Tolerância a Intrusões em Sistemas Informáticos

Nuno Ferreira Neves

DI-FCUL TR–05–7

May 2005

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749-016 Lisboa
Portugal

Technical reports are available athttp://www.di.fc.ul.pt/tech-reports. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Tolerância a Intrusões em Sistemas Informáticos

Nuno Ferreira Neves

O uso generalizado dos computadores pela sociedade para realização das mais variadas tarefas, é uma tendência que se tem verificado ao longo da última década, e que tudo indica, continuará nos anos vindouros. Esta mudança, no entanto, é acompanhada pelas suas vicissitudes, em especial no que toca à (in)segurança informática – a prática tem demonstrado que é extremamente difícil a construção de sistemas complexos totalmente imunes a intrusões. À medida que a complexidade cresce vão-se esgotando os paradigmas prevalentes: o de prevenir ou evitar as intrusões; e o de reagir às mesmas com a intervenção humana. Em resposta a esta questão, este trabalho propõe dotar os sistemas de mecanismos que lhes permitam tolerar as intrusões de um modo automático, impedindo que aquelas violem as suas propriedades.

O trabalho apresenta uma solução para a concretização de serviços replicados baseados em protocolos distribuídos, que assentam num modelo híbrido de sistema. Neste modelo, os protocolos são executados num ambiente com hipóteses muito fracas (e.g., quanto à sincronia e quanto ao modelo de ataque), podendo recorrer esporadicamente a um componente especial para o processamento de operações simples. A solução preconizada por esta arquitectura é atraente sob vários ângulos, nomeadamente porque possibilita o projecto de protocolos tolerantes a intrusões sem hipóteses temporais explícitas, com um bom desempenho, e com um elevado nível de segurança.

1 Introdução

A sociedade tem ao longo dos últimos anos ampliado a sua dependência nos sistemas computacionais para a realização das mais diversas actividades. A Internet tem servido como um importante catalizador nesta mudança, facilitando por um lado a integração de novos utilizadores, e por outro lado proporcionando um meio eficaz para a provisão de serviços. Assim sendo, tudo leva a crer, que esta tendência se irá manter e que um maior número de tarefas passará a ser realizado com auxílio dos computadores.

A experiência actual, no entanto, indica que esta transição não é isenta de vicissitudes. Os aspectos da segurança informática estão na ordem do dia, como indicam as notícias de ataques bem sucedidos, que tão frequentemente são descritas na imprensa. Como exemplos típicos destes ataques temos: «uma nova estirpe de verme espalha-se pela Internet, degradando significativamente o desempenho dos sistemas em várias empresas»; «uma intrusão numa base de dados de comércio electrónico resultou no furto de vários números de cartões de crédito»; «um grupo de piratas informáticos penetrou num servidor web e colocou um conjunto de mensagens obscenas nas páginas públicas da organização». Todos estes exemplos constituem indícios de um problema genérico e de difícil resolução – a construção de um sistema informático complexo completamente seguro, ou seja, sem vulnerabilidades que possam ser exploradas por um ataque de um adversário.

No entanto, a perspectiva clássica que tem sido seguida pela área da segurança informática é a da prevenção de intrusões, ou quando muito, da detecção de intrusões, embora ainda sem

formas sistematizadas para o processamento das mesmas. Esta perspectiva assenta no princípio de que é possível concretizar sistemas informáticos sem defeitos, e de os dotar de capacidades que detenham qualquer forma de ataque. À luz do que foi referido acima, esta visão é temerária, em particular porque actualmente se continuam a usar más práticas de programação e teste do software, os ciclos de desenvolvimento estão cada vez mais apertados, e é fácil aos mais variados utilizadores disponibilizar e/ou aceder aos produtos informáticos.

Em resposta a estas questões, pretende-se com este trabalho desenvolver sistemas que tolerem automaticamente as intrusões. Sucintamente, deseja-se empregar mecanismos capazes de tratar um conjunto diversificado de problemas (ou faltas), incluindo os ataques e intrusões, impedindo-se que estes causem a falha das propriedades de segurança dos sistemas. Na solução proposta, em vez de se prevenirem todas as intrusões, assume-se à partida que algumas delas irão ter sucesso, e depois constroem-se os sistemas de modo a que elas não produzam danos (ou os façam de uma forma limitada).

A classe dos sistemas ou aplicações que irão ser considerados é constituída por um conjunto de entidades (e.g., processos, servidores e clientes) que se encontram distribuídas por máquinas diferentes e que usam uma rede para trocarem dados, com o propósito de cooperarem e realizarem uma tarefa. Como se pretende projectar soluções que permitam tolerar quer as faltas acidentais (e.g., a paragem de uma máquina) como as maliciosas (e.g., as intrusões), é aconselhável que se assumam adversários realistas mas com capacidades alargadas, e que se façam hipóteses fracas em relação ao ambiente computacional. Em princípio, será mais difícil assegurar na prática hipótese fortes do que fracas, em particular porque os adversários muitas vezes dirigem os seus ataques com o intuito de as violarem.

A classe de faltas que vai ser assumida é a mais genérica de todas (abarca as outras) e é chamada de *arbitrária* (ou *Bizantina* [LA82]). As faltas arbitrárias possibilitam às entidades afectadas um qualquer comportamento, inclusive o ataque aos protocolos executados pelas aplicações. Como os ataques também podem ser direccionados a atrasar a prossecução das operações que se desenrolam no sistema (e.g., com ataques de negação de serviço), é obrigatória a adopção de um modelo de sincronia muito fraco, o *assíncrono* [FI85], [LY96]. No modelo assíncrono não é possível definir limites para o tempo que demoram as actividades do sistema, o que impede o uso de hipóteses simplificadoras, tais como a existência de intervalos máximos para a transmissão de mensagens, no projecto dos protocolos.

Neste trabalho é apresentada uma solução para a construção de sistemas distribuídos tolerantes a intrusões. Esta solução inclui dois protocolos tolerantes a intrusões, em que o primeiro resolve uma das variantes dum problema clássico na área dos sistemas distribuídos -- o acordo vectorial [DO97], e o segundo efectua a coordenação de servidores replicados. Ambos os protocolos têm uma relevância considerável porque, por um lado, o acordo serve de base para resolução de muitos problemas importantes em sistemas distribuídos [GU96], e por outro lado, como o protocolo de replicação é genérico, pode ser empregue na concepção de um número variado de serviços (e.g., servidores de ficheiros ou de nomes).

No acordo vectorial, os processos arrancam com um valor, e quando terminam, devem decidir num mesmo vector. O vector é caracterizado por conter uma maioria de valores de processos correctos (ou seja, não falhados). Este problema, embora tenha uma especificação relativamente simples, não é passível de uma solução determinista em sistemas assíncronos

sujeitos a faltas por paragem[1] [FI85]. Assim, ao longo dos anos, alguns investigadores propuseram mecanismos para contornar este resultado, que podem ser agrupados em duas classes de soluções. A primeira recorre à aleatoriedade [RA83], [BE83], e a segunda adiciona hipótese temporais ao modelo assíncrono. As hipóteses temporais tanto têm aparecido como extensões explícitas ao modelo de sistema [DL87], [DW88], como através do seu encapsulamento em algum artefacto, por exemplo os detectores de falhas [CH96]. Neste último caso, o sistema continua a ser assíncrono, com excepção do detector de falhas que efectua hipóteses temporais na sua concretização.

Até agora, apenas dois outros protocolos foram propostos para a resolução do problema do acordo vectorial [DO97], [BA00]. Ambos se baseiam na existência de detectores de falhas capazes de descobrir condutas maliciosas, ou seja, faltas do tipo arbitrário. Dum ponto de vista pragmático, é muito complicado construir estes detectores porque é intrinsecamente difícil a especificação de todas as acções maliciosas e ao mesmo tempo a sua separação das honestas[2]. Por acréscimo, os piratas informáticos tendencialmente atacam os sistemas de detecção de intrusões, por exemplo com ferramentas que simulam um extenso número de ataques desirmanados [DE01], tornando o seu uso inútil.

Este trabalho segue uma aproximação diversa, que se baseia em enriquecer o sistema com um componente distribuído com propriedades mais fortes, preconizando a ideia dum modelo híbrido de sistema. O componente, a que se chama um *wormhole*, fornece uns poucos serviços às aplicações, que normalmente não seriam realizáveis no ambiente que se está a considerar. Por exemplo, poderia oferecer uma operação segura, que contrariamente às operações "normais", devolveria sempre o resultado correcto, ou no pior dos casos, não retornaria nada. Para que estes comportamentos sejam garantidos com um grau de confiança satisfatório (ou com uma taxa de cobertura elevada), é imprescindível que haja um cuidado especial na construção do wormhole. Em particular, ele deve ser mantido simples e pequeno, para que a sua funcionalidade seja validada.

