Capítulo 3

Serviços Distribuídos Tolerantes a Intrusões: resultados recentes e problemas abertos

Miguel Pupo Correia
LASIGE, Faculdade de Ciências da Universidade de Lisboa

Abstract

The idea of using dependability *concepts, mechanisms and architectures in the* security *domain is generating a lot of interest on both communities under the name of* intrusion tolerance. *Much of this attention has been created by the European MAFTIA project and the American OASIS program around 2000, although the notion comes from earlier. Albeit these projects have ended, much relevant work in the area has appeared in recent years, and now it is possible to know how to build* intrusion tolerant distributed services. *The objective is to have services with the properties of integrity, availability and confidentiality, even if some servers are attacked and controlled with success by hackers or malicious code. The chapter presents the state of the art in this area, clarifying the problems it solves and topics that remain open and have to be researched.*

Resumo

A idéia de aplicar conceitos, mecanismos e arquiteturas da área da confiança no funcionamento *no domínio da* segurança *tem gerado muito interesse em ambas as comunidades sob a designação de* tolerância a intrusões. *Muita da atenção foi criada pelo projeto europeu MAFTIA e pelo programa americano OASIS por volta do ano 2000, embora a noção venha de bem mais longe. Apesar desses projetos terem terminado, muito trabalho relevante tem surgido recentemente, sendo já possível ter idéias claras sobre como se podem concretizar* serviços distribuídos tolerantes a intrusões. *O objetivo consiste em garantir a integridade, disponibilidade e confidencialidade desses serviços mesmo que alguns servidores sejam atacados e controlados com sucesso por atacantes ou código nocivo. Este capítulo apresenta o estado da arte na área, clarificando os problemas que permite resolver e os tópicos que permanecem abertos e que precisam de ser pesquisados.*

3.1. Introdução

A *segurança* e a *confiança no funcionamento* (*security* e *dependability* em inglês) são duas áreas de pesquisa já com cerca de três décadas de existência. Quando se fala de *segurança*, está sempre presente a idéia de uma intenção maliciosa por parte de alguém, quer atuando diretamente, quer através da criação e distribuição de código nocivo (virus, vermes, etc.). O objetivo da *segurança* consiste em evitar que essa vontade de fazer mal prejudique dois tipos de bens: informação e serviços. Mais precisamente, as principais propriedades que a *segurança* pretende garantir são a confidencialidade, integridade e disponibilidade da informação e de serviços computacionais. Na *confiança no funcionamento*, que inclui a *tolerância a faltas*, a idéia subjacente é a de que o sistema se comporte de acordo com a sua especificação, perante os inúmeros problemas que podem ocorrer na prática: desde catástrofes naturais a *bugs* no software ...

Apesar de terem seguido caminhos diferentes ao longo destas três décadas, as duas áreas têm muito em comum. Na realidade, o objetivo das duas áreas é o mesmo: garantir que os sistemas computacionais funcionam corretamente. A ênfase da *segurança* tem sido em problemas de origem maliciosa (ataques, código nocivo), enquanto que o foco da *confiança no funcionamento* tem sido nos problemas de origem acidental. No entanto, as disciplinas não se excluem, pois a *segurança* pode tratar problemas de origem acidental e a *confiança no funcionamento* pode incluir problemas de origem maliciosa.

A *tolerância a intrusões (TI)* surge precisamente do encontro dessas duas áreas. Resumidamente, a idéia consiste em *aplicar o paradigma da tolerância a faltas no domínio de segurança*. Uma pequena estória pode ajudar a compreender em que consiste a TI:

> *O pirata pegou a luneta e observou o torreão da fortaleza. Era preciso conquistá-lo para chegar ao famoso tesouro do Rei daquele país, que tantos antes dele tinham cobiçado! O olho de vidro brilhou ... se não de felicidade pelo menos com o reflexo do sol.*
>
> *Os obstáculos para chegar ao torreão eram difíceis, mas não intransponíveis. Primeiro, teria de remar centenas de metros em plena noite tentando não ser vislumbrado. Depois iria acostar aos rochedos afiados contra os quais o mar se esmagava violentamente.*
>
> *Para descansar das manobras náuticas, um agradável exercício de escalada de 50 metros de escarpa rochosa, completado com a passagem da primeira muralha e dos seus guardas. Um fosso povoado por jacarés iria ser um bom momento para refrescar as idéias ... e a subida da parede da fortaleza iria dar-lhe tempo para secar a roupa.*
>
> *A tarefa não era simples, mas uma vez no torreão ...*

Se nesta pequena estória substituirmos "pirata" por "atacante", "fortaleza" por "sistema" e "tesouro" por informação ou serviço, percebemos que há um paralelo evidente com a *segurança* de sistemas computacionais. A abordagem clássica em *segurança* consiste em usar cuidadosamente todos os obstáculos da estória para dificultar a vida do pi-

rata: mar e rochedos, escarpa, muralha, guardas, jacarés ... que podem ser *firewalls*, controle de acesso, mecanismos biométricos, criptografia, redes privadas virtuais, etc. Passar por todos esses mecanismos é difícil, mas todos os dias muitos piratas em todo o mundo gritam de alegria quando chegam ao seu tesouro [CERT/CC, 2005, Turner et al., 2004].

A área da *confiança no funcionamento* tem uma abordagem algo diferente desta da *segurança*. Por exemplo, não basta usar as melhores técnicas de engenharia para que o computador do Airbus A380 não pare; é preciso ter vários computadores a bordo para *tolerar* esses eventos. A *tolerância a intrusões* pega nesta mesma idéia: não basta que o pirata tenha que ultrapassar obstáculos difíceis, embora essa dificuldade – todos os mecanismos clássicos da *segurança* – seja essencial. O que seria desejável é que o pirata tivesse que penetrar em vários torreões diferentes, em fortalezas diferentes, para conseguir pegar o tesouro!

O conceito de TI foi introduzido há já duas décadas por Fraga e Powell [Fraga and Powell, 1985][1]. No entanto, a TI começou a gerar maior interesse só mais recentemente, em parte devido ao projeto europeu MAFTIA[2] e ao programa americano OASIS [Lala, 2003][3], ambos iniciados por volta do ano 2000[4].

Apesar destes projetos terem terminado, muitos trabalhos relevantes têm surgido recentemente, sendo já possível ter idéias claras sobre como se podem concretizar *serviços distribuídos tolerantes a intrusões*. O objetivo consiste em garantir a integridade, disponibilidade e confidencialidade de serviços constituídos por diversos servidores ligados através de uma rede, mesmo que alguns desses servidores sejam atacados e controlados com sucesso por atacantes (*hackers*, *crackers*) ou por código nocivo (virus, vermes, etc.). Alguns exemplos de serviços que podem assim se tornar seguros são PKIs, sistemas de arquivos distribuídos, comércio eletrônico ou serviços de autorização. A TI não se reduz aos serviços distribuídos, mas essa é provavelmente a área onde os trabalhos mais relevantes têm sido realizados.

O objetivo deste capítulo consiste em apresentar o estado da arte na área, clarificando os problemas que esta permite resolver e os tópicos que permanecem abertos e que precisam de ser pesquisados.

O capítulo está organizado da seguinte forma. A seção 3.2 apresenta os principais conceitos de TI e a sua relação com a *segurança*. A seção 3.3 apresenta as soluções e algoritmos para concretizar serviços distribuídos TI com replicação. Usando estas técnicas – que incluem a replicação de máquinas de estados e os quoruns – consegue-se aumentar a disponibilidade e a integridade desses serviços. A seção 3.4 introduz as abordagens que permitem garantir também a confidencialidade dos dados usando fragmentação. A seção 3.5 explica algumas técnicas que permitem processar as intrusões dos servidores, aumentando assim o número de intrusões que é possível tolerar. A seção 3.6 passa dos algoritmos para a arquitetura de serviços distribuídos TI, introduzindo diversas arquiteturas e sistemas propostos na bibliografia. Finalmente, a seção 3.7 apresenta os principais

[1] Logo podemos dizer que tem sangue brasileiro ...

[2] http://www.maftia.org

[3] http://www.tolerantsystems.org

[4] O termo *survivability* tem também sido usado para apelidar alguns trabalhos na área, sobretudo quando a origem é americana.

problemas abertos da área, procurando dar pistas sobre a pesquisa que é preciso fazer. A seção 3.8 apresenta algumas conclusões.

3.2. Conceitos básicos de Tolerância a Intrusões

Esta seção explica as bases da TI, começando por uma introdução à *confiança no funcionamento* [Avizienis et al., 2004, Veríssimo and Rodrigues, 2001]. Em relação a esta área seguiremos a terminologia para português de Veríssimo e de Lemos, na variação brasileira quando existirem duas versões de um termo [Veríssimo and de Lemos, 1989].

3.2.1. Confiança no funcionamento

Um *sistema* é uma entidade que interage com outros sistemas – computacionais, mecânicos, físicos, seres humanos – através da sua *fronteira*. Tudo o que está fora do sistema constitui o seu *ambiente*. Um sistema computacional contém diversos *componentes* e é caracterizado pelo que faz – a sua *funcionalidade* – e por um conjunto de propriedades *não funcionais*, como o seu desempenho, a sua *segurança*, a sua confiabilidade, etc. Um sistema fornece um determinado *serviço*, através de uma *interface*, a um *utilizador*, e tem um *estado* que muda com o passar do tempo.

Um sistema fornece um serviço *correto* se este obedece à especificação do sistema. Caso contrário existe uma *falha* do serviço. O objetivo da *confiança no funcionamento* consiste em fazer com que o serviço permaneça correto, ou seja, que não falhe. Para que isso seja possível é preciso entender o processo que leva à falha.

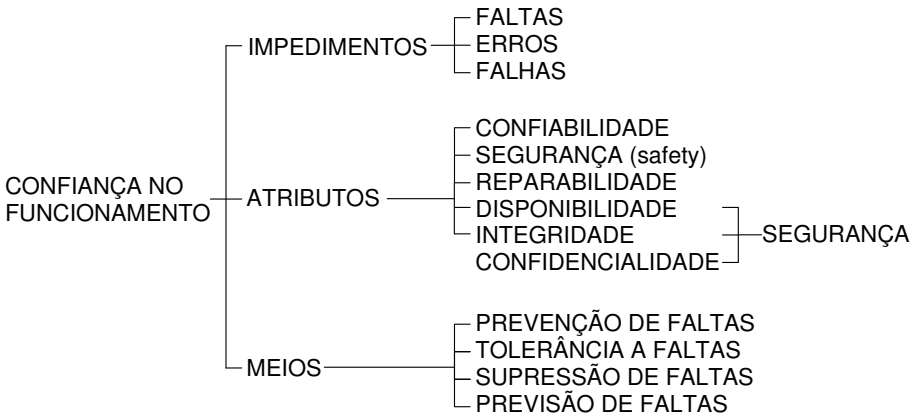

Figura 3.1. Conceitos de *confiança no funcionamento* [Avizienis et al., 2004].

Os impedimentos à *confiança no funcionamento* assumem três facetas (v. fig. 3.1): falta, erro e falha (já referida). Uma *falta* é a causa remota de uma falha. Uma falta pode ser interna (p. ex. um defeito na memória RAM) ou externa (p. ex. um operador que tropeça e desliga um cabo). Um *erro* é a consequência de uma falta no estado do sistema (p. ex. um registo corrompido por ter sido lido da memória RAM defeituosa). Uma falta pode ficar *dormente*, pode não gerar imediatamente um erro (quando gera diz-se *ativa*); um erro pode ou não gerar a falha do sistema (p. ex. se o registo não for lido o sistema não falha devido a esse erro). Uma falta interna corresponde à falha de um componente do sistema. Se olharmos para esse componente como um sistema, também a sua falha pode

ser causado pela falha de um dos seus componentes, podendo existir uma sequência que conduza à falha do sistema:

falta → erro → falha → falta → erro → falha ...

A *confiança no funcionamento* pretende garantir um conjunto de *atributos*: *confiabilidade* (continuidade do serviço correto); *segurança (safety)* (ausência de consequências catastróficas sobre os utilizadores); *reparabilidade* (capacidade de receber modificações e reparações); *disponibilidade* (prontidão do serviço correto); *integridade* (ausência de alterações inadequadas ao sistema). É evidente que os dois últimos atributos coincidem com propriedades básicas da *segurança*. Uma propriedade de *segurança* geralmente não considerada no âmbito da *confiança no funcionamento* é a *confidencialidade*, a ausência de revelação inadequada de informação.

Uma questão especialmente importante é a dos *meios* para procurar garantir a *confiança no funcionamento*. Os meios são muitos, fruto de muitos anos de pesquisa, mas podem ser agrupados em quatro categorias:

- *Prevenção de faltas* - meios para prevenir a introdução de faltas. Faz parte do processo normal da engenharia de sistemas, tanto da engenharia de software como da de hardware.
- *Tolerância a faltas* - meios para evitar a falha do serviço quando ocorrem faltas. Os meios desta categoria podem ser divididos em duas sub-categorias:
  - *mascaramento de faltas* - usar redundância para garantir que as faltas não causem a falha do sistema;
  - *detecção e processamento* - detectar a ocorrência de erros e processá-los de forma a os neutralizar.
- *Supressão de faltas* - meios usados durante o projeto do sistema para reduzir o número e/ou a severidade das faltas. Estes meios incluem diversas técnicas de verificação e validação de sistemas, tanto de hardware como de software.
- *Previsão de faltas* - meios para estimar o número de faltas no sistema, e prever o número e consequências de faltas futuras. As principais técnicas podem ser divididas em modelagem e teste.

A *confiança no funcionamento* é consequência da combinação eficaz destes meios. Os mecanismos clássicos de *segurança*, como o controle de acesso e a autenticação, poderiam ser englobados nos meios para *prevenção de faltas*. Já a TI, tenta usar os mecanismos e conceitos da *tolerância a faltas* no domínio da *segurança*. A *supressão de faltas* e a *previsão de faltas* são ortogonais aos outros meios e não vão ser mais considerados.

3.2.2. Tolerância a intrusões

Depois do que foi dito sobre *confiança no funcionamento* conclui-se facilmente em que consiste aplicar o paradigma da *tolerância a faltas* no domínio da *segurança* – a *tolerância a intrusões* [Veríssimo et al., 2003, Adelsbach et al., 2002]:

- assumir e aceitar que o sistema permanece sempre mais ou menos vulnerável;
- assumir e aceitar que os componentes do sistema podem ser atacados e que alguns desses ataques terão sucesso;
- garantir que o sistema como um todo permanece seguro e operacional, ou seja, que não falha.

O nome *tolerância a intrusões* dá a entender que *intrusões* são faltas. Na realidade, não só as intrusões mas também as vulnerabilidades e os ataques são faltas. Uma *vulnerabilidade* é uma falta de projeto ou de configuração, geralmente acidental (i.e., não intencional), que pode ser explorada com fins maliciosos. Um *ataque* é uma falta intencional, maliciosa, que visa explorar uma ou mais vulnerabilidades. Uma *intrusão* é o resultado de um ataque que tem sucesso em explorar uma ou mais vulnerabilidades.

Como já foi dito, no domínio da *confiança no funcionamento* geralmente são consideradas apenas faltas acidentais. Muitas vezes são tratadas apenas as mais simples entre essas faltas, as faltas de parada (p. ex. o *crash* de processos ou máquinas). No domínio da *segurança* geralmente é irrealista, logo perigoso, levantar hipóteses sobre o modo como um componente falha. Por isso, as faltas maliciosas são geralmente consideradas como podendo ser de qualquer tipo, logo sendo englobadas na categoria de faltas mais geral: as *faltas arbitrárias*, também denominadas de *faltas bizantinas*[5]. Em *tolerância a intrusões* os termos *intrusão* e *falta bizantina* são usados geralmente como sinônimos.

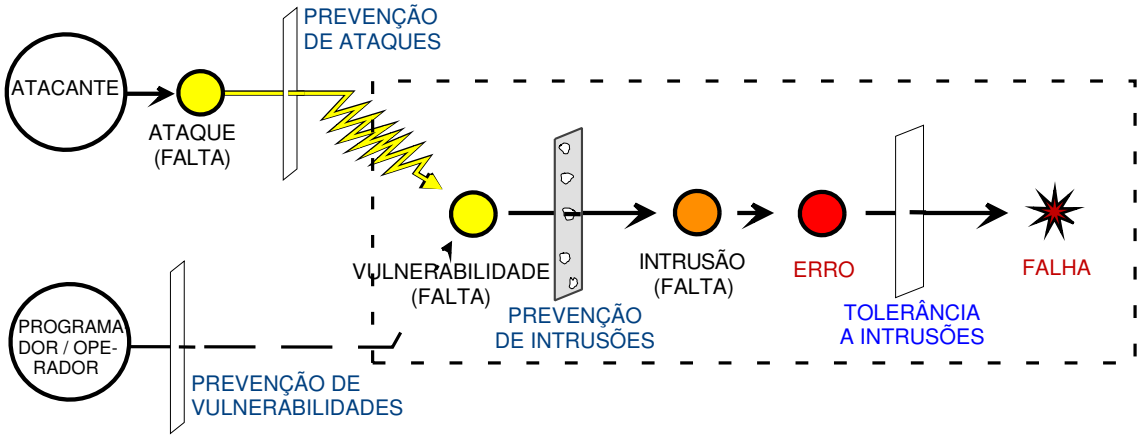

Figura 3.2. O modelo AVI e os mecanismos para evitar a falha [Veríssimo et al., 2003].

Esta relação entre as noções de ataque-vulnerabilidade-intrusão e falta não deve ser descartada como uma questão de nomenclatura. Pelo contrário, serve para entender o processo de falha de um sistema e, consequentemente, os mecanismos que se podem usar para evitar que isso aconteça. Essa relação é ilustrada pela figura 3.2, que é auto-explicativa.

Em *segurança*, a noção de *trustworthiness* diz a que ponto um componente ou sistema satisfaz um conjunto de propriedades (de segurança). Se generalizarmos esta

[5] A denominação *faltas bizantinas* vem de um artigo clássico que apresenta um protocolo tolerante a faltas maliciosas através de um problema envolvendo generais bizantinos [Lamport et al., 1982].

noção para incluir os outros *atributos* da figura 3.1, então *trustworthiness* torna-se um sinônimo de *confiança no funcionamento* (*dependability*). Outra noção relacionada é a de confiança (*trust*): a dependência de um componente em relação às propriedades de segurança de outro componente. Também esta noção pode ser extendida para incluir as outras propriedades de *confiança no funcionamento*.

