [Mendonça 2002]

7.4.1. Pontos de controle (checkpoint)

O *checkpoint* é realizado a cada tempo t, que pode ser definido pelo usuário ou fixado na implementação do sistema. Durante este intervalo de tempo t, várias requisições de usuários são realizadas e gravadas no arquivo de *log*. Um exemplo da utilização da técnica de recuperação é uma transação bancária onde um cliente requisita a transferência de um determinado valor de uma conta X para uma conta Y. Nesta transação ocorrem duas operações: uma de débito da conta X, e uma de crédito na conta Y.

Suponhamos que no intervalo de tempo entre o débito e o crédito ocorra uma falha e só tenha sido realizado o débito da conta X, então haveria aí uma inconsistência de dados, pois o valor debitado da conta X estaria perdido. Mas, com o uso do checkpoint e *logging* isto é resolvido da seguinte maneira. Suponhamos que temos partes do sistema que realiza esta operação de transferência em vários meios físicos confiáveis, como por exemplo: em dois servidores de aplicação (um ativo e um outro como backup). Quando o cliente requisita a operação de transferência, o servidor ativo assume a operação e o mecanismo de logging grava esta

requisição em um servidor de *log*. Se o intervalo de *checkpoint* for suficientemente curto, esta requisição será repassada para um arquivo de *log* no servidor backup.

No momento em que ocorre uma falha como a descrita anteriormente, o cliente é logo redirecionado para o servidor backup e o mecanismo de recuperação (*recovery*) consulta o *log* para verificar qual foi a última operação realizada, e a partir deste ponto continuar a operação, sem causar transtornos para o cliente.

8. Segurança

Uma rede de computadores, por natureza, tem várias ameaças. Num sistema informático o principal objetivo é proteger a informação, ou seja, controlar a capacidade de acessar, alterar ou executar um arquivo, um registro, uma base de dados, etc. Mas o principal objetivo de um sistema distribuído é compartilhamento de informação. Isto conduz a um dilema uma vez que também é necessário facilitar o acesso.

A segurança está relacionada à necessidade de proteção contra o acesso ou manipulação, intencional ou não, de informações confidenciais por elementos não autorizados e a utilização não autorizada do computador ou de seus dispositivos periféricos. A necessidade de proteção deve ser definida em termos das possíveis ameaças e riscos e dos objetivos de uma organização, formalizados nos termos de uma política de segurança.

A solução passa por definir uma política de segurança, na qual se deve ter sempre em conta vários fatores como é o caso do custo, valor da informação, o que se pretende proteger, que mecanismos de proteção podem ser utilizados e claro, fazer uma avaliação das possíveis ameaças e formas de ataque. Esta política teve origem nos sistemas militares, em que o principal problema era a divulgação de informação não autorizada.

Para além do controle do acesso à informação também é necessário garantir a sua integridade. É necessário ter em consideração as várias ameaças a definir na política de segurança. Entre elas destacam-se o acesso e divulgação não autorizado de informação, modificação ilegítima da informação, operações não autorizados sob recursos do sistema, bloqueio de acesso legítimo à informação e vandalismo.

8.1. Política de Segurança e Mecanismos de Segurança

Uma política de segurança é um conjunto de leis, regras e práticas que regulam como uma organização gerencia, protege e distribui suas informações e recursos. E os mecanismos de segurança são os métodos que devem ser empregados para executar a política de segurança. Assim como uma fechadura numa porta não garante a segurança de um prédio se não houver uma política para seu uso, os mecanismos de segurança que nós descreveremos não garantem a segurança de um sistema a menos que existam políticas para seu uso.

A distinção entre políticas e mecanismos de segurança é útil quando estamos projetando um sistema seguro, mas frequentemente é difícil garantir que um dado conjunto de mecanismos de segurança execute completamente a política de segurança desejada. Note que as políticas de segurança são independentes da tecnologia utilizada.

A política de segurança deveria representar sucintamente como é a política de sua organização com relação à:

- Quem tem permissão para usar o sistema;
- Quando eles têm permissão para usá-lo;
- O que eles têm permissão para fazer (grupos diferentes podem ter diferentes níveis de acesso);
- Procedimentos para garantir acesso ao sistema;
- Procedimentos para revogar o acesso de determinadas pessoas (por exemplo quando um funcionário deixa de trabalhar na empresa);
- O que constitui o uso aceitável do sistema;
- Métodos para *login* local e remoto;
- Sistema de monitoramento de procedimentos;
- Protocolos para reação às possíveis falhas na segurança.