O protocolo de acordo vectorial é executado na parte assíncrona do sistema, onde podem surgir faltas arbitrárias. Logo, o seu objectivo é garantir que os processos correctos conseguem decidir num vector, ainda que um subconjunto de processos maliciosos (no máximo f num total de $N \geq 3f+1$) os tentem enganar, e que as mensagens sejam atacadas durante a transmissão. Para atingir este objectivo, os processos recorrem a um serviço de votação oferecido por um wormhole, num ponto crítico do seu programa.

A replicação é uma técnica bem conhecida para melhorar a disponibilidade de um serviço e/ou aumentar o seu desempenho. De uma forma simplista, um servidor é copiado para várias máquinas, passando qualquer uma delas a responder aos pedidos dos clientes. Assim, se alguma das máquinas tiver uma paragem ou ficar sobrecarregada, as outras podem tomar o seu lugar. A concretização da replicação terá maior ou menor complexidade conforme as operações que são executadas pelos serviços. Por exemplo, se as operações forem unicamente de leitura, ou seja, apenas devolvem a informação que está guardada nos servidores, então é relativamente fácil a replicação. Se pelo contrário, as operações causarem alterações, então é necessário que exista

[1] A este resultado chama-se muitas vezes a *impossibilidade FLP*, em honra dos seus autores – M. Fischer, N. Lynch, e M. Paterson.

[2] Por exemplo, o detector de falhas observa um acesso ao sítio www.piratas-informáticos.net de onde são descarregadas aplicações de ataque. Será que esta é uma acção maliciosa? Depende das intenções do seu autor (e.g., se é um pirata ou um técnico de segurança), mas isto não é do conhecimento do detector.

uma coordenação entre os servidores que assegure que os seus estados (ou seja, os dados armazenados) evoluem de um modo semelhante ao longo do tempo.

A maioria dos protocolos de coordenação só tolera faltas por paragem, o que os torna completamente indefesos face a intrusões. Normalmente, basta a um adversário controlar um dos servidores para que seja capaz de enganar os clientes com respostas erradas, e consiga corromper o estado dos outros servidores. Recentemente foram propostos alguns mecanismos de replicação tolerantes a faltas arbitrárias. A solução que mais tem sido descrita consiste em basear a replicação num sistema de comunicação em grupo tolerante a intrusões, destacando-se nesta vertente o Rampart [RE95] e o Sintra [CA02]. Outra possibilidade consiste em se desenvolverem soluções para aplicações específicas, como o COCA [ZH02] para as autoridades de certificação, e o CODEX [MA04] para a distribuição de chaves. O BFT [CS02] é um protocolo que emprega um servidor primário para organizar os outros.

O protocolo de replicação que irá ser descrito segue uma aproximação diferente das anteriores – usa o acordo vectorial para coordenar o processamento dos servidores. Os servidores ao receberem os pedidos recorrem ao acordo para decidirem por que ordem os devem tratar, garantindo assim que os seus estados evoluem de um modo análogo. Uma das características interessantes do protocolo é que tem um bom comportamento face ao aumento da carga, porque cada acordo permite a ordenação de um número elevado de pedidos.

O wormhole utilizado neste trabalho pode ser realizado de diferentes maneiras. Por exemplo, com recurso a hardware suplementar ou então apenas com alterações ao software que corre na máquina. Independentemente da forma de concretização que se adoptar, é necessário que seja projectado um protocolo que materialize o serviço de votação. Para tornar este documento auto-suficiente, é apresentado um protocolo de votação para um modelo de sincronia muito fraco (embora mais potente que o assíncrono). As hipóteses efectuadas por este modelo são facilmente verificadas na realidade desde que as propriedades de segurança do wormhole sejam asseguradas.

Principais contribuições deste trabalho:

1) Um modelo híbrido para sistemas distribuídos, que assenta na existência de componentes com propriedades mais fortes, que normalmente não poderiam ser garantidas no resto do sistema, por este ter hipóteses muito fracas quanto à sincronia e tipos de faltas. Os wormholes são os artefactos que materializam esta noção. O modelo tem servido de base a várias publicações em revistas e conferências internacionais, tendo o autor deste documento participado em oito delas. Este modelo foi também um dos mais importantes resultados dum projecto que recentemente foi finalista num prestigiado prémio internacional.
2) Um protocolo de acordo vectorial que explora um serviço de votação oferecido por um wormhole. Duas características distinguem esta solução: a) contorna o importante resultado de impossibilidade FLP sem ter que fazer explicitamente qualquer hipótese temporal (algo que ainda não se tinha conseguido com os outros wormholes); b) do ponto de vista de desempenho, tem uma boa complexidade temporal quando comparado com as duas outras propostas para acordo vectorial. Esta solução vai ser este ano objecto de publicação numa revista de alto prestígio.

3) Um protocolo para a gestão dum sistema replicado tolerante a intrusões, e um protocolo de votação que materializa o serviço do wormhole. Ambos os protocolos têm características interessantes, salientando-se para o primeiro a sua independência face a hipóteses temporais (a maioria dos mecanismos descritos anteriormente necessitam destas hipóteses, e por isso são vulneráveis a certos tipos de ataques) e o bom comportamento em relação ao aumento de carga; para o segundo realçam-se as fracas hipóteses de sincronia em que é baseado. Esta parte do trabalho foi desenvolvida nos últimos meses, esperando-se que seja submetida para publicação ao longo deste ano.

Principais contribuições do autor do trabalho: Com a excepção da primeira contribuição em que também participou, as outras são fundamentalmente da responsabilidade do autor deste documento. O modelo híbrido foi e está a ser desenvolvido por um grupo de investigadores, tendo o autor deste trabalho feito contribuições nos aspectos da interface, segurança e enfraquecimento da sincronia do wormhole, entre outros.

2 Modelo do Sistema

O sistema é composto por um conjunto de máquinas que usam uma rede para trocarem dados e cooperarem (ver topo de Figura 1). As máquinas executam um número variado de softwares, incluindo um sistema operativo, bibliotecas e servidores de suporte (e.g., serviços de autenticação e autorização) e aplicações diversas (e.g., servidores de web e bases de dados). Adicionalmente, as máquinas contêm um *wormhole*. O wormhole é um componente distribuído com uma parte local em cada máquina – o *wormhole local*, e uma rede privada – a *rede de controlo* (ver discussão sobre a concretização do wormhole na Secção 5). As aplicações tratam o wormhole como um qualquer outro software de suporte, utilizando os seus serviços sempre que o desejarem. O wormhole, no entanto, tem uma característica fundamental que o separa do resto do sistema – possui propriedades mais fortes de sincronia e de segurança.

As aplicações são executadas por um grupo $P = \{p_1, p_2, \ldots, p_N\}$ de N processos. Os processos correm em máquinas diferentes e transmitem mensagens apenas pela rede normal (ver fundo da Figura 1). Ocasionalmente, recorrem aos serviços do wormhole através de chamadas às operações exportadas pela sua interface (ver Secção 2.5). A realização dos serviços requer às vezes a comunicação entre wormholes locais. Sempre que tal acontece, é usada a rede de controlo em vez da rede habitual.

