# Analisando a Viabilidade da Implementação Prática de Sistemas Tolerantes a Intrusões

**Rafael R. Obelheiro[*], Alysson Neves Bessani[†] e Lau Cheuk Lung[‡]**

Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina
Email: rro@das.ufsc.br, neves@das.ufsc.br

Programa de Pós-Graduação em Informática
Pontifícia Universidade Católica do Paraná
Email: lau@ppgia.pucpr.br

***Resumo.*** *A construção de sistemas seguros e invioláveis usando mecanismos tradicionais vem se tornando um objetivo cada vez mais difícil de ser atingido. O reconhecimento desse fato tem aumentado o interesse em abordagens alternativas de segurança, como a tolerância a intrusões, que aplica conceitos e técnicas de tolerância a faltas em problemas de segurança. Uma das principais questões envolvendo os sistemas tolerantes a intrusões é que muitos dos algoritmos usados pressupõem que os componentes do sistema falham ou são comprometidos de forma independente, premissa que vem sendo seguidamente questionada. Neste artigo nós mostramos que a diversidade possibilita a construção de sistemas tolerantes a intrusões reais. Nós examinamos várias formas de diversidade e discutimos como elas podem auxiliar nessa tarefa. Além disso, é mostrado um exemplo prático de como um sistema tolerante a intrusões pode ser construído usando a diversidade.*
***Palavras chave****: tolerância a intrusões, diversidade*

***Abstract.*** *Building secure, inviolable systems using traditional mechanisms is increasingly becoming an unattainable goal. Recognition of this fact has increased interest in alternative approaches to security such as intrusion tolerance, which applies fault tolerance concepts and techniques to security problems. One of the main issues surrounding intrusion-tolerant systems is that many of the algorithms used for building them assume that system components fail or are compromised independently, and this assumption has been repeatedly called into question. In this paper we show that diversity enables us to build real intrusion-tolerant systems. We investigate several kinds of diversity and discuss how they can help with this task. Furthermore, we provide a practical example of how an intrusion-tolerant system can be built using diversity.*
***Keywords****: intrusion tolerance, diversity*

[*]Bolsista de doutorado CAPES.
[†]Bolsista de doutorado CNPq.
[‡]Bolsista de Produtividade em Pesquisa do CNPq – Nível 2.

## 1. Introdução

A segurança dos sistemas computacionais atuais é constantemente posta à prova por vários tipos de ataques, incluindo ação de invasores e de diversas espécies de *malware*, como *worms*, vírus e cavalos de Tróia (*Trojan horses*). As ações de defesa contra esses ataques geralmente se concentram na prevenção de incidentes de segurança através de ferramentas tais como autenticação e controle de acesso. Entretanto, com freqüência sistemas supostamente seguros são comprometidos, devido a problemas como erros de configuração e a presença de vulnerabilidades no *software*. Tais problemas são fatores virtualmente inescapáveis face à complexidade dos sistemas e à sua baixa qualidade de construção. Uma alternativa à abordagem preventiva, que surgiu na comunidade de segurança do funcionamento (*dependability*) [Avizienis et al. 2004] mas que vem se difundindo no âmbito de segurança computacional, é a **tolerância a intrusões** [Fraga e Powell 1985; Veríssimo et al. 2003]. O objetivo da tolerância a intrusões é garantir que um sistema continue funcionando corretamente mesmo que algumas partes desse sistema sejam comprometidas.

A tolerância a intrusões pressupõe um sistema distribuído, e subjacente a essa distribuição do sistema está uma premissa de **independência de falhas**, isto é, que os diferentes componentes do sistema não serão comprometidos simultaneamente. Essa premissa é bastante razoável quando se lida com faltas acidentais, sejam elas de *hardware* ou *software*, posto que a freqüência de faltas dessa natureza é estatisticamente baixa (desde que os componentes tenham um mínimo de qualidade). Entretanto, quando o objetivo é tolerar intrusões, não se pode presumir que um ataque não seja desferido de forma coordenada contra os diversos componentes; na verdade, torna-se necessário construir o sistema de forma que ele se comporte de forma satisfatória mesmo em tal situação. Se a arquitetura do sistema utilizar replicação, por exemplo, torna-se muito difícil garantir a independência de falhas: como as réplicas são idênticas, uma vulnerabilidade que afete uma delas irá afetar todas. A solução para garantir independência de falhas é empregar técnicas de diversidade.

A necessidade de garantir independência de falhas não é a única dificuldade na construção de sistemas tolerantes a intrusões. Um outro fator importante é o desempenho, já que as técnicas usadas em tolerância a intrusões são muitas vezes dispendiosas do ponto de vista computacional. Na maioria dos casos, esse custo é inerente à complexidade da tarefa de garantir um sistema funcional a despeito de intrusões. Diversos trabalhos tratam especificamente da melhoria do desempenho de mecanismos que podem ser usados em sistemas tolerantes a intrusões, como o PBFT (*Practical Byzantine Fault-Tolerance*) [Castro e Liskov 2002], a entidade certificadora COCA (*Cornell Online Certification Authority*) [Zhou et al. 2002] e protocolos baseados em *wormholes* [Correia et al. 2002], e esse aspecto não será discutido em maiores detalhes neste artigo.

Os trabalhos que lidam com tolerância a intrusões geralmente assumem diversidade de implementação de tal forma que os componentes falham de forma independente [Reiter 1995; Cachin e Poritz 2002; Zhou et al. 2002; Castro e Liskov 2002]. Apesar desta ser uma premissa comum, e até aceitável do ponto de vista do estudo de protocolos e mecanismos, não se conhecem estudos sobre a viabilidade da validade desta premissa na implementação de sistemas práticos.