8.2. Conceitos de segurança

Os métodos através dos quais as violações de segurança podem ser perpetradas em sistemas distribuídos dependem da obtenção de acesso aos canais de comunicação existentes ou do estabelecimento de canais que acrescentam às conexões de um cliente alguma autoridade desejada. Eles incluem:

- *Eaveadropping*: obtenção de cópias de mensagens sem autorização. Isto pode ser feito diretamente pela rede ou examinando informações que estejam protegidas inadequadamente durante a armazenagem. Por exemplo, na Internet uma estação de trabalho pode ajustar seu endereço na rede, como sendo o endereço de uma outra estação. Assim, a primeira estação pode receber as mensagens que são endereçadas para a segunda estação.
- *Masquerading*: enviar ou receber mensagens usando a identidade de outro usuário sem a autorização dele. Isto pode ser feito utilizando a senha e a identidade desse usuário ou usando um *token* de acesso (ou a habilidade) depois que a autorização para o uso tenha expirado.
- *Message tampering*: interceptar mensagens ou alterar seu conteúdo antes que elas cheguem ao destinatário. É difícil conseguir evitar esse tipo de violação dentro de um meio *broadcast*, como um *Ethernet*, uma vez que a camada física de comunicação garante a entrega de mensagens para todas as estações, mas é relativamente simples em redes *store-and forward*.
- *Replaying*: armazenar mensagens e enviá-las atrasadas. Por exemplo, depois do esgotamento do prazo para utilização de um recurso. O *Replaying* não pode ser combatido por simples encriptação, uma vez que o reenvio de mensagens pode ser utilizado para consumir tempo de utilização de recursos e vandalismo, mesmo quando as mensagens não podem ser interpretadas pelo perpetrador.

8.2.1. Segurança em um SD

Para violar um sistema em um dos modos acima, é necessário acesso ao sistema. Virtualmente, todos os computadores possuem canais de comunicação que podem ser usados para conseguir acesso não-autorizado. Em sistemas distribuídos, os computadores estão

ligados a uma rede e seus sistemas operacionais oferecem uma interface padrão de comunicação que permite que canais de comunicação virtuais podem ser estabelecidos.

As principais ameaças à segurança em sistemas distribuídos derivam da abertura dos canais de comunicação (isto é, as pontes usadas para legitimar a comunicação entre processos de clientes e servidores) e sua consequente vulnerabilidade ao *eavesdropping, masquerading, tampering* e *replaying*. Devemos assumir que todo canal de comunicação em qualquer nível do hardware ou software do sistema está sujeito a essas ameaças.

Violadores em potencial (humanos ou programas) não são facilmente identificáveis, então nós devemos adotar uma visão do mundo que não admite confiança. Mas nós devemos começar com alguns componentes confiáveis. Uma abordagem efetiva para o projeto é assumir que toda a comunicação tem origem não confiável até que se prove o contrário. A confiabilidade de parceiros de comunicação deve ser demonstrada sempre que um canal de comunicação for utilizado.

Os mecanismos usados para a implementação de segurança devem ser validados para um padrão alto - por exemplo, protocolos de comunicação segura e os softwares que os implementam deveriam ser demonstravelmente corretos para todas as sequências possíveis de mensagens. As demonstrações deveriam ter o rigor de provas formais.

8.2.2. Ataques de segurança

Infiltração
Para lançar tais ataques em um sistema distribuído, o atacante deve ter acesso ao sistema para poder executar o programa que implementa o ataque. A maioria dos ataques são lançados por usuários de dentro do sistema. Eles abusam de sua autoridade executando programas que são designados para acarretar umas das formas de ataque [Flemming 2005]. Para usuários de fora do sistema, um método simples é o de adivinhação de senhas, ou do uso de programas para quebrar a senha de um usuário conhecido. Tais ataques podem ser prevenidos pelo uso de senhas bem escolhidas e de tamanho adequado.

Além dessas formas diretas de infiltração, existem vários métodos sutis que tem se tornado bem conhecidos através da grande publicidade que gira em torno dos ataques bem sucedidos. Esses métodos são:

- Vírus: um programa que é anexado a um programa hospedeiro qualquer e se instala no ambiente alvo sempre que o hospedeiro é executado. Uma vez instalado, o vírus executa suas ações criminosas a vontade, frequentemente utilizando uma data como gatilho. Como o nome indica, uma de suas ações é replicar-se, anexando-se sozinho a todos os programas que encontrar no ambiente alvo. Eles se transferem para outras máquinas sempre que um programa hospedeiro é movido, seja através de comunicação de rede ou através de armazenamento físico (disquetes principalmente).
- *Worm*: um programa que explora as facilidades para execução remota de processos em sistemas distribuídos. Tais facilidades podem existir tanto acidentalmente quanto propositadamente: O Internet *Worm* explora uma combinação de características

intencionais e acidentais para executar programas remotamente em sistemas Unix BSD.

- Cavalo de Tróia: um programa que é oferecido para os usuários como um sistema que executa tarefas funcionais, mas tem uma segunda função escondida dentro dele. O exemplo mais comum é o "*spoof login*", um programa que apresenta aos usuários *prompts* que são idênticas as caixas normais para validação de login e senha do usuário, mas na realidade, armazena os dados do usuário ingênuo num arquivo conveniente para uso ilícito. Tal programa pode ser deixado, executando, em uma estração desocupada, onde ele simulará a aparência de uma tela sem usuários logados.

