Concretização de um Sistema de Comunicação em Grupo Tolerante a Intrusões *

Tiago Jorge José Pascoal Miguel Correia Nuno F. Neves Paulo Veríssimo
Faculdade de Ciências da Universidade de Lisboa
{*tjpj,jcmp*}*@lasige.di.fc.ul.pt* {*mpc,nuno,pjv*}*@di.fc.ul.pt*

Resumo

O desenho de sistemas distribuídos tolerantes a faltas pode ser simplificado mediante o recurso a sistemas de comunicação em grupo. Este tipo de *middleware* fornece primitivas de comunicação tolerantes a faltas, com semânticas mais ou menos fortes conforme as necessidades da aplicação.

O artigo apresenta a concretização de um sistema de comunicação em grupo tolerante a faltas arbitrárias, uma categoria que inclui as intrusões. O sistema suporta a comunicação de grupos de máquinas, garantindo a correcção da comunicação apesar da ocorrência de faltas benignas (paragens, omissões na rede) ou maliciosas, como as causadas por *hackers*. O sistema tem fundamentalmente duas componentes: um serviço de filiação encarregue de fornecer a lista das máquinas correctas em cada momento; e um serviço de comunicação com uma primitiva de difusão atómica que entrega mensagens de forma ordenada.

O sistema foi concretizado utilizando a linguagem Java e o Appia, uma plataforma de suporte à composição modular de protocolos. O sistema é baseado numa componente distribuída segura, a *Trusted Timely Computing Base*. O desempenho do sistema é analisado.

1 Introdução

A aproximação predominante com vista à obtenção de sistemas informáticos seguros tem sido sempre a prevenção. Por outras palavras, a ênfase tem sido posta no desenho de sistemas correctos, sem vulnerabilidades que possam ser exploradas por *hackers* ou código malicioso (*worms* ou *virus*). A realidade mostra que isso é impossível e que os sistemas vivem num ciclo permanente: vulnerabilidade descoberta / sistema atacado / remendo (*patch*) / nova vulnerabilidade descoberta /...

A tolerância a faltas de sistemas críticos é baseada noutra perspectiva. Nesta disciplina assume-se que as componentes falham, mas que essas componentes que falham têm de ser usadas para construir sistemas que não falhem. Se bem que as duas aproximações pareçam antagónicas, os ataques e as intrusões (problemas de segurança) podem ser considerados como sendo faltas, englobadas na categoria de faltas arbitrárias ou bizantinas [18]. O problema da tolerância a estes tipos de faltas tem recebido muita atenção nos últimos anos sob a designação de Tolerância a Intrusões [15, 29, 1].

Este artigo descreve a concretização de um sistema tolerante a intrusões baseado numa componente denominada *Trusted Timely Computing Base (TTCB)* [13]. Esta componente é um subsistema distribuído

*Este trabalho foi parcialmente financiado pela FCT através dos projectos POSI/1999/CHS/33996 (DEFEATS) e POSI/CHS/39815/2001 (COPE), e do Large-Scale Informatic Systems Laboratory (LASIGE).

seguro e tempo-real (síncrono), com o seu próprio canal de comunicação privado. A TTCB fornece apenas um conjunto limitado de serviços, entre os quais um serviço de acordo distribuído, que é a base do sistema descrito no artigo. Este tipo de componentes com propriedades de segurança e/ou tempo mais fortes do que o resto do sistema têm sido designadas por *wormholes* [28].

O desenvolvimento de sistemas distribuídos tolerantes a faltas pode ser simplificado mediante o recurso a um sistema de comunicação em grupo. Este tipo de middleware fornece primitivas de comunicação tolerantes a faltas, com semânticas mais ou menos fortes, conforme as necessidades da aplicação. O artigo descreve a concretização de um sistema de comunicação em grupo denominado *Worm-IT (Wormhole-based Intrusion-Tolerant Group Communication System)*. O sistema suporta a comunicação de grupos de processadores (máquinas) garantindo a correcção da comunicação, ainda que haja faltas benignas (paragens, omissões na rede) ou maliciosas, como as causadas por processadores controlados por *hackers*. O sistema tem fundamentalmente duas componentes. O serviço de filiação está encarregue de fornecer a lista dos processadores correctos em cada momento. Este serviço aceita a entrada e saída de membros de um grupo, e remove membros falhados com base em informação fornecida por um módulo detector de falhas. O serviço de comunicação fornece uma primitiva de difusão atómica responsável por entregar todas as mensagens pela mesma ordem a todos os membros de um grupo. O desenho do sistema foi apresentado em [9, 11].