2.1 Hipóteses de Falha

O sistema, com a excepção do wormhole, está sujeito a faltas arbitrárias. Logo, é permitido aos processos exibirem um comportamento indiscriminado sempre que falhem. Por exemplo, podem interromper a sua execução, saltar alguns passos dos protocolos, fornecer informação errada aos outros processos e wormhole, ou entrar em conluio com os demais processos falhados para tentarem quebrar as propriedades dos protocolos. Com o intuito de tornar esta hipótese a mais conservadora possível, vai-se assumir que um adversário controla todos os processos falhados, e que os faz actuar em cada instante da maneira mais perniciosa para os protocolos. Mais ainda, sempre que o adversário corrompe um novo processo, passa a conhecer

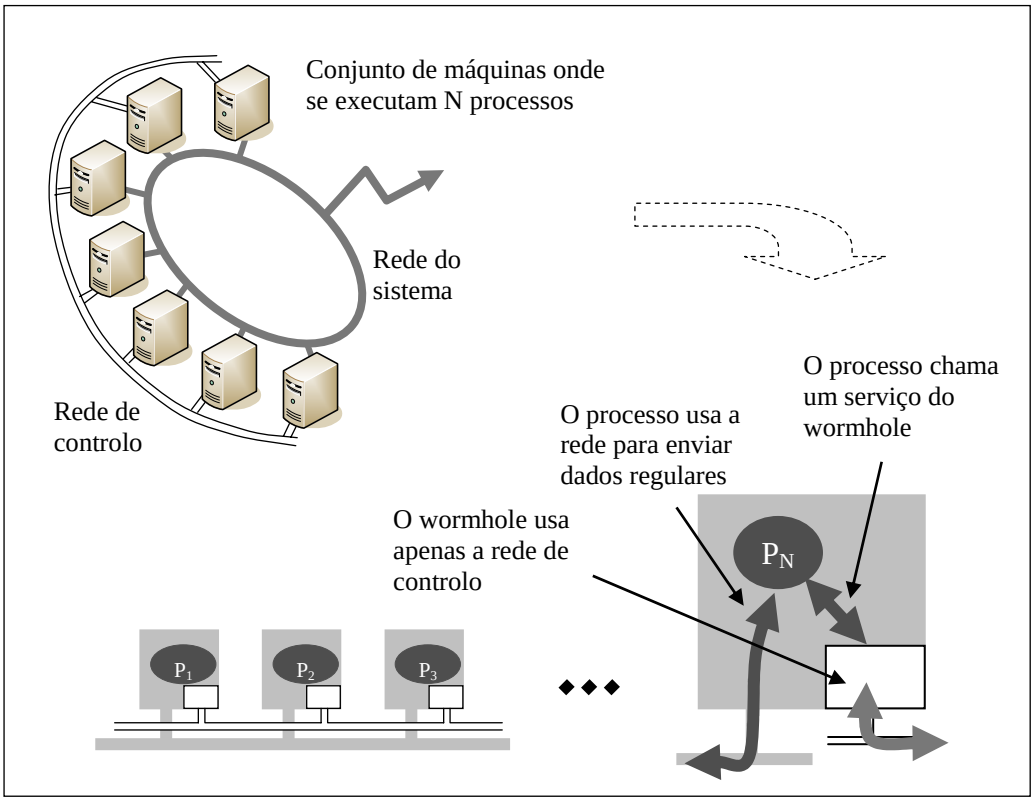

Figura 1: Arquitectura de um sistema com um wormhole distribuído (o wormhole encontra-se representado a branco, e o resto do sistema a cinzento).

todos os seus segredos (as chaves) e a usá-las como quiser.

A rede também tem faltas arbitrárias. Mais uma vez, para se ser conservador, vai-se considerar que um adversário controla a rede, com a capacidade por exemplo de remover, atrasar, ou alterar as mensagens enquanto estão em trânsito. O envio e/ou recepção de dados por um processo pode igualmente ser interrompido durante um período incerto mas limitado. Se a interrupção fosse permanente, então o processo seria incapaz de comunicar, e logo de executar a aplicação, estando sob o ponto de vista prático falhado. Faz-se então uma hipótese muito fraca em relação aos canais de comunicação dos processos correctos:

- ***Canais imparciais:*** se um processo correcto envia uma mensagem um número infinito de vezes para um dado destino, então ela vai ser recebida correctamente um número infinito de vezes.

As chamadas aos serviços do wormhole correspondem a uma forma de comunicação, embora na mesma máquina, e por consequência, também sofrem de faltas arbitrárias (e.g., os argumentos e/ou resultados são alterados maliciosamente). Para simplificar a descrição no resto do documento, vai assumir-se que se a invocação do serviço for afectada por uma falta, o processo fica contaminado e falha.

Deve-se notar, que este modelo de ataque atribui ao adversário um poder considerável, superior ao necessário para perpetrar muitas das intrusões bem sucedidas nos sistemas em funcionamento hoje em dia. Por maioria de razão, um sistema que, neste modelo de ataque, continue a funcionar correctamente, será consideravelmente mais robusto que as tecnologias correntes.

O wormhole, que inclui a rede de controlo, apenas tem falhas por paragem. Logo, um wormhole local ou devolve as respostas certas ou simplesmente deixa de se executar. Esta hipótese deve-se verificar ainda que um adversário consiga penetrar numa máquina e ataque o wormhole local.

Quer o protocolo de acordo vectorial como o de replicação toleram um limite máximo de f processos falhados, para um total de $N \geq 3f+1$. Este nível de robustez é o tipicamente necessário na resolução de problemas análogos e em sistemas com hipóteses semelhantes.

2.2 Hipóteses de Sincronia

O sistema assume um tipo de sincronia muito fraco, o modelo assíncrono [FI85], [LY96]. Neste modelo, as várias actividades do sistema não têm uma duração bem definida, embora esta duração seja limitada. Por exemplo, não é possível identificar um intervalo máximo para a transmissão de uma mensagem ou para a execução de uma operação por um processo (o que inclui as chamadas aos serviços do wormhole). Ao longo do tempo o sistema vai progredindo na direcção de concluir as tarefas que lhe foram pedidas, mas a uma velocidade desconhecida. Este modelo é particularmente interessante para sistemas com faltas arbitrárias porque os atrasos introduzidos propositadamente por ataques são facilmente tratados.

O wormhole tem o nível de sincronia necessário para garantir que os seus serviços terminam e que um resultado é devolvido aos processos. Neste trabalho vai-se adoptar um modelo bastante fraco, que é definido pela seguinte propriedade (este modelo corresponde a um enfraquecimento descrito por [CH96] dos modelos originalmente propostos por [DW88]):

- ***Modelo parcialmente síncrono do wormhole:*** Em cada execução, a partir de um certo instante desconhecido chamado *TEG* (de *Tempo de Estabilização Global*), passam a existir limites para o tempo de transmissão de mensagens pela rede de controlo e velocidade relativa dos wormholes locais, embora não se saibam os valores destes limites.

Este modelo assume basicamente que, após um período de instabilidade inicial, a rede de controlo e os wormholes locais começam-se a comportar de uma maneira favorável, o que possibilita a conclusão dos protocolos. É de notar, no entanto, que o intervalo de instabilidade (fixado por TEG) é potencialmente distinto por cada execução dum protocolo, mas admitindo que a rede e as máquinas normalmente funcionam bem, na maioria dos casos terá o valor zero. Por outro lado, os limites para a transmissão de mensagens e velocidade relativa apenas necessitam de se manter até que o protocolo termine, e também estes serão diferentes por cada execução.

2.3 Operações Criptográficas

Os protocolos utilizam algumas operações criptográficas para restringirem as capacidades de ataque dos piratas informáticos. É assumido ao longo do documento que estas operações não são subvertidas, ou seja., que as suas propriedades nunca são violadas. Muito brevemente, as operações que vão ser usadas são[3]:

- ***Operação de Síntese (do inglês* hash function*):*** é uma operação que recebe um conjunto de dados de tamanho arbitrário e devolve uma síntese com um tamanho fixo. A síntese serve de representante dos dados originais. Normalmente é calculada de uma forma muito eficiente e tem um tamanho reduzido (16 ou 20 bytes). Esta operação deve verificar um conjunto de propriedades, entre elas a resistência fraca e forte a colisões.
- ***Operação de cifrar e de decifrar:*** a operação de *cifrar* transforma um texto legível (ou em claro) num texto obscurecido (ou cifrado, ou criptograma), e a operação de *decifrar* faz a transformação inversa. Usualmente, as duas operações são configuradas por chaves. Caso a *chave de cifrar* seja igual à *chave de decifrar*, está-se na presença dum sistema de criptografia simétrica. No caso oposto, está-se a usar um sistema de criptografia assimétrica. Na criptografia assimétrica, a chave de cifrar é tornada pública (e por isso é chamada de *chave pública*) para que qualquer entidade possa enviar dados cifrados para um qualquer destino. A chave de decifrar nunca é mostrada, e por isso chama-se *chave privada*. Existem algumas propriedades asseguradas por estas operações. Por exemplo, é impossível obter-se o texto em claro a partir de um cifrado sem se conhecer a chave de decifrar.
- ***Operação de assinatura digital:*** possibilita que uma entidade associe a sua identificação a um conjunto de dados. A operação de *assinar* recebe um conjunto de dados e devolve uma assinatura, e a operação de *verificar* confirma que uma assinatura foi criada por uma dada entidade. Normalmente a assinatura tem um tamanho pequeno e é transferida com os dados. O receptor depois verifica a assinatura para comprovar a identidade do emissor. As assinaturas digitais verificam algumas propriedades, tais como: qualquer alteração nos dados impede a verificação da assinatura; é inexequível forjar uma assinatura em nome de outra entidade.
- ***Operação MAC (do inglês* Message Autentication Code*):*** pode ser vista como uma forma de assinatura digital com propriedades mais fracas, mas com um cálculo significativamente mais eficiente. Os MACs são habitualmente gerados com um algoritmo de síntese e uma chave partilhada entre os interlocutores. O emissor cria um MAC baseando-se nos dados, e envia ambos numa mensagem. O receptor usa os dados que lhe chegam para calcular um MAC, e depois compara-o com o MAC que veio na mensagem. Caso sejam distintos, a mensagem deve ser eliminada porque sofreu algum ataque na sua integridade e/ou autenticidade. A principal propriedade em que os MACs diferem das assinaturas digitais, é a não-repudiação.