Estes nomes alcançaram certa notoriedade, mas deveria ser observado que nem todos os programas que exibem essas características são necessariamente maldosos, e de fato o *worm* tem sido utilizado com sucesso na exploração de um mecanismo para a alocação de tarefas computacionais para processadores em sistemas distribuídos.

8.3. Criptografia

De forma de evitar a escuta e falsificação de mensagens surge a cifra de mensagens – Criptografia. Ou seja, é estabelecido um canal seguro onde passam os dados cifrados. As mensagens a serem criptografadas, são transformadas por uma função que é parametrizada por uma chave. O texto criptografado é então transmitido e, no destino, o processo inverso ocorre, isto é, o método de criptografia é aplicado novamente para decodificar o texto criptografado [UENO 2003].

A segurança de sistemas distribuídos é uma necessidade e a utilização de técnicas que envolvem a criptografia assimétrica garantem uma solução elegante para a confidencialidade e assinatura digital de mensagens. Muitas aplicações conhecidas utilizam criptografia assimétrica direta ou indiretamente. A sua simplicidade e forte segurança possibilitaram a adesão por parte das empresas e da comunidade.

Convém referir que apesar das mensagens estarem cifradas, existe uma pequena probabilidade de um atacante deduzir a chave, embora esta seja muito pequena. O tamanho da chave de cifra é fulcral, pois uma chave pequena é mais facilmente descoberta (através de “força bruta” por exemplo) que uma chave maior.

8.3.1. DES (Data Encryption Standand)

Um dos principais métodos de criptografia baseado em chave secreta é o DES desenvolvido pela IBM e adotado pelo governo dos EUA como método de criptografia padrão. O método DES codifica blocos de 64 bits de texto normal gerando 64 bits de texto criptografado. O algoritmo de codificação é parametrizado por uma chave K de 56 bits e possui 19 estágios diferentes.

Num algoritmo de criptografia simétrica, os processos de encriptação e desencriptação recorrem a uma mesma chave. Tipicamente, o emissor e o receptor partilham uma chave mantida em segredo a todo o custo. Na criptografia assimétrica o processo de encriptação utiliza uma chave forçosamente diferente da chave utilizada na desencriptação.

O primeiro estágio realiza uma transposição dos bits do texto independente da chave. O último estágio realiza uma transposição inversa a do primeiro estágio. O penúltimo estágio realiza a permutação dos 32 bits mais significativos com os 32 bits menos significativos do bloco de dados. Os outros 16 estágios são funcionalmente idênticos (transposições e substituições), porém são parametrizados por chaves Ki, obtidas pela aplicação de funções que variam de um estágio i para outro, nos bits da chave K original. O método permite que a decodificação seja feita com a mesma chave usada na codificação, através da execução das mesmas etapas na ordem inversa.

O principal problema do método DES, e de todos os algoritmos de criptografia simétricos é a exigência que o transmissor e o receptor de uma mensagem conheçam a chave secreta e única usada na codificação e na decodificação. O acordo em torno da chave secreta entre transmissor e receptor, quando eles estão distantes um do outro, não é um problema trivial, pois envolve a transmissão da chave e nem sempre é possível garantir que essa transmissão não seja interceptada [UENO 2003].

Se for interceptada, o responsável pelo ataque poderá ler todas as mensagens que serão criptografadas utilizando a referida chave secreta. Um complicador para o problema é o fato de que em um sistema com n usuários, comunicando-se dois a dois, são necessárias n2 chaves secretas. A tarefa de gerar, transmitir e armazenar chaves em um sistema de segurança é denominada gerenciamento de chaves.

8.3.2. RSA (Rivest, Shamir e Adleman)

O mais importante método de criptografia assimétrico é o RSA, cujo nome deriva das iniciais dos autores Rivest, Shamir e Adleman. O método RSA baseia-se na dificuldade de se fatorar números muito grandes. Para usar o RSA, deve-se tomar dois números primos p e q, com centenas de bits de comprimento, e calcular n = p x q. Em seguida deve-se obter um número d, tal que d e (p-1)x(q-1) sejam primos entre si [UENO 2003]. Deve-se obter também um número e tal que e x d = 1 mod Z.

Uma vez escolhidos números que satisfaçam as condições apresentadas anteriormente, pode-se utilizar o par (e,n) como chave pública e o par (d,n) como chave privada (secreta). A codificação do texto normal P é realizada através da aplicação da operação:

$$C \leftarrow P^{e}(\text{mod } n)$$

A decodificação é executada aplicando-se a mesma operação utilizando como expoente o d:

$$P \leftarrow C^{d}\ (\text{mod } n)$$

O único modo conhecido de recuperar o valor de d conhecendo o valor de e envolve a fatoração de n. Fatorando n é possível encontrar p e q e, com base nesses valores, os demais. A segurança do método RSA apoia-se na enorme dificuldade de fatorar números muito grandes. Os métodos de criptografia assimétricos apresentam dois inconvenientes: o tamanho das chaves e a lentidão dos procedimentos de codificação e decodificação.