O artigo apresenta uma concretização do Worm-IT em Java usando o Appia, uma plataforma de suporte à composição modular de protocolos. Alguns dos benefícios desta concretização são os que se seguem. Em primeiro lugar, a concretização do sistema em Appia torna simples a integração com novos protocolos e com o nível aplicacional, inclusive com diversos protocolos já disponíveis para esta plataforma. Em segundo lugar, a linguagem Java previne algumas vulnerabilidades frequentes através de um conjunto de mecanismos como a *sandbox* (que permite definir com precisão as permissões de um programa) ou a verificação dos limites dos vectores em tempo de execução (que evita ataques por *buffer overflow*) [30]. Em terceiro lugar, apesar de concretizado em Java, o sistema apresenta um desempenho comparável ao de outros sistemas, como o Rampart [25].

A estrutura do artigo é a seguinte. Na Secção 2 é descrito o contexto em que se insere este trabalho. Na Secção 3 é apresentada a arquitectura e o modelo do sistema. A Secção 4 apresenta o sistema e a Secção 5 a sua concretização. Por fim, são apresentadas medidas de desempenho e algumas conclusões.

2 Contexto

A maior parte dos sistemas de comunicação em grupo na literatura assumem o modelo de faltas por paragem [6]. Alguns destes sistemas evoluíram para suportar um modelo mais forte no qual a comunicação na rede pode ser atacada, mas não as máquinas: Horus [26], Secure Spread [2] e Ensemble [27].

Há três sistemas de comunicação em grupo tolerantes a intrusões na literatura: Rampart [25], SecureRing [17] e SecureGroup [22]. Os protocolos do Rampart assentam num processo sequenciador que ordena as mensagens e realiza as mudanças de filiação do grupo usando um protocolo de *three-phase commit*. A concretização inicial do Rampart foi feita em C sobre o sistema Horus. Mais tarde, foi realizada uma nova concretização sobre C-Ensemble no âmbito do projecto ITUA [24]. O SecureRing é baseado num anel lógico imposto sobre uma LAN. Os protocolos são baseados num token que circula nesse anel e o protótipo foi escrito em C. O Secure Group é também destinado a LANs, baseando-se num protocolo de ordenação total probabilístico. Não há menção a uma concretização deste sistema.

Existem alguns sistemas de outro tipo que merecem uma breve menção dado estarem ao nível do middleware e serem tolerantes a intrusões. O Enclaves é também um sistema de comunicação em grupo

mas que não fornece primitivas de comunicação fiável [14]. O BFT é um algoritmo muito eficiente para a concretização de serviços tolerantes a intrusões usando replicação de máquinas de estados [5]. O SINTRA é um conjunto de protocolos destinados a suportarem serviços replicados tolerantes a intrusões [3].

3 Arquitectura e Modelo do Sistema

Os sistemas tolerantes a intrusões referidos na secção anterior, como o Rampart ou o SecureRing, consideram um modelo de faltas arbitrário ou bizantino, ou seja, assumem que os processos e máquinas podem falhar por paragem, omitindo algumas mensagens, enviando mensagens com formatos inválidos, entrando em conluio com outros processos, etc. Igualmente, a rede pode modificar, remover ou introduzir mensagens, de forma acidental ou maliciosa. O modelo temporal considerado é (quase) o modelo assíncrono, ou seja, não são considerados limites para os tempos de processamento ou comunicação, com excepção dos necessários para detectar falhas de processos. Estes modelos de faltas e temporal são homogéneos, ou seja, aplicam-se igualmente a todas as componentes do sistema.

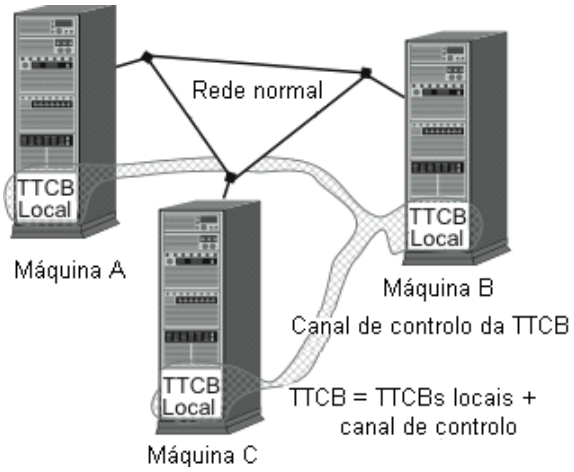

Figura 1: Arquitectura de um sistema com uma TTCB.