Na seção anterior falámos dos *meios* para obter *confiança no funcionamento*. O projeto de *serviços distribuídos tolerantes a intrusões* baseia-se numa conjugação dos quatro meios. Em relação à *tolerância a faltas*, a maior parte das soluções que veremos no capítulo são baseadas em *mascaramento de faltas*. Os serviços baseados nesses tipo de mecanismos *mascaram*, ou escondem, a existência de faltas. Para isso, usam-se não um mas vários servidores – redundância – em conjunto com protocolos de comunicação que permitam fazer esse mascaramento (v. fig. 3.3). Esses *protocolos* (ou *algoritmos distribuídos*) têm também de ser tolerantes a intrusões. Os serviços TI podem também usar o outro tipo de mecanismos de *tolerância a faltas*, o *processamento de erros*, para remover as intrusões que ocorram e assim evitar que o sistema falhe.

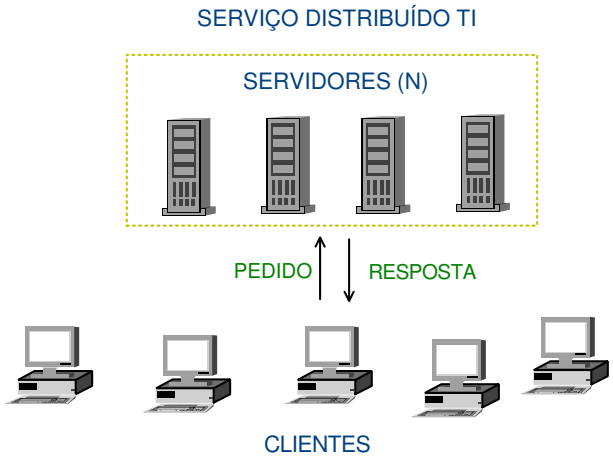

Figura 3.3. Arquitetura genérica de um sistema com um *serviço distribuído tolerante a intrusões*.

3.2.3. Sistemas distribuídos

Como o tema do capítulo é a TI em *sistemas distribuídos*, é importante fazer uma breve introdução a este tema. A definição clássica de Lamport diz que *um sistema distribuído é aquele que não o deixa trabalhar por causa da falha de um computador do qual nunca ouviu falar*. Considerando essa definição, própria de um investigador em *tolerância a faltas*, é importante apontar que o comportamento dos sistemas distribuídos é complexo, logo para raciocinar sobre esses sistemas são usados *modelos*, como em qualquer outro ramo da ciência.

O *modelo topológico* diz como as máquinas são interligadas por uma rede. Todos os trabalhos que vamos abordar usam um modelo topológico simples com conectividade total: todas as máquinas têm uma canal de comunicação com todas as outras[6].

[6]Na realidade algumas arquiteturas mais complexas que veremos na seção 3.6 dividem a rede em segmentos independentes e filtram o tráfego que passa entre eles. No entanto os algoritmos distribuídos são

O *modelo de falhas* define hipóteses sobre como podem falhar os componentes do sistema. Nos trabalhos que vamos analisar geralmente consideram-se faltas bizantinas, como já apontado, embora por vezes sejam usados determinados mecanismos para excluir *a priori* certos tipos de faltas (p. ex. usando canais SSL/TLS podem-se excluir as faltas na rede, exceto a quebra total de comunicação). Os modelos híbridos consideram diferentes hipóteses de faltas sobre diferentes partes do sistema, assumindo por exemplo a existência de certos componentes simples seguros [Veríssimo et al., 2000, Correia et al., 2002a].

O *modelo temporal* consiste num conjunto de hipóteses sobre o comportamento do sistema em termos de tempo. Os trabalhos que vamos estudar geralmente consideram o *modelo assíncrono*, que não assume qualquer hipótese sobre os tempos de processamento e de comunicação no sistema[7]. Este modelo é escolhido por prudência, digamos assim, pois muitas hipóteses temporais podem ser quebradas através de certos ataques (p. ex. uma hipótese sobre o atraso de comunicação pode ser quebrada através de um ataque de negação de serviço). No entanto, esta questão é algo mais complicada. Um problema importante em sistemas distribuídos é chamado de *consenso*. O problema pode ser formulado informalmente da seguinte forma: dado um conjunto de *processos*[8] cada um com um valor inicial; como fazer com que todos os processos corretos (ou seja, que não falhem) cheguem a acordo sobre um único valor? O problema parece simples, mas na realidade foi provado que não tem solução determinista num sistema assíncrono nem que apenas um processo possa falhar por parada [Fischer et al., 1985]. Esse resultado (FLP) tem consequências em inúmeros problemas de sistemas distribuídos que são equivalentes ao consenso, por exemplo, a entrega de mensagens com ordem total ou a comunicação em grupo com sincronia de vistas. Para contornar este resultado é preciso usar protocolos aleatórios [Ben-Or, 1983, Rabin, 1983], detectores de falhas [Chandra and Toueg, 1996] ou outras técnicas. Algumas destas técnicas escondem hipóteses temporais que podem introduzir vulnerabilidades.

3.2.4. Criptografia de limiar

O termo *criptografia de limiar* denomina um conjunto de algoritmos tipicamente de tolerância a faltas/intrusões mas que surgiram no âmbito da *segurança*[9]. Esses algoritmos por vezes constituem componentes importantes dos trabalhos que vamos apresentar mais à frente, logo é importante introduzi-los desde já. A bibliografia sobre o tema é extensa, mas um bom resumo encontra-se em [Gemmell, 1997].

A criptografia de limiar assume duas formas básicas. Sejam dados N processos, cada um detendo uma determinada *parte* secreta. O objetivo de um algoritmo de *partilha*

---

executados separadamente em cada segmento, logo o modelo de conectividade total continua a aplicar-se.

[7] No outro extremo do espetro dos modelos temporais está o *modelo síncrono*, que assume limites de tempo de processamento e comunicação. Entre os dois extremos há diversos modelos intermédios, geralmente chamados de modelos de sincronia parcial [Dwork et al., 1988].

[8] Ao longo de todo o capítulo vamos usar o termo *processo* para significar uma entidade que participa num algoritmo ou protocolo. Alguns termos usados com o mesmo significado são: processador, participante, parte, jogador. Um processo no qual tenha ocorrido uma intrusão e se desvie do comportamento especificado diz-se *malicioso*. Caso contrário diz-se *correto*.

[9] Nesta seção, como em todo o capítulo, vamos assumir que o adversário ou atacante – a entidade que procura quebrar o funcionamento de um serviço ou protocolo – é limitado computacionalmente, ou seja, que não consegue quebrar as primitivas criptográficas usadas.

de segredos é permitir que k processos combinem as suas partes e revelem determinado segredo s, garantindo simultaneamente que um conluio de até $k-1$ processos maliciosos não consegue fazer outro tanto, nem sequer obter qualquer informação relevante sobre s. Um algoritmo de *partilha de funções* permite que k processos apliquem determinada função F, não sendo possível até $k-1$ processos fazerem o mesmo. Um tipo especialmente importante de algoritmos deste último tipo são os algoritmos de *assinatura de limiar*, que permitem a um conjunto de processos criar uma assinatura criptográfica sem revelar a chave privada. No entanto, convém notar que um algoritmo de criptografia de limiar pode ser substituído por vetores com assinaturas, p. ex. RSA, uma por cada processo, inclusive com melhor desempenho [Cachin, 2002].

O algoritmo original de partilha de segredos de Shamir pode ajudar a entender como funciona a criptografia de limiar [Shamir, 1979] (este resultado foi desenvolvido ao mesmo tempo que outro semelhante de Blakley [Blakley, 1979]). O esquema é baseado em duas propriedades dos polinômios:

- dados quaisquer $d+1$ pontos distintos da curva definida pelo polinômio, é possível determinar qualquer outro ponto do polinômio;
- se os índices a_i do polinômio forem todos desconhecidos, o conhecimento de até d pontos da curva não revela nenhuma informação sobre outros pontos.

Seja dado um *polinômio* de grau d:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_d x^d \tag{1}$$

O algoritmo de Shamir considera a existência de um processo que pretende guardar o segredo s. Esse processo define um polinômio de grau $d = k-1$ de índices a_i aleatórios, exceto $a_0 = p(0) = s$. Depois, calcula $p(x)$ para N valores diferentes e aleatórios de x e distribui cada uma dessas *partes* por um dos N processos. Atendendo às duas propriedades dos polinômios acima, o segredo pode ser reconstituído por k processos mas não por $k-1$. A reconstituição é feita usando a interpolação de Lagrange:

$$p(0) = \sum_{i=1}^{k} (p(x_i) \prod_{j \neq i} \frac{0 - x_j}{x_i - x_j}) \tag{2}$$

Esta é a idéia básica do funcionamento do algoritmo de Shamir. A única simplificação feita é a de que na prática não se podem usar valores arbitrariamente grandes, logo o algoritmo considera um número primo grande e faz todos os cálculos *módulo* esse número (ou seja, usa sempre o resto da divisão do resultado por esse número).

O algoritmo de Shamir tem duas limitações. A primeira é a de que um processo não tem como saber se a sua parte é "boa", ou seja, se combinada com outras $k-1$ partes boas reconstrói o segredo. Essa lacuna foi preenchida mais tarde por algoritmos de *partilha de segredos verificável*. A segunda limitação é que se uma das partes usadas para reconstituir o segredo estiver corrompida, p. ex. por ser fornecida por um processo

malicioso, não é reconstituído o verdadeiro segredo e pode nem sequer ser possível compreender que este está errado. Para resolver este problema foram desenvolvidos algoritmos *robustos*, baseados em *provas de conhecimento zero*. Existem ainda algoritmos de criptografia de limiar *proativos* mas isso é um tema para a seção 3.5.

3.3. Replicação: garantindo disponibilidade e integridade

A idéia básica da replicação consiste em distribuir cópias do código e dos dados de determinado serviço por um conjunto de servidores. A replicação tem sido amplamente usada em *tolerância a faltas* para garantir a disponibilidade e a confiabilidade de serviços distribuídos. Muitos dos trabalhos em *serviços distribuídos TI* são também baseado em replicação. Este tipo de soluções permite garantir a *disponibilidade* e a *integridade* do serviço se houver intrusões num número limitado de réplicas, geralmente menos de um terço.

Voltando à nossa estória do pirata, a idéia consiste em ter diversos torreões. O tesouro não é propriamente monetário (ouro, jóias ...) mas um serviço fornecido pelo sistema de fortalezas, digamos um serviço de vigia da entrada no único porto daquele país. O objetivo do pirata é interromper o serviço de informações, ou fazê-lo dar informação errada de modo a que os navios embandeirados pelo crânio e as tíbias possam entrar no porto. Para isso terá de invadir *vários torreões*, caso contrário o serviço fornecido pelo sistema de fortalezas permanecerá íntegro e disponível.

Os principais trabalhos nesta área podem ser classificados como os que fazem replicação de máquinas de estados [Lamport, 1978, Schneider, 1990] e os que usam quoruns bizantinos [Malkhi and Reiter, 1998a].

3.3.1. Replicação de máquinas de estados

A *replicação de máquinas de estados (RME)* é uma solução genérica para a concretização de *serviços* tolerantes a faltas [Schneider, 1990]. Um serviço oferece um conjunto de *operações* aos seus *clientes*, que os invocam através de *pedidos*. Um serviço é concretizado através de um conjunto de *N servidores* $s_i \in U$ (também chamados de *réplicas* neste contexto)[10]. A figura 3.3 ilustra estes conceitos.

Cada servidor é uma *máquina de estados*, definida por *variáveis de estado* que definem o seu estado, e por *comandos* que modificam esse estado. Os comandos têm de ser atômicos, ou seja, não podem interferir uns com os outros. Todos os servidores seguem a mesma sequência de estados, para o que é suficiente satisfazer quatro propriedades:

- *Estado inicial.* Todos os servidores começam no mesmo estado.
- *Acordo.* Todos os servidores executam os mesmos comandos.
- *Ordem total.* Todos os servidores executam os comandos pela mesma ordem.
- *Determinismo.* O mesmo comando executado no mesmo estado inicial gera o mesmo estado final.

[10] A nomenclatura usada nos diversos trabalhos varia bastante. Neste capítulo usaremos uma nomenclatura coerente.

A primeira propriedade é geralmente simples de garantir. A segunda e a terceira podem ser forçadas usando um *protocolo de difusão atômica* (ou de difusão com ordem total). Quanto à quarta propriedade, vamos considerá-a uma premissa por ora mas voltaremos a ela na seção 3.7.

Um parâmetro importante quando de fala de um serviço tolerante a faltas/intrusões é a *resistência* (*resilience*), o número máximo de servidores que podem falhar para o serviço se manter correto. Em sistemas assíncronos TI baseados em replicação de máquinas de estado este limite é imposto pelo protocolo de difusão atômica, cuja resistência máxima é de $f = \lfloor \frac{N-1}{3} \rfloor$ em N servidores [Bracha and Toueg, 1985, Hadzilacos and Toueg, 1994, Cachin et al., 2001]. Uma forma mais clara de dizer o mesmo, que por isso é a que vamos usar ao longo do texto, é a de que são necessários (pelo menos) $3f+1$ servidores para tolerar f servidores que falham. O problema da difusão atômica é, como já foi referido, equivalente ao do consenso [Cachin et al., 2001]. Diversas soluções têm sido apresentadas na bibliografia para resolver consenso tolerante a faltas bizantinas [Ben-Or, 1983, Rabin, 1983, Bracha and Toueg, 1985, Malkhi and Reiter, 1997b, Doudou and Schiper, 1997, Cachin et al., 2000, Baldoni et al., 2000, Doudou et al., 2002, Kihlstrom et al., 2003, Correia et al., 2005a, Neves et al., 2005], como aliás também para difusão atômica [Reiter, 1994, Moser and Melliar-Smith, 1999, Kihlstrom et al., 2001, Castro and Liskov, 2002, Cachin and Poritz, 2002].

3.3.1.1. BFT

Um algoritmo ideal para aprofundar o tema da RME tolerante a intrusões é o BFT (*Byzantine Fault Tolerance*), um dos mais citados trabalhos de TI [Castro and Liskov, 2002]. Este algoritmo tem a resistência máxima possível em sistemas assíncronos: $N \geq 3f+1$. Na discussão que se segue vamos considerar o caso mais restrito: $N = 3f+1$[11].

O grande objetivo visado no BFT foi ter um protocolo correto e com bom desempenho. Em relação à correção, o BFT satisfaz sempre as suas propriedades de segurança mas o progresso do algoritmo, nomeadamente quando há mudança de vistas, depende de uma hipótese temporal fraca: o atraso na rede não cresce exponencialmente. Quanto ao desempenho, um sistema de arquivos distribuído NFS TI baseado na biblioteca BFT teve um desempenho entre 2% melhor e 24% pior do que soluções não seguras nem replicadas.

A principal opção que tornou o bom desempenho possível foi evitar o uso de criptografia de chave pública [Diffie and Hellman, 1976, Rivest et al., 1978] durante o funcionamento normal do sistema, ou seja, quando não há intrusões. As mensagens são assinadas usando *message authentication codes*, MACs [Menezes et al., 1997]. Cada par cliente-servidor partilha uma chave secreta. Cada mensagem ponto-a-ponto leva um MAC calculado com a chave partilhada pelo emissor/destinatário e cada mensagem por difusão leva um vetor com um MAC calculado com a chave compartilhada pelo emissor e cada destinatário. Com os servidores passa-se o mesmo, mas cada par de servidores partilha duas chaves secretas, uma para a comunicação em cada direção.

[11] Este é o caso mais restrito pois podia-se também ter $N = 3f+2$, $N = 3f+3$, etc.

Quando um cliente pretende enviar um pedido ao serviço, difunde uma mensagem com o comando, uma estampilha temporal (*timestamp*) e um vetor de MACs para todos os servidores. A estampilha serve para garantir que cada pedido de um cliente é executado precisamente uma vez. Cada servidor processa a mensagem (aliás, qualquer mensagem) apenas se o MAC que lhe corresponde estiver correto. Também um cliente só processa mensagens com MAC correto. O cliente aceita a resposta ao seu pedido quando recebe $f+1$ cópias vindas de diferentes servidores, o que garante que pelo menos uma das cópias vem de um servidor correto (parte-se da hipótese de que no máximo f sofrem intrusões). Se a resposta não chega, o cliente retransmite o pedido.

O algoritmo é baseado numa mistura de *replicação passiva* (ou *primário-secundário*) e *replicação ativa*. As réplicas vão mudando de configuração, sendo cada uma das configurações denominada uma *vista*. Em cada vista um servidor é o *primário* e os restantes são os *secundários*. A ordem de execução dos pedidos é definida pelo primário, atribuindo-lhe o próximo número de sequência a cada pedido recebido e reenviando-o para os secundários. Se o primário for malicioso, pode dar o mesmo número a dois pedidos, parar de atribuir números, ou deixar intervalos entre os números. Por isso, os secundários verificam os números de sequência atribuídos pelo primário e marcam o tempo para ver se ele pára. Quando os secundários suspeitam que o primário falhou, mudam de vista, logo de primário.

O protocolo de difusão atômica em operação normal tem três fases: *pre-prepare*, *prepare* e *commit* (v. fig. 3.4). As duas primeiras servem para ordenar pedidos enviados numa mesma vista, mesmo que o primário seja malicioso. A terceira fase garante a ordenação dos pedidos entre vistas diferentes.

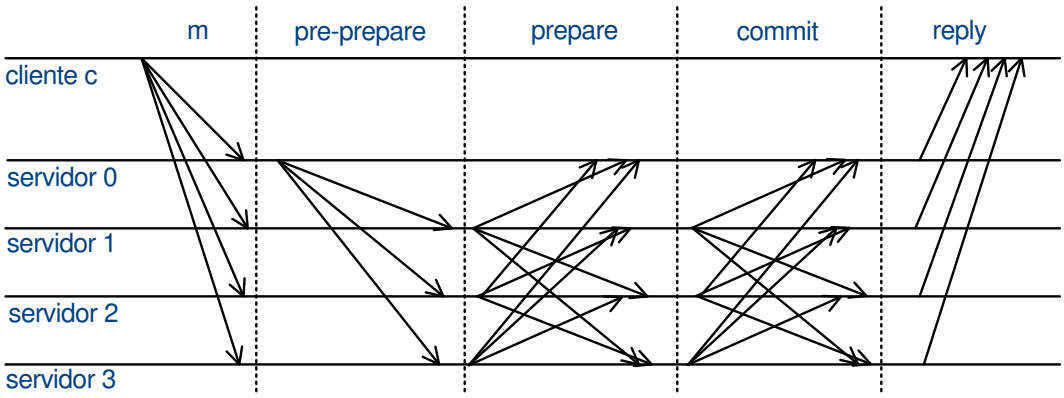

Figura 3.4. Algoritmo BFT em funcionamento normal [Castro and Liskov, 2002].