8.3.3. MD5 (Message Digest 5)

As funções Hashes são funções de compressão e quando estas são irreversíveis são chamadas de Secure Hashes ou algoritmos de *Message Digest* (MD). Um algoritmo de *Message Digest* comprime um texto em um bloco de tamanho fixo que normalmente tem 128 bits e passaremos a representar a função que gera o valor hash hm a partir da mensagem M por hm = H(M).

As propriedades básicas de uma função de Secure Hash são:

- Dado hm, é impossível determinar uma mensagem M tal que hm = H(M);
- Dada uma mensagem M, é impossível determinar uma mensagem M' tal que H(M) = H(M');
- É impossível determinar um par de mensagens (M,M') tal que H(M) = H(M').

A primeira propriedade determina que o valor hash representa a mensagem sem revelar seu conteúdo enquanto que a segunda e terceira propriedades determinam que o Secure Hash é um mecanismo apropriado para garantir sua integridade tendo em vista que o valor hash é único e ninguém produzirá outra mensagem que gere o mesmo valor hash.

O MD5 é uma função de *Message Digest* criada por Rivest em 1992 que é mais segura e com performance ligeiramente inferior em relação ao seu predecessor MD4 [Moitinho 2001]. O MD5 opera sua função sobre a mensagem dividindo-a em blocos de 512 bits e gera uma saída de 128 bits, executando o algoritmo quatro (4) vezes em cada bloco e usa a saída de cada etapa como entrada da próxima. O MD5 está representado na Figura 33:

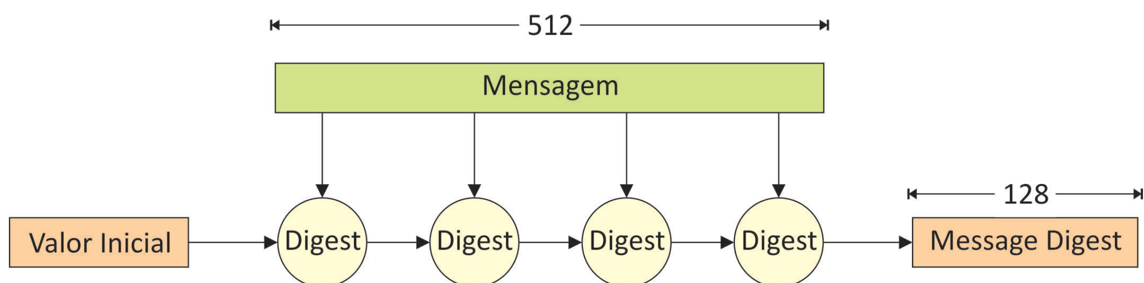


Figura 33: Message Digest [Moitinho 2001]

8.3.4. Autenticação

A autenticação é o mecanismo utilizado para se garantir que um usuário ou processo sob controle ou processos que trabalham de forma cooperativa estão se identificando de acordo com suas reais identidades.

Existem três formas básicas de autenticação que devem ser utilizadas de acordo com situação especifica. A autenticação pode ser feita de forma unilateral, mútua ou com a mediação de terceiros.

A autenticação unilateral acontece quando apenas um parceiro da comunicação autentica-se perante o outro, mas a recíproca não é verdadeira. Na autenticação mútua os parceiros da comunicação se autenticam um perante o outro [Moitinho 2001]. A autenticação com a mediação de terceiros é utilizada quando os parceiros da comunicação não se conhecem e

portanto não podem se autenticar mutuamente, mas conhecem um terceiro com quem se autenticam e recebem credenciais para procederem assim a autenticação mútua. Na Figura 34 estão os diagramas representando as três formas básicas de autenticação apresentadas.

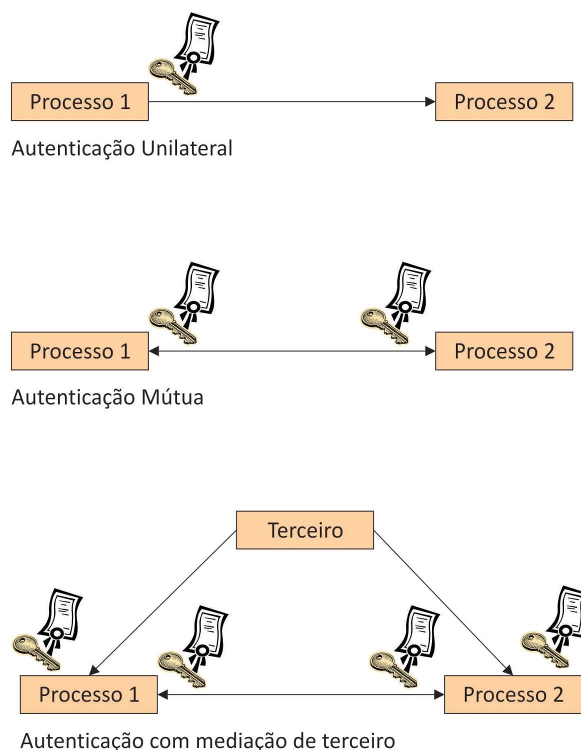


Figura 34: Formas básicas de autenticação [Moitinho 2001]