Os modelos de faltas e temporal usados neste artigo não são homogéneos mas híbridos. A maior parte do sistema pode também falhar de forma arbitrária e é assíncrona, mas existe uma componente distribuída que só pode falhar por paragem – pois é construída de forma a ser segura – e que é síncrona, ou tempo-real. Essa componente tem a denominação de Trusted Timely Computing Base (TTCB) [13].

A arquitectura do sistema com a TTCB está representada na Figura 1. O sistema é essencialmente um conjunto de máquinas interligadas por uma rede normal, insegura e assíncrona. No entanto, nalgumas máquinas existe uma TTCB local, e estes módulos são interligados por um canal privado da TTCB. Este conjunto das TTCBs locais mais o canal privado formam a TTCB propriamente dita.

O objectivo desta aproximação é que os processadores continuem a comunicar usando a rede normal, mas que ocasionalmente usem algum dos serviços da TTCB para executarem algumas operações de forma segura. Assim, a TTCB fornece apenas um conjunto de serviços simples e de baixo nível, de forma a que seja possível verificar formalmente a sua correcção. Os serviços fornecidos incluem a geração de estampilhas temporais e acordo distribuído sobre um valor pequeno, actualmente 160 bits [13].

O modelo de sistema baseado na TTCB tem sido usado para desenhar diversos protocolos tolerantes a intrusões eficientes, como um protocolo de difusão fiável denominado BRM-M [8] e um protocolo de

consenso [10]. Recentemente foi proposto um serviço tolerante a intrusões baseado em replicação de máquinas de estados, que tem a vantagem importante de precisar de um número de réplicas inferior ao mínimo em sistemas assíncronos [12].

4 O Sistema Worm-IT

O Worm-IT é um sistema de comunicação para grupos de processadores [9, 11]. O sistema tem fundamentalmente duas componentes: o serviço de filiação e o serviço de comunicação. Se bem que conceptualmente sejam componentes distintas, na realidade partilham código. Os dois serviços são concretizados por um conjunto de três protocolos denominados COLLECT, PICK e RCAST (versões simplificadas encontram-se na Figura 2). Abaixo descreve-se sucintamente o objectivo e o funcionamento de cada um. O serviço de filiação utiliza informação de uma terceira componente, que não vai ser aqui descrita: o detector de falhas. Dado um grupo com n processos, o Worm-IT tolera $f = \lfloor \frac{n-1}{3} \rfloor$ processadores falhados.

O protocolo COLLECT é utilizado tanto pelo serviço de filiação como pelo serviço de comunicação. O seu objectivo é processar quatro tipo de eventos: *suspeita* de que um membro falhou, pedido de *entrada* de um novo membro, *saída* voluntária de um membro, e recepção de uma *mensagem* de comunicação. O protocolo tem dois estados: *Normal* e *Acordo*. O estado habitual é o primeiro mas, quando se verifica determinada condição, passa para o estado de acordo, no qual é executado o protocolo PICK (v. Figura 2).

Da mesma forma que o protocolo COLLECT, o protocolo PICK é utilizado tanto pelo serviço de filiação como pelo serviço de comunicação. O seu objectivo é o de fazer acordo entre todos os processadores correctos sobre as modificações a realizar à filiação do grupo (eventos *suspeita*, *entrada* e *saída*) e sobre as mensagens a entregar. O protocolo baseia-se no serviço de acordo da TTCB, o *Trusted Block Agreement (TBA)* [13]. Esse serviço é usado para chegar a acordo sobre uma síntese criptográfia – um *hash* [20] – dos eventos a realizar.

O protocolo RCAST é exclusivo do serviço de comunicação. O seu papel consiste em difundir uma mensagem por todos os processadores, garantido que todos os que estão correctos a recebem. A entrega de uma mensagem por este protocolo é tratada como um evento pelo COLLECT, que vai usar o PICK para decidir a sua entrega.

5 Concretização

Nesta secção vai ser descrita a concretização do sistema, começando por uma breve menção ao Appia e à concretização da TTCB que foi utilizada para as medidas de desempenho.