[3] Para um tratamento completo sobre criptografia, recomenda-se a consulta de um dos muitos livros que versam sobre este tema. Um livro relativamente completo sobre esta matéria é [ME97].

2.4 Canais Seguros

As operações criptográficas são usualmente utilizadas na construção de canais de comunicação com propriedades mais fortes do que aquelas que são oferecidas pela rede. Para facilitar a exposição dos protocolos, vai-se assumir que cada par de processos está ligado por um *canal seguro* com as seguintes características:

- ***Fiabilidade:*** se p_i e p_k são processos correctos e se p_i envia a mensagem M a p_k, então p_k irá mais tarde ou mais cedo receber M.
- ***Integridade:*** se p_i e p_k são processos correctos e se p_k recebe a mensagem M em que é indicado como emissor o p_i, então M foi realmente enviada por p_i e M não foi alterada na rede.

Estas duas propriedades podem ser concretizadas de um modo relativamente simples. A fiabilidade é conseguida se as mensagens forem periodicamente retransmitidas até que uma confirmação seja recebida (não esquecer a hipótese de canais imparciais). A adição de MACs às mensagens permite a realização da segunda propriedade. O uso desta operação criptográfica, no entanto, obriga a que cada par de processos partilhe uma chave. Existem diversas maneiras de entregar estas chaves aos processos, por exemplo, manualmente num ficheiro de configuração em cada máquina, ou através dum qualquer serviço de distribuição de chaves.

2.5 Interface do Wormhole

O wormhole fornece um número reduzido de serviços aos processos. Os protótipos destes serviços encontram-se na Tabela 1. O *Serviço de Autenticação Local* normalmente não é usado directamente pelos protocolos, mas pelo código que configura as aplicações, servindo para iniciar a comunicação entre o processo e o wormhole local. Este serviço devolve um identificador único para o processo, *eid*, que deverá ser usado em todas as interacções futuras com o wormhole.

Tabela 1: Interface to wormhole.

Serviço de Autenticação Local: *eid ← W_Local_Auth()*
Serviço de Votação: *error, value, voted-ok ← W_Voter(eid, elist, vid, quorum, value)*

No *Serviço de Votação* os processos propõem um valor e recebem como resultado o valor mais votado (em caso de empate, é escolhido aleatoriamente um dos valores mais votados). Como os recursos do wormhole são restritos e precisam de ser partilhados por todas as aplicações, é imposto um limite máximo ao tamanho dos valores (20 bytes). Em acréscimo, o serviço deve ser usado apenas em alturas críticas na execução dos protocolos. O wormhole emprega mecanismos de controlo de admissão, para se precaver contra a exaustão dos seus recursos, que atrasam o processamento de pedidos dos processos abusadores. Assinala-se, no entanto, que o atraso não deverá pôr em causa a correcção dos protocolos, uma vez que eles foram projectados para sistemas assíncronos.

O serviço assegura as seguintes propriedades:

- ***Integridade:*** Os vários wormholes locais correctos no máximo decidem num resultado.
- ***Concordância:*** Dois wormholes locais correctos não decidem de forma diferente.
- ***Validade:*** Se um wormhole local decide um resultado, então este resultado baseia-se no valor mais votado (ou em caso de empate, num dos valores mais votados) num conjunto com pelo menos *quorum* propostas de valores.
- ***Terminação:*** Se chegarem *quorum* propostas aos wormholes locais correctos, então todos os wormholes locais correctos decidem num resultado.

Os parâmetros do serviço de votação têm os seguintes significados (ver Tabela 1). O *eid* é o identificador mencionado atrás, e *elist* é uma lista com os *eids* dos processos que participam na votação. Cada votação para um mesmo grupo de processos necessita de ter um identificador diferente, indicado no parâmetro *vid*. A votação deve incluir no mínimo *quorum* propostas de processos distintos. O resultado da votação contém a seguinte informação: (1) um código de erro[4]; (2) o valor mais votado; e (3) uma máscara *voted-ok* com um bit seleccionado por cada processo que propôs o valor retornado.

Uma vez que as máquinas não se executam à mesma velocidade, sendo influenciadas por factores como as cargas respectivas, pode suceder que um processo proponha o seu valor depois de uma votação já ter terminado. Neste caso, tal como era de esperar, o valor proposto não é utilizado, mas o processo obtém o resultado da votação. Para simplificar a descrição do protocolo de acordo, será assumido que o wormhole armazena resultados de votações durante muito tempo. Na realidade, o wormhole terá de eliminar periodicamente os resultados mais antigos, o que faz com que certos processos atrasados sejam incapazes de recolher alguns resultados de votações. Quando tal acontece, e uma vez que os resultados são indispensáveis para a execução dos protocolos, os processos afectados devem ser forçados a falhar através de uma paragem. Nota-se que esta acção não irá comprometer a correcção dos protocolos, uma vez que eles são tolerantes a faltas arbitrárias (aquele comportamento apenas corresponde a um tipo específico de falta).

3 O Acordo Vectorial

O acordo é um problema importante porque pode servir de base para resolução de um conjunto grande de outros problemas em sistemas distribuídos. Ao longo dos anos, foi usado na concepção de diversos protocolos tolerantes a faltas, tais como a confirmação em duas fases (do inglês *two phase commit*), gestão da filiação em grupos ou difusão de mensagens com ordem total [GU96].

Nesta secção descreve-se um protocolo para o acordo vectorial num sistema assíncrono. O sistema, com a excepção do wormhole, pode sofrer de faltas arbitrárias. A rede é usada para transmissão da maioria dos dados, incluindo os valores e os vectores dos processos. Contudo, num certo ponto da execução, o protocolo utiliza o serviço de votação do wormhole para a selecção de um conjunto reduzido de dados, o que assegura que uma decisão é mais tarde ou mais cedo adoptada. Por razões de limitação de espaço não vão ser incluídas as provas de correcção deste e dos próximos protocolos.

[4] Existem algumas ocasiões em que pode ser preciso devolver uma indicação de erro. Por exemplo, é um erro fornecer uma *elist* com um número de elementos inferior a *quorum*.

3.1 O Problema do Acordo Vectorial

O problema do acordo vectorial tem uma especificação relativamente simples. Inicialmente, cada um dos processos do grupo tem um valor. Quando o protocolo termina, todos os processos correctos devem decidir num mesmo vector, que é calculado com os valores originais. Mais precisamente, o acordo vectorial é definido pelas seguintes propriedades [DO97]:

- ***Validade:*** Cada processo correcto decide num vector *vect* com tamanho *N*, tal que:
  - Para cada $1 \leq i \leq N$, se o processo p_i é correcto, então *vect[i]* é o valor original de p_i ou então um valor inválido $\perp$, e
  - pelo menos $f+1$ elementos do vector *vect* são os valores originais de processos correctos.
- ***Concordância:*** Nenhum processo correcto decide vectores diferentes.
- ***Terminação:*** Todos os processos correctos decidem inevitavelmente.

A primeira propriedade estabelece o tipo de vectores que podem ser decididos pelo acordo. Sabendo-se à partida que existem $N \geq 3f+1$ processos, então o vector só é válido se tiver uma maioria de valores de processos correctos – os maliciosos podem no máximo preencher *f* elementos, e os restantes são os valores originais dos correctos ou o valor inválido $\perp$. As outras duas propriedades asseguram que uma decisão é tomada mais cedo ou mais tarde, em que todos os correctos têm vectores idênticos.