Na fase *pre-prepare* o primário (*servidor* 0 na figura) difunde o pedido recebido, com um número de sequência n e o número da vista v, para todos os secundários. Um secundário aceita esta mensagem se estiver na vista v e não tiver aceitado outra mensagem *pre-prepare* com os mesmos v e n. Caso aceite a mensagem e tenha recebido esse mesmo pedido do cliente, difunde uma mensagem *prepare* com um resumo criptográfico (*hash* [Menezes et al., 1997]) da mensagem para todas as réplicas (caso contrário não faz nada). Desta forma a réplica aceita atribuir o número de sequência n a esse pedido. Quando um servidor receber $2f$ mensagens *prepare* de outras réplicas, o pedido diz-se

preparado.

Este esquema força diversas propriedades importantes *numa mesma vista*. Primeiro, um primário malicioso não pode "criar" um pedido do nada pois as réplicas corretas só processam pedidos que tenham recebido de um cliente. Segundo, um cliente ou um primário maliciosos não podem provocar a execução de dois comandos diferentes pela mesma ordem em duas réplicas corretas diferentes. A justificativa é a seguinte. Uma réplica correta só executa o comando se receber $2f+1$ mensagens *prepare* (contando com a sua), logo pelo menos $f+1$ dessas mensagens vêm de réplicas corretas. Mesmo que todas as réplicas maliciosas – no máximo f – enviem mensagens diferentes com o mesmo número a réplicas diferentes, isso não é suficiente para fazer dois quoruns de $2f+1$ mensagens: $2(2f+1) > 3f+1$.

Este esquema garante a ordem dentro de uma vista mas não quando há mudança de vista. A fase de *commit* resolve esse problema. Quando uma réplica tem um pedido preparado, difunde uma mensagem *commit* para as outras réplicas. Quando uma réplica tem $2f+1$ mensagens *commit* então o pedido diz-se confirmado e pode ser executado. Se a vista mudar o novo primário têm de propagar os pedidos confirmados para a vista seguinte. Uma réplica executa um comando num pedido quando este tiver sido confirmado e todos os comandos em pedidos com número de ordem inferior tiverem sido executados. Quando o comando termina é enviada uma mensagem *reply* ao cliente que o solicitou.

Como já vimos, para garantir que o serviço continua a funcionar se o primário falhar é preciso mudar de vista. Vimos acima que os secundários podem suspeitar que o primário falhou. Fala-se em *suspeitar* em lugar de *detectar* pois em sistemas assíncronos pode existir incerteza em relação a essa falha. O modelo assíncrono não impõe limites temporais para a comunicação e processamento, logo o fato de o primário não responder durante algum tempo, não significa que falhou: ele ou a comunicação podem estar simplesmente lentos. Esse, aliás, é o ponto fraco do BFT: um atacante pode atrasar o sistema atrasando a comunicação e forçando a mudança constante de primário. O protocolo de mudança de vista é apresentado esquematicamente na figura 3.5. A questão crucial é garantir que os pedidos já confirmados são processados na nova vista.

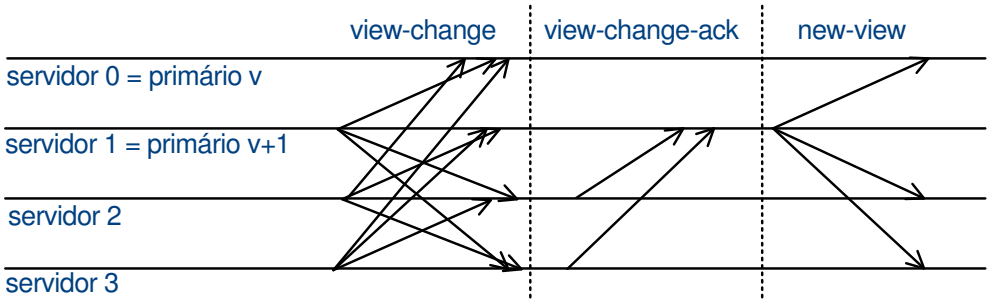

Figura 3.5. Mudança de vista no BFT [Castro and Liskov, 2002].

O bom desempenho do BFT deve-se não apenas ao uso de criptografia simétrica mas também a diversas otimizações. Duas delas merecem ser mencionadas. Para garantir a propriedade de ordem total da RME é essencial ordenar as escritas mas não as leituras.

Assim, o protocolo de leitura pode ser simplificado: o cliente envia um pedido de leitura para todas as réplicas e estas devolvem-lhe o valor pedido. Depois o cliente tenta recolher $f+1$ respostas idênticas. Se isso acontecer, a operação terminou. Caso haja escritas concorrentes e não seja possível recolher respostas idênticas, o cliente volta a fazer o pedido usando o protocolo que vimos atrás. No entanto, é fácil compreender que a probabilidade deste segundo caso acontecer é muito baixa.

A segunda otimização consiste em fazer processamento em pacotes de pedidos (*batching*). Em lugar de o primário enviar cada pedido recebido usando o protocolo explicado acima, tenta juntar diversos pedidos num só e enviá-lo.

Muito mais poderia ser dito sobre o BFT. As réplicas têm de guardar diversos registos (*logs*) que têm de ser limpos para evitar que cresçam indefinidamente. O BFT resolve a questão com uma espécie de *garbage collection* baseada em provas de que a informação sobre uma mensagem pode ser descartada. Uma versão do BFT, BFT-PR, utiliza recuperação proativa para processar as intrusões em servidores, uma questão para a seção 3.5.

Um artigo recente apresenta uma solução para RME TI com menor número de passos do que o BFT [Martin and Alvisi, 2005]. No entanto, isso é feito à custa de piorar a resistência para $N \geq 5f+1$.

3.3.1.2. Rampart

O sistema Rampart é um sistema que suporta RME TI que surgiu antes do BFT [Reiter, 1995, Reiter, 1994]. Na realidade o Rampart é mais genérico do que o BFT, pois é um sistema de comunicação em grupo que oferece primitivas de comunicação como difusão fiável e difusão atômica com sincronia de vistas. Tem um protocolo de filiação (*membership*) que permite a entrada e a saída de membros num grupo e a remoção de membros maliciosos, logo o conjunto de servidores que concretizam um serviço não é fixo. Neste contexto a *vista* é o conjunto de membros do grupo num determinado momento. No âmbito do projeto ITUA do programa OASIS foi feita uma concretização completa do sistema [Ramasamy et al., 2002]. A resistência é a mesma do BFT: $N \geq 3f+1$.

O protocolo de difusão atômica do Rampart é semelhante ao do BFT. No Rampart, o papel do primário é desempenhado pelo *sequenciador*, que pode ser definido em cada vista, por exemplo, como o servidor com menor identificador. O Rampart tem muito pior desempenho do que o BFT pois usa mensagens assinadas com criptografia de chave pública, em vez de MACs. No entanto, a maior fraqueza do Rampart, que aliás é partilhada com os demais sistemas de comunicação em grupo, é a de que as suspeitas de falha levam à remoção dos suspeitos do grupo. Assim, um atacante pode tentar obter uma maioria de processos maliciosos no grupo atrasando os processos corretos e causando a sua expulsão. Pelo contrário, no BFT se há suspeitas sobre o primário, este passa a secundário, mas nunca é expulso, logo o problema não existe.

Um ponto interessante do Rampart é que pode votar os outputs das réplicas de duas formas diferentes [Reiter, 1995]. A primeira usa um esquema de *assinatura de limiar-(k,N)*: o cliente aceita a resposta se esta estiver assinada usando esse esquema. No entanto,

este esquema apresentou um desempenho fraco de forma a que na prática é usado um esquema de votação semelhante ao BFT.

3.3.1.3. SINTRA, SecureRing, SecureGroup, Worm-IT

Antes de vermos alguns sistemas que melhoram a resistência do BFT e do Rampart, vamos referir quatro sistemas: SINTRA, SecureRing, SecureGroup e Worm-IT. Os quatro têm em comum oferecerem uma primitiva de difusão atômica TI. Não constituem uma solução completa para fazer RME TI, mas o que lhes falta é apenas a comunicação com os clientes. Todos têm resistência $N \geq 3f + 1$.

O sistema SINTRA oferece um conjunto de primitivas de difusão (fiável, atômica, causal), considerando um conjunto estático de máquinas, como o BFT [Cachin and Poritz, 2002]. Ao contrário do BFT e do Rampart que usam detecção de falhas/intrusões para garantir o progresso do sistema, o SINTRA contorna o FLP mediante um protocolo de consenso binário baseado em aleatoridade [Cachin et al., 2000]. O protocolo usa criptografia de limiar e criptografia de chave pública, logo o seu desempenho é fraco quando o tempo de comunicação é "pequeno", como numa LAN; numa WAN o tempo de processamento pode ser negligenciável face ao de comunicação, logo o desempenho pode ser aceitável.

O SecureRing é um sistema de comunicação em grupo especialmente vocacionado para redes de pequena dimensão, já que é baseado num anel lógico de máquinas [Kihlstrom et al., 2001]. O protocolo de ordenação é baseado num *token* que circula no anel, só podendo enviar mensagens quem tiver esse *token*. O sistema usa assinaturas baseadas em criptografia de chave pública, mas em menor quantidade do que o Rampart. Por exemplo, em vez das mensagens serem todas assinadas, é assinado o *token* que transporta um resumo criptográfico das mensagens já enviadas mas ainda não entregues. Máquinas maliciosas são removidas com base em informação fornecida por um detector de falhas bizantinas [Kihlstrom et al., 2003].

O SecureGroup usa um protocolo de difusão atômica baseado em aleatoridade [Moser et al., 2000, Moser and Melliar-Smith, 1999]. Sendo um sistema de comunicação em grupo como o SecureRing, tem o inconveniente de a resistência ser menos de um terço de todas as máquinas do universo das que podem teoricamente entrar no grupo, não das que fazem parte do grupo num determinado instante. Assim, na prática a resistência é menor do que a dos outros sistemas.

O Worm-IT é um sistema de comunicação em grupo baseado na noção de *wormhole* seguro [Correia et al., 2005b, Veríssimo, 2003]. O sistema é baseado num *modelo de falhas híbrido*: a maior parte do sistema pode sofrer intrusões mas cada nó é extendido com um componente seguro, a TTCB [Correia et al., 2002b]. Sobre este tipo de componentes e modelos veremos mais na próxima seção. O que é relevante neste ponto é referir que o protocolo de difusão atômica tem a vantagem de ser eficiente (não usa criptografia de chave pública) e totalmente distribuído (não tem um primário/sequenciador), o que lhe permite evitar os ataques referidos atrás a propósito do BFT.

3.3.1.4. Otimização da resistência

Todos os sistemas mencionados até agora têm um ponto em comum: toleram menos de $1/3$ de réplicas maliciosas, ou seja, $N \geq 3f+1$. Essa proporção pode parecer tão boa como outra qualquer mas, se fizermos as contas, significa que para tolerar intrusões num servidor são precisos $N = 4$, para tolerar intrusões em 2 são precisos $N = 7$ e assim por diante. Estes números são consideravelmente altos, já que cada servidor tem um custo em termos de hardware e software. Mais, para garantir que as réplicas não têm vulnerabilidades, um problema de que falaremos mais tarde, pode ser necessário desenvolver software específico para cada réplica, o que implica um custo significativo. Por tudo isto, reduzir o número de réplicas necessárias para tolerar f intrusões não é uma questão menor.

Uma contribuição engenhosa para esta questão foi proposta em [Yin et al., 2003]. A idéia consiste em separar o *acordo sobre a ordenação de mensagens* da *execução do serviço*. Porque é que todos os sistemas acima precisavam de pelo menos $3f+1$ réplicas? Porque em sistemas assíncronos a difusão atômica – o acordo sobre a ordenação das mensagens – é impossível com menos réplicas. Uma vez ordenados os pedidos/comandos, quantas réplicas são necessárias para executar o serviço? Bastam $2f+1$ para se fazer uma votação simples (conta o que disser a maioria, i.e., $f+1$).

O esquema apresentado nesse artigo é arquitetural: o serviço passa a ter $3f+1$ réplicas que fazem *acordo* sobre a ordenação e $2f+1$ réplicas que *executam* o serviço. Esta arquitetura é apresentada na figura 3.6. A grande vantagem desta solução é que reduz o número de réplicas que executam o serviço que, em geral, serão mais complexas e caras do que as que fazem acordo (a execução do serviço pode envolver por exemplo um base de dados de grande dimensão). É também dada uma solução para proteger a confidencialidade dos dados nos servidores através de uma *firewall*, mas isso ficará para mais tarde (seção 3.6).

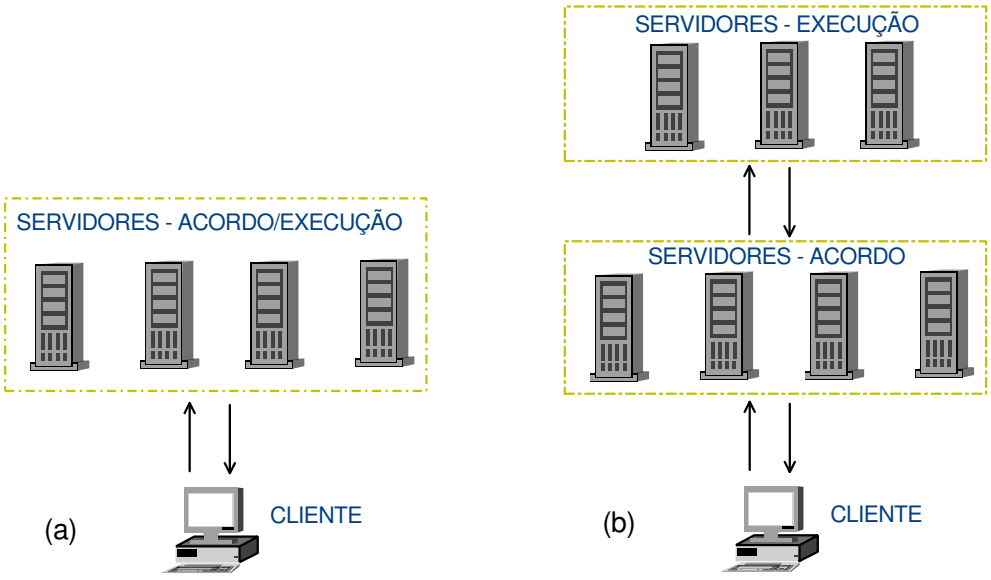

Figura 3.6. Separação acordo-execução. (a) Arquitetura típica. (b) Arquitetura com separação [Yin et al., 2003].

Um sistema que permite realmente diminuir o número de réplicas para $N \geq 2f+1$ é apresentado em [Correia et al., 2004]. A solução consiste em usar um *modelo de falhas híbrido* baseado num *wormhole* seguro para ordenar as mensagens[12]. Este *wormhole*, chamado TTCB, é um componente distribuído e seguro. A arquitetura do sistema está representada na figura 3.7. A TTCB é um núcleo de segurança distribuído, que é suficientemente simples para ser construída de forma a que seja segura, ou seja, para que não seja possível que aí ocorram intrusões. Na realidade, para suportar este serviço o componente precisa de ter apenas um serviço de ordenação de mensagens: o TMO (*Trusted Multicast Ordering*). Como esse serviço é executado num componente seguro, são precisos apenas $2f+1$ servidores para o executar.

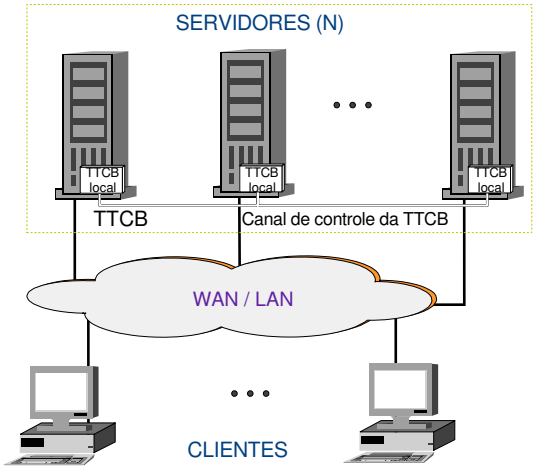

Figura 3.7. Replicação de máquinas de estados com um *wormhole* [Correia et al., 2004].

O algoritmo funciona esquematicamente da seguinte forma. Um cliente envia um pedido para um dos servidores à sua escolha. Um servidor malicioso pode tentar não processar o pedido, logo ao fim de um certo tempo o cliente reenvia o pedido para outros f servidores, o que garante que pelo menos um servidor correto o recebe. O pedido vai assinado com um vetor de MACs de forma a evitar que um servidor malicioso modifique o pedido.

Quando um servidor correto recebe o pedido, difunde-o por todos os outros servidores e passa um resumo criptográfico desse mesmo pedido ao TMO. Quando outro servidor recebe esse pedido, passa também uma síntese ao TMO. Quando o TMO descobre que $f+1$ servidores têm o mesmo pedido, atribui-lhe um número de ordem e entrega esse número a todos os servidores. O TMO espera por $f+1$ servidores para garantir que pelo menos um é correto e tem mesmo a mensagem. Como o TMO é seguro, não pode "mentir", todos os servidores recebem o mesmo número de ordem para o pedido. Quando um servidor não tem pedidos com número menor por executar, executa o comando nesse pedido e envia a resposta ao cliente. Quando o cliente recebe $f+1$ cópias da mesma res-

[12] A metáfora do *wormhole* vem de um conceito da astrofísica que alguma fição científica tem apresentado como atalhos que permitiram viajar em pouco tempo entre pontos afastados do universo. Uma introdução ao tema está em http://en.wikipedia.org/wiki/Wormhole. A idéia explorada neste trabalho consiste em usar um componente seguro por onde a informação pode "viajar" em segurança. Os *wormholes* têm sido usados também para aplicações com requisitos temporais [Veríssimo and Casimiro, 2002].

posta vindas de servidores diferentes tem a certeza de que essa é a resposta correta pois pelo menos um dos servidores é correto. Logo, aceita essa resposta.

O algoritmo tem muito mais detalhes que não podem ser aqui explicados. Convém apenas dizer que não usa criptografia de chave pública em tempo de execução (pode ser necessária para distribuir inicialmente as chaves secretas).

3.3.2. Quoruns

Um *sistema de quoruns* $\mathcal{Q}$ é um conjunto de subconjuntos de servidores denominados *quoruns* tal que $\forall Q_1, Q_2 \in \mathcal{Q}, Q_1 \cap Q_2 \neq \emptyset$. Esta definição, apesar de comum, tem tanto de precisa quanto de obscura. Para que servirá definir *um conjunto de subconjuntos de servidores*?