Este trabalho tenta preencher justamente esta lacuna, investigando os tipos de diversidade existentes e verificando sua aplicabilidade na implementação de sistemas tolerantes a in-

trusões. Dentre as principais contribuições do estudo apresentado neste artigo está a introdução dos conceitos de eixo e grau de diversidade, assim como um apanhado dos eixos de diversidade que podem ser usados na implementação de sistemas tolerantes a intrusões que utilizem mecanismos que requerem independência de falhas, suas limitações e um exemplo de como esses eixos de diversidade podem ser aplicados na construção e projeto de um sistema com a qualidade de serviço em questão.

O restante do artigo está organizado da seguinte forma: a seção 2 apresenta alguns aspectos da tolerância a intrusões. Na seção 3 a técnica de diversidade é definida e vários eixos de diversidade são analisados considerando a implementação de sistemas tolerantes a intrusões. A seção 4 apresenta um estudo de caso que ilustra um serviço Web crítico projetado levando-se em consideração a diversidade. Finalmente, a seção 5 apresenta as considerações finais do trabalho bem como algumas perspectivas futuras.

## 2. Tolerância a Intrusões

Um sistema tolerante a intrusões é um sistema que é capaz de fornecer um serviço seguro de forma continuada a despeito de intrusões em um determinado número de seus componentes. Este conceito admite uma degradação na funcionalidade oferecida pelo sistema, desde que a sua segurança não seja violada.

O ponto de partida para a construção de sistemas tolerantes a intrusões é eliminar pontos únicos de falha, ou seja, elementos que, se tiverem sua segurança violada, provocam o comprometimento do sistema como um todo. Isso implica a distribuição da informação armazenada no sistema e das funcionalidades que esse sistema implementa. A coordenação entre os componentes do sistema pode ser feita através de protocolos tolerantes a faltas bizantinas [Lamport et al. 1982], que são capazes de tolerar faltas arbitrárias desses componentes. Conforme já discutido na introdução, é fundamental que os componentes do sistema falhem ou sejam comprometidos de forma independente.

Elementos essenciais em muitos mecanismos de segurança, as chaves criptográficas tradicionais também podem ser consideradas um ponto único de falha, tanto em termos de confidencialidade (quando uma chave privada ou secreta é obtida por um elemento não autorizado) quanto em termos de disponibilidade (quando uma chave privada ou secreta é destruída de forma não autorizada). A criptografia de limiar (*threshold cryptography*) [Gemmell 1997; Desmedt 1997] minimiza este problema, possibilitando que um segredo (tal como uma chave privada ou uma chave simétrica) seja compartilhado de forma segura (aumentando a sua disponibilidade sem necessariamente comprometer sua confidencialidade). Em um esquema criptográfico de limiar, um segredo é fracionado e repartido de tal forma que só possa ser reconstruído a partir de um número mínimo de frações ou imagens (*shadows*). A criptografia de limiar possui duas variantes básicas. Uma é o compartilhamento de segredos, onde uma chave é fracionada, as frações são distribuídas pelo sistema e, quando necessário, o segredo original é reconstruído a partir de um subconjunto dessas frações. A outra é a computação multiparte segura, onde cada componente do sistema utiliza um segredo que só ele possui para efetuar uma computação e os resultados das diferentes computações são combinados segundo uma determinada função para obter um resultado desejado.

## 3. Diversidade

A diversidade de projeto é um mecanismo clássico de tolerância a faltas de *software*, proposto inicialmente na década de 70 [Randell 1975; Avizienis e Chen 1977]. A idéia central desse mecanismo é usar diferentes implementações de um mesmo sistema (ou subsistema) para obter tolerância a faltas, partindo da premissa que projetos e implementações independentes apresentem falhas de *software* também independentes.

As subseções seguintes examinam diferentes formas de implementar a diversidade de projeto em sistemas reais. A ênfase da discussão reside em como obter, na prática, independência de falhas para os componentes de um sistema distribuído com o propósito de torná-lo mais resiliente a intrusões. Para cada técnica individual, são apresentados a sua importância e principais benefícios, assim como eventuais desvantagens e as implicações de cada uma em termos de custo para o sistema.

Na discussão a seguir, damos o nome de **eixos de diversidade** a componentes ou subsistemas de um sistema que podem ser diversificados. Por exemplo, se um determinado sistema requer o uso de um sistema gerenciador de banco de dados (SGBD), o SGDB é um eixo de diversidade no projeto deste sistema. Chamamos de **grau de diversidade** a quantidade de possíveis escolhas que podem ser feitas considerando um eixo de diversidade em particular. No exemplo do banco de dados, se existem dois SGBDs diferentes que satisfazem as necessidades do sistema, dizemos que o grau de diversidade deste eixo em nosso sistema é dois.

Uma outra taxonomia de técnicas de diversidade, de acordo com o nível em que elas são implementadas, foi proposta em [Deswarte et al. 1998]. Essa classificação tem um caráter mais sintético, privilegiando a identificação de aspectos comuns das diferentes técnicas. Neste trabalho, porém, pretendemos explorar da forma mais ampla possível o leque de opções de implementação de diversidade, razão pela qual preferimos não seguir essa taxonomia.

### 3.1. Implementação

A diversidade de implementação de *software* de aplicação é a forma mais comum de diversidade de projeto. Isso se explica porque o *software* de aplicação é, muitas vezes, o único componente sobre o qual uma organização possui total controle, tendo acesso a suas especificações completas.