8.4. Assinatura digital

A assinatura digital é um processo que utiliza basicamente a criptografia assimétrica e a função *hashing*, e tem como principal propósito garantir o sigilo, integridade e autenticidade dos documentos envolvidos em transações eletrônicas.

As propriedades da assinatura digital são:

a) assinatura autêntica: quando o receptor utiliza a chave pública do emissor para decifrar um documento, ele confirma que o documento provem do emissor e somente do emissor;
b) assinatura não pode ser forjada: somente o receptor conhece sua chave secreta;
c) documento assinado não pode ser alterado: se houver alteração no texto criptografado, o mesmo não poderá ser restaurado com o uso da chave pública do receptor;
d) assinatura não reutilizável: assinatura é particular de cada documento e não poder ser transferida para outro documento;
e) assinatura não poder ser repudiada: a assinatura pode ser reconhecida por quem as recebe, verificando sua validade e caso seja válida, ela não pode ser negada pelo seu proprietário.

A assinatura digital é um mecanismo que envolve duas etapas, a assinatura da mensagem M propriamente dita $S_{Ks}(M)$ e a verificação da referida assinatura $V_{Kp}(M)$. Este mecanismo tem

por objetivo garantir a autenticidade e a integridade de uma mensagem de tal forma que ninguém possa alterá-la ou reutilizá-la como um todo ou mesmo parcialmente e que seu remetente possa posteriormente negar que a tenha enviado, ou seja, garantir a não-repudiação da mensagem [Moitinho 2001].

O mecanismo da assinatura digital baseia-se na criptografia assimétrica, onde o usuário que queira assinar uma mensagem usa sua chave privada para criptografar a mensagem e o receptor poderá verificar a autenticidade da assinatura descriptografando a mensagem com a chave pública do remetente e com isto ter a certeza da autenticidade e integridade da mensagem. Vejamos o exemplo de um usuário A que envia uma mensagem assinada M para um usuário B.

1. Usuário A assina a mensagem M e a envia para B: $S_{Ks}{}^{A}(M) = Ks^{A}(M)$
2. Usuário B recebe a mensagem $K_s{}^{A}(M)$ e verifica a assinatura de A:

   $V_{Kp}{}^{A}(M) = K_p{}^{A}(K_s{}^{A}(M)) = M$

É importante salientar que no exemplo anterior não há confidencialidade porque qualquer pessoa poderá descriptografar a mensagem com a chave pública do remetente, como já foi dito. Caso queira-se garantir a confidencialidade na comunicação, o remetente após assinar a mensagem deverá criptografa-la com a chave pública do destinatário e este ao recebe-la deve primeiramente descriptografar a referida mensagem com sua chave secreta e, então, verificar a autenticidade da assinatura descriptografando o resultado anterior com a chave pública do remetente e finalmente obtendo a mensagem original M.

Vejamos um exemplo onde o usuário A envia uma mensagem assinada M de forma confidencial para B.

1. Usuário A assina a mensagem M: $S_{Ks}{}^{A}(M) = Ks^{A}(M)$
2. Usuário A criptografa a mensagem assinada M e a envia para B:

   $$Kp^{B}(S_{Ks}{}^{A}(M)) = Kp^{B}(Ks^{A}(M)) = C$$

3. Usuário B recebe a mensagem C e a descriptografa com sua chave secreta:

   $$Ks^{B}(C) = Ks^{B}(Kp^{B}(Ks^{A}(M))) = Ks^{A}(M) = S_{Ks}{}^{A}(M)$$

4. Usuário B verifica a assinatura de A e obtém a mensagem M:

   $$V_{Kp}{}^{A}(M) = Kp^{A}(Ks^{A}(M)) = M$$

8.5. Circuitos de Criptografia

Os circuitos de criptografia podem ser implementados em qualquer uma das camadas da arquitetura de protocolos. Quando um circuito de criptografia é estabelecido nas camadas inferiores da arquitetura de protocolos, todas as mensagens que fluem através do enlace estabelecido entre dois hosts são criptografadas, incluindo-se os cabeçalhos das mensagens dos níveis superiores, e por isto são chamados de circuitos físicos ou *Physical Circuit*

Encryption. Os circuitos de criptografia podem ser implementados nas camadas superiores, fim-a-fim, com base no fluxo de dados e por isto são chamados de circuitos virtuais ou *Virtual Circuit Encryption* [Veríssimo 01].