Dado um conjunto de servidores U, um sistema de quoruns permite raciocinar sobre esses servidores e definir *objetos* distribuídos com diferentes semânticas, por exemplo, variáveis compartilhadas, objetos de exclusão mútua e objetos de consenso. Esta explicação não exclui os algoritmos de RME, que no fundo concretizam um tipo de variáveis compartilhadas com semântica forte (leituras e escritas ordenadas). A diferença é que enquanto que a RME é uma solução genérica para concretizar *serviços* tolerantes a faltas/intrusões, os quoruns geralmente são usados para construir *repositórios de dados* tolerantes a faltas/intrusões, o que constitui um caso particular dos referidos serviços.

Ao servirem para concretizar algo de mais simples do que RME, muitas vezes os trabalhos com quoruns evitam a necessidade de realizar consenso e, como tal, não são circunscritos pelo FLP podendo os algoritmos ser totalmente assíncronos. No entanto, a principal diferença entre a RME e os sistemas de quoruns é que as operações na RME envolvem sempre todos os servidores, enquanto que nos sistemas de quoruns as operações são geralmente feitas sobre um quorum – um subconjunto dos servidores – o que torna os algoritmos mais escaláveis.

Nesta seção vamos considerar o uso de sistemas de quoruns para concretizar repositórios de dados. Uma forma de caracterizar um repositório é pensando nele como concretizando um conjunto de *variáveis de memória compartilhada* (*shared-memory*), ou seja, de memória que pode ser lida e escrita por diversos processos (clientes). Em sistemas distribuídos há um trabalho vasto em *algoritmos sobre memória compartilhada*, desenvolvido em paralelo com todo o trabalho em *algoritmos com comunicação por mensagens* (*message passing*)[13]. No entanto, apesar de serem duas linhas de pesquisa paralelas, num sistema distribuído as variáveis de memória compartilhada têm necessariamente de ser concretizadas usando clientes e servidores que se comunicam por mensagens. Os sistemas de quoruns fornecem uma forma conveniente de raciocinar sobre os conjuntos de servidores com vista à definição de algoritmos que implementem os repositórios.

Lamport apresentou uma classificação de variáveis de memória compartilhada que continua a ser amplamente utilizada [Lamport, 1986]. Um primeiro ponto dessa classificação é o de quantos processos podem acessar uma variável para leitura e escrita. Na discussão que se segue vamos considerar sempre variáveis com múltiplos-escritores/múltiplos-leitores (*multi-writer/multi-reader*), embora existam muitos traba-

[13]Os algoritmos mencionados neste capítulo são todos baseados em comunicação por mensagens.

lhos em variáveis para um-escritor/múltiplos-leitores (*single-writer/multi-reader*).

O segundo ponto da classificação é a *semântica de consistência* da variável, que pode ser uma de três: segura (*safe*), regular e atômica. Diz-se que a operação o_1 *acontece antes* da operação o_2 se o_1 termina antes de o_2 começar. Duas operações o_1 e o_2 dizem-se *concorrentes* se nem o_1 *acontece antes* de o_2 nem o_2 *acontece antes* de o_1. As três semânticas de consistência podem ser definidas informalmente da seguinte forma [Lamport, 1986, Martin et al., 2002a]:

- *segura:* uma leitura que não seja concorrente com nenhuma escrita retorna o último valor escrito; uma leitura concorrente com uma ou mais escritas retorna qualquer valor;
- *regular:* garante a semântica segura e também que se uma leitura é concorrente com várias escritas, o valor retornado é um dos valores dessas operações de escrita ou o valor escrito pela última escrita que tenha terminado antes da leitura;
- *atômica:* garante a semântica regular e também que as escritas e leituras retornam valores como se tivessem sido feitas de acordo com uma ordem definida; uma variável com esta semântica também se diz *linearizável* [Herlihy and Wing, 1990][14].

Estas definições seguem as originais de [Lamport, 1986] que não consideram o caso de múltiplos escritores, apenas múltiplos leitores. Alguns trabalhos mais recentes, como [Martin et al., 2002a], consideram também este caso. A especificação da semântica no caso de escritas concorrentes depende de cada algoritmo. É interessante notar que se fosse usada replicação de máquinas de estados para concretizar um repositório de dados, p. ex. o BFT, a semântica obtida seria a mais forte, a atômica, incluindo ordem nas escritas concorrentes.

Os trabalhos em quoruns para tolerância a faltas acidentais tem vários anos e é até anterior à replicação de máquinas de estados [Gifford, 1979]. Muito mais recentemente surgiu um interesse considerável no estudo de sistemas de quoruns tolerantes a faltas bizantinas, ou tolerantes a intrusões, começando em [Malkhi and Reiter, 1997a, Malkhi et al., 1997][15]. Estes sistemas são chamados de *sistemas de quoruns de mascaramento*, já que o objetivo é *mascarar* a ocorrência de faltas em alguns servidores.

3.3.2.1. Variáveis compartilhadas com quoruns

O tipo de objeto distribuído mais óbvio é a *variável compartilhada*. O trabalho em sistemas de quoruns é fértil em algoritmos para a concretização deste tipo de variáveis. Vamos ilustrar este tipo de algoritmos com variantes das variáveis compartilhadas definidas em [Malkhi and Reiter, 1998a, Malkhi and Reiter, 1998b][16].

[14] Na realidade o conceito de linearizável é uma generalização da semântica atômica de variáveis compartilhadas para objetos genéricos.

[15] Há um trabalho anterior que combina quoruns e faltas bizantinas mas que considera apenas clientes maliciosos [Naor and Wool, 1996].

[16] O trabalho original usa a noção de *fail-prone system* para generalizar a idéia de que *no máximo f servidores podem falhar* [Malkhi and Reiter, 1998a]. Na prática tal generalização conduz a algoritmos

Para cada variável compartilhada x existe em cada servidor u uma cópia da variável x_u e uma estampilha temporal $t_{x,u}$. No sistema existe um conjunto de escritores W_x que contém os identificadores dos clientes que podem escrever em x. O valor de W_x é conhecido por todos os clientes e servidores (pode estar codificado no nome da variável ou ser guardado em outra variável compartilhada). A estampilha temporal indica quando a variável foi escrita pela última vez e os conjuntos de estampilhas temporais usadas pelos clientes não se intersectam (p. ex. as próprias estampilhas incluem o identificador do cliente nos bits menos significativos).

Os clientes comunicam com os servidores usando uma *chamada a procedimento remoto quorum*. A chamada Q-RPC(m) envia o pedido m a um subconjunto dos servidores e recolhe respostas de um quorum. A operação pode implicar reenviar o pedido e excluir servidores maliciosos, já que estes podem tentar boicotar o funcionamento do sistema de diferentes modos. Nem os clientes e nem os servidores comunicam diretamente entre si, o que é uma característica comum à maioria dos trabalhos de quoruns publicados, que traz benefícios em termos de escalabilidade [Malkhi and Reiter, 2000].

Os sistemas de quoruns podem ser divididos em várias classes, das quais veremos duas. Começaremos pela classe que conduz a algoritmos mais simples, os sistemas de quoruns de *disseminação-f*. A principal característica destes sistemas é que os dados armazenados são assinados pelo cliente que os escreve – *dados auto-verificáveis* – logo os servidores não podem forjar ou modificar esses dados. Nos algoritmos de escrita e leitura que vamos ver, o sistema de quoruns tem de obedecer a duas propriedades:

- *Consistência.* $\forall Q_1, Q_2 \in \mathscr{Q}, |Q_1 \cap Q_2| \geq f+1$
- *Disponibilidade.* $\forall Q \in \mathscr{Q}, |Q| \leq n-f$

Comecemos por assumir que *os escritores são corretos*[17], que $N \geq 3f+1$ (consequência das duas propriedades acima) e que o tamanho dos quoruns é $\forall Q \in \mathscr{Q}, |Q| = \lceil \frac{N+f+1}{2} \rceil$. O algoritmo que permite escrever o valor v na variável compartilhada x com *semântica regular* é bastante simples:

1. fazer Q-RPC para obter o conjunto de estampilhas temporais $\{t_{x,u}\}_{u \in Q_1}$ do quorum Q_1 de servidores;
2. escolher uma estampilha t maior do que todas as obtidas e fazer Q-RPC para enviar o par $\langle v,t \rangle$ para um quorum Q_2.

É fácil compreender que f servidores maliciosos não podem interferir com o funcionamento do algoritmo pois os valores que não estiverem corretamente assinados por um escritor de W_x são descartados pelos leitores que os recebam, e assumiu-se que os escritores são corretos, logo os valores serão assinados corretamente. O algoritmo que lê o conteúdo de x é igualmente simples:

---

mais difíceis de compreender, logo não vamos usá-la.

[17] Podíamos falar de *clientes* corretos, mas sob o ponto de vista do algoritmo é irrelevante os leitores serem corretos ou maliciosos já que não alteram o estado dos servidores.

1. fazer Q-RPC para obter o conjunto de pares assinados valor/estampilha $\{\langle v_u, t_u\rangle_{w_u}\}_{u\in Q_1}$ no quorum Q_1 de servidores;

2. retornar o valor com a maior estampilha escrito por um escritor de W_x.

Consideremos agora a possibilidade de existirem *escritores maliciosos*. O problema é complicado pois um escritor malicioso pode facilmente deixar o sistema num estado inconsistente, p. ex. escrevendo um par $\langle v_i, t_i\rangle$ diferente em cada servidor. A primeira parte da solução consiste em usar o protocolo *echo* [Reiter, 1994] para garantir que todos os servidores que aceitem a escrita de um valor, aceitem o mesmo par $\langle v,t\rangle$. O protocolo funciona assim: (1) o escritor envia o par para os servidores e obtém ecos assinados vindos de um quorum inteiro; (2) o escritor envia o par e as assinaturas para os servidores do mesmo quorum. Como todos os quoruns se intersectam é impossível o servidor escrever em dois quoruns dois valores diferentes com a mesma estampilha, ou seja, dois pares $\langle v,t\rangle$ e $\langle v',t\rangle$ com $v \neq v'$.

Consideremos agora o caso genérico em que não temos dados assinados pelos escritores, ou seja, em que os dados não são *auto-verificáveis*. Uma justificativa para esta generalização é a de que para verificar as assinaturas seria preciso distribuir as chaves públicas dos clientes pelos outros clientes, o que pode não ser conveniente.

Neste caso genérico é necessário que a interseção de dois quoruns tenha sempre uma maioria de servidores corretos:

- *Consistência.* $\forall Q_1, Q_2 \in \mathscr{Q}, |Q_1 \cap Q_2| \geq 2f+1$

Assim, é necessário aumentar a redundância de servidores: $N \geq 4f+1$ e $\forall Q \in \mathscr{Q}, |Q| = \lceil \frac{N+2f+1}{2} \rceil$. Este tipo de sistema de quoruns é denominado de *mascaramento-f*.

Um algoritmo que permite a um escritor $w \in W_x$ escrever o valor v na variável compartilhada x com *semântica segura* é:

1. fazer Q-RPC para obter o conjunto de estampilhas temporais $\{t_{x,u}\}_{u\in Q_1}$ do quorum Q_1 de servidores;

2. escolher uma estampilha t maior do que todas as obtidas;

3. fazer Q-RPC para enviar $\langle v,t\rangle$ e receber ecos assinados de um quorum Q_2;

4. fazer Q-RPC para enviar os ecos para Q_2, escrevendo $\langle v,t\rangle$ nesse quorum.

O protocolo que permite a um cliente ler a variável x é:

1. fazer Q-RPC para obter o conjunto de pares valor/estampilha $\{\langle v_u, t_{x,u}\rangle_{w_u}\}_{u\in Q_1}$ no quorum Q_1 de servidores, cada par assinado pelo servidor onde se encontra;

2. retornar o valor v do par $\langle v,t\rangle$ com maior estampilha que apareça em pelo menos $f+1$ respostas (ou $\perp$ se não houver nenhum).

Caso uma leitura não seja concorrente com uma ou mais escritas, o valor retornado é o último escrito. Caso contrário, a variável pode retornar um valor qualquer dos que estão a ser escritos ou $\perp$. Essa é a razão pela qual a semântica é a mais fraca, a semântica *segura*.

Os algoritmos apresentados têm em comum a forma como os quoruns aí aparecem explicitamente (p. ex. Q_1, Q_2). Esta forma de especificação torna os algoritmos muito genéricos já que não dependem do conteúdo dos quoruns. Uma forma menos genérica mas mais simples de especificar quoruns é a que referimos atrás a propósito do BFT, por exemplo, "um quorum de (quaisquer) $f+1$ servidores", "um quorum de $2f+1$ servidores" [Castro and Liskov, 2002, Martin et al., 2002a, Martin and Alvisi, 2004].

Em relação às variáveis compartilhadas vale a pena dizer ainda que alguns algoritmos mais eficientes dos que aqui apresentados encontram-se em [Martin et al., 2002a]. A tabela 3.1 apresenta uma comparação de alguns algoritmos. Um protocolo *não-confirmável* difere dos que vimos em que não é possível determinar quando uma escrita termina.

referência	disseminação-f confirmável	mascaramento-f confirmável	disseminação-f não-confirmável	mascaramento-f não-confirmável
[Malkhi and Reiter, 1998a]	regular, 3f+1	segura, 4f+1		
[Malkhi and Reiter, 1998b]	atômica, 3f+1	segura, 4f+1		
[Martin et al., 2002a]	atômica, 3f+1	atômica, 3f+1	regular, 2f+1	regular, 2f+1
[Martin et al., 2002b]	regular, 3f+1	segura, 4f+1	regular, 2f+1	segura, 3f+1

Tabela 3.1. Comparação das semânticas e resistências de variáveis compartilhadas com quoruns [Martin et al., 2002a].

3.3.2.2. Outros objetos de memória compartilhada com quoruns

Vários outros objetos de memória compartilhada têm sido concretizados usando quoruns. Uma primitiva importante quando se fala de programação concorrente é a *exclusão mútua*. No tipo de sistemas que estamos a ver, o objetivo é permitir a um cliente reservar recursos (p. ex. objetos) para seu uso exclusivo enquanto realiza determinada operação. Um algoritmo que não garante que o recurso seja reservado caso haja vários clientes a concorrer é apresentado em [Malkhi and Reiter, 1998b]. Um algoritmo de exclusão mútua sem esta restrição baseado no algoritmo do confeiteiro de Lamport encontra-se em [Bessani et al., 2005]. Apesar de os algoritmos com quoruns tentarem geralmente evitar o consenso, por vezes é mesmo necessário ter um objeto que realize essa operação. Um objeto de consenso aleatório baseado em quoruns encontra-se em [Malkhi and Reiter, 2000].

Por último, vale a pena referir um trabalho recente que modifica uma premissa que todos os sistemas de quoruns até então assumiam: f e N serem constantes. Martin e Alvisi introduziram uma metodologia genérica que permite transformar protocolos de quoruns como os dados acima em protocolos dinâmicos, nos quais f e N podem crescer ou diminuir [Martin and Alvisi, 2004]. A idéia básica consiste em substituir a primitiva Q-RPC por uma primitiva DQ-RPC que lida com as variações desses parâmetros.

3.4. Fragmentação: garantindo disponibilidade, integridade e confidencialidade

Voltemos à nossa estória do pirata. As soluções de replicação que foram apresentadas resolvem o problema da disponibilidade e da integridade de um serviço se um máximo de f torreões forem invadidos pelo pirata. Imaginemos agora que o que o pirata procura é informação, por exemplo, um mapa do tesouro. Nesse caso, replicar informação em todos os torreões, em lugar de aumentar a segurança acaba por diminuí-la, já que basta ao pirata tomar um dos torreões para o obter o mapa! Para garantir também a *confidencialidade* – além da disponibilidade e integridade – os dados terão de ser *fragmentados* ou *disseminados* pelos diversos servidores/torreões, podendo ser reconstruídos apenas por quem tiver autorização para o fazer. Se o pirata invadir um torreão obterá apenas um pedaço de mapa sem significado.

A fragmentação de dados para obter disponibilidade, integridade e confidencialidade é o tema desta seção. Convém referir que alguns trabalhos que veremos têm outra motivação: fragmentar os dados pelos servidores para diminuir o espaço de armazenamento total, evitando ter uma réplica de todos os dados em todos os servidores. Em qualquer dos casos, nesta linha de trabalho o objetivo é o *armazenamento de dados*, como vimos a propósito dos sistemas de quoruns, não a concretização de serviços genéricos, como na replicação de máquinas de estados.

O trabalho seminal na área é de Fraga e Powell, que cunharam originalmente o termo *tolerância a intrusões* [Fraga and Powell, 1985]. Esse trabalho tratou de muitos dos problemas que os trabalhos seguintes também trataram, logo é por ele que vamos começar.

Esse trabalho apresenta um sistema de arquivos TI e introduz uma técnica chamada *fragmentação-redundância-dispersão* (*FRS, fragmentation-redundancy-scattering*). A idéia básica é fácil de compreender. Um sistema de arquivos é concretizado através de um conjunto de servidores. Cada arquivo F, antes de ser armazenado, é fragmentado em m fragmentos, que depois são espalhados pelos N servidores, de tal modo que cada fragmento fique guardado em vários servidores (para garantir a disponibilidade) e nenhum servidor tenha fragmentos suficientes para que um intruso possa reconstruir F (para contribuir para a confidencialidade) – v. fig. 3.8[18]. Desta breve explicação percebe-se imediatamente que o sistema não fornece propriamente confidencialidade em sentido estrito, já que um intruso pode obter alguma informação atacando um servidor, embora para reconstruir o arquivo completo sejam necessários m fragmentos. Para usar as palavras do artigo, este esquema "reduz o significado da informação disponível a um intruso". Em relação à integridade, são sugeridas duas soluções alternativas para detectar fragmentos corrompidos: (1) quando é feita a leitura são lidas diversas cópias de cada fragmento e faz-se uma votação; (2) junta-se um MAC a cada fragmento.

A informação sobre a localização dos arquivos está armazenada num serviço que é também distribuído para tolerar intrusões. Um ponto importante gerido por esse serviço

[18]Nos trabalhos desta área é usual falar-se de fragmentação, ou dispersão, de um arquivo F, em vez de acesso a uma variável compartilhada x, como nos trabalhos sobre quoruns. Nesta seção vamos seguir essa nomenclatura.