Uma desvantagem da diversidade de projeto é o custo de implementar diferentes versões (variantes) de um *software*. Esse custo, porém, não cresce de forma linear: estudos comprovam que a implementação de cada variante custa em torno de 70–80% do custo da versão inicial [Deswarte et al. 1998]. Essa redução é esperada, uma vez que o levantamento e análise de requisitos exigidos na especificação do sistema, que correspondem a uma parte custosa do processo de desenvolvimento de *software*, podem ser reaproveitadas em todas as versões. Os testes caixa preta do sistema também podem ser aproveitados em todas as variantes de um *software*.

### 3.2. Administração

É sabido que grande parte das invasões de segurança são fruto de engenharia social [Winkler e Dealy 1995]. A distribuição dos componentes de um serviço tolerante a intrusões em diversos domínios administrativos visa reduzir esse problema e dificultar o uso de engenharia social por parte de um atacante.

Além disso, diferentes sistemas operam com diferentes administradores que aplicam diferentes políticas de gerenciamento de segurança. Essas políticas compreendem tanto o uso de *software* na proteção dos domínios, como na configuração e localização de *firewalls* e sistemas de detecção de intrusões, quanto a organização desses domínios e as políticas de segurança às quais os usuários dos domínios estão sujeitos.

### 3.3. Localização

A diversidade de localização consiste na dispersão dos vários componentes físicos de um sistema por diferentes sítios de instalação. Essa distribuição é uma importante defesa contra ameaças físicas, sejam elas de natureza maliciosa (roubo ou destruição de *hardware* ou da infra-estrutura elétrica, por exemplo) ou acidental (os chamados "atos de Deus", como enchentes, terremotos, incêndios, animais peçonhentos[1] ou faxineiras que desconectam servidores da tomada de força para ligar uma enceradeira durante a limpeza[2]). Violações de natureza física são, na imensa maioria dos casos, eventos isolados, o que assegura a independência de falhas. Por outro lado, sistemas suficientemente importantes para estarem sujeitos a um ataque coordenado contra diferentes localizações geográficas necessitam de uma política de segurança física bastante robusta.

É evidente que manter um número de instalações físicas adequadas para hospedar sistemas computacionais possui um custo. Entretanto, muitas vezes uma organização que possui múltiplas unidades administrativas já dispõe de tais instalações, ou pode adaptar as instalações existentes sem gastos excessivos. Além disso, a diversidade de localização pode ser combinada de forma sinérgica com a diversidade de administração (seção 3.2). Componentes situados em locais distintos e administrados por pessoas diferentes tendem a apresentar maior independência de falhas, e a conjugação dessas medidas ajuda a racionalizar os custos de adotar esses dois eixos de diversidade.

### 3.4. COTS

Componentes prontos, ditos de prateleira ou COTS (*Commercial Off-The-Shelf*)[3], são parte integrante da maioria dos sistemas atuais. Tais componentes geralmente são usados como parte de um sistema maior, seja na construção do sistema, como compiladores e bibliotecas de rotinas, seja para implementar grandes subsistemas, como um sistema gerenciador de bancos de dados (SGBD) ou um *middleware*.

O uso de COTS oferece uma boa oportunidade de aplicação da diversidade, uma vez que pode-se dispor de vários componentes que implementam um determinado conjunto de funcionalidades. Dentre os inúmeros tipos de COTS comumente usados na implementação de sistemas, podemos destacar alguns:[4]

- **SGDBs**: Existe uma série de SGBDs que podem ser acionados através de interfaces de programação (APIs) padronizadas e *plugins* de conectividade. Por exemplo, a plataforma JAVA suporta a API JDBC (*Java Data Base Connectivity*) que permite a integração transparente de qualquer base de dados que aceite SQL desde que o *driver* para

[1] Um dos autores foi testemunha de um disco rígido de um servidor que foi inutilizado após uma barata ter posto seus ovos sobre a placa controladora do disco em uma ocasião em que o servidor estava parado para reparos.

[2] Fato igualmente verídico.

[3] Apesar da definição usar o termo "*Commercial*", o conceito de COTS se aplica igualmente a *software* livre.

[4] Sistemas operacionais também são considerados COTS, mas são discutidos separadamente na seção 3.5.

esta base de dados esteja disponível. Praticamente todos os bancos de dados têm *drivers* JDBC (alguns até mais de um, suportando diversidade também nesse nível) e podem ser usados por aplicações baseadas nessa tecnologia desde que as consultas e atualizações na base sigam o padrão SQL-92 (suportado por todos os SGBDs relacionais modernos);