5.1 Appia

A concretização do Worm-IT é baseada na plataforma Appia [21][1], realizada em Java. Esta plataforma permite compor pilhas de micro-protocolos em tempo de execução. O objectivo do Appia é permitir composição de protocolos de forma versátil e com bom desempenho. Um micro-protocolo é especificado através de uma classe Java. A comunicação entre os protocolos de uma pilha é feita por eventos, que podem ser descendentes (p. ex., mensagem a ser enviada) ou ascendentes (mensagem recebida). Os

[1] http://appia.di.fc.ul.pt

COLLECT

1. Quando receber um evento, difundi-lo numa mensagem INFO para todo o grupo.
2. Quando receber $f + 1$ mensagens INFO com um mesmo evento que ainda não tenha difundido, difundi-lo numa mensagem INFO para todo o grupo.
3. Quando receber $2f + 1$ mensagens INFO com um mesmo evento de *suspeita*, *entrada* ou *saída*, ou $2f + 1$ mensagens INFO de $Nmsg$ eventos *mensagem*, passar para o estado *Acordo* e executar o protocolo PICK.
4. Quando o protocolo PICK terminar, fazer as alterações acordadas à filiação do grupo e entregar as mensagens acordadas por ordem das suas estampilhas temporais. Mudar para o estado *Normal*.

PICK

1. Propor ao TBA a síntese dos eventos reunidos pelo COLLECT. Aguardar que o TBA termine. Repetir as mesmas operações até que $2f + 1$ processadores tenham proposto.
2. Se tiver proposto a síntese decidida pelo TBA, difundir numa mensagem PICKED o conjunto de eventos para todo o grupo. Retornar.
3. Caso contrário, aguardar por uma mensagem PICKED cuja síntese dos eventos seja a decidida pelo TBA. Retornar quando esta mensagem for recebida.

RCAST

1. (Emissor) Quando for solicitado o envio de uma mensagem, juntar-lhe uma estampilha temporal, dar uma síntese ao TBA e difundi-la pelo grupo. Gerar um evento com a mensagem.
2. (Receptores) Quando for recebida uma mensagem, obter a síntese do emissor a partir do TBA. Caso a síntese da mensagem seja igual à obtida a partir do TBA, difundir a mensagem para todo o grupo e gerar um evento com a mensagem.

Figura 2: Protocolos do Worm-IT (executados por cada processador)

eventos são também especificados através de classes Java. Nas secções que se seguem a palavra *evento* vai ser reservada para os eventos tratados pelo protocolo COLLECT, enquanto que os do Appia serão denominados *evento-appia*.

5.2 TTCB

A concretização corrente da TTCB baseia-se em PCs convencionais [13][2]. A TTCB tem de ser segura, logo tem de ser isolada do resto do sistema. A melhor solução para realizar a TTCB local seria executá-la dentro de um módulo de hardware fisicamente isolado do resto do PC (por exemplo, uma placa PC/104 com processador e memória). No entanto, a solução adoptada foi outra. Na concretização corrente, a TTCB local é executada dentro de um núcleo tempo-real baseado em Linux, o RTAI [7]. Este núcleo é reforçado usando o mecanismo de *capabilities* do Linux, de forma a garantir a sua segurança [13]. Esta solução não é a ideal mas tem a vantagem de ser fácil de disponibilizar para outros investigadores que a pretendam utilizar.

[2]Disponibilizada em: http://www.navigators.di.fc.ul.pt/software/ttcb/

A solução usada para o canal de controlo é uma LAN Ethernet usada exclusivamente pela TTCB. Os PCs têm pois duas placas de rede: uma para a rede normal, outra para o canal da TTCB. A rede da TTCB pode ser considerada segura se se assumir que o atacante não lhe pode aceder fisicamente, o que faz sentido se se considerar o típico *hacker* da Internet. Esta rede tem um comportamento tempo-real dentro de certas condições, a principal das quais é o tráfego ser controlado, o que é garantido por um mecanismo de controle de admissão da TTCB [4].

```
public class TTCB {
  public TTCB() {}

  public native int openTCBConnection(descriptor_t descr);
  public native int closeTCBConnection(descriptor_t descr);
  public native int getTCBGlobalTimestamp(descriptor_t descr, timestamp_out_t ts);

  // TBA service
  public native propose_out_t propose(descriptor_t descr, short[] elist,
                                      long tstart, byte decision, byte[] value);
  public native decide_out_t decide(descriptor_t descr, short tag);
  ...
}
```

Figura 3: Interface da biblioteca Java da TTCB (excerto).