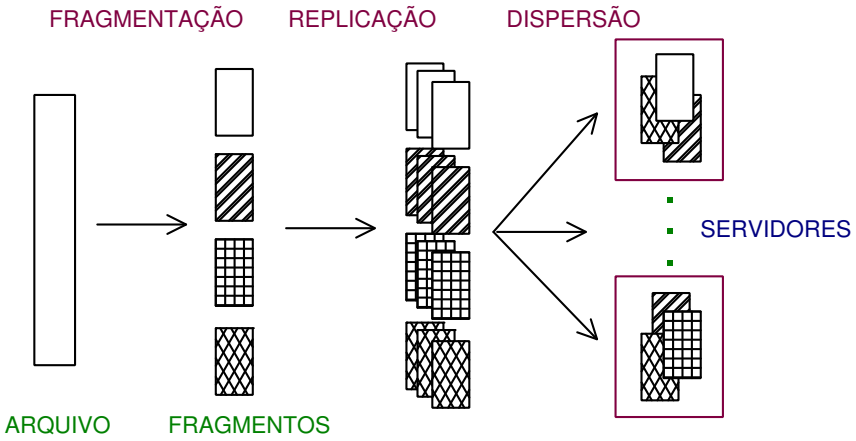

Figura 3.8. Fragmentação-redundância-dispersão [Deswarte et al., 1991].

é o da autorização de acesso a um arquivo. A solução, que é apenas esboçada, usa um esquema de partilha de segredos para construir a chave necessária para acessar ao arquivo. O sistema tem a limitação de não tolerar clientes maliciosos.

3.4.1. Códigos de apagamento

Pouco depois de [Fraga and Powell, 1985], Rabin publicou uma solução para fragmentação – a que chama um *algoritmo de dispersão de informação* – que otimiza o espaço ocupado com base em *códigos de apagamento* (*erasure codes*) [Rabin, 1989]. Estes códigos são semelhantes aos códigos de correção de erros usados em telecomunicações, mas enquanto nos primeiros a informação pode apenas ser apagada, nos segundos pode também ser modificada.

A idéia consiste em dividir um arquivo em N fragmentos de forma a que seja suficiente ter k fragmentos para reconstruí-lo, mas $k-1$ fragmentos não cheguem para o fazer. Para o efeito usa-se um *código de apagamento-(k,N)*. Esse trabalho não fornece propriamente um protocolo para concretizar o esquema num sistema distribuído. Esse passo foi dado em outros trabalhos que dele derivam [Krawczyk, 1993, Alon et al., 2000, Garay et al., 2000]. Mais tarde, no âmbito do projeto PASIS (programa OASIS) foi definido um sistema eficiente para dispersão de informação com semântica atômica, que tolera clientes maliciosos [Goodson et al., 2004]. Nenhum destes trabalhos aborda a questão da confidencialidade, sendo os códigos de apagamento usados unicamente para otimizar o espaço ocupado pelos dados.

O único trabalho dessa linha que considera o problema da confidencialidade dos dados é bastante recente [Cachin and Tessaro, 2004, Cachin and Tessaro, 2005]. O primeiro dos dois artigos não permite acessos concorrentes, e cada arquivo só pode ser escrito uma vez (não lida com versões). Mesmo assim, esse trabalho vai servir para ilustrar este tipo de soluções.

O primeiro mecanismo apresentado em [Cachin and Tessaro, 2004] é denominado AVID (*asynchronous verifiable information dispersal*) e não fornece confidencialidade, apenas integridade e disponibilidade. Os clientes podem ser maliciosos.

Um cliente que quer armazenar um arquivo F começa por o codificar como um vetor $[F_1,...,F_N]$ usando um código de apagamento-(k,N). Além disso obtém um conjunto de *impressões digitais* [Krawczyk, 1993] calculando um vetor com sínteses criptográficas (*hashes*) de cada F_i: $D = [D_1,...,D_N]$. Depois, toda essa informação é enviada para os servidores usando um protocolo de difusão fiável, que é uma variante do protocolo clássico de Bracha [Bracha, 1984]. Um protocolo de difusão fiável garante duas propriedades: (1) todos os servidores entregam os mesmos pedidos; (2) se o cliente é correto, o pedido é entregue (se for malicioso, pode não ser entregue). Este protocolo não é usado *ipsis verbis* mas modificado para diminuir a quantidade de dados enviados: o cliente envia apenas F_i para o servidor s_i, mas depois estes servidores trocam entre si os F_i para garantirem que todos têm o seu fragmento. Se o cliente for malicioso e alguns dos fragmentos estiverem corrompidos há duas possibilidades: o número de fragmentos disponíveis permite reconstruir os fragmentos omitidos, o que é feito; ou não é possível reconstruir esses fragmentos e o arquivo não é armazenado. Quando a operação é terminada os servidores apagam todos os fragmentos que não lhe pertencem.

A operação de leitura consiste simplesmente em pedir fragmentos aos servidores até se obterem os k necessários para reconstruir F. O parâmetro k tem de verificar a condição: $f+1 \leq k \leq N-2f$. A melhor resistência é obtida quando $N = 3f+1$, logo $k = f+1$.

O mesmo artigo apresenta o esquema cAVID que garante também a confidencialidade dos dados armazenados. Para garantir a confidencialidade é necessário haver controle de acesso ao arquivo. Para o efeito junto do arquivo é guardada uma lista de controle de acesso L com os identificadores dos clientes que a ele podem acessar.

A forma como é conseguida a confidencialidade é simples: o arquivo é cifrado usando criptografia simétrica antes de ser armazenado usando o esquema AVID. O problema é o que se faz da chave. Se o cliente ficasse com a chave para si, só ele poderia recuperar o arquivo, o que em geral não é o objetivo. Para resolver este problema usa-se um esquema de *criptografia de limiar*. Este esquema fornece essencialmente:

- um algoritmo para cifrar dados usando uma *chave pública PK*;
- um algoritmo para decifrar dados que usa uma *chave privada* SK_i para obter um fragmento σ dos dados cifrados;
- um algoritmo que permite verificar se um fragmento σ é válido usando uma *chave de verificação* VK;
- um algoritmo que permite obter os dados iniciais combinando k fragmentos decifrados usando VK.

Cada servidor tem uma chave privada SK_i e todos os clientes têm a chave pública PK. Para armazenar o arquivo F, o cliente gera uma chave secreta K, cifra o arquivo com essa chave (usando criptografia simétrica), e cifra K com PK. Depois, usa o algoritmo de dispersão AVID para armazenar o arquivo, a chave K cifrada e a lista de controle de acesso L. Para ler o arquivo, é necessário obter fragmentos σ de k servidores para reconstruir K.

Note que este esquema é ortogonal à fragmentação, podendo ser adicionado a um serviço baseado em replicação de máquinas de estados ou quoruns para garantir a confidencialidade dos dados armazenados. No entanto, se o serviço for baseado em fragmentação, após o processo de armazenamento do arquivo é necessário penetrar em k servidores para poder executar um ataque com o objetivo de quebrar a cifra de K, enquanto que com RME ou quoruns basta penetrar em um, logo a segurança oferecida é ligeiramente superior.

Este esquema é extendido para concretizar uma variável compartilhada para múltiplos escritores e múltiplos leitores e com semântica atômica em [Cachin and Tessaro, 2005]. A solução é baseada em estampilhas temporais de forma semelhante ao usado em sistemas de quoruns. Por exemplo, para fazer uma escrita o cliente começa por obter a maior estampilha armazenada e depois escreve o arquivo com uma estampilha superior. Para evitar ataques de negação de serviço através da utilização de estampilhas muito grandes, é usado o esquema de *non-skipping timestamps* que não permite que estas tomem um valor superior ao número de escritas já realizadas [Bazzi and Ding, 2004].

3.4.2. Partilha de segredos

Uma solução bastante evidente para garantir a confidencialidade dos dados seria usar um esquema de *partilha de segredos*, como os de Shamir ou Blakley [Shamir, 1979, Blakley, 1979]. No entanto, só foi encontrado um trabalho que usa essa técnica para tolerar intrusões em servidores, a *secure store* [Lakshmanan et al., 2003].

A razão pela qual não surgiram mais sistemas deste tipo é fácil de conjeturar: dificilmente o desempenho de um esquema de partilha de segredos é compatível com um sistema de armazenamento de dados de tamanho arbitrário. Para resolver esse problema, o trabalho que vamos estudar combina essa técnica com replicação, necessitando assim de mais servidores do que os sistemas que vimos até agora. Os servidores formam uma matriz com c colunas e r linhas, num total de $N = rc$ servidores (v. fig. 3.9).

Quando um cliente pretende armazenar um arquivo, divide-o em c fragmentos usando um *mecanismo de partilha de segredos (k,c)*, sendo k o número de fragmentos necessário para reconstruir o segredo (os mínimos são $k = f + 1$ e $c = 2f + 1$). Depois, cada fragmento é replicado por uma coluna de servidores (v. figura). Para recuperar um arquivo é preciso obter k fragmentos de diferentes colunas.

O sistema oferece um mecanismo ingênuo de recuperação proativa, o tipo de mecanismo que veremos na próxima seção. Em vez de simplesmente assumir que o número máximo de intrusões é f, o sistema considera que esse é o número máximo de intrusões durante um intervalo de tempo T_v. Para garantir essa hipótese, os fragmentos são renovados cada $T < T_v$, ou seja, são novamente cifrados usando um *protocolo de renovação de fragmentos*. Obviamente isso não é particularmente útil pois no mínimo é também necessário remover a intrusão ou o número de servidores corrompidos irá sempre aumentando.

A *secure store* não considera clientes maliciosos, que podem deixar os servidores num estado inconsistente. As semânticas de consistência oferecidas são fracas.

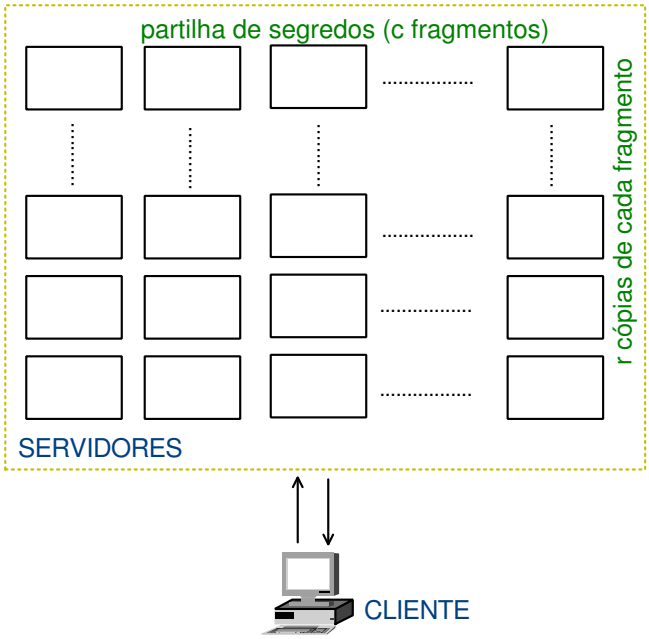

Figura 3.9. Arquitetura da *secure store* [Lakshmanan et al., 2003].

Um comentário final sobre a relação de todas estas soluções para TI com confidencialidade. Alguns dos artigos discutidos falam de *quoruns*. De fato, faz todo o sentido falar de sistemas de quoruns também neste contexto, embora com a diferença de que agora os dados não são replicados pelos servidores mas fragmentados. Na realidade, a utilização de sistemas de quoruns é ortogonal a todos os esquemas que temos visto, embora por vezes surjam apenas implicitamente.

3.5. Recuperação proativa

O mascaramento de intrusões através de replicação (seção 3.3) ou fragmentação (seção 3.4) permite tolerar um número máximo de f intrusões em servidores durante o tempo de vida de um serviço. Esse tempo normalmente será longo, p. ex. de meses, tornando essa premissa de existir um número máximo de intrusões difícil de substanciar. Lembremo-nos do nosso pirata: se lhe for dado tempo suficiente ele pode acabar por invadir muitos torreões!

Para ultrapassar essa limitação é necessário utilizar uma técnica de *tolerância a faltas* que referimos na seção 3.2.1: o *processamento de erros*. No contexto da TI, a idéia consiste em remover as intrusões que vão ocorrendo de tal forma a que o número de servidores corrompidos nunca ultrapasse f. Apesar de alguns trabalhos sugerirem a utilização de *detectores de intrusões* para descobrir quando é necessário remover uma intrusão, os sistemas desse tipo atualmente disponíveis dificilmente poderão ser usados para fazer remoção automática de intrusões dado o número elevado de falsos positivos e falsos negativos que produzem [Lippmann et al., 2000].

Na prática, na TI tem sido usado um mecanismo de processamento de erros denominado *recuperação proativa*. A idéia consiste em fazer periodicamente uma renovação de cada servidor de forma a deixá-lo num estado correto. Em outras palavras,

periodicamente são removidas quaisquer alterações ao estado ou código de um servidor no qual tenha ocorrido uma intrusão; esta renovação é feita mesmo que não tenham acontecido intrusões. Esta técnica não faz o sistema ficar invulnerável, mas o pirata teria de ser muito rápido para conseguir violar a sua segurança. Na prática o risco de serem corrompidos mais do que f servidores fica circunscrito a uma *janela de vulnerabilidade* que depende do período de recuperação.

Um atacante que realiza este tipo de ataques que procura corromper sucessivos servidores é por vezes chamado de *adversário móvel*, seguindo o primeiro trabalho na área [Ostrovsky and Yung, 1991]. Um interessante resumo dos trabalhos mais antigos nesta linha encontra-se em [Canetti et al., 1997]. Os trabalhos que veremos em seguida são mais recentes.

3.5.1. BFT-PR

O sistema BFT (seção 3.3.1.1) foi extendido para fazer recuperação proativa, passando a chamar-se BFT-PR [Castro and Liskov, 2002]. A idéia básica desta recuperação é a que acabamos de ver. As soluções de recuperação proativa anteriores ao BFT-PR, começando em [Ostrovsky and Yung, 1991], exigiam que o código da réplica estivesse em memória só de leitura. O BFT-PR precisa apenas de um pequeno *monitor* em memória deste tipo. No BFT-PR a recuperação consiste em realizar três operações:

1. renovar as chaves secretas usadas para a comunicação cliente-servidor e servidor-servidor (usadas para obter os MACs);
2. repor o código do sistema caso tenha sido corrompido;
3. repor o estado do sistema caso tenha sido corrompido.

O artigo em questão aponta diversas premissas para este tipo de recuperação ser possível. Em cada réplica tem de existir:

- um coprocessador criptográfico que armazene a chave privada da réplica e assine e decifre mensagens sem expor essa chave;
- uma memória não volátil só de leitura onde sejam guardadas as chaves públicas de cada uma das outras réplicas e o monitor de recuperação (p. ex. a BIOS);
- um temporizador seguro para disparar a recuperação (existem temporizadores em hardware que podem ser usados com esse fim).

Todos estes mecanismos exigem a hipótese adicional de que o atacante não tem acesso físico ao servidor. O BFT-PR exige ainda uma hipótese temporal mais forte do que o BFT: existe um instante desconhecido a partir do qual o atraso da comunicação é inferior a um dado Δ (ou seja, um modelo de sincronia parcial [Dwork et al., 1988]).

A primeira operação de recuperação referida acima é a renovação das chaves secretas. O protocolo usado é extremamente simples pois consiste no envio periódico (p. ex. cada minuto) de uma única mensagem com a nova chave. A mensagem enviada

pelo servidor s_i ao servidor s_j tem a forma: $\langle \text{NEW-KEY}, i, ..., \{k_{j,i}\}_{\varepsilon_j}, ..., t \rangle_{\sigma_i}$. A chave $\{k_{j,i}\}$ é a que será usada por s_j para calcular os MACs enviados a s_i. A chave é cifrada com a chave pública de s_j (ε_j) de forma a que só s_j a possa decifrar. A mensagem inclui um contador t que evita ataques de repetição de mensagens antigas. A mensagem é assinada com a chave privada de s_i (σ_i). Estas chaves são usadas para proteger a comunicação numa única direção. Já a comunicação com o cliente é protegida nas duas direções usando uma única chave, que lhe é enviada pelo servidor usando uma mensagem do mesmo tipo.

A segunda operação de recuperação consiste em repor o código do servidor. Esta recuperação é feita de forma proativa sempre que o temporizador dispara. Quando isso acontece, o monitor cria uma imagem do código e do estado da réplica em disco. Depois, força o reinício da máquina (*reboot*). Para verificar se o sistema operacional ou o código do serviço foram corrompidos o monitor usa as suas sínteses criptográficas que se encontram guardadas na memória só de leitura. Se tiverem sido corrompidos, o monitor obtém uma cópia de outros servidores.

A terceira operação começa por executar um protocolo com os outros servidores para determinar se o estado da réplica foi corrompido. Se tiver sido corrompido, o estado é transferido de outras réplicas corretas.

A janela de vulnerabilidade do BFT-PR é $T_v = 2T_k + T_r$, sendo T_k o período máximo de renovação das chaves e T_r o período de recuperação do servidor.

3.5.2. COCA

O sistema COCA (*Cornell On-line Certification Authority*) é uma autoridade de certificação *on-line* TI desenvolvida no âmbito do programa OASIS [Zhou et al., 2002a]. Esse sistema é bem mais simples do que o BFT-PR mas tem o interesse de ser um dos primeiros a usar recuperação proativa e de o fazer de uma forma diferente da que vimos.

O objetivo do COCA é fornecer certificados com associações entre nomes e chaves públicas. O sistema oferece apenas duas operações:

- *Update:* criar, atualizar ou invalidar certificados.
- *Query:* obter o certificado correspondente a um nome.

O COCA não usa RME mas um sistema de quoruns de disseminação. O número de servidores é de $N \geq 3f + 1$ e o tamanho do quorum é $|Q| = 2f + 1$. O sistema usa um esquema de *assinatura de limiar-(k,N)* com $k = f + 1$. Todos os clientes e servidores têm a chave pública do serviço. A chave privada está repartida por todos os servidores e é necessário um quorum de k para assinar um resultado do serviço, ou seja, um certificado.

O funcionamento básico do sistema é ilustrado pela figura 3.10. Um cliente envia um pedido para um dos servidores (o delegado) que o envia para outros $2f$, perfazendo $2f + 1$, ou seja, um quorum. Cada servidor obtém de alguma forma o certificado e assina-o com a sua parte da chave privada. Se ao fim de algum tempo o servidor não responder, o pedido é reenviado para outros $f + 1$ servidores de forma a que pelo menos um servidor correto o receba. Como cada quorum tem $2f + 1$ servidores, pelo menos $f + 1 = k$ deles irão devolver o certificado corretamente assinado, o que é suficiente para o cliente verificar a assinatura usando o esquema de assinatura de limiar-(k,N).

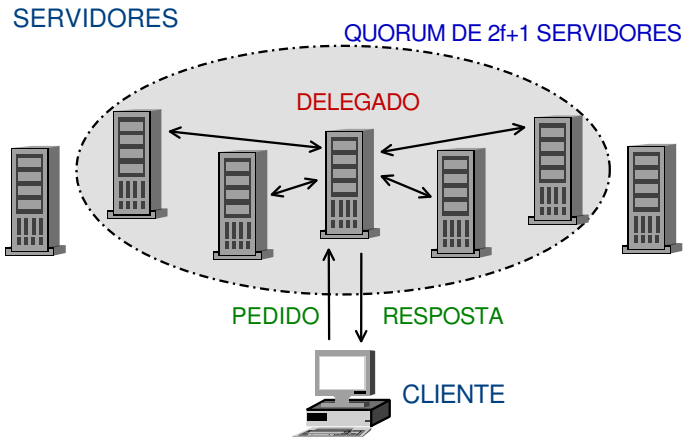

Figura 3.10. Funcionamento básico do COCA [Zhou et al., 2002a].