- ***Middleware***: Os sistemas distribuídos quase sempre se utilizam de plataformas de *middleware* para esconder a complexidade das interações entre as diversas partes do sistema. Dentre as plataformas de *middleware* mais utilizadas estão os serviços Web, CORBA (*Common Object Request Broker Archtecture*) e RPC (*Remote Procedure Call*). Essas plataformas em geral se baseiam em especificações padronizadas que garantem a interoperabilidade entre partes implementadas usando diferentes plataformas. Como o *middleware* é usado para comunicação entre as partes do sistema, vulnerabilidades nesse componente tendem a comprometer a integridade da aplicação de forma cabal. Desta forma a utilização de diferentes implementações de um padrão de *middleware* em que a aplicação se baseia permite tolerar falhas causadas pela implementação comprometida desses padrões. Um exemplo representativo de como esse tipo de diversidade pode ser obtido facilmente é o padrão CORBA. A OMG (*Object Management Group*) define uma série de especificações e diversas organizações implementam estas especificações da forma que desejarem. Considerando apenas as linguagens de programação Java e C++, existem pelo menos quatro implementações de qualidade e livres para este padrão (JacORB, OpenORB, TAO e MICO).
- **VMs**: Linguagens como Java, C#, Python e LISP (entre outras) se baseiam no uso de máquinas virtuais para execução de aplicações. Uma máquina virtual controla quase todos os recursos (memória, arquivos, *sockets*, etc.) do sistema operacional usados por aplicativos escritos nessas linguagens além de implementar características fundamentais para a integridade da aplicação (como monitores de segurança). Desta forma, problemas relacionados a máquinas virtuais podem também ser considerados críticos em um sistema e mais uma vez aplicações tolerantes a intrusões podem se beneficiar de diversidade. No caso de Java, temos a JVM (*Java Virtual Machine*) que, dentre outras coisas, gerencia memória, transforma *bytecode* em código nativo e implementa monitores de segurança do (poderoso) modelo de segurança desta linguagem. Como no caso das especificações CORBA, existem diversas máquinas virtuais de qualidade (entre proprietárias e livres) que podem ser usadas em ambientes de produção [Doederlein 2005].
- **Compiladores**: Compiladores não são parte de uma aplicação tolerante a intrusões mas são ferramentas fundamentais na construção de praticamente toda aplicação. O uso de diferentes compiladores leva à geração de código objeto (ou intermediário) diferente. Assim, o uso de diversidade de compiladores traz pelo menos duas consequências imediatas para a aplicação resultante: evita comprometimento total da aplicação em caso de geração de código com *bugs* que se revelam vulnerabilidades de segurança[5] e código gerado por compiladores diferentes pode não ser afetado da mesma forma por um único *exploit*[6], já que a geração de código define, por exemplo, a ordem de parâmetros na pilha.

[5]Embora a geração de código com *bugs* não seja tão freqüente, ela ocorre por vezes, especialmente com determinadas opções de otimização de código. Um exemplo é a otimização `-frename-registers` do GCC (*GNU Compiler Collection*), desabilitada por *default* em versões recentes do compilador justamente por gerar código com *bugs* [Stallman 2005, pp. 79–80].

[6]Um *exploit* é uma ferramenta automatizada que explora vulnerabilidades específicas com o intuito de comprometer a segurança de um sistema.

- **Bibliotecas**: Tendo em vista que grande parte de um *software* é composta de bibliotecas fornecidas por terceiros, é de se esperar que a aplicação de diversidade de bibliotecas confira alguma independência de falhas ao sistema. Vários são os tipos de bibliotecas disponíveis, algumas acessadas através de interfaces padrão e outras através de APIs proprietárias. As primeiras podem ter diversidade de aplicação apenas trocando a implementação da interface (como JAX—*Java API for XML Processing* e JCE—*Java Cryptography Extensions* no Java) enquanto as APIs proprietárias podem ser acessadas através de adaptadores.

Além do uso de diferentes COTS em diferentes partes de um sistema tolerante a intrusões, pode-se aplicar diferentes configurações destes COTS, como, por exemplo, opções de compiladores JIT (*Just In Time*) e algoritmos de coleta de lixo nas máquinas virtuais.

A diversidade de COTS talvez seja o eixo de diversidade com melhor relação custo/benefício para um sistema crítico. Ela é barata pois COTS comumente utilizados nas aplicações, como SGDBs e compiladores, podem ser obtidos com relativa facilidade (o que possibilita também obter um alto grau de diversidade), e é eficiente pois muitas vezes sistemas são comprometidos não através de vulnerabilidades nas aplicações em si, mas através de vulnerabilidades nos componentes COTS usados como suporte da aplicação.

Obviamente, a utilização de diferentes versões de um componente torna o *software* mais complexo, pois a administração de componentes diversos é mais difícil uma vez que nem todos os componentes podem ser administrados da mesma forma. Outro problema diz respeito à interoperabilidade dos COTS. Os desenvolvedores da aplicação devem se certificar que os mecanismos usados por eles estão disponíveis nas diferentes implementações do componente que espera-se usar. Por exemplo, se o sistema requer um SGDB, os desenvolvedores da aplicação devem utilizar um conjunto de comandos SQL que estejam disponíveis em todos os diferentes SGDBs que forem utilizados pela aplicação.

### 3.5. Sistema Operacional

O sistema operacional (SO) desempenha um papel essencial na segurança de um sistema. Como é ele que controla os recursos de máquina usados pelas aplicações, uma vulnerabilidade em um sistema operacional pode subverter a segurança do sistema como um todo, mesmo que as aplicações que executam sobre esse SO não possuam qualquer falha de projeto ou implementação. Isso ocorre porque as aplicações lidam com abstrações criadas ou gerenciadas pelo sistema operacional (como arquivos, processos e segmentos de memória), e uma violação da segurança do SO oferece a um atacante a possibilidade de manipular livremente essas abstrações. Por essas razões, um sistema operacional vulnerável se torna o "calcanhar-de-aquiles" de qualquer sistema, por mais robustos que sejam os componentes que executam sobre esse SO. Além disso, os sistemas operacionais atuais são geralmente grandes e complexos, o que os torna mais propensos a possuir vulnerabilidades do que *software* de menor porte. Sendo assim, em um sistema distribuído este eixo de diversidade se torna crítico na garantia da independência de falhas entre as diferentes partes do sistema.

Existem algumas facilidades para implementar diversidade em nível de sistema operacional. Uma delas é a existência de interfaces de programação (APIs—*Application Programming Interfaces*) padronizadas, como no caso do POSIX (*Portable Operating System Interface*). A disponibilidade de uma API padronizada significa que uma aplicação pode ser portada para um sistema operacional diferente apenas recompilando-a no novo sistema, reduzindo custos de

desenvolvimento e viabilizando um maior grau de diversidade. Por outro lado, se alguma operação de uma API padronizada tiver falhas na sua semântica—por exemplo, permitir a ocorrência de corridas críticas (*race conditions*)—, esse tipo de falha pode ser explorada com sucesso em diferentes SOs.