3.2 Protocolo de Acordo Vectorial

Os processos, antes de começarem a usar o protocolo, necessitam de executar algumas operações de configuração. Nomeadamente, devem chamar o serviço de autenticação local do wormhole para obterem um identificador no sistema, *eid*, e devem seleccionar o grupo de processos que irá participar no acordo. Adicionalmente, os processos precisam de arranjar um identificador único para cada execução do protocolo.

Um processo inicia o protocolo quando chama a função *consensus()* (ver Figura 2). Esta função tem três argumentos em que os dois primeiros devem ser iguais em todos os processos. Caso contrário, mais do que uma instância do protocolo é começada, acabando estas por serem realizadas em paralelo. Os argumentos têm os seguintes significados: *elist* é um vector que contém os identificadores de todos os processos que irão fazer o acordo, *cid* é o identificador único deste acordo, e $value_i$ é o valor que é proposto pelo processo p_i. O protocolo não impõe quaisquer limitações ao tamanho do valor, o que indica que este poderia ter, por exemplo, um bit ou um bilião de bytes.

A execução do protocolo tem duas fases. Na primeira, um processo assina o seu valor e difunde o valor assinado pela rede (Linha 6 e mensagem **B-value** em 8). Em seguida, constrói um vector com $2f+1$ entradas, preenchidas com os valores iniciais assinados de diversos processos (Linha 7 e 9-13). A posição *i* do *vector-of-vectors* contem o vector do processo p_i, e a posição *k* do vector *vector-of-vectors[i]* tem o valor assinado que foi recebido por p_i do processo p_k. O vector produzido por p_i possui duas características importantes: a) os valores armazenados são iguais ao enviado, sem qualquer adulteração na rede, visto que foram transmitidos por canais seguros (ver Secção 2.4); b) inclui pelo menos $f+1$ valores de processos correctos, uma vez que no máximo existem *f* processos maliciosos. No entanto, os vectores produzidos pelos vários processos deverão apresentar diferenças porque as mensagens **B-value** podem não chegar

```
1   function consensus(elist, cid, value_i)
2   round ← 0                                                    {número da iteração}
3   hash-v ← ⊥                                                   {síntese do vector seleccionado}
4   decide-set ← ∅          {conjunto com as mensagens Decide que foram recebidas}
5   vector-of-vectors ← ((⊥, ..., ⊥), ..., (⊥, ..., ⊥))   {vector com um vector por cada p_k}

6   signed-value_i ← sign(i, value_i)                            {assinar o meu valor}
7   vector-of-vectors[i][i] ← signed-value_i                     {guardar o valor no meu vector}
8   broadcast B-value = < i, cid, signed-value_i >               {usar a rede normal}
9   repeat
10      receive B-value = < k, cid, signed-value_k >
11      if (signed-value_k is correctly signed) then
12          vector-of-vectors[i][k] ← signed-value_k             {guardar o valor de p_k}
13  until (vector-of-vectors[i] has 2f+1 values from different processes)
14  broadcast B-vector = < i, cid, vector-of-vectors[i] >        {usar rede normal}
15  activate task(T1, T2)                                        {iniciar duas tarefas concorrentes}

16  Task T1:
17  when receive B-vector = < k, cid, vector_k > do
18      vector-of-vectors[k] ← vector_k
19  when receive Decide = < k, cid, vector_k > do
20      decide-set ← decide-set ∪ {vector_k}
21  when (hash-v ≠ ⊥) and (∃ vector_k ∈ decide-set : Hash(vector_k) = hash-v)) do
22      return (vector_k)

23  Task T2:
24  while (true) do
25      index ← (round mod n) + 1
26      round ← round+1
27      vid ← (cid, round)
28      v ← getNextGoodVector(vector-of-vectors, index)          {obter vector a propor}
29      out ← W_VOTER(eid, elist, vid, 2f+1, Hash(v))
30      if (at least f+1 proposed the same value) then
31          if (out.value = Hash(v)) then
32              if (not all processes proposed the same value) then
33                  send Decide = < i, cid, v > to all processes that did not propose v
34              return (v)
35          else
36              hash-v = out.value
37              break()                                          {terminar execução desta tarefa}
```

Figura 2 : Protocolo de acordo vectorial (executado por cada processo p_i)

pela mesma ordem (e.g., se conforme o destinatário tiverem atrasos diversos), e também porque os emissores maliciosos podem transmitir valores distintos conforme os receptores.

Na segunda fase, o protocolo escolhe um dos vectores construídos pelos processos. Um processo começa por difundir o seu vector pela rede (mensagem **B-vector** na Linha 14), e

depois inicia duas tarefas que irão ser executadas em paralelo. A tarefa T1 recebe e trata dos diversos tipos de mensagens que chegam ao processo (Linhas 17-20), e termina o protocolo quando uma decisão for tomada (Linhas 21-22). A outra tarefa garante que todos os processos correctos seleccionam o mesmo vector, usando para atingir este objectivo, um procedimento iterativo.

Em cada iteração é escolhido e votado um dos vectores dos processos. Se um destes vectores tiver uma votação satisfatória, *f+1* votos, então pode-se completar o protocolo e devolver esse vector. Caso contrário, inicia-se uma nova iteração onde um outro vector é considerado para aprovação. Um vector pode não receber uma votação aceitável por causa de duas razões principais: ele ainda não chegou aos seus destinatários, e por consequência não existem suficientes processos para votarem nele, ou porque ele foi enviado por um processo malicioso que está a atacar o protocolo, através da difusão de vectores errados ou pela transmissão do vector para pequenos subconjuntos de processos.

Um processo a executar a tarefa T2 realiza os seguintes passos. Começa por actualizar o contador de iterações, *round*, e o identificador único da votação corrente, *vid* (Linhas 26-27). Em seguida, calcula um indíce baseado no valor do contador de iterações, *index*, e escolhe o vector a ser proposto (Linhas 25 e 28). Mais abaixo, explica-se como é que *getNextGoodVector()* selecciona o vector e como é que garante que este verifica a especificação do acordo vectorial. Posteriormente, o processo chama o serviço de votação do wormhole com uma síntese do vector (obtida com a função *Hash()*) e um quórum de *2f+1*, e fica à espera do resultado da votação.

Embora os processos possam propor diferentes sínteses, todos eles recebem o mesmo resultado do serviço de votação. Logo, se eles se basearem neste resultado para determinarem se podem ou não terminar o protocolo, então todos os processos correctos deverão tomar decisões concordantes.

Um processo começa por verificar se a síntese mais votada teve *f+1* ou mais votos, usando para isso a máscara *out.voted-ok* (Linha 30, em que não se representou a máscara para facilitar a leitura). Caso esta condição seja falsa, é iniciada uma nova iteração da tarefa T2. No caso de ser verdadeira, existe a certeza que pelo menos um processo correcto tem o vector correspondente a esta síntese (relembra-se que no máximo *f* são maliciosos), seleccionando-se portanto esse vector como decisão do acordo. Cada processo verifica se tem o vector escolhido, comparando a síntese do seu vector com o valor devolvido pelo serviço de votação (Linha 31), terminando o protocolo caso o possua (Linha 34). Se o processo não tiver o vector, então limita-se a guardar a síntese na variável *hash-v* e a interromper a execução da tarefa T2 (Linhas 36-37).

Um processo que não tenha o vector seleccionado pode acabar por nunca o receber numa mensagem **B-vector**. Um adversário pode transmitir um vector válido para apenas um subconjunto dos processos, tentando desta forma impedir a terminação dos restantes. Para obviar a esta situação, o protocolo necessita de garantir que todos os processos correctos acabam por receber o vector escolhido. Este objectivo é atingido, obrigando cada processo antes de terminar, a enviar uma mensagem **Decide** com o vector seleccionado (Linhas 32-33). Esta mensagem apenas precisa de ser transmitida para os processos que não propuseram a síntese correcta, pois apenas estes poderão não receber o vector escolhido (a máscara *out.voted-ok* é usada para se encontrar este subconjunto de processos).

Um processo que não tenha o vector pode então terminar o protocolo se ficar à espera da chegada de mensagens **Decide** (Linhas 19-20), e retornar como decisão um vector $vector_k$ com uma síntese igual àquela que armazenou em *hash-v* (a síntese do vector mais votado).