O esquema de recuperação proativa do COCA consiste em realizar periodicamente três operações:

1. renovar as partes das chaves privadas de cada servidor;
2. repor o código do servidor caso tenha sido corrompido;
3. repor o estado do servidor caso tenha sido corrompido.

Destas operações, a segunda e a terceira são semelhantes às do BFT-PR, sendo mais interessante explicar a primeira. A informação crítica que um atacante pode capturar se penetrar num servidor é a sua parte da chave privada, já que o COCA não assume que esta está em local seguro (ao contrário do BFT). Se o *adversário móvel* conseguisse capturar $f+1$ partes conseguiria personificar o serviço, logo a recuperação proativa tem de *renovar* essas partes.

Essa renovação das partes é baseada num *protocolo de partilha de segredos proativa* denominado APSS [Zhou et al., 2002b], que aliás é o primeiro protocolo assíncrono desse tipo. O protocolo é executado periodicamente, gerando de cada vez novas partes da chave privada. A chave privada, que se mantém sempre a mesma, nunca é materializada em nenhum dos servidores, tomando estes conhecimento apenas da sua parte. Um protocolo assíncrono de partilha de segredos proativa mais eficiente do que o APSS foi proposto em [Cachin et al., 2002].

Por fim, vale a pena referir que na sequência do COCA surgiu o sistema CODEX, uma evolução do COCA para armazenamento de dados [Marsh and Schneider, 2004]. A recuperação proativa usada no CODEX é semelhante à do COCA. Os dois sistemas têm um problema de modelo que fica claro através de duas afirmações contraditórias em [Zhou et al., 2002a]. Em diversos lugares do artigo é afirmado que o sistema é assíncrono mas, na seção 2.2, é também dito que "na prática, esperamos que possam ser feitas hipóteses temporais sobre partes do sistema que não tenham sido comprometidas" (pg. 334). Esta ambiguidade causa uma vulnerabilidade subtil que analisaremos de seguida.

3.5.3. Recuperação proativa em sistemas assíncronos?

Quando na seção 3.2.3 falámos de *modelos temporais*, referimos que o modelo assíncrono tinha a vantagem de não levantar hipóteses temporais, logo assumi-lo evita criar vulnerabilidades desse tipo. No entanto, recentemente foi mostrado que não é possível fazer recuperação proativa de forma segura (*safe*) em sistemas assíncronos [Sousa et al., 2005a, Sousa et al., 2005b].

A demonstração é elegante e vale a pena resumi-la.

Um protocolo ou algoritmo distribuído é sempre definido com base num conjunto de premissas. Exemplos de premissas incluem as que são feitos sobre o tipo e quantidade de faltas que podem ocorrer (p. ex. bizantinas e não mais do que f) e a sincronia da execução (p. ex. modelo assíncrono). Estas hipóteses são de fato uma abstração dos recursos de que o protocolo necessita para a sua execução. Por exemplo, quando se assume que não mais do que f servidores podem falhar, isso significa que o protocolo precisa de $N - f$ servidores corretos para funcionar corretamente.

O tipo de hipóteses que nos interessam são precisamente as deste tipo, as *hipóteses quantitativas de segurança sobre recursos*. Interessam-nos as hipóteses de segurança (*safety*), não as de progresso, pois é delas que depende a correção de um protocolo.

Dado um recurso qualquer *r*, esse recurso está *exausto* se tiver sido violada uma hipótese quantitativa de segurança sobre *r*. Um sistema diz-se *seguro-de-exaustão-r* (*r-exhaustion-safe*) se for capaz de garantir que o recurso *r* não será exausto.

Dado um sistema *A*, definimos $A_{t_{start}}$ como o instante de início da sua execução, $A_{t_{end}}$ o instante de terminação e $A_{t_{exaust}}$ o instante no qual determinado recurso *r* é exausto. Todos estes instantes são instantes de tempo real. O sistema *A* é seguro-de-exaustão-*r* sse $A_{t_{end}} < A_{t_{exaust}}$.

Consideremos um sistema composto por um serviço com vários servidores e clientes, como os que vimos acima. Esse sistema é *seguro-de-exaustão-servidor* se o número de servidores que podem falhar – no máximo f – não for exausto. Podemos dizer que existe um valor para o instante $A_{t_{exaust}}$ desse sistema, apesar de ser desconhecido e depender do "poder" do atacante. Como num sistema assíncrono os relógios podem derivar (*drift*) de forma ilimitada, e os tempos de processamento e comunicação são também ilimitados, não é possível garantir que o sistema terminará em determinado intervalo de tempo, ou seja, $A_{t_{end}}$ poderá tomar valores arbitrariamente grandes. Sendo assim, não é possível garantir que $A_{t_{end}} < A_{t_{exaust}}$ e o sistema não é seguro-de-exaustão-servidor.

O objetivo da recuperação proativa é garantir que nunca se atinge $A_{t_{exaust}}$ através da remoção das intrusões que possam ter ocorrido. Isso exige que a recuperação seja realizada *periodicamente* e que essa recuperação não demore mais do que um *tempo* determinado. Obviamente que para que isso seja possível é preciso levantar hipóteses *temporais* sobre o funcionamento do sistema, hipóteses essas que podem sempre ser violadas num sistema assíncrono. Na prática estas hipóteses podem ainda ser mais facilmente violadas se existirem ataques. Um atacante pode violar essas hipóteses, por exemplo, atrasando os relógios do sistema, ou atrasando a comunicação na rede (um ataque deste estilo ao sistema CODEX é explicado em [Sousa et al., 2005a]). Assim temos a impossibilidade

de recuperação proativa em sistemas estritamente assíncronos.

Uma solução para este problema consiste em usar um modelo de falhas híbrido baseado num *wormhole*, uma abordagem que já introduzimos atrás (v. seção 3.3.1.4). Ao contrário do resto do sistema, este componente deve ser síncrono de forma a que a recuperação possa ser executada a tempo. Como este componente é construído de forma a que seja seguro, as hipóteses de sincronia feitas em relação a ele não podem ser violadas devido a ataques. Convém notar que se continua a assumir que as intrusões nos servidores não ocorrem instantaneamente (no tempo real), mas de forma progressiva. O que esta solução permite garantir é que se não for possível num período $T_{rejuvenation}$ (tempo real) corromper $f+1$ servidores, então nunca serão corrompidos $f+1$ servidores e o sistema será *seguro-de-exaustão-servidor*.

3.6. Arquiteturas e sistemas

Apresentamos acima quatro tipos de soluções para tolerar intrusões: replicação de máquinas de estados, sistemas de quoruns, fragmentação e recuperação proativa. Como vimos, estes quatro tipos de mecanismos podem ser combinados de várias formas para concretizar *serviços distribuídos tolerantes a intrusões*, de que aliás vimos alguns exemplos. No entanto, existem outras técnicas que podemos considerar como sendo de *tolerância a intrusões* e que podem ser usadas para construir serviços distribuídos TI. Exemplos são a detecção de intrusões, a remoção dessas intrusões (quando ocorrem, não de forma proativa) e a reconfiguração. Outras técnicas de *prevenção* podem também ser usadas, como as *firewalls*. Nesta seção vamos ver diversas arquiteturas e sistemas da bibliografia que combinam diversos desses mecanismos para concretizar serviços TI. Voltando à nossa estória do pirata: o sistema pode ser bem mais complicado do que ter apenas um conjunto de torreões! Podem existir vigias para detectar intrusões, artilharia para disparar contra os piratas, etc.

A arquitetura comum a quase todos os sistemas que vimos até agora é a apresentada na figura 3.3. Duas exceções foram as arquiteturas da *secure store* (fig. 3.9) e a de separação de execução e acordo (fig. 3.6). Vimos ainda o modelo de falhas híbrido baseado numa arquitetura com um *wormhole* (seções 3.3.1.3 e 3.3.1.4).

3.6.1. Três variantes da arquitetura geral

Três arquiteturas ligeiramente mais complicadas do que a da figura 3.3 são comparadas em [Gupta et al., 2003].

A primeira arquitetura é chamada de *encaminhamento centralizado / gestão centralizada* e é apresentada na figura 3.11. O objetivo consiste em proteger um conjunto de servidores usando um pequeno número de componentes de confiança (*trusted*). Os pedidos recebidos são filtrados por uma *firewall*. Depois, chegam a uma *gateway* de confiança que os encaminha aleatoriamente para um dos servidores ativos. Cada servidor inclui um CMDaemon (*configuration management daemon*) encarregue de fazer detecção de intrusões localmente. Estes componentes informam um *gestor de configuração* centralizado e de confiança sobre o estado do servidor. O gestor limpa e recupera os servidores nos quais ocorram intrusões.

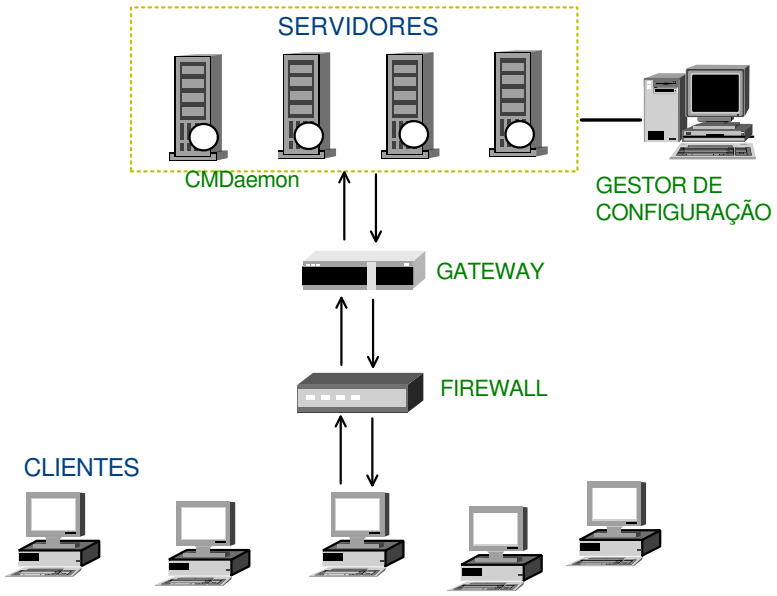

Figura 3.11. Arquitetura encaminhamento centralizado / gestão centralizada [Gupta et al., 2003].

A segunda arquitetura é denominada *encaminhamento por difusão / gestão centralizada* e tem muito em comum com a primeira. As principais diferenças são que não existe a *gateway* e que a *firewall* à entrada é substituída por uma *firewall* em cada servidor (figura 3.12). Nesta arquitetura cada servidor decide quais os pedidos que processa, p. ex. de acordo com uma política de balanceamento de carga.

Estas duas primeiras arquiteturas têm o inconveniente de poderem perder pedidos devido ao encaminhamento de mensagens para servidores com intrusões ainda não detectadas e removidas. No entanto, só são necessários $f + 1$ servidores e os pedidos são processados em paralelo o que, em princípio, permite obter melhor débito (pedidos processados por unidade de tempo).

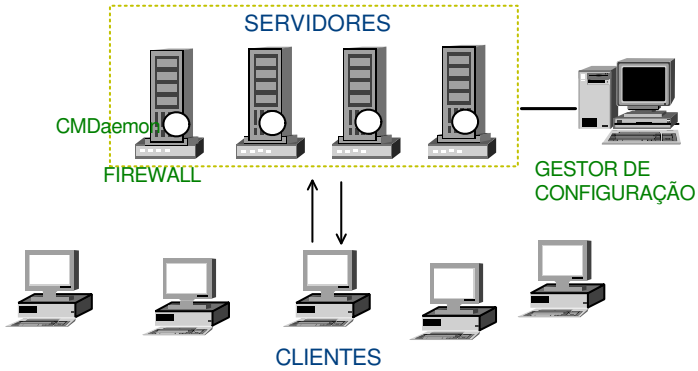

Figura 3.12. Arquitetura encaminhamento por difusão / gestão centralizada [Gupta et al., 2003].

A terceira arquitetura é chamada de *encaminhamento por difusão / gestão descentralizada* e difere da anterior pela não existência de um gestor de configuração (fig. 3.13). Esta arquitetura procura combinar o bom desempenho das duas anteriores com redundân-

cia de todos os componentes. Cada pedido é processado por apenas um dos servidores, mas os CMDaemons detectam intrusões e chegam a acordo sobre as reconfigurações que seja necessário realizar. A base desse mecanismo pode ser um sistema de comunicação em grupo, como o Rampart, sendo necessários $3f + 1$ servidores.

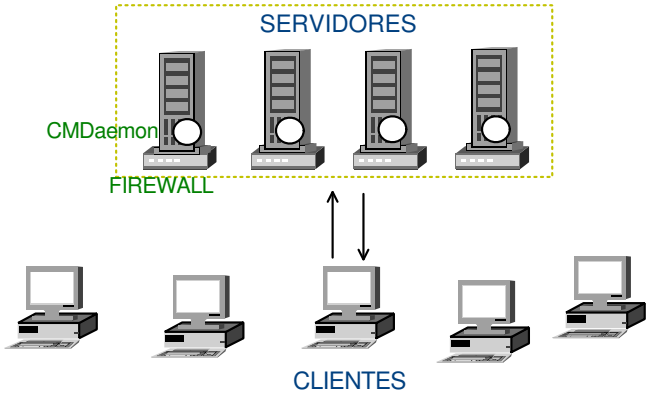

Figura 3.13. Arquitetura encaminhamento por difusão / gestão descentralizada [Gupta et al., 2003].

3.6.2. Arquitetura com *firewall* de privacidade

Voltemos à arquitetura que faz a separação entre servidores de execução e servidores de acordo. O artigo desta última arquitetura, apresenta uma versão mais complicada que não apresentamos na seção 3.6 por ser ortogonal ao tema aí discutido [Yin et al., 2003]. Esta outra arquitetura, que inclui uma *firewall de privacidade*, é apresentada na figura 3.14.

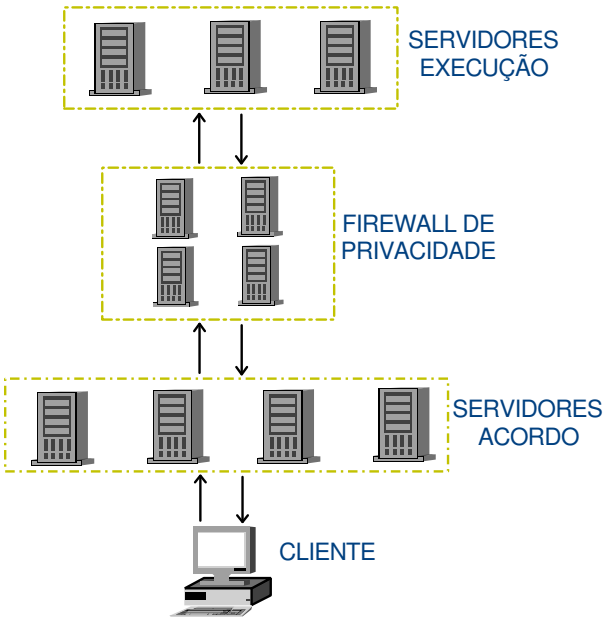

Figura 3.14. Arquitetura com separação execução-acordo e *firewall* de privacidade [Yin et al., 2003].

Como vimos atrás, um serviço baseado em replicação de máquinas de estados como o BFT não garante a confidencialidade da informação nos servidores pois um servi-

dor no qual tenha ocorrido uma intrusão pode difundir informação que aí se encontre. A *firewall* de privacidade pretende resolver este problema. Caso haja uma intrusão num *servidor de execução*, o componente filtra a informação que daí sai, garantindo assim a confidencialidade – ou privacidade caso se trate de informação pessoal. Só respostas bem formadas são deixadas passar por esse filtro. A idéia consiste em que a *firewall* tenha $h+1$ níveis, cada um com $h+1$ réplicas. Assim, é possível tolerar h intrusões nessas réplicas mantendo a disponibilidade, integridade e confidencialidade. Em outras palavras, são necessárias intrusões em $h+1$ dessas réplicas (e uma nos servidores de execução) para que o atacante possa obter informação confidencial.

3.6.3. Arquitetura SITAR

No âmbito do programa OASIS foi desenvolvida uma arquitetura abstrata denominada SITAR (*Scalable Intrusion-Tolerant Architecture for Distributed Systems*) com o objetivo de servir de base a serviços distribuídos TI [Wang et al., 2001]. A arquitetura pretende ser passível de ser introduzida entre clientes e servidores COTS (*commercial of-the-shelf*) sem que estes sejam modificados (v. fig. 3.15).

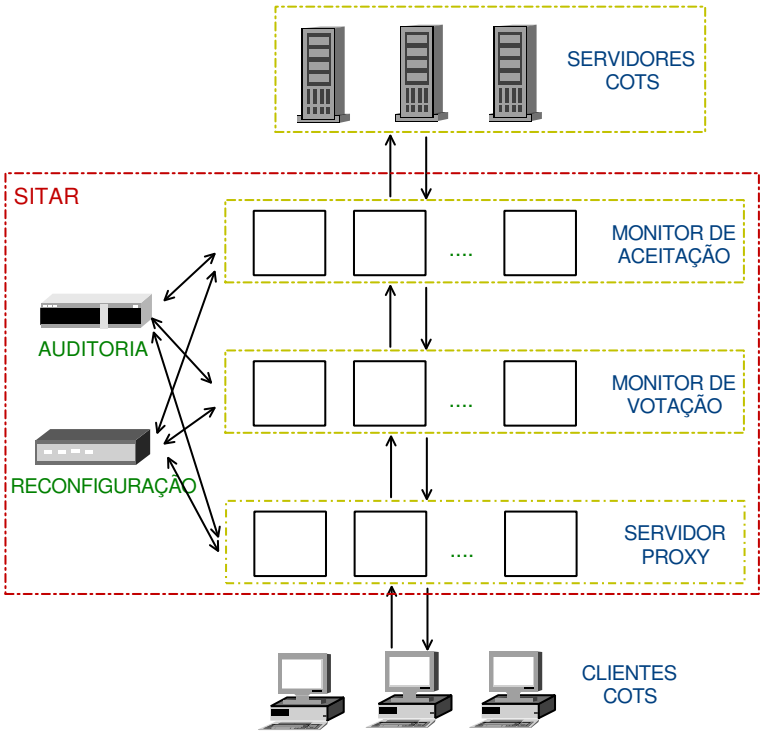

Figura 3.15. A arquitetura SITAR [Wang et al., 2001].