A diversidade de sistemas operacionais possui um custo visível em termos de recursos humanos. Isso porque de pouco adianta usar sistemas operacionais variados se eles não podem ser configurados e administrados de forma segura. Na verdade, na ausência de pessoal capacitado o uso de diversidade de SO torna-se uma temeridade, trazendo potencialmente mais riscos do que benefícios (o mesmo vale para alguns tipos de COTS, como SGBDs).

Alguns sistemas operacionais oferecem uma solução intermediária, que garante uma certa independência de falhas sem exigir o uso de SOs diversos. Essa solução consiste em técnicas para aleatorizar a utilização do espaço de endereçamento de memória, tornando difícil a exploração bem sucedida de vulnerabilidades baseadas em estouro de *buffer* [Cowan et al. 2000] de forma simultânea. Isso acontece porque a exploração dessas vulnerabilidades exige a modificação de ponteiros de memória para endereços de escolha do atacante, o que requer que ele possa antecipar o mapa de alocação de memória, uma tarefa surpreendentemente simples em um SO convencional. Quando o espaço de endereçamento é aleatorizado, porém, o atacante não consegue prever onde estão os ponteiros a serem sobrescritos nem quais os endereços que podem lhe beneficiar. Isso retira do atacante a possibilidade de usar um mesmo *exploit* para comprometer várias máquinas, fazendo com que ele seja obrigado a varrer diferentes endereços em um ataque por força bruta, o que é ineficiente, potencialmente demorado e pode chamar muita atenção. Técnicas de aleatorização de memória estão disponíveis, por exemplo, no OpenBSD [OpenBSD] e no PaX [PaX], uma modificação do núcleo do Linux. Uma discussão mais profunda sobre a segurança oferecida por técnicas de aleatorização de memória pode ser encontrada em [Shacham et al. 2004].

### 3.6. Métodos

A diversidade de métodos [Hiltunen et al. 2003] envolve o uso de métodos distintos para garantir um determinado atributo de segurança, de modo que o atributo permaneça válido mesmo que um desses métodos seja comprometido. Por exemplo, para garantir confidencialidade é possível cifrar sucessivamente uma mesma mensagem com dois algoritmos criptográficos diferentes, o que garante o segredo da mensagem cifrada mesmo que um dos algoritmos ou uma das chaves usadas (que devem ser distintas) seja comprometido. Outro exemplo é o uso de métodos complementares de autenticação, tais como uma senha e uma impressão digital, ou uma senha temporária enviada para um telefone celular ou *pager* cujo número tenha sido previamente cadastrado.

Da mesma maneira que os outros eixos de diversidade, a diversidade de métodos busca a maximização da independência entre os métodos redundantes. Por exemplo, se dois métodos $m_1$ e $m_2$ são usados para autenticação de usuário, o comprometimento do método $m_1$ não deve facilitar o comprometimento do método $m_2$. Isso é especialmente importante no caso de mecanismos criptográficos, já que a aplicação sucessiva desses mecanismos pode não aumentar a segurança da cifra adotada [Schneier 1996].

### 3.7. Hardware

Tradicionalmente, sistemas usando redundância de *hardware* não tinham grande preocupação com diversidade, já que a ocorrência de faltas de projeto em *hardware* costumava ser bem menos freqüente do que em *software*, chegando a um nível considerado aceitável por parte dos implementadores de sistemas. Entretanto, com o aumento da complexidade do *hardware* houve um incremento na ocorrência de faltas de projeto em componentes. Exemplos com impacto na segurança de sistemas incluem o *bug* F00F do Pentium [Collins 1998], que permite a um usuário não privilegiado travar o processador, e a recentemente descoberta vulnerabilidade no mecanismo de *hyperthreading* de processadores Intel, que possibilita a revelação não autorizada de informações (incluindo o roubo, por parte de um usuário não privilegiado, de chaves privadas RSA sendo usadas na mesma máquina) [Percival 2005].

Além de faltas de projeto, a diversidade de *hardware* é um eixo atraente por outra razão. Os *exploits* encontrados na Internet são normalmente específicos para uma determinada arquitetura de *hardware* (na sua imensa maioria a arquitetura Intel i386, a mais popular atualmente). Caso um atacante tente usar um *exploit* para uma arquitetura em um sistema de outra arquitetura, muito provavelmente ele não causará mais do que um *crash* no sistema atacado, ainda que o *exploit* seja capaz de fornecer um acesso privilegiado quando lançado contra um sistema da arquitetura correta. Portanto, um sistema que use uma arquitetura de *hardware* diferente não terá sua segurança comprometida por um desses *exploits*, e o administrador responsável pela máquina terá uma janela de tempo maior para corrigir as vulnerabilidades que forem descobertas. Pelas mesmas razões, a diversidade de *hardware* é uma forma eficaz de reduzir a exposição de uma rede a vermes (*worms*).

### 3.8. Discussão

Em todas os eixos de diversidade discutidos acima, é necessário considerar o nível de independência das diferentes implementações de um componente, que pode impactar o grau de diversidade no eixo em questão. As variantes que possuem elementos em comum (como as diferentes máquinas virtuais Java que se baseiam no código da Sun [Doederlein 2005] ou placas que compartilham determinados *chips*, por exemplo) são menos diversas—isto é, possuem nível menor de diversidade—do que variantes completamente independentes. O grau de diversidade de um eixo também é afetado por fatores como custo e disponibilidade: quanto mais variantes de um eixo puderem ser usadas a um baixo custo de aquisição e manutenção, maior poderá ser o grau de diversidade desse eixo.