A TTCB local é constituída por um conjunto de módulos escritos em C inseridos no núcleo RTAI. Os serviços da TTCB podem ser acedidos através de duas bibliotecas, uma em C e outra em Java, que comunicam com os módulos do núcleo usando FIFOs tempo-real. A biblioteca Java faz chamadas às funções da bilioteca C usando a Java Native Interface [19]. Um excerto da interface da biblioteca Java é apresentado na Figura 3.

5.3 Worm-IT

Esta secção descreve a concretização do Worm-IT, cuja arquitectura é apresentada na Figura 4. O sistema é uma pilha de protocolos Appia, com excepções do protocolo PICK que é uma *thread* Java separada, por razões que serão discutidas mais adiante, e da TTCB local.

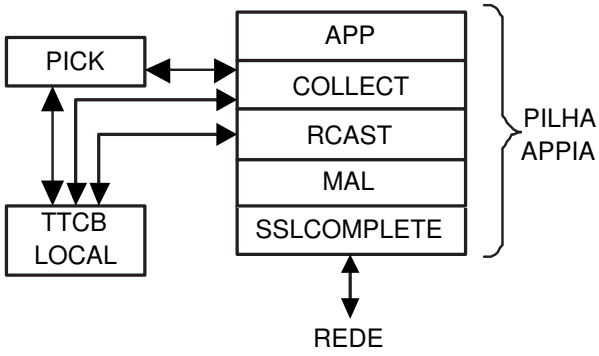

Figura 4: Arquitectura do sistema Worm-IT numa máquina.

evento-appia	evento Worm-IT
JoinGroupEvent	pedido de entrada de um processador
LeaveGroupEvent	pedido de saída de um membro
SuspectGroupEvent	suspeita de que um membro falhou
RCASTSendableMessageEvent	mensagem do protocolo de difusão atómica

Tabela 1: Correspondências entre eventos-appia e eventos do sistema Worm-IT

O Worm-IT processa quatro tipos de eventos, cuja correspondência a eventos Appia é mostrada na Tabela 1. Quando uma máquina pretende entrar num grupo, a sua camada aplicação (APP) difunde um evento-appia *JoinGroupEvent* para todo o grupo. Quando um processador pretende sair do grupo, a sua camada aplicação difunde um evento-appia *LeaveGroupEvent* para todo o grupo, incluindo para si próprio. A arquitectura não representa o detector de falhas dado que este não foi concretizado, mas o seu papel seria o de gerar eventos-appia *SuspectGroupEvent* sempre que suspeitasse da falha de algum processador. A camada aplicação inicia a difusão de uma mensagem de dados enviando uma evento-appia *RCASTMessageEvent*, que na camada RCAST é transformado num evento-appia *RCASTSendableMessageEvent* e difundido para o grupo. Todos estes eventos são processados pela camada COLLECT. As próximas secções descrevem as diversas camadas do sistema.

5.3.1 SSLComplete e MulticastAbstractionLayer

A camada SSLComplete garante a fiabilidade e a integridade da comunicação entre os processadores (v. Figura 4). Mais precisamente, garante que: (1) uma mensagem enviada por um processador correcto é sempre recebida por todos os processadores correctos (*Fiabilidade*); (2) nenhuma mensagem é modificada na rede (*Integridade*). Esta camada, que faz parte da distribuição do Appia, e garante estas duas propriedades através do estabelecimento de um canal SSL entre a máquina e cada uma das outras do grupo. O protocolo SSL [16] concretiza um canal seguro, podendo fazer autenticação das partes envolvidas e garantido a integridade e a confidencialidade da comunicação. Esta camada fornece uma primitiva de difusão que envia uma mensagem para cada uma das máquinas através de cada um dos canais SSL.

O objectivo da camada *MulticastAbstractionLayer (MAL)* é fundamentalmente o de permitir às camadas superiores do protocolo de comunicação utilizado. Os eventos enviados pelas camadas superiores têm um identificador de grupo, que esta camada traduz num conjunto de endereços IP específicos. Estes endereços são depois colocados num evento-appia *SendableEvent* que é enviado pela rede. Esta camada tem também o papel de serializar os dados dos eventos-appia descendentes e colocá-los num vector no *SendableEvent*. Para os eventos-appia ascendentes faz o contrário.