A arquitetura SITAR define um conjunto de módulos que podem ser instanciados de diferentes formas. O objetivo do servidor proxy é o de representar os servidores COTS perante os clientes COTS quando estes fazem pedidos. O monitor de aceitação e o monitor de votação, pelo contrário, atuam quando o servidores COTS respondem a um cliente. O monitor de aceitação faz algumas verificações de validade às respostas usando regras dependentes de cada serviço e faz detecção de intrusões nos servidores COTS. O monitor de votação recebe as respostas vindas do monitor de aceitação e faz votação para mascarar

intrusões. As respostas escolhidas são enviadas ao proxy que as reencaminha para os clientes.

3.6.4. Arquitetura DPASA

Quando o programa OASIS terminou, o DARPA criou um novo programa chamado OASIS Dem/Val para demonstrar e avaliar algumas das tecnologias aí pesquisadas. Nesse programa foi desenvolvida a arquitetura DPASA (*Designing Protection and Adaptation into a Survivability Architecture*) com o objetivo de demonstrar a TI num sistema real. A arquitetura DPASA é provavelmente a arquitetura de TI mais complexa e interessante desenvolvida até agora.

Os textos disponíveis sobre a arquitetura DPASA não descrevem propriamente a arquitetura mas antes a sua utilização numa aplicação que está a ser desenvolvida pela força aérea americana, a *Joint Battlespace Infosphere (JBI)* [Stevens et al., 2004, Atighetchi et al., 2005]. O objetivo de uma JBI é fornecer serviços de publicação-subscrição-interrogação (*publish-subscribe-query, PSQ*) a clientes de forma a que estes possam trocar informação sob a forma de objetos de informação (OIs).

O sistema IT-JBI – a versão TI da JBI – consiste num núcleo central que fornece serviços de comunicação a um conjunto de clientes. Uma versão simplificada da arquitetura do sistema encontra-se na figura 3.16.

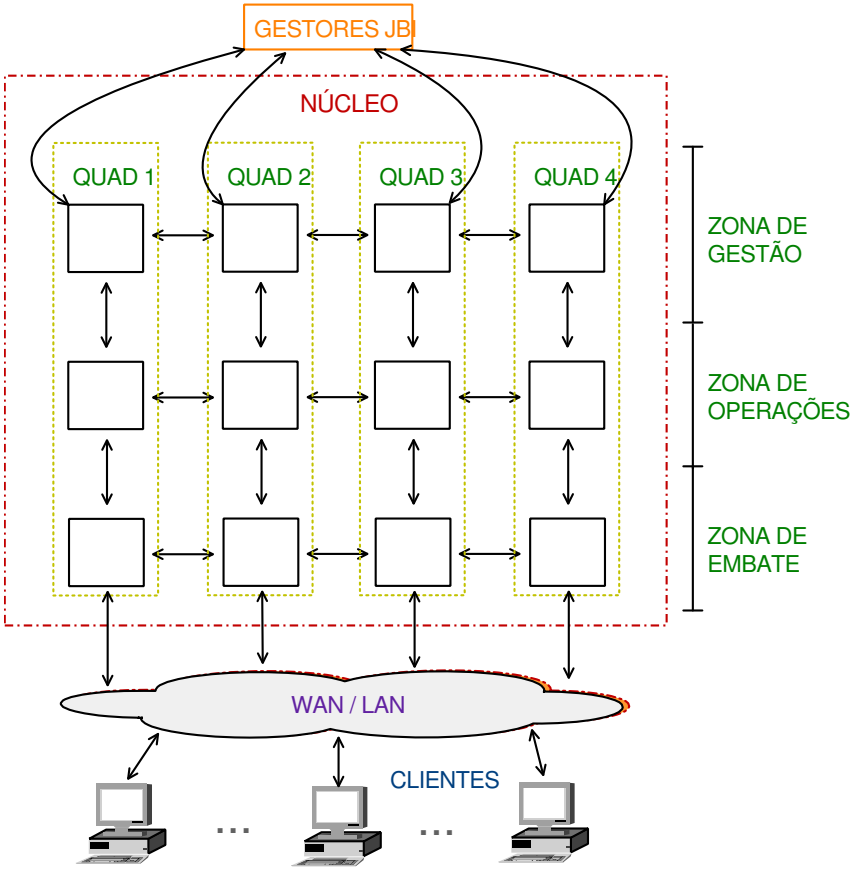

Figura 3.16. O sistema IT-JBI baseado na arquitetura DPASA [Stevens et al., 2004].

O núcleo é replicado em quatro quadrantes (*quad 1* a *quad 4*). As máquinas dos quatro quadrantes correm sistemas operacionais diferentes: SELinux, Solaris e Windows XP e 2000. Os adaptadores de rede das máquinas têm uma *firewall embutida* [Markham et al., 2003] que faz filtragem de pacotes e suporta redes privadas virtuais (VPNs) com máquinas com adaptadores do mesmo tipo.

O núcleo está também dividido em três zonas (v. figura). A *zona de embate* é a zona onde se dá o primeiro impato de um potencial ataque. A *zona de operações* é onde se encontra a concretização do serviço de publicação-subscrição-interrogação. A *zona de gestão* contém funções de gestão e controle do serviço. A comunicação entre zonas é controlada através de *switches* geridos e através de VPNs suportadas pelos adaptadores de rede.

A zona de embate contém um *proxy* de acesso (AP). O seu papel é fundamentalmente fazer de interface entre os protocolos e *middleware* de comunicação usados pelos clientes (p. ex. RMI, CORBA) e o JBI. A comunicação entre os clientes e os APs é feita através de VPNs de forma a evitar que o tráfego seja escutado na rede. Os APs têm dois adaptadores de rede. O adaptador exterior só recebe ligações vindas de clientes legítimos através de uma VPN e limita as portas de comunicação UDP e TCP. O adaptador interior serve para comunicar com os servidores da zona de operações.

A zona de operações contém os componentes que desempenham as principais funções do núcleo: os serviços de PSQ e a supervisão da segurança do núcleo e dos clientes a ele ligados. O processamento dos OIs é feito por três componentes desta zona: o servidor PSQ, o controlador do sentido descendente e o guardião. O papel do primeiro é evidente. O objetivo do segundo é verificar as assinaturas dos pedidos enviados pelos clientes para garantir a integridade desses mesmos pedidos. O guardião usa informação específica da aplicação para detectar OIs corrompidos.

A zona de gestão contém um componente que correlaciona a informação sobre intrusões obtida por sensores colocados nos clientes e nas zonas de tampão e operações. Essa informação é filtrada de forma a serem descartados alarmes falsos ou redundantes com base no contexto do sistema. Os alarmes importantes são enviados para outro componente da mesma zona denominada gestor de sistemas, SM. Este componente gera respostas às intrusões. O funcionamento do IT-JBI é supervisionado por gestores humanos (v. topo da figura).

O sistema IT-JBI foi testado no âmbito do programa OASIS Dem/Val por duas *red teams* com conhecimento total do sistema. O sistema resistiu bem aos ataques infligidos, exceto dois ataques de negação de serviço que o conseguiram bloquear temporariamente.

3.7. Problemas abertos

Depois do que tudo o que foi dito até agora, fica claro que a *tolerância a intrusões* já atingiu uma certa maturidade. Dos inúmeros exemplos que vimos é óbvio que já é possível construir *sistemas distribuídos tolerantes a intrusões* que ofereçam diversas propriedades e graus de segurança. Se aquilo que se pretende proteger for suficientemente valioso, já é possível criar um sistema de torreões dificilmente controlável pelo nosso pirata!

Apesar disso, a área da *tolerância a intrusões* está longe de estar fechada. Há

ainda um conjunto de problemas não triviais com os quais o arquiteto de sistemas tem de se defrontar. Esta seção irá listar os principais, dando pistas sobre a pesquisa que precisa de ser feita para resolver cada um deles.

3.7.1. Independência de faltas e diversidade

Todas as soluções de replicação e de fragmentação que vimos ao longo do capítulo partem sempre da mesma hipótese: é significativamente mais difícil penetrar em *m* servidores do que em um. Se o grau de dificuldade fosse idêntico, não valia a pena replicar ou fragmentar pois o atacante poderia tomar todos os servidores em vez de um só. Em outras palavras, tem de existir independência de faltas.

Quando se fala de intrusões o problema de garantir independência de faltas não é simples. Como vimos no modelo AVI, que caracteriza as faltas no domínio da *segurança* (fig. 3.2), uma intrusão é causada por um ou mais ataques que exploram com sucesso uma ou mais vulnerabilidades. Claramente a independência de faltas tem de ser garantida do lado das vulnerabilidades, já que é arriscado levantar hipóteses sobre os ataques. No entanto, se há algo que caracteriza as vulnerabilidades é que elas são problemas desconhecidos de um sistema computacional (salvo raras e desonrosas exceções). Logo, como vamos garantir que *N* servidores não sofrem do mesmo problema se nem sabemos do que é que sofre cada um deles?

Uma solução intuitiva é o recurso à *diversidade*. É evidente que os servidores devem ter sistemas operacionais diferentes, concretizações diferentes do código do sistema, etc. Este tipo de abordagens é usada em *tolerância a faltas*, p. ex. através do recurso à *programação com n versões* [Avizienis, 1985]. Também em *segurança* há algum debate sobre o tema, p. ex. em relação à monocultura MS-Windows na Internet e como esta favorece a difusão de código nocivo.

Diversos níveis de diversidade são sugeridos em [Deswarte et al., 1998, Castro and Liskov, 2002] e num estudo mais aprofundado sobre o tema apresentado em [Obelheiro et al., 2005]:

- diversidade de software da aplicação;
- diversidade de software de suporte à execução (sistema operacional, *middleware*, bases de dados, . . .);
- diversidade de hardware;
- diversidade de administradores dos servidores;
- diversidade de localização.

O significado dos diferentes níveis é bastante evidente. A diversidade do software da aplicação pode ser conseguida usando a já referida *programação com n versões*. A idéia consiste em ter *N* equipes de programadores que desenvolvem em paralelo versões de software com a mesma funcionalidade, uma para cada um dos servidores. Apesar do custo elevado que isso representa, em geral o custo de cada versão pode rondar apenas 0,7

a 0,85 o custo de uma versão de software normal, devido aos custos que não se multiplicam (estabelecimento de requisitos, modelagem do sistema) [Deswarte et al., 1998].

A conjugação de todos estes níveis de diversidade com técnicas de análise estática de código [Viega and McGraw, 2002] podem levar a níveis de confiança elevados mas qual é a independência de faltas que essa combinação realmente dá? Como garantir essa independência com determinado nível de confiança?

Outro tema interessante que merece ser pesquisado é a criação automática de diversidade. Será que é possível criar automaticamente diversas versões de software, possivelmente para diversos sistemas operacionais, com independência de vulnerabilidades?

Esta forma de diversidade entre as diversas réplicas apesar de tudo é simples quando comparada com outro tipo de diversidade desejável para a *recuperação proativa*: a diversidade num mesmo servidor após cada recuperação. A recuperação proativa parte do princípio que atacar cada servidor toma um determinado tempo mas isso não é bem verdade. O trabalho demorado é descobrir como se pode atacar um servidor; o processo de ataque em si geralmente é rápido se forem usado ferramentas automáticas (p. ex. um ataque *buffer overflow* demora . . .). Se um atacante ao fim de alguns dias descobrir como atacar $f + 1$ servidores e o ataque demorar alguns milisegundos, a recuperação proativa é inútil (o período típico de recuperação ronda um minuto). Uma solução seria que cada recuperação modificasse o servidor de tal forma que o atacante tivesse que recomeçar o processo de compreender como atacá-lo. Como conseguir isso?

3.7.2. Determinismo

A diversidade das réplicas no caso da *replicação de máquinas de estados* conduz a um problema complexo: o seu determinismo. Na seção 3.3.1 referimos a necessidade de que os servidores fossem determinísticos, ou seja, que o mesmo comando executado no mesmo estado inicial gerasse o mesmo estado final independentemente do servidor onde fosse executado. O problema é mais simples no caso dos sistemas de quoruns ou da fragmentação já que os servidores não executam comandos genéricos mas sobretudo armazenam dados em variáveis.

Uma experiência de concretização de um serviço distribuído TI relativamente simples mostrou como o problema do determinismo é particularmente relevante [Ferraz et al., 2004]. A idéia era concretizar um servidor de web TI usando replicação de máquinas de estados. As páginas consideradas estavam em HTML (não foi usado PHP ou ASPs) e não foi considerado o uso de HTTP-S, apenas HTTP. As réplicas eram idênticas (Linux e Apache). Mesmo com todas estas restrições o cabeçalho do HTTP tem diversos campos que não são determinísticos. Três campos geraram problemas: uma estampilha temporal com a data e a hora em que a página é retornada; um campo *Etag* que identifica univocamente a resposta ao pedido HTML; e um identificador do servidor. Muitos outros parâmetros podem gerar falta de determinismo, desde a representação de números reais às respostas a erros, passando por todo o tipo de operações que dependam do tempo.

Um sistema que pretende resolver este problema é uma evolução do BFT chamada BASE (*BFT with Abstract Specification Encapsulation*) [Castro et al., 2003]. O BASE

permite o uso de réplicas diferentes ou que não sejam deterministas. A solução consiste em fazer uma especificação abstrata do serviço, com um estado abstrato e um conjunto de operações que manipulam esse estado. Depois, concretiza-se um *wrapper* de conformidade que esconde cada réplica atrás de uma interface conforme com a especificação abstrata do serviço. Finalmente, caso seja necessário fazer transferência de estado entre réplicas, como no BFT, define-se uma função de abstração que traduz o estado de uma concretização para o estado abstrato e vice-versa.

O BASE resolve parte do problema mas deixa em aberto diversas questões. Como é que se lida com dados com estrutura complexa? E com dados comprimidos? E a principal questão: dado que em sistemas seguros muitas vezes é preciso cifrar a comunicação, como é que se lida com dados cifrados?

3.7.3. Detecção e remoção de intrusões

A detecção de intrusões pode ser considerada como fazendo parte da TI, já que é uma forma de *detecção de faltas* (seção 3.2.1). No entanto, a pesquisa em *sistemas de detecção de intrusões* é uma área com mérito reconhecido e que seria injusto reduzir a uma parte da TI.

Diversos sistemas TI têm englobado detecção de intrusões em soluções baseadas em replicação ou fragmentação, por exemplo, [Knight et al., 2001, Keromytis et al., 2003, Atighetchi et al., 2005]. No entanto os sistemas de detecção de intrusões continuam a ter uma elevada taxa de falsos positivos e falsos negativos o que na prática impede a resposta automática a intrusões. Mas se reduzirmos as intrusões que se pretendem detectar às que podem ocorrer num determinado tipo específico de serviço TI, não será possível fazer essa detecção com precisão suficiente para fazer resposta automática?

3.7.4. Avaliação

Em *sistemas distribuídos* existem diversas métricas que permitem comparar dois algoritmos que resolvem o mesmo problema. As métricas mais usadas são o número (ou a complexidade) de ciclos de comunicação e o número (ou a complexidade) de mensagens trocadas. As comparações podem ser matizadas (*o algoritmo X é melhor do que o Y no caso Z mas não no caso W*) mas em geral essas métricas permitem avaliar algoritmos sem ambiguidade.

Em *confiança no funcionamento* foram também desenvolvidos diversos métodos para avaliar a tolerância a faltas de sistemas e algoritmos. Um exemplo são os métodos estocásticos que permitem determinar a probabilidade de um sistema falhar dada a probabilidade de ocorrerem faltas em certos componentes e dessas faltas se propagarem [Nicol et al., 2004].

Em *tolerância a intrusões* é igualmente importante avaliar a segurança de sistemas e algoritmos. Alguns trabalhos recentes têm feito esse tipo de análise usando métodos estocásticos [Gupta et al., 2003, Singh et al., 2003, Stevens et al., 2004] e redes de Petri [Wang et al., 2003]. No entanto, essas avaliações assumem comportamentos probabilísticos dos ataques e intrusões o que não parece ser realista. O ideal seria existirem métricas simples que permitissem avaliar esses sistemas como é feito com os algoritmos distribuídos.

3.7.5. Negação de serviço e privacidade

Os ataques de negação de serviço (DoS) e as questões de privacidade em sistemas TI têm sido pouco tratadas na bibliografia e são geralmente importantes em sistemas reais.

Os ataques DoS ganharam fama mundial com o grande ataque de Fevereiro de 2000 que deixou serviços como o Yahoo, CNN e eBay inoperantes durante horas. Estes ataques consistiram em usar aquilo que hoje se chamaria uma *botnet* para bombardear esses serviços com tráfego. O problema é tanto mais grave quanto maior for o processamento feito por cada pedido. No caso dos serviços TI é evidente que cada pedido exige algum processamento: os pedidos são executados por vários servidores, são feitas diversas operações criptográficas, ...

Como lidar com ataques DoS em serviços TI? Uma solução baseada numa rede *overlay* é usada no sistema SABER [Keromytis et al., 2003]. A idéia consiste em usar a porção da rede que cerca o servidor para filtrar agressivamente o tráfego que lhe é dirigido com base no endereço de origem. Em cada momento existe um número limitado de endereços de origem aceites, de que só os clientes válidos devem ter conhecimento. A idéia é interessante mas parece necessitar de um número elevado de nós na rede *overlay*. Também o sistema COCA procura lidar com este tipo de ataque usando um conjunto de mecanismos bastante simples [Zhou et al., 2002a]. O que nenhum desses trabalhos considera é a relação entre serviços que têm por objetivo garantir, entre outras propriedades, a disponibilidade, e ataques realizados precisamente contra essa disponibilidade, que é o que constituem os ataques DoS. Será que não seria possível aproveitar as próprias características dos serviços distribuídos TI para combater esses ataques? Qual a arquitetura adequada a esse fim?

A questão da privacidade não tem sido tratada nos trabalhos sobre serviços TI com a exceção de [Yin et al., 2003]. É evidente que as soluções que permitem obter confidencialidade (seção 3.4) também contribuem para a privacidade de dados pessoais aí armazenados. No entanto, a privacidade face ao próprio administrador do serviço não é garantida. Soluções?

3.8. Conclusão

Uma vez terminada a exposição sobre *serviços distribuídos tolerantes a intrusões* – conceitos básicos, mecanismos, arquiteturas, problemas abertos – é interessante opinar sobre a utilização que será dada a estes resultados de vários anos de pesquisa. A questão não é certamente a necessidade de serviços seguros, que é evidente. O que nos podemos perguntar é se (1) a TI permite obter essa segurança, (2) a TI está pronta para ser usada já hoje, e (3) em que tipo de aplicações será realmente usada.