Outro ponto a considerar é que a diversidade pode ajudar nas fases de testes e validação do sistema, já que é possível comparar os resultados dessas fases entre as variantes. Essa comparação pode propiciar a descoberta de *bugs* que permaneceriam de outra forma dormentes (possivelmente até que o sistema entrasse em produção) e a identificação de aspectos subespecificados do sistema (já que diferentes desenvolvedores podem dar interpretações divergentes para pontos obscuros das especificações).

Por fim, é importante também compreender as limitações da diversidade. Por exemplo, a diversidade de implementação não pode ser eficaz se os requisitos usados nas diferentes implementações são incompletos, incoerentes, inseguros ou incorretos.

## 4. Estudo de Caso: Serviço Web Crítico

A fim de ilustrar a aplicação da diversidade na implementação de um sistema distribuído real, consideramos nesta seção o projeto de um serviço Web crítico construído de forma tolerante a intrusões.

O objetivo é projetar um serviço Web que permaneça disponível o máximo de tempo possível, mesmo com a ocorrência de falhas, ataques e intrusões. Um serviço como esse poderia ser usado, por exemplo, para o sistema de reserva de vôos de uma companhia aérea, para que seus agentes façam reservas em vôos domésticos e internacionais. É importante destacar que a arquitetura apresentada nesta seção pode ser empregada em outras aplicações além da que é descrita aqui.

A arquitetura proposta para o serviço é apresentada na figura 1. Nessa figura temos o serviço espalhado em quatro localidades (diferentes escritórios da empresa pelo mundo). Cada uma dessas localizações é composta de uma interface Web do serviço, acessível via SOAP (*Simple Object Access Protocol*) [W3C 2003] e uma réplica da aplicação que controla as reservas nos vôos. As réplicas da aplicação sincronizam seus estados utilizando **replicação ativa** [Schneider 1990]. Neste modelo de replicação todas as réplicas do sistema executam todas as requisições ao serviço na mesma ordem, garantindo assim que após a execução de cada requisição todas permanecem no mesmo estado. Este tipo de comportamento requer a execução de um protocolo de **difusão com ordem total**[7]. A literatura é vasta em protocolos eficientes que oferecem este nível de garantia mesmo com processos sujeitos a faltas bizantinas [Castro e Liskov 2002; Zielinski 2004].

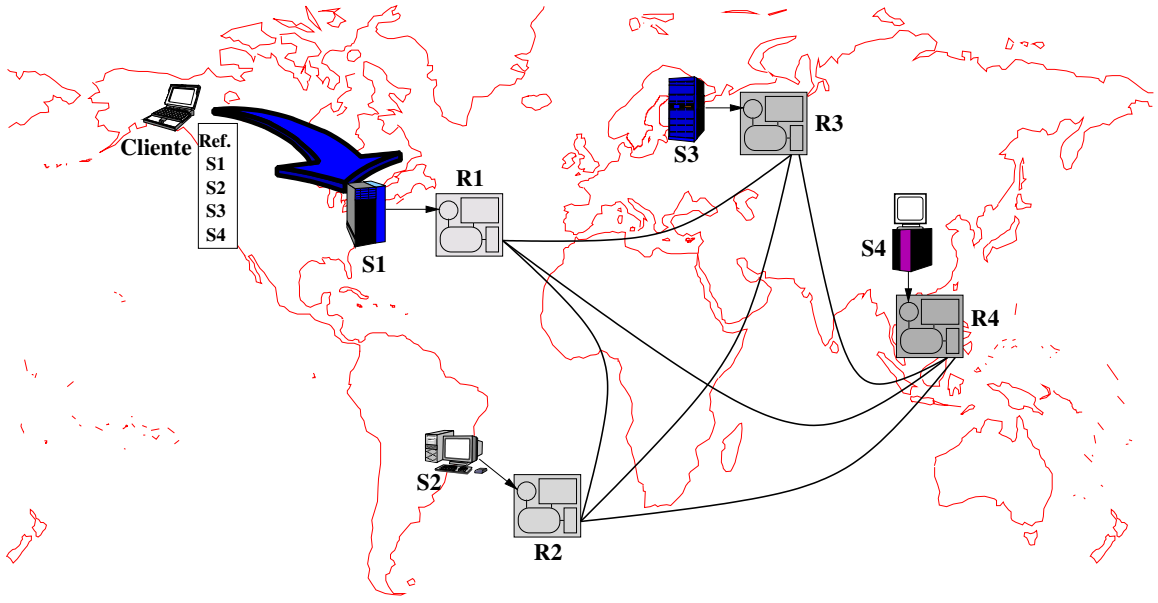


**Figura 1: Arquitetura do Serviço Web Tolerante a Intrusões.**

Um cliente que deseja acessar o sistema de reservas obtém a descrição do serviço em um registro WSDL (*Web Services Description Language*) [Christensen et al. 2001] publicado em algum serviço UDDI (*Universal Description, Discovery and Integration*) [OASIS 2004] na

[7]Também chamado **difusão atômica**. O protocolo tolerante a faltas bizantinas mais eficiente para a implementação desta primitiva de comunicação é o apresentado em [Zielinski 2004].

Internet. Nesta descrição do serviço existe a lista com os quatro servidores Web que podem ser usados para se ter acesso ao sistema em questão. O cliente pode acessar o sistema a partir de qualquer um dos servidores.