5.3.2 COLLECT e PICK

Quando o protocolo COLLECT recebe um evento Worm-IT sob a forma de um dos quatro eventos-appia na Tabela 1, difunde uma mensagem INFO para todo o grupo (v. Figura 2). Na concretização, esta mensagem INFO é um de quatro eventos-appia, correspondendo a cada um dos quatro eventos Worm-IT: *JoinInfoEvent*, *LeaveInfoEvent*, *SuspectInfoEvent* e *DataMsgEvent*. Quando estes eventos INFO são recebidos, são tratados pelo protocolo como esquematizado na Figura 2. Caso se verifique a condição da linha 3 do protocolo e caso o PICK não esteja a correr, é lançada uma *thread* Java para executar o PICK.

A concretização do protocolo PICK consiste num ciclo que faz uma série de chamadas ao serviço

TBA da TTCB. Para propor um valor é usado o método *propose* e para receber o resultado o método *decide*, ambos da classe TTCB (v. Figura 3). O valor proposto ao TBA é uma síntese dos eventos Worm-IT sobre os quais se pretende chegar a acordo. A função de síntese usada é a concretização distribuída com o Java do algoritmo SHA-1 [23]. O protocolo foi implementado numa *thread* à parte pois tem de se bloquear à espera do resultado da TBA. O Appia não suporta funções bloqueantes pois tem uma única *thread* que não pode ser bloqueada, sob o risco de parar a execução de todos os protocolos.

Quando o PICK termina, introduz na pilha um evento-appia *PICKResultEvent*. Quando o COLLECT recebe este evento-appia, começa por separar os eventos relativos a alterações de vista dos de mensagens de dados. Quanto a estas últimas, a camada envia à camada APP um evento-appia *RCASTToDeliverEvent* por cada uma por ordem de entrega. Quanto ao tratamento de eventos relativos a mudança de vista, é criada uma nova vista com base na decisão do PICK, o estado do sistema passa a Normal e a nova vista é enviada para a camada APP e para os novos membros do grupo. Por fim, é verificado se existem eventos que deviam ter sido propostos e não foram. Se isso acontecer, serão novamente propostos.

Um grupo é criado pelo seu primeiro membro. Enquanto não houver pelo menos quatro ($n = 4$) máquinas no grupo não é possível tolerar nenhuma falta, dado que o sistema tolera $f = \lfloor \frac{n-1}{3} \rfloor$ faltas. Logo, enquanto não existe pelo menos esse número de membros, o grupo está num estado especial no qual para ser tomada uma decisão, todos os membros têm de propor o mesmo.

5.3.3 RCAST

A camada RCAST é usada exclusivamente pelo protocolo de difusão atómica. Quando recebe um evento-appia *RCASTMessageEvent*, constrói um evento *RCASTSendableMessageEvent* e usa o método *propose()* da TTCB para entregar uma síntese da mensagem a todos os membros do grupo. Depois, gera um evento-appia *DataMsgEvent* que envia pela rede para todos os processadores do grupo.

Ao receber um *RCASTSendableMessageEvent* vindo da rede, cada processador verifica se já tinha recebido a mensagem e usa o método *propose()* para descobrir se alguém propôs a síntese. Se a mensagem corresponder à síntese, envia para todos os membros um evento-appia *DataMsgEvent*. Este reenvio provoca que cada receptor receba diversas cópias da mesma mensagem, mas é necessário para garantir que uma mensagem difundida por um processador malicioso não é recebida só por alguns dos processadores correctos. A decisão das mensagens a entregar não cabe a esta camada mas às camadas COLLECT e PICK.

5.3.4 Camada aplicação

A camada aplicação (APP) não faz propriamente parte do sistema mas fornece uma interface com o utilizador. A interacção directa com o utilizador é feita usando uma *thread* à parte, mais uma vez com o objectivo de não bloquear o Appia. Esta camada permite solicitar a entrada e a saída do grupo e provocar a suspeição de determinado membro. Permite também enviar um conjunto de mensagens medindo o débito e a latência média do sistema.

6 Medidas de desempenho

Esta secção apresenta medidas de desempenho do protocolo de difusão atómica. As experiências envolveram cinco PCs com processador Intel Pentium III a 500 Mhz, e 256MB SDRam PC133. Cada PC tinha dois adaptadores de rede 3Com 10/100. A rede normal e o canal de controle de TTCB eram