Físico

Os circuitos físicos são normalmente implementados nas camadas Física ou de Enlace e podem ser implementadas por hardware ou software. Portanto, se o tráfego de mensagens passar por roteadores faz-se necessário descriptografar o pacote para que o mesmo possa ser roteado e criptografado novamente antes de ser encaminhado. As tarefas de criptografar e descriptografar podem ser realizadas por hosts dedicados a tal finalidade ou não. De qualquer forma, os circuitos físicos são pouco eficientes para tráfego de mensagens que passem por muitos roteadores devido ao número de operações de criptografia e descriptografia (Figura 35).

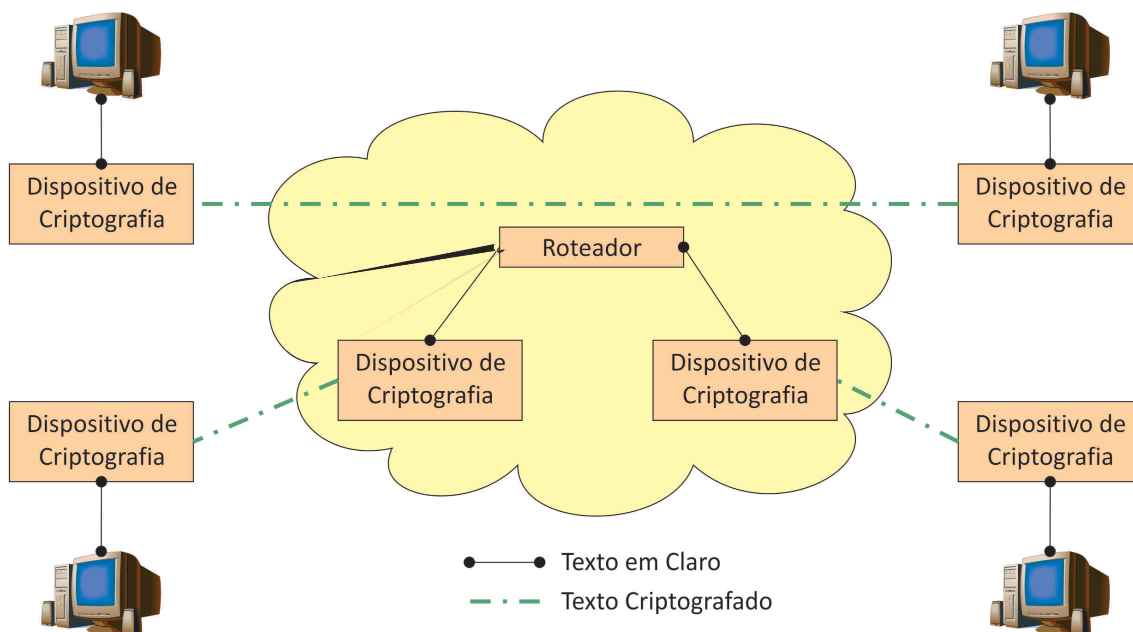


Figura 35: Circuitos Físicos [Moitinho 2001]

Virtual

Nos circuitos virtuais as mensagens são criptografadas normalmente na camada de Aplicação e são encapsuladas e descapsuladas pelos protocolos das camadas inferiores e podem ser multiplexadas em uma conexão de Transporte como qualquer outro tipo de mensagem. Os circuitos virtuais de criptografia têm sido largamente usados na Internet e especialmente para proteger transações financeiras.

8.6. Proteção de Dados e Serviços

A autenticação de usuários ou processos sob seu controle pode ser feita mutuamente ou através de um terceiro que seja confiável perante os participantes da comunicação. Os participantes da comunicação em Sistemas Distribuídos não necessariamente confiam um no outro e mesmo que exista esta confiança mútua ou em um terceiro. Sendo assim, devem fazer uso de mecanismos de autenticação que impeçam ataques do tipo personificação e *spoofing*. O uso de certificados de autenticidade fornecidos por entidades especializadas e oficiais

chamadas de *Certification Authority* (CA), provêm meios para que haja esta cooperação entre participantes da comunicação de forma confiável [Veríssimo 01].

Os Circuitos Virtuais de Criptografia protegem os dados contra praticamente todos os tipos de ataques, porém a criptografia assimétrica exige um grande esforço computacional e a criptografia simétrica apresenta um grande problema de gerenciamento de chaves. O problema de gerenciamento de chaves na criptografia simétrica pode ser minimizado através de uma entidade centralizada e confiável que irá gerar e distribuir chaves válidas para sessões de comunicação. Estas entidades são chamadas de *Key Distribution Center* (KDC) e veremos com detalhes adiante.

Para proteção dos dados e dos serviços devem ser empregadas técnicas que tornem o sistema tolerante a ataques e eventuais invasões. As ações devem ser focadas em três atividades básicas para tornarem os sistemas mais seguros:

- Prevenção a ataques;
- Detecção e reação a invasões;
- Tolerância a invasões.