A resposta à primeira questão é positiva. Obviamente, como tão bem ilustra o sistema IT-JBI, uma arquitetura TI completa inclui numerosos componentes e mecanismos que vêm da *segurança* clássica, não são nenhuma novidade da TI. No entanto a possibilidade de multiplicar por N (ou $f+1$ ou ...) a dificuldade de atacar e controlar um serviço traz claramente segurança acrescida.

Quanto à segunda questão, a resposta é também positiva. Existem lacunas que podem ser pesquisadas e a TI tornada muito mais prática, mas as soluções que vimos são

reais e utilizáveis já hoje.

Algumas pistas sobre a terceira questão podem vir da forma como a *tolerância a faltas* é usada hoje em dia. Muitas soluções complexas e caras são usadas em aplicações que justificam o seu custo, p. ex. por envolverem risco de vidas humanas. Exemplos incluem a aviação, a indústria espacial e centrais de produção de energia. No entanto, algumas soluções tolerantes a faltas mais simples estão disponíveis um pouco por toda a parte: desde cabos de telecomunicações replicados a soluções tolerantes a faltas em discos rígidos (RAID), passando pela detecção de erros na memória dos PCs comuns.

Em relação à *tolerância a intrusões* podemos esperar que algo de semelhante aconteça. Algumas soluções complexas e caras serão certamente usadas em aplicações críticas, nem que o sejam só sob o ponto de vista financeiro. Exemplos que ocorrem são a rede SWIFT, que interliga mais de 7000 bancos de todo o mundo e que movimenta triliões de euros todos os dias, ou aplicações militares como o IT-JBI. Na realidade, já há alguns anos que a chave de raiz do sistema SET da MasterCard/VISA se encontrava distribuída por diversas empresas diferentes usando criptografia de limiar [Frankel and Yung, 1998]. Além disso, muitas soluções simples poderão ser usadas para tornar mais seguros componentes de sistemas de menor custo. Por exemplo, com o sucesso das soluções de armazenamento de dados em rede (NAS, SAN) poderão surgir produtos comerciais que forneçam disponibilidade, integridade e confidencialidade sem grande desperdício de espaço de disco graças ao uso de códigos de apagamento.

A *tolerância a intrusões* não é certamente a panaceia para todos os problemas de segurança dos sistemas computacionais do nosso tempo. No entanto fornece um conjunto de soluções válidas que podem certamente contribuir para a segurança de algumas fortalezas nesse mundo dominado por piratas em que se tornou a Internet.

Agradecimentos

Este capítulo surge após seis anos de pesquisa em *tolerância a intrusões*. Durante esse tempo trabalhei sempre com duas pessoas a quem estou agradecido por muitas razões: Paulo Veríssimo e Nuno Ferreira Neves. Trabalhei também com vários outros colegas dos quais não posso deixar de mencionar um: Lau Cheuk Lung. O Alysson Bessani Neves e o Paulo Sousa tiveram a paciência de fazer uma revisão do manuscrito e dar muitas sugestões de melhoramentos, fato pelo qual estou muito agradecido.

Referências

[Adelsbach et al., 2002] Adelsbach, A., Alessandri, D., Cachin, C., Creese, S., Deswarte, Y., Kursawe, K., Laprie, J. C., Powell, D., Randell, B., Riordan, J., Ryan, P., Simmonds, W., Stroud, R., Veríssimo, P., Waidner, M., and Wespi, A. (2002). *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21*.

[Alon et al., 2000] Alon, N., Kaplan, H., Krivelevich, M., Malkhi, D., and Stern, J. (2000). Scalable secure storage when half the system is faulty. In Montanari, U., Rolim, J., and Welzl, R., editors, *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 576–587. Springer-Verlag.

[Atighetchi et al., 2005] Atighetchi, M., Rubel, P., Pal, P., Chong, J., and Studin, L. (2005). Case study: The intrusion tolerant JBI. Technical report, BBN Technologies.

[Avizienis, 1985] Avizienis, A. (1985). The N-version approach to fault tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[Baldoni et al., 2000] Baldoni, R., Helary, J., Raynal, M., and Tanguy, L. (2000). Consensus in Byzantine asynchronous systems. In *Proceedings of the International Colloquium on Structural Information and Communication Complexity*, pages 1–16.

[Bazzi and Ding, 2004] Bazzi, R. and Ding, Y. (2004). Non-skipping timestamps for Byzantine data storage systems. In Guerraoui, R., editor, *Proceedings of the 18th International Conference on Distributed Computing*, volume 3274 of *Lecture Notes in Computer Science*, pages 405–419. Springer-Verlag.

[Ben-Or, 1983] Ben-Or, M. (1983). Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30.

[Bessani et al., 2005] Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2005). O confeiteiro bizantino: Exclusão mútua em sistemas abertos sujeitos a faltas bizantinas. In *Anais do 23o. Simpósio Brasileiro de Redes de Computadores*.

[Blakley, 1979] Blakley, G. R. (1979). Safeguarding cryptographic keys. In *Proceedings of the AFIPS National Computer Conference*, volume 48, pages 313–317.

[Bracha, 1984] Bracha, G. (1984). An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162.

[Bracha and Toueg, 1985] Bracha, G. and Toueg, S. (1985). Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840.

[Cachin, 2002] Cachin, C. (2002). Personal communication.

[Cachin et al., 2002] Cachin, C., Kursawe, K., Lysyanskaya, A., and Strobl, R. (2002). Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97.

[Cachin et al., 2001] Cachin, C., Kursawe, K., Petzold, F., and Shoup, V. (2001). Secure and efficient asynchronous broadcast protocols (extended abstract). In Kilian, J., editor, *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer-Verlag.

[Cachin et al., 2000] Cachin, C., Kursawe, K., and Shoup, V. (2000). Random oracles in Contanstinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 123–132.

[Cachin and Poritz, 2002] Cachin, C. and Poritz, J. A. (2002). Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 167–176.

[Cachin and Tessaro, 2004] Cachin, C. and Tessaro, S. (2004). Asynchronous verifiable information dispersal. RZ 3569, IBM Research.

[Cachin and Tessaro, 2005] Cachin, C. and Tessaro, S. (2005). Optimal resilience for erasure-coded Byzantine distributed storage. RZ 3575, IBM Research.

[Canetti et al., 1997] Canetti, R., Gennaro, R., Herzberg, A., and Naor, D. (1997). Proactive security: Long-term protection against break-ins. *RSA CryptoBytes*, 3(1):1–8.

[Castro and Liskov, 2002] Castro, M. and Liskov, B. (2002). Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461.

[Castro et al., 2003] Castro, M., Rodrigues, R., and Liskov, B. (2003). BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269.

[CERT/CC, 2005] CERT/CC (2005). CERT coordination center statistics 1988-2005. http://www.cert.org/stats/.

[Chandra and Toueg, 1996] Chandra, T. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

[Correia et al., 2002a] Correia, M., Lung, L. C., Neves, N. F., and Veríssimo, P. (2002a). Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 2–11.

[Correia et al., 2005a] Correia, M., Neves, N. F., Lung, L. C., and Veríssimo, P. (2005a). Low complexity Byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249.

[Correia et al., 2005b] Correia, M., Neves, N. F., Lung, L. C., and Veríssimo, P. (2005b). Worm-IT – a wormhole-based intrusion-tolerant group communication system. Submitted for publication.

[Correia et al., 2004] Correia, M., Neves, N. F., and Veríssimo, P. (2004). How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 174–183.

[Correia et al., 2002b] Correia, M., Veríssimo, P., and Neves, N. F. (2002b). The design of a COTS real-time distributed security kernel. In *Proceedings of the Fourth European Dependable Computing Conference*, pages 234–252.

[Deswarte et al., 1991] Deswarte, Y., Blain, L., and Fabre, J. C. (1991). Intrusion tolerance in distributed computing systems. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 110–121.

[Deswarte et al., 1998] Deswarte, Y., Kanoun, K., and Laprie, J. C. (1998). Diversity against accidental and deliberate faults. In *Computer Security, Dependability, & Assurance: From Needs to Solutions*. IEEE Press.

[Diffie and Hellman, 1976] Diffie, W. and Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654.

[Doudou et al., 2002] Doudou, A., Garbinato, B., and Guerraoui, R. (2002). Encapsulating failure detection: From crash-stop to Byzantine failures. In *International Conference on Reliable Software Technologies*, pages 24–50.

[Doudou and Schiper, 1997] Doudou, A. and Schiper, A. (1997). Muteness detectors for consensus with Byzantine processes. Technical Report 97/30, EPFL.

[Dwork et al., 1988] Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323.

[Ferraz et al., 2004] Ferraz, R., Goncalves, B., Sequeira, J., Correia, M., Neves, N. F., and Veríssimo, P. (2004). An intrusion-tolerant web server based on the DISTRACT architecture. In *Proceedings of the Workshop on Dependable Distributed Data Management*.

[Fischer et al., 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

[Fraga and Powell, 1985] Fraga, J. S. and Powell, D. (1985). A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218.

[Frankel and Yung, 1998] Frankel, Y. and Yung, M. (1998). Risk management using theshold RSA cryptosystems. *Usenix ;login: online*.

[Garay et al., 2000] Garay, J. A., Gennaro, R., Jutla, C., and Rabin, T. (2000). Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389.

[Gemmell, 1997] Gemmell, P. S. (1997). An introduction to threshold cryptography. *Cryptobytes*, 2(3):7–12.

[Gifford, 1979] Gifford, D. K. (1979). Weighted voting for replicated data. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 150–162.

[Goodson et al., 2004] Goodson, G., Wylie, J., Ganger, G., and Reiter, M. (2004). Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*.

[Gupta et al., 2003] Gupta, V., Lam, V., Ramasamy, H., Sanders, W., and Singh, S. (2003). Dependability and performance evaluation of intrusion-tolerant server architectures. In *Proceedings of the First Latin-American Symposium on Dependable Computing*, pages 81–101.

[Hadzilacos and Toueg, 1994] Hadzilacos, V. and Toueg, S. (1994). A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science.

[Herlihy and Wing, 1990] Herlihy, M. P. and Wing, J. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492.

[Keromytis et al., 2003] Keromytis, A., Gross, P., Kaiser, G., Misra, V., Nieh, J., Rubenstein, D., and Stolfo, S. (2003). A holistic approach to service survivability. In *Proceedings of the ACM Workshop on Survivable and Self-Regenerative Systems*, pages 11–22.

[Kihlstrom et al., 2001] Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. (2001). The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406.

[Kihlstrom et al., 2003] Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. (2003). Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35.

[Knight et al., 2001] Knight, J., Heimbigner, D., Wolf, A., Carzaniga, A., Hill, J., and Devanbu, P. (2001). The Willow survivability architecture. In *Proceedings of the 4th Information Survivability Workshop*.

[Krawczyk, 1993] Krawczyk, H. (1993). Distributed fingerprints and secure information dispersal. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 207–218.

[Lakshmanan et al., 2003] Lakshmanan, S., Ahamad, M., and Venkateswaran, H. (2003). Responsive security for stored data. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):818–828.

[Lala, 2003] Lala, J. H., editor (2003). *Foundations of Intrusion Tolerant Systems*. IEEE Computer Society Press.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[Lamport, 1986] Lamport, L. (1986). On interprocess communication (part II: Algorithms). *Distributed Computing*, 1:86–101.

[Lamport et al., 1982] Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

[Lippmann et al., 2000] Lippmann, R., Haines, J., Fried, D., Korba, J., and Das, K. (2000). Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In Debar, H., Mé, L., and Wu, S. F., editors, *Recent Advances in Intrusion Detection - Third International Workshop*, volume 1907 of *Lecture Notes in Computer Science*, pages 162–182. Springer-Verlag.

[Malkhi and Reiter, 1997a] Malkhi, D. and Reiter, M. (1997a). Byzantine quorum systems. In *Proceedings of the 29th ACM Symposium in Theory of Computing*, pages 569–578.

[Malkhi and Reiter, 1997b] Malkhi, D. and Reiter, M. (1997b). Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 116–124.

[Malkhi and Reiter, 1998a] Malkhi, D. and Reiter, M. (1998a). Byzantine quorum systems. *Distributed Computing*, 11:203–213.

[Malkhi and Reiter, 1998b] Malkhi, D. and Reiter, M. (1998b). Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*.

[Malkhi et al., 1997] Malkhi, D., Reiter, M., and Wool, A. (1997). The load and availability of Byzantine quorum systems. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 249–257.

[Malkhi and Reiter, 2000] Malkhi, D. and Reiter, M. K. (2000). An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202.

[Markham et al., 2003] Markham, T., Meredith, L., and Payne, C. (2003). Distributed embedded firewalls with virtual private groups. In *DARPA Information Survivability Conference and Exposition - Volume II*.

[Marsh and Schneider, 2004] Marsh, M. A. and Schneider, F. B. (2004). CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47.

[Martin and Alvisi, 2004] Martin, J. P. and Alvisi, L. (2004). A framework for dynamic Byzantine storage. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 325–334.

[Martin and Alvisi, 2005] Martin, J. P. and Alvisi, L. (2005). Fast Byzantine consensus. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*.

[Martin et al., 2002a] Martin, J. P., Alvisi, L., and Dahlin, M. (2002a). Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, volume 2508 of *LNCS*, pages 311–325. Springer-Verlag.

[Martin et al., 2002b] Martin, J. P., Alvisi, L., and Dahlin, M. (2002b). Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 374–383.

[Menezes et al., 1997] Menezes, A. J., Oorschot, P. C. V., and Vanstone, S. A. (1997). *Handbook of Applied Cryptography*. CRC Press.

[Moser and Melliar-Smith, 1999] Moser, L. E. and Melliar-Smith, P. M. (1999). Byzantine-resistant total ordering algorithms. *Information and Computation*, 150:75–111.

[Moser et al., 2000] Moser, L. E., Melliar-Smith, P. M., and Narasimhan, N. (2000). The SecureGroup communication system. In *Proceedings of the IEEE Information Survivability Conference*, pages 507–516.

[Naor and Wool, 1996] Naor, M. and Wool, A. (1996). Access control and signatures via quorum secret sharing. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 157–168.

[Neves et al., 2005] Neves, N. F., Correia, M., and Veríssimo, P. (2005). Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*. Accepted for publication.

[Nicol et al., 2004] Nicol, D. M., Sanders, W. H., and Trivedi, K. S. (2004). Model-based evaluation: From dependability to security. *IEEE Transactions on Dependable and Secure Computing*, 1(1):48–65.

[Obelheiro et al., 2005] Obelheiro, R. R., Bessani, A. N., Fraga, J. S., and Lung, L. C. (2005). Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In *Anais do 5o Simpósio Brasileiro de Segurança*.

[Ostrovsky and Yung, 1991] Ostrovsky, R. and Yung, M. (1991). How to withstand mobile virus attacks. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 51–59.

[Rabin, 1983] Rabin, M. O. (1983). Randomized Byzantine generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409.

[Rabin, 1989] Rabin, M. O. (1989). Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348.

[Ramasamy et al., 2002] Ramasamy, H., Pandey, P., Lyons, J., Cukier, M., and Sanders, W. H. (2002). Quantifying the cost of providing intrusion tolerance in group communication systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 229–238.

[Reiter, 1994] Reiter, M. (1994). Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80.

[Reiter, 1995] Reiter, M. K. (1995). The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. M. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

[Schneider, 1990] Schneider, F. B. (1990). Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

[Shamir, 1979] Shamir, A. (1979). How to share a secret. *Communications of the ACM*, 22(1):612–613.

[Singh et al., 2003] Singh, S., Cukier, M., and Sanders, W. H. (2003). Probabilistic validation of an intrusion-tolerant replication system. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 615–624.

[Sousa et al., 2005a] Sousa, P., Neves, N. F., and Verissimo, P. (2005a). How resilient are distributed f fault/intrusion-tolerant systems? In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*.

[Sousa et al., 2005b] Sousa, P., Neves, N. F., and Veríssimo, P. (2005b). Proactive resilience through architectural hybridization. DI/FCUL TR 05–8, Department of Informatics, University of Lisbon.

[Stevens et al., 2004] Stevens, F., Courtney, T., Singh, S., Agbaria, A., Meyer, J. F., Sanders, W. H., and Pal, P. (2004). Model-based validation of an intrusion-tolerant information system. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 184–194.

[Turner et al., 2004] Turner, D., Entwisle, S., Friedrichs, O., Hanson, D., Fossi, M., Ahmad, D., Gordon, S., Szor, P., and Chien, E. (2004). Symantec Internet security threat report. Trends for January 1, 2004 – June 30, 2004. Volume VI.

[Veríssimo, 2003] Veríssimo, P. (2003). Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 108–113. Springer-Verlag.

[Veríssimo and Casimiro, 2002] Veríssimo, P. and Casimiro, A. (2002). The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930.

[Veríssimo and de Lemos, 1989] Veríssimo, P. and de Lemos, R. (1989). Confiança no funcionamento: Proposta para uma terminologia em português. Technical Report RT48-89, INESC.

[Veríssimo et al., 2000] Veríssimo, P., Neves, N. F., and Correia, M. (2000). The middleware architecture of MAFTIA: A blueprint. In *Proceedings of the Third IEEE Information Survivability Workshop*.

[Veríssimo et al., 2003] Veríssimo, P., Neves, N. F., and Correia, M. (2003). Intrusion-tolerant architectures: Concepts and design. In Lemos, R., Gacek, C., and Romanovsky, A., editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 3–36. Springer-Verlag.

[Veríssimo and Rodrigues, 2001] Veríssimo, P. and Rodrigues, L. (2001). *Distributed Systems for System Architects*. Kluwer Academic Publishers.

[Viega and McGraw, 2002] Viega, J. and McGraw, G. (2002). *Building Secure Software*. Addison Wesley.

[Wang et al., 2003] Wang, D., Madan, B., and Trivedi, K. (2003). Security analysis of the SITAR intrusion tolerance system. In *Proceedings of the ACM Workshop on Survivable and Self-Regenerative Systems*, pages 23–32.

[Wang et al., 2001] Wang, F., Gong, F., Sargor, C., Goseva-Popstojana, K., Trivedi, K., and Jou, F. (2001). SITAR: A scalable intrusion tolerance architecture for distributed services. In *Proceedings of the IEEE Second SMC Information Assurance Workshop*, pages 38–45.

[Yin et al., 2003] Yin, J., Martin, J., Venkataramani, A., Alvisi, L., and Dahlin, M. (2003). Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267.

[Zhou et al., 2002a] Zhou, L., Schneider, F., and van Renesse, R. (2002a). COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368.

[Zhou et al., 2002b] Zhou, L., Schneider, F., and van Renesse, R. (2002b). Proactive secret sharing in asynchronous systems. TR 1877, Cornell University.