Deve-se notar que as propriedades dos algoritmos de síntese, resistência a colisões fraca e forte, têm relevância na correcção do protocolo de acordo. Elas garantem dois aspectos importantes: a) se dois processos p_i e p_k têm $out.value = Hash(v_i)$ e $out.value = Hash(v_k)$ na Linha 30 do protocolo, então $v_i = v_k$, e portanto ambos decidem o mesmo vector; b) se um processo p_i com um $hash\text{-}v \neq \perp$ recebe uma mensagem **Decide** na Linha 19 com um $vector_k$ tal que $Hash(vector_k) = hash\text{-}v$, então $vector_k$ é o vector decidido pelos processos correctos, e não uma qualquer informação errada transmitida por um processo malicioso.

```
1    function getNextGoodVector(vector-of-vectors, index)
2    while (true) do
3        new-v ← (⊥, ..., ⊥)
4        for (each k non empty entry of vector-of-vectors[index]) do
5            if (vector-of-vectors[index][k] is correctly signed) then
6                new-v[k] ← vector-of-vectors[index][k].value      {guardar só o valor}
7        if (new-v has at least 2f+1 non empty entries) then
8            return (new-v)
9        index ← (index mod N) + 1
```

Figura 3 : Selecção do vector a ser proposto

Escolha do vector: Um processo ao receber um vector não consegue determinar se o emissor foi malicioso ou correcto. O processo apenas tem a certeza de uma coisa – o vector não sofreu alterações durante a sua transmissão porque se utilizaram canais seguros. No entanto, se o vector foi emitido por um processo malicioso então ele pode conter informação incorrecta (e.g., pode dizer que o processo p_k propôs o valor X quando na realidade tinha enviado Y). A função *getNextGoodVector()* é usada para evitar este tipo de ataques, garantindo que apenas vectores que verifiquem a especificação do acordo são votados pelo serviço do wormhole. Basicamente isto implica que o vector deve ter pelo menos $2f+1$ entradas adequadamente preenchidas, o que leva a que $f+1$ delas sejam provenientes de processos correctos.

A função *getNextGoodVector()* percorre *vector-of-vectors* de uma forma circular (*vector-of-vectors[index]*, *vector-of-vectors[(index mod N) + 1]* …) até encontrar um vector satisfatório (ver Figura 3). Um conjunto limitado de testes é efectuado em cada iteração. Primeiro lugar, verifica-se se as entradas do vector contêm valores válidos, ou seja, os valores que foram originalmente propostos pelos processos (Linhas 4-6). Como cada valor tem associado uma assinatura que não pode ser forjada por um adversário, através da verificação da assinatura é possível confirmar a validade do valor. Em seguida, descobre-se se o vector resultante tem pelo menos $2f+1$ entradas preenchidas, podendo neste caso ser devolvido (Linhas 7-8). É de notar que este algoritmo relativamente simples tem a vantagem de nunca devolver vectores vazios, o que evita tentativas de votação em vectores pertencentes a processos que sofreram uma paragem, melhorando potencialmente o desempenho do protocolo de acordo.

Como as mensagens sofrem atrasos, quando um processo chama a função *getNextGoodVector()* pela primeira vez, pode ainda não ter recebido qualquer vector dos outros

processos. No entanto, o processo nunca irá ficar bloqueado nesta função porque por um lado a tarefa T1, que é executada em paralelo, continua a adicionar vectores a *vector-of-vectors* (Linhas 17-18 da Figura 2), e por outro lado, pelo menos um vector válido existe sempre – o vector deste processo.

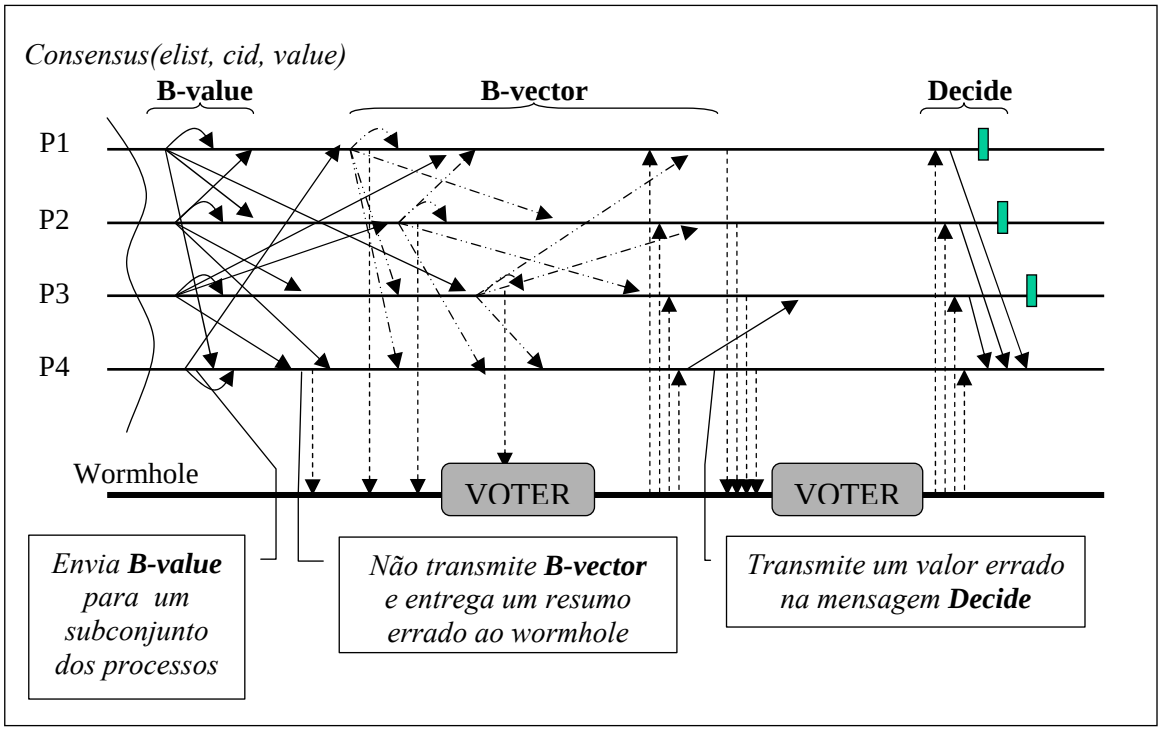

Figura 4 : Exemplo de uma execução do protocolo (com $n=4$ e $f=1$)

3.3 Exemplo de uma Execução do Protocolo

A Figura 4 apresenta num diagrama espaço-tempo um exemplo de uma execução de quatro processos e do wormhole. A execução de cada processo é ilustrada com uma linha em que o tempo cresce da esquerda para a direita. O wormhole, embora seja um componente distribuído, é representado com apenas uma linha. Optou-se por não se mostrar a execução dos protocolos internos ao wormhole com o fim de se reduzir a complexidade da figura. As linhas verticais dos processos para o wormhole correspondem a chamadas aos serviços do wormhole, e as setas entre processos indicam a transmissão de mensagens.

No exemplo três dos processos são correctos, p_1 a p_3, e um quarto é malicioso, p_4. Os processos correctos actuam de acordo com o algoritmo da Figura 2, enquanto que o malicioso tem toda a liberdade para agir como lhe aprouver, podendo tentar enganar os outros processos para quebrar o protocolo.

Os processos correctos começam por difundir os seus valores assinados numa mensagem **B-value**. Nesta altura o processo p_4 decide não seguir o protocolo, e envia o seu valor apenas para ele próprio e para o processo p_1. Um processo correcto, depois de receber três valores ($2f+1$ com $f=1$), constrói o seu vector e envia-o numa mensagem **B-vector**. Em seguida, propõe a síntese de um vector ao wormhole. Os processos p_1 a p_3 propõem respectivamente os vectores

de: p_1, p_2 (uma vez que o vector de p_1 sofreu um atraso e este é o primeiro vector não vazio em *vector-of-vectors*), e p_1. O processo p_4 mais uma vez procedeu de uma maneira incorrecta, e propôs uma síntese errada. O serviço de votação do wormhole (VOTER) pode devolver uma resposta mal receba três sínteses ($2f+1$), o que leva a que a proposta do processo p_3 seja rejeitada porque chega atrasada.

Como todas as propostas apenas colheram um voto, o serviço VOTER pode devolver a síntese do vector de p_1 ou p_2 ou a síntese errada de p_4. Vamos assumir que o VOTER seleccionou a síntese do vector de p_1 (os outros casos dariam algo semelhante). Os processos p_1 a p_3 obtêm este resultado e imediatamente notam que a proposta apenas recebeu um voto. Logo, como não é atingido o quórum mínimo de dois votos ($f+1$), eles iniciam uma nova iteração do ciclo. Para tentar confundir os outros processos, o p_4 transmite uma mensagem **Decide** com um vector errado, mas este não é aceite porque *hash-v* continua igual a $\perp$.