Os protocolos requeridos para implementação de replicação ativa (difusão com ordem total) na Internet assumem que menos de 1/3 das réplicas do sistema serão faltosas em um instante de tempo. Desta forma, o sistema da figura 1 tolera apenas uma réplica falha. Para fazer com que faltas e intrusões sejam independentes neste sistema temos de utilizar de diversidade na implementação do sistema.

Para este exemplo em especial, consideramos cinco eixos de diversidade para as quatro réplicas do sistema ($R1$-$R4$): implementação, suporte de execução (COTS), banco de dados (COTS), sistema operacional, *hardware* e localização. A tabela 1 apresenta algumas possíveis escolhas para a implementação deste sistema, com cada eixo tendo grau de diversidade igual a quatro.

| **Eixo de Diversidade** | $R1$ | $R2$ | $R3$ | $R4$ |
|---|---|---|---|---|
| Implementação | Java 1 | Java 2 | C# 1 | C# 2 |
| Ambiente de Execução | Java Sun[a] | Java Livre[b] | .NET | Mono |
| Banco de Dados | MySQL | PostgreSQL | IBM DB2 | Firebird |
| Sistema Operacional | Solaris | FreeBSD | Windows | Linux |
| Hardware | Ultra SPARC | Pentium | Athlon | Mac |
| Localização | EUA | Brasil | Rússia | China |

[a] J2SRE + JWSDK (*Java Web Services Development Kit*).
[b] Kaffe + GNU Classpath + Apache Axis.

**Tabela 1: COTS que poderiam ser utilizados na implementação do serviço Web.**

Alguns comentários podem ser feitos a respeito da tabela 1. Em primeiro lugar, para satisfazer a diversidade de implementação seriam construídas quatro variantes do *software* de aplicação: duas usando Java e duas usando C#. Essas diferentes versões seriam executadas em diferentes ambientes de execução que compreendem uma máquina virtual e um suporte a invocação de serviços Web. Outro aspecto interessante é que no exemplo considerado, a diversidade de localização praticamente implica uma diversidade de administração, uma vez que dificilmente ambientes tão distantes seriam gerenciados pela mesma equipe.

A aplicação desenvolvida seguindo o projeto aqui proposto seria tolerante a intrusões e manteria independência de falhas utilizando-se de componentes já disponíveis e largamente utilizados. Isso ocorre porque vulnerabilidades existentes em quaisquer dos componentes individuais do sistema só poderão ser exploradas em uma única réplica. Ataques baseados em engenharia social não afetarão mais de uma réplica devido à diversidade de localização/administração. Esse eixo de diversidade também confere ao sistema resistência contra ataques físicos à infra-estrutura de computação e comunicação, além de resistência limitada contra ataques de negação de serviço.

### A importância do número quatro

Cabe notar que o sistema proposto nessa seção contém quatro réplicas e que para cada eixo de diversidade consideramos grau de diversidade quatro (ou seja, quatro versões diferentes). Isso não ocorre por acaso: protocolos de acordo tolerantes a faltas bizantinas (base para a

maioria dos sistemas tolerantes a intrusões e usados na replicação ativa do exemplo) toleram $f$ faltas se o número $n$ de réplicas do sistema for $n \geq 3f + 1$ [Lamport et al. 1982; Toueg 1984]. Desta forma, o mínimo de réplicas que um sistema deve ter é quatro. A fim de manter a maior independência de falhas possível, o projetista de um sistema tolerante a intrusões deve procurar um grau de diversidade em cada eixo considerado de pelo menos quatro, e idealmente que seja igual ao número de réplicas. Por exemplo, se no exemplo anterior tivéssemos duas réplicas executando o sistema operacional Windows, uma vulnerabilidade neste poderia levar a intrusão nestas duas réplicas, o que poderia destruir o serviço uma vez que ele tolera apenas uma falha.

## 5. Conclusões

Pelo estudo apresentado neste texto podemos concluir que, mesmo com os problemas decorrentes da complexidade de se utilizar diversidade (alto custo, incompatibilidade entre diferentes COTS, administração complexa, etc...), **é possível** implementar sistemas tolerantes a intrusões **na prática** se a diversidade for aplicada de forma correta.

Uma contribuição deste trabalho é a introdução dos conceitos de eixo e grau de diversidade, bem como um levantamento de vários eixos de diversidade que podem ser considerados na implementação de serviços tolerantes a intrusões. Uma segunda contribuição é a proposta de uma arquitetura simples e eficaz para a implementação de um serviço Web tolerante a intrusões na Internet utilizando COTS amplamente disponíveis na Web.

O trabalho apresentado neste artigo abre interessantes questões a respeito da utilização de diversidade na implementação de sistemas tolerantes a intrusões. Em particular, nos parece de fundamental importância examinar o uso sistemático de diversidade [Castro et al. 2003], de preferência com o apoio de metodologias e ferramentas, de forma a facilitar a provisão de independência de falhas. Outra possível continuação do estudo proposto neste trabalho é a implementação de um sistema como o apresentado na seção 4 de tal forma a avaliar as dificuldades práticas de se implementar um sistema tolerante a intrusões que emprega diversidade. Neste contexto, uma questão ainda em aberto é quais métricas e métodos seriam os mais adequados para avaliar o grau de tolerância a intrusões e o custo desse sistema.

## Referências

Algirdas Avizienis e L. Chen. “On the Implementation of N-Version Programming for Software Fault Tolerance During Execution”. In *Proceedings of the IEEE COMPSAC*, pp. 149–155, Chicago, IL, USA, November 1977.

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, e Carl Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, March 2004.