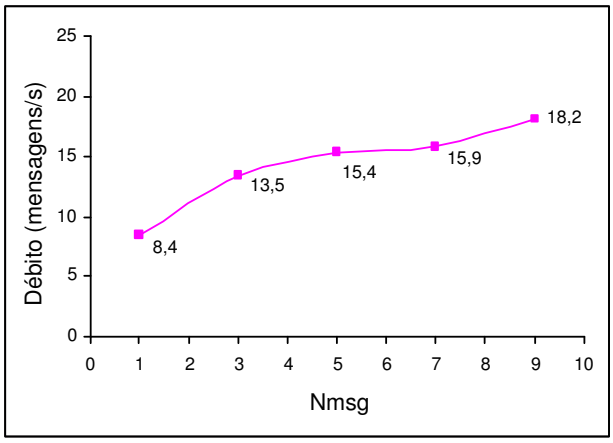

Figura 5: Variação do débito da difusão atómica com $Nmsg$ (mensagem vazia)

duas redes Fast-Ethernet switched a 100Mbps. As versões do software usadas foram: TTCB 1.11, RTAI 24.1.10, Java SDK 1.4.0 e Appia 1.9-2. O SSL usado foi o distribuído com o Java, na configuração por omissão.

Foram realizadas três experiências, tendo cada uma envolvido o envio de um mínimo de mil mensagens. A rede normal não tinha qualquer tráfego para além do gerado pelo sistema. Na primeira experiência foi medida a variação do débito do protocolo de difusão atómica com o parâmetro $Nmsg$, que indica o número de mensagens pelas quais o protocolo COLLECT espera antes de chamar o PICK para decidir a entrega de mensagens (v. Figura 2). A mensagem enviada não continha dados, apenas o cabeçalho. A Figura 5 permite concluir que o débito melhora com o aumento de $Nmsg$, o que seria expectável pois uma única execução do PICK decide a entrega de mais mensagens.

Na segunda experiência foi medida a variação do débito e da latência média do protocolo com o comprimento da mensagem, sendo o parâmetro $Nmsg$ fixo em $Nmsg = 10$ (Figura 6). Os resultados mostram que o débito diminui com o comprimento da mensagem, o que seria de esperar dado que maior comprimento implica maior consumo de tempo de processamento, por exemplo no cálculo das sínteses e na cifra das mensagens. Dado o peso do Java e do Appia, e tendo em conta anteriores avaliações de protocolos semelhantes concretizados em C [11], presume-se que a influência da largura de banda seja aqui negligenciável. A latência apresentada na figura é uma latência média. Esta grandeza pode variar bastante entre mensagens pois estas podem ser mais ou menos atrasadas enquanto esperam pela execução do protocolo PICK, o que só acontece quando chegam $Nmsg$ mensagens.

A terceira experiência compara o desempenho do protocolo de difusão atómica com todas as máquinas correctas versus o desempenho com uma máquina parada, ou seja, falhada (Figura 7). Os resultados mostram um prejuízo em termos de latência média e débito, o que se deve ao TBA perder algum tempo quando nem todos os processadores envolvidos propõem um valor [8].

7 Conclusões

Este artigo descreve a concretização de um sistema de comunicação em grupo tolerante a intrusões. Este tipo de sistemas é importante para simplificar o desenho e a programação de sistemas distribuídos. Por seu lado, a tolerância a intrusões é uma área que tem tido alguma atenção nos últimos anos, dados os

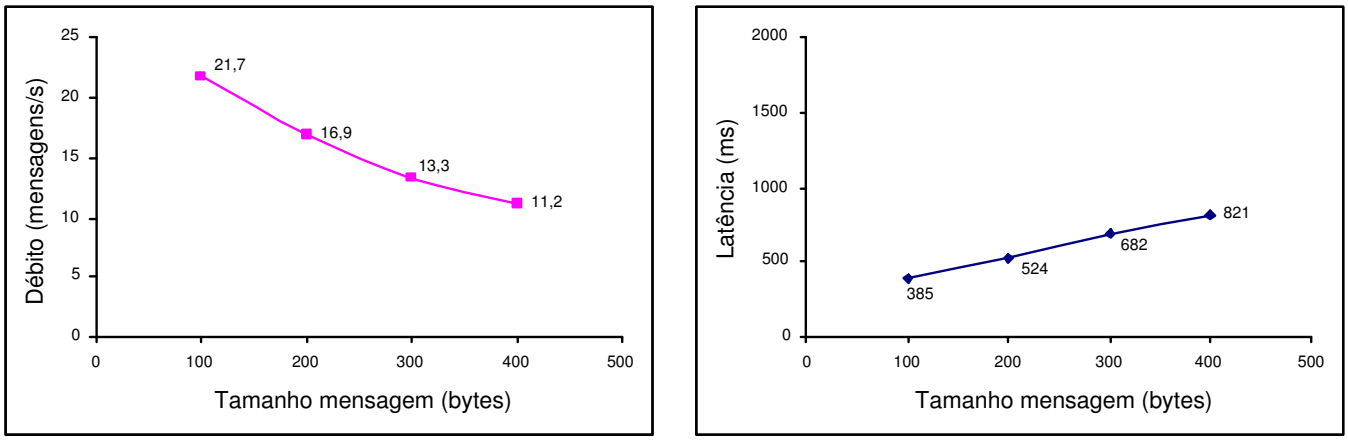

Figura 6: Variação do débito e da latência da difusão atómica com o comprimento da mensagem ($Nmsg = 10$)