As medidas de prevenção a ataques têm como objetivo evitar que ataques ao sistema sejam bem sucedidos e que ocorram invasões. A implementação da Política de Segurança e a escolha da Topologia da Rede de Computadores, com o uso de elementos ativos como switches, roteadores, filtros, firewalls e proxies, são essenciais para se reduzir o risco da organização ser invadida por hackers.

8.7. Gerenciamento de Chaves

O gerenciamento das chaves criptográficas é essencial para o uso eficaz das técnicas de criptografia. Qualquer exposição ou perda das chaves criptográficas pode levar ao comprometimento da confidencialidade, da autenticidade e/ou da integridade da informação. Convém que um sistema de gerenciamento de chaves seja implantado para garantir o uso pela organização dos dois tipos de técnicas criptográficas, que são:

- Técnicas de chave secreta, onde duas ou mais partes compartilham a mesma chave e esta chave é utilizada tanto para codificar, como para decodificar, a informação. Estas chaves necessitam ser mantidas em segredo uma vez que qualquer pessoa que tiver acesso à chave será capaz de decodificar toda informação codificada com aquela chave, ou introduzir informações não autorizadas;
- Técnicas de chave pública, onde cada usuário possui um par de chaves, uma chave pública (que pode ser revelada a qualquer pessoa) e uma chave privada (que tem que ser mantida em segredo). As técnicas de chave pública podem ser utilizadas para codificar e para produzir assinaturas digitais.

Convém que todas as chaves sejam protegidas contra modificação e destruição. As técnicas de criptografia podem ser utilizadas para este propósito. Convém que proteção física seja utilizada para a proteção de equipamentos usados na geração, armazenamento e arquivamento de chaves.

Convém também que um sistema de gerenciamento de chaves seja baseado em um acordo sobre um conjunto convencionado de padrões, procedimentos e métodos seguros para:

- Geração de chaves para diferentes sistemas criptográficos e diferentes aplicações;
- Geração e obtenção de certificados de chave pública;
- Distribuição de chaves para usuários predeterminados, inclusive com a informação de como devem as chaves ser ativadas quando recebidas;
- Armazenamento de chaves, inclusive com a informação de como os usuários autorizados podem obter acesso às chaves;
- Modificação ou atualização de chaves, incluindo regras sobre quando as chaves devem ser modificadas e como isto pode ser feito;
- Tratamento de chaves comprometidas;
- Revogação de chaves, incluindo como as chaves devem ser recolhidas ou desativadas, por exemplo, quando as chaves forem comprometidas ou quando um usuário deixar a organização (nestes casos as chaves também devem ser arquivadas);
- Recuperação de chaves que estão perdidas ou corrompidas como parte do gerenciamento da continuidade do negócio, por exemplo, para a recuperação de informações codificadas;
- Arquivamento de chaves, por exemplo, para informações arquivadas ou de reserva (“backup”);
- Destruição de chaves;
- Registros (“logs”) e auditoria das atividades relacionadas com o gerenciamento de chaves.

Para poder reduzir a probabilidade de comprometimento, convém que as chaves tenham data de ativação e desativação definidas, de forma que somente possam ser usadas por um período limitado de tempo e que este período de tempo dependa das circunstâncias sob as quais os controles de criptografia estão sendo utilizados.

Pode ser necessário considerar certos procedimentos para o tratamento de requisitos legais para acessar chaves criptográficas, por exemplo, as informações codificadas podem ser necessárias na sua forma não codificada como produção de prova em um julgamento de determinada ação judicial.

Adicionalmente à questão do gerenciamento seguro das chaves secretas e das privadas, convém que a proteção de chaves públicas também seja considerada. Existe a ameaça de alguém falsificar uma assinatura digital através da troca da chave pública do usuário por uma falsa. Este problema é solucionado com o uso de certificados de chave pública. Convém que estes certificados sejam produzidos de forma que relacione de um único modo as informações do proprietário do par de chaves pública / privada com a chave pública considerada. Além disto é importante que o processo de gerenciamento que gerou este certificado possa ser confiável. Este processo é normalmente implementado por uma autoridade certificadora que deve ser uma organização reconhecida e com controles e procedimentos implementados para fornecer o grau de confiabilidade necessário [Moitinho 2001].

Além disso é necessários que o conteúdo do acordo do nível de serviço ou contrato com fornecedores externos de serviços de criptografia, por exemplo, com uma autoridade certificadora, cubra questões relacionadas com a responsabilidade cível, a confiabilidade dos serviços e o tempo de resposta para o fornecimento dos serviços contratados.

Exercícios

As questões abaixo devem ser respondidas em forma dissertativa e argumentativa com pelo menos uma lauda. Devem também refletir a interpretação da leitura do texto juntamente com pesquisas sobre o tema arguido.