Na segunda iteração do ciclo, os três processos correctos propõem a síntese do vector de p_2 uma vez que a variável *index* tem o valor 2. Mais uma vez p_4 propõe um valor errado. Como as propostas chegam aproximadamente ao mesmo tempo, o serviço VOTER inclui na sua decisão todos os votos. Em seguida, os vários processos recebem como resultado a síntese do vector de p_2, e uma vez que esta tem uma votação de três, eles podem terminar o protocolo devolvendo o vector de p_2. Antes de retornarem, os processos correctos enviam o vector de p_2 ao processo p_4 numa mensagem **Decide**, porque este não tinha proposto a síntese correcta na votação (e poderia por isso não ter o vector escolhido).

4 Replicação de um Serviço

Nesta secção é explicado como é que o acordo vectorial apresentado anteriormente pode ser integrado na concretização de um sistema replicado tolerante a intrusões. A solução que se vai descrever é genérica, podendo ser empregue na concepção de um número variado de serviços, tornando-os tolerantes a intrusões. Por exemplo, ela pode ser usada para a criação de um serviço de ficheiros replicado que ofereça a semântica do NFS, ou na replicação de um servidor de nomes DNS, ou na construção de uma autoridade de certificação replicada.

4.1 Replicação como Forma de Tolerar Intrusões num Serviço

No topo da Figura 5 é representado um sistema replicado típico, constituído por um conjunto de servidores $S = \{ s_1, s_2, \ldots, s_N \}$ que correm em máquinas separadas, e que fornecem serviços a um número indeterminado de clientes. Cada servidor mantem um conjunto de dados – o seu *estado*, que podem ser consultados ou alterados pelos clientes. Para aceder a estes dados, o cliente começa por enviar um pedido, onde indica a operação que deseja realizar, e depois fica à espera de uma resposta com o resultado. A operação em si pode ser tão genérica quanto o programador dos serviços o desejar, necessitando no entanto de verificar os seguintes pressupostos: não deve existir interferência entre operações concorrentes, a operação deve criar sempre as mesmas alterações no estado, e o resultado deve apenas depender do estado actual.

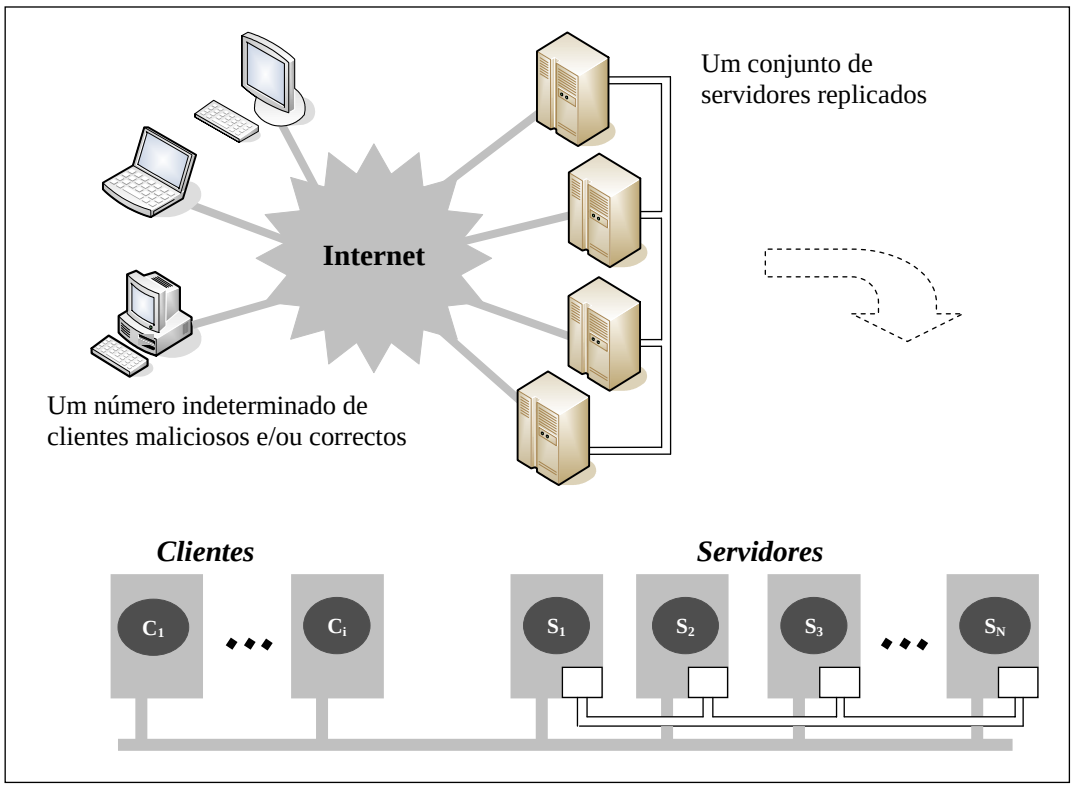

Figura 5 : Arquitectura da replicação de um serviço, onde existe um número indeterminado de clientes e N servidores com acesso ao wormhole.

Se forem construídos adequadamente, os sistemas replicados são capazes de tolerar intrusões nos servidores, continuando a fornecer resultados certos aos clientes. Para atingir este objectivo, o pedido do cliente, em vez de ser enviado apenas a um servidor, tem de ser difundido pelos vários. Cada um destes servidores executa a operação e devolve uma resposta. Como algumas respostas possivelmente são inválidas porque foram produzidas por servidores maliciosos, o cliente tem de esperar pela chegada de um número suficiente de respostas, de modo a aplicar um mecanismo simples de votação para seleccionar a correcta. No entanto, este modo de funcionamento requer que os servidores correctos devolvam a mesma resposta, tornando-se obrigatório que o mecanismo de replicação satisfaça as seguintes propriedades:

- ***Estado inicial.*** Todos os servidores são iniciados com o mesmo estado
- ***Concordância.*** Todos os servidores correctos executam as mesmas operações
- ***Ordem total.*** As operações são executadas pela mesma ordem nos servidores correctos
- ***Determinismo.*** A mesma operação executada sobre estados iniciais idênticos produz estados finais iguais

A primeira propriedade estipula que os servidores quando começam a executar-se, partem todos com estados iguais. Caso um servidor seja introduzido quando o sistema já está em actividade, então o seu estado tem de ser actualizado para que fique igual ao dos outros, antes de iniciar o processamento dos pedidos dos clientes. A duas propriedades seguintes estabelecem que todos os servidores correctos executam iguais pedidos pela mesma ordem. Deste modo, garante-se que os vários estados vão evoluindo de forma idêntica, sempre que é executado o mesmo número de operações, uma vez que estas são deterministas.

4.2 O Protocolo de Replicação

O protocolo de replicação define as interacções entre os clientes e os servidores, e a maneira como os servidores se coordenam para executar as operações. Na descrição que se segue, é assumido o modelo de sistema apresentado na Secção 2, encontrando-se os servidores em máquinas com um wormhole. Os clientes, no entanto, como podem enviar pedidos a partir de um qualquer dispositivo com ligação à rede, não carecem de ter acesso ao wormhole (ver fundo da Figura 5). É também assumido que no máximo *f* servidores podem falhar num total de $N \geq 3f+1$, e que existe um número indeterminado de clientes, em que um subconjunto deles é malicioso.

4.2.1 Interacções dos Clientes com o Sistema de Replicação

Um cliente quando quer executar uma operação, envia um pedido a um dos servidores. O formato do pedido é o seguinte:

$$\textbf{\textit{C-req}} = < client\_id, client\_addr, rid, op, other\_info >_{client\_sign}$$

Os dois primeiros elementos do pedido possibilitam aos servidores saber quem os está a contactar e para onde devem devolver a resposta (*client_id* é o identificador do cliente e *client_addr* é o seu endereço na rede). O *rid* identifica univocamente o pedido. O modo como este identificador é gerado pode ser baseado num contador local ao cliente. O elemento *op* especifica a operação e inclui toda a informação necessária à sua execução. Dependendo do serviço prestado pelo servidor, pode ser necessário incluir outro tipo de informação, *other_info*, por exemplo credenciais usadas por um mecanismo de controlo de acesso. Neste caso seria exequível a aplicação de políticas de segurança que limitariam o acesso dos clientes a partes do estado e apenas com certos tipos de operações. Para evitar ataques de personificação, o pedido tem de ser assinado pelo cliente.