enormes problemas de segurança que se continuam a verificar nos sistemas distribuídos correntes. Os autores do artigo têm estado envolvidos em investigação nesta área usando a abordagem aqui apresentada e que já mostrou benefícios relevantes em termos de resistência a faltas – permite tolerar mais faltas com o mesmo número de máquinas [8, 12] – e de desempenho [8, 11]. Estes artigo apresenta a concretização de um sistema usando Java e a plataforma Appia. Esta solução tem as vantagens de prevenir alguns ataques dado ser baseada em Java, e permitir a fácil integração com outros protocolos dado usar Appia. Apesar de o desempenho ser comparável ao de outros sistemas de comunicação tolerantes a intrusões como o Rampart [25], é claramente inferior a protótipos escritos em C [11].

Referências

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

[2] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 330–343, April 2000.

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

[4] A. Casimiro, P. Martins, and P. Veríssimo. How to build a Timely Computing Base using Real-Time Linux. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 127–134, September 2000.

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

[6] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

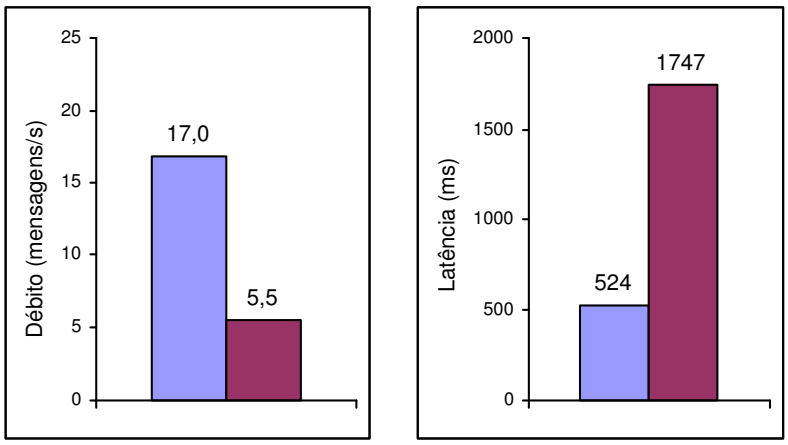

Figura 7: Débito e latência da difusão atómica sem máquinas falhas (claro) e com uma máquina falhada (escuro)

[7] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. DIAPM-RTAI position paper. In *Real-Time Linux Workshop*, November 2000.

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

[9] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. A wormhole-based intrusion-tolerant group communication system - WIT-GCS. In *The 5th Cabernet Plenary Workshop*, November 2003.

[10] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Low complexity Byzantine-resilient consensus. *Distributed Computing*, 2004. Aceite para publicação.

[11] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Worm-IT – a wormhole-based intrusion-tolerant group communication system. Submetido para publicação, 2004.

[12] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, October 2004.

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

[14] B. Dutertre, V. Crettaz, and V. Stavridou. Intrusion-tolerant Enclaves. In *Proceedings of the IEEE International Symposium on Security and Privacy*, pages 216–226, May 2002.

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

[16] K. Hickman. The SSL protocol. Netscape Communications Corp., February 1995.

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

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

[19] S. Liang. *Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley, 1999.

[20] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[21] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems Workshops*, pages 707–710, April 2001.

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

[23] NIST. Announcement of weakness in the secure hash standard, May 1994.

[24] H. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. 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, June 2002.

[25] M. Reiter. 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, November 1994.

[26] M. K. Reiter, K. P. Birman, and R. van Rennesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, November 1994.

[27] O. Rodeh, K. Birman, and D. Dolev. The architecture and performance of security protocols in the Ensemble group communication system. *ACM Transactions on Information and System Security*, 4(3):289–319, August 2001.

[28] P. Veríssimo. Traveling through wormholes: Meeting the grand challenge of distributed systems. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*, pages 144–151, June 2002.

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

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