1. Qual a diferença entre falha, erro e defeito?
2. Quais as diferenças entre falhas por queda, omissão, temporal e Arbitrária?
3. O que é redundância?
4. Qual a diferença entre redundância por Software e por Hardware?
5. Explique o que é looging e checkpoint.
6. Diferencie criptografia simétrica e assimétrica.
7. Como funciona o esquema de Assinatura digital?
8. O que são circuitos de criptografia Físicos e Virtuais?

Resumo

A unidade IV focou em tópicos relacionados a segurança em sistemas distribuídos. Para que se possa ter um entendimento mais abrangente sobre esse aspecto, foram mostrados conceitos de tolerância e tipos de falhas que podem ocorrer em um SD. Ainda sobre este aspecto, mecanismos de redundância também foram mostrados de forma a evitar que falhas que por ventura ocorram no sistema possam parar seu funcionamento. Além disso, discorremos sobre mecanismos para recuperar falhas, casos elas aconteçam.

Por fim, foram apresentados aspectos de segurança no que tange a defesa dos sistemas contra ataques e acessos indevidos. Para isso, tópicos como criptografia e assinatura digital forma explanados.

Bibliografia

FILHO, Nélio Alves Pereira. **Serviços de Pertinência para clusters de alta disponibilidade**. Dissertação de Mestrado. São Paulo: Universidade de São Paulo, 2004.

FINKEL, V. S. et al. **Instrumentação Industrial**. 2ª. ed. Rio de Janeiro: Interciência, 2006.

JAFFE, M.S.; LEVESON, N.G.; HEIMDAHL, M.P.E.M.; BONNIE, E. **Software requirements analysis for real time process control systems**. , v.17, n.3, p.241-258, 1991.

LAPRIE, Jean-Claude. **Dependable computing and fault tolerance. Concepts and teminology**, *15th Internal Symposium on Fault Tolerant Computing Systems*, June 1985.

PHAM, H. **Handbook of Reliability Engineering**. 1ª. ed. New Jersey, USA: Springer, 2003.

PIAZZA, G. **Introdução à Engenharia da Confiabilidade**. 1ª. ed. Caxias do Sul, RS: EDUCS, 2000.

SOPPA, Eduardo L. **Sistema Tolerante a Falhas Utilizando Windows CE Focando Controladores Industriais**. Dissertação de Mestrado. UFPR, 2009

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems principles and paradigms**. 1. ed. Prentice Hall, 2002.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas Distribuídos: Princípios e Paradigmas**. 2. ed., Prentice-Hall, 2008.

Weblografia

FILIPPI, Adriano Joel – **Um estudo sobre mecanismos de tolerância a intrusão** – 2007
http://www.upf.br/computacao/images/stories/TCs/arquivos_20072/adriano_filippi.pdf
Acessado em: 15/11/2010

Mendonça, Fábio S. – **Tolerância a Falhas em Sistemas Distribuídos** – 2002
http://www.acso.uneb.br/marcosimoes/TrabalhosOrientados/mendonca2002.pdf
Acessado em: 07/08/2010

WEBER, Taisy Silva. **Tolerância a falhas: conceitos e exemplos**. 2005.
http://www.inf.ufrgs.br/~taisy/disciplinas/textos/ConceitosDependabilidade.PDF
Acessado em: 11/12/2010

SOUZA, Carlos R. K.; CAMPOS, Paulo C. G. – **Alta disponibilidade em bando de dados utilizando tecnologia Oracle -** 2008
http://www.kich.com.br/wp-content/uploads/Monografia-formatada.pdf
Acessado em: 11/12/2010

FLEMING, Erick – **Segurança em Sistemas Distribuídos** – 2005 http://www.daninfor.com/pdf/computacao/aplicacao_sistema_distribuidos/NP1-Seguranca-em-Sistemas-Distribuidos.pdf
Acessado em: 11/12/2010

UENO, Wagner Hiroshi – **Segurança em Redes** – 2003
http://www2.dc.uel.br/nourau/document/?down=240
Acessado em: 14/10/2010

MOITINHO, Stoessel Dourado – **Segurança em Sistemas Distribuídos**
http://wiki.dcc.ufba.br/pub/GESI/WebHome/SeguranaemSistemasDistribudos.pdf
Acessado em: 10/12/2010

Vinicius Ponte Machado

CV. http://lattes.cnpq.br/9385561556243194

Doutor em Engenharia Elétrica de Computação pela Universidade Federal do Rio Grande do Norte (2009) e Mestre em Informática Aplicada pela Universidade de Fortaleza (2003). Atualmente é professor adjunto da Universidade Federal do Piauí sendo professor permanente do Programa de Pós-Graduação em Ciência da Computação-UFPI. Tem experiência na área de Ciência da Computação, com ênfase em Gestão do Conhecimento e Inteligência Artificial, atuando principalmente nos seguintes temas: sistemas multiagente, redes neurais artificiais e Redes Industriais.