Cristian Cachin e Jonathan A. Poritz. “Secure Intrusion-Tolerant Replication on the Internet”. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN’2002)*, Washington, DC, USA, 2002.

Miguel Castro e Barbara Liskov. “Practical Byzantine Fault Tolerance and Proactive Recovery”. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

Miguel Castro, Rodrigo Rodrigues, e Barbara Liskov. "BASE: Using Abstraction to Improve Fault Tolerance". *ACM Transactions on Computer Systems*, 21(3):236–269, August 2003. A preliminary version appeared in the *18th Symposium on Operating Systems Principles*, 2001.

Erik Christensen, Francisco Curbera, Greg Meredith, e Sanjiva Weerawarana. *Web Services Description Language 1.1*. W3C Working Group, March 2001.

Robert R. Collins. "The Pentium F00F Bug". *Dr. Dobb's Journal of Software Tools*, 23(5):62, 64–66, May 1998.

Miguel Correia, Lau Cheuk Lung, Nuno F. Neves, e Paulo Veríssimo. "Efficient Byzantine-Resilient Reliable Multicast on a Hybrid Failure Model". In *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'2002)*, Suita, Japan, October 2002.

Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, e Jonathan Walpole. "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade". In *DARPA Information Survivability Conference and Expo (DISCEX)*, Hilton Head Island, SC, USA, January 2000.

Yvo G. Desmedt. "Some Recent Research Aspects of Threshold Cryptography". In E. Okamoto, G. Davida, e M. Mambo (Eds.), *Proceedings of the First International Workshop on Information Security (ISW'97)*, LNCS 1396, pp. 158–173, Ishikawa, Japan, September 1997. Springer-Verlag.

Yves Deswarte, Karama Kanoun, e Jean-Claude Laprie. "Diversity Against Accidental and Deliberate Faults". In P. Ammann, B. H. Barnes, S. Jajodia, e E. H. Sibley (Eds.), *Computer Security, Dependability, and Assurance: From Needs to Solutions*, pp. 171–181, Williamsburg, VA, USA, November 1998. IEEE Computer Press.

Osvaldo Pinali Doederlein. "JVMs Alternativas: Explore Implementações do J2SE". *Java Magazine*, (24):32–45, maio de 2005.

Joni S. Fraga e David Powell. "A Fault- and Intrusion-Tolerant File System". In *Proceedings of the 3rd International Congress on Computer Security (IFIP/SEC'85)*, pp. 203–218, Dublin, Ireland, August 1985.

Peter S. Gemmell. "An Introduction to Threshold Cryptography". *Cryptobytes—The Technical Newsletter of RSA Laboratories*, 2(3):7–12, Winter 1997. `ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto2n3.pdf`.

Matti A. Hiltunen, Richard D. Schlichting, e Carlos A. Ugarte. "Building Survivable Services Using Redundancy and Adaptation". *IEEE Transactions on Computers*, 52(2):181–194, February 2003.

Leslie Lamport, Robert Shostak, e Marshall Pease. "The Byzantine Generals Problem". *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

OASIS. *Universal Description, Discovery and Integration v3.0.2 (UDDI)*. Organization for the Advancement of Structured Information Standards (OASIS), October 2004.

OpenBSD. "The OpenBSD Project". `http://www.openbsd.org/`. Acessado em 05 de junho de 2005.

PaX. "Homepage of the PaX Team". `http://pax.grsecurity.net/`. Acessado em 05 de junho de 2005.

Colin Percival. "Cache Missing for Fun and Profit", May 2005. Disponível em `http://www.daemonology.net/papers/htt.pdf`. Acessado em 05 de junho de 2005.

Brian Randell. “System Structure for Software Fault Tolerance”. *IEEE Transactions on Software Engineering*, SE-1:220–232, June 1975.

Michael K. Reiter. “The Rampart Toolkit for Building High-Integrity Services”. In *Theory and Practice in Distributed Systems*, LNCS 938, pp. 99–110. Springer-Verlag, 1995.

Fred B. Schneider. “Implementing Fault-Tolerant Service Using the State Machine Aproach: A Tutorial”. *ACM Computing Surveys*, 22(4):299–319, December 1990.

Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons, New York, NY, USA, 2nd edition, 1996.

Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, e Dan Boneh. “On the Effectiveness of Address-Space Randomization”. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS’04)*, pp. 298–307, Washington, DC, USA, October 2004.

Richard Stallman. *Using the GNU Compiler Collection (version 4.0.0)*. Free Software Foundation, Boston, MA, 2005. Disponível em `http://gcc.gnu.org/onlinedocs/`. Acessado em 05 de junho de 2005.

Sam Toueg. “Randomized Byzantine Agreements”. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 163–178, 1984.

Paulo Veríssimo, Nuno Ferreira Neves, e Miguel Correia. “Intrusion-Tolerant Architectures: Concepts and Design”. In R. Lemos, C. Gacek, e A. Romanovsky (Eds.), *Architecting Dependable Systems*, LNCS 2677, pp. 3–36. Springer-Verlag, 2003.

W3C. *SOAP 1.2 – W3C Recommendation*. W3C, June 2003. `http://www.w3.org/TR/soap12/`. Acessado em 05 de junho de 2005.

Ira S. Winkler e Brian Dealy. “Information Security Technology? . . . Don’t Rely on It—A Case Study in Social Engineering”. In *Proceedings of the 5th USENIX UNIX Security Symposium*, Salt Lake City, UT, USA, June 1995.

Lidong Zhou, Fred B. Schneider, e Robbert Van Rennesse. “COCA: A Secure Distributed Online Certification Authority”. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.

Piotr Zielinski. “Paxos at War”. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, June 2004.