Pode acontecer que o cliente envie o pedido para um servidor malicioso que o elimina ou atrasa o seu processamento (não o consegue alterar sem detecção por causa da assinatura). Para lidar com esta situação, o cliente deve retransmitir periodicamente o pedido para servidores diferentes, até que mais cedo ou mais tarde lhe cheguem as respostas. A não ser que a rede esteja muito congestionada, em princípio o cliente deverá ter de contactar no máximo $f+1$ servidores.

A resposta enviada pelos servidores tem um formato e conteúdo semelhante ao do pedido, em que os identificadores e assinaturas são os do servidor. O *rid* é igual ao do pedido e o *result* contém o resultado da execução da operação:

$$\textbf{\textit{C-reply}} = < serv\_id, serv\_addr, rid, result, other\_info >_{serv\_sign}$$

Ao chegarem as respostas ao cliente, ele deve esperar por $f+1$ resultados iguais de servidores distintos. Assim, garante-se que pelo menos um dos servidores era correcto, e que em consequência o resultado é válido.

4.2.2 O Funcionamento dos Servidores

Os servidores necessitam de realizar um conjunto de operações de iniciação antes de começarem a processar os pedidos dos clientes. Cada servidor deve obter o seu identificador único, *eid*, através do serviço de autenticação local do wormhole, e em seguida deve difundir o mesmo para que possa ser construída uma lista com todos os servidores envolvidos na

```
1    function serviceReplication(elist, smid)
2    id ← 0                                   {contador para gerar identificadores únicos}
3    accept-order ← 1                         {ordem do próximo pedido aceite para execução}
4    execute-order ← 1                        {ordem do próximo pedido a ser executado}
5    R_Set ← ∅                                {conjunto com pedidos do cliente}
6    A_Set ← ∅                                {conjunto com pedidos aceites para execução}
7    activate task(T1, T2)

8    Task T1:
9    when receive C-req = <client_id, client_addr, rid, op, other_info >client_sign do
10       if (C-req is correctly signed) then
11           if (C-req was received directly from client) then
12               broadcast C-req to servers S \ {si}
13           if (C-req was never received) then
14               R_Set ← R_Set ∪ {C-req}

15   Task T2:
16   while true do
17       id ← id + 1
18       cid ← (smid, id)                     {identificador para o próximo acordo}
19       C ← selectRequests(R_Set)            { conjunto dos pedidos C-req a processar}
20       CH ← {hashes of C-req in set C}
21       vector-of-sets ← consensus(elist, cid, CH)   {o valor a acordar é um conjunto
                          de sínteses e o resultado é um vector de conjuntos de sínteses}
22       AH ← {hashes in f+1 or more sets of vector-of-sets}
23       AH ← storeInDeterministicOrder(AH)
24       for (each hashReq in AH) do
25           A_Set ← A_Set ∪ {(accept-order, hashReq)}
26           accept-order ← accept-order + 1
27           if ( ∃C-req ∈ R_Set: Hash(C-req)= hashReq) then
28               broadcast C-req to S \ {servers with hashReq in a set of vector-of-sets}
29       while (∃(order, hash) ∈ A_Set: (order = execute-order) and
                              (∃C-req ∈ R_Set: Hash(C-req) = hash)) do
30           execute-order ← execute-order+1
31           A_Set ← A_Set \ {(order, hash)}
32           R_set ← R_set \ {C-req}
33           Execute C-req and send reply with result to client
```

Figura 6 : Protocolo de replicação de um serviço (executado por cada servidor $s_i \in S$).

replicação. O estado dos dados armazenados tem também de ser o mesmo em todos os servidores.

Um servidor executa o protocolo da Figura 6 enquanto se encontra a processar os pedidos. A função *serviceReplication()* recebe dois argumentos que devem ser iguais em todos os servidores – a lista dos servidores, *elist*, e um identificador único para esta execução do serviço de replicado, *smid*. Esta função começa por colocar algumas variáveis com os seus valores de omissão (Linhas 2-6), e depois inicia duas tarefas que são executadas em paralelo.

Um servidor ao executar a tarefa T1, recebe os pedidos **C-req** enviados directamente pelos clientes ou que foram redireccionados através de algum outro servidor. Depois, verifica se o **C-req** está correctamente assinado pelo cliente (Linha 10), prevenindo assim ataques com pedidos forjados, quer por outros clientes quer por servidores maliciosos. Caso a mensagem tenha sido transmitida directamente pelo cliente, o servidor reenvia-a para os outros servidores (Linhas 11-12), para que mais tarde possa ser tratada por todos. Por fim, confirma que o **C-req** nunca foi recebido anteriormente (usando o identificador único *rid*) e depois armazena-o no conjunto de pedidos recebidos, *R_set* (Linhas 13-14). Esta condição deve obrigatoriamente ser testada para evitar a re-execução do pedido, uma vez que o cliente retransmite-o caso a resposta não chegue em tempo útil, e a operação pode não ser idempotente (e.g., subtrai uma constante a uma variável de estado).

Na segunda tarefa, um servidor selecciona alguns pedidos e atribui-lhes uma ordem pela qual serão tratados. Este procedimento recorre ao protocolo de acordo vectorial para garantir que o seguinte invariante é verificado – *todos os servidores correctos executam os mesmos pedidos por uma ordem idêntica*.

O servidor começa por criar um identificador único para o próximo acordo, usando o identificador desta replicação, *smid*¸ mais o valor de um contador, *id* (Linhas 17-18). Depois, selecciona um subconjunto dos pedidos recebidos (a função *selectRequest()* é descrita mais abaixo) e calcula as suas sínteses, armazenando-as num conjunto *CH* (Linhas 19-20). O acordo recebe como valor este conjunto (Linha 20), e devolve um vector *vector-of-sets* com *2f+1* conjuntos de servidores distintos, em que *f+1* deles pertencem a correctos.

Cada servidor utiliza a resposta do acordo para escolher um novo grupo de pedidos a serem aceites para execução. Em primeiro lugar, cria um conjunto *AH* com as sínteses que aparecem em *f+1* ou mais entradas de *vector-of-sets* (Linha 22), o que assegura que pelo menos um servidor correcto recebeu o pedido correspondente à síntese. Desta forma impede-se que servidores maliciosos possam inserir sínteses de pedidos errados ou que nunca venham a ser recebidos. Os elementos do conjunto são depois arrumados numa ordem determinista pela função *storeInDeterministicOrder()* (Linha 23). A ordenação pode ser concretizada de uma maneira simples, se cada síntese for tratada como um inteiro – por exemplo, do número menor para o maior. Em segundo lugar, o servidor processa os elementos de *AH* pela sua ordem e guarda-os no conjunto de sínteses de pedidos aceites para execução, *A_Set* (Linhas 24-26). A cada síntese é associada a ordem com que foram aceites, *accept-order*. Caso o servidor já tenha o pedido **C-req** correspondente à síntese, este disponibiliza-o aos servidores que carecem dele, ou seja, os que ainda o não receberam (Linhas 27-28). Este passo é importante para evitar ataques sofisticados em que, por exemplo, um servidor malicioso retransmite um pedido apenas para um subconjunto dos correctos (nas Linhas 11-12), impedindo que os demais alguma vez o executem.

O servidor termina a iteração actual da tarefa T2 tratando, pela ordem de aceitação, tantos pedidos quanto lhe for possível. Por outras palavras, verifica se o próximo pedido já chegou (Linha 29), e no caso afirmativo executa-o (Linha 33). Caso contrário, interrompe o processamento e inicia outra iteração, enquanto aguarda que o pedido seja recebido.

É de notar que os servidores correctos executam os mesmos acordos e obtêm respostas análogas (ver propriedades no acordo em Secção 3.1). Logo, como usam um processo

determinista baseado na resposta do acordo, para escolherem os pedidos a executar e lhes darem uma ordem, todos os servidores chegam a resultados coincidentes. Desta forma, consegue-se assegurar o invariante acima mencionado.

Selecção dos pedidos: A função *selectRequests()* escolhe os pedidos que são fornecidos ao acordo em cada iteração. Na concretização mais simples, a função poderia devolver o conjunto *R_Set*, ou seja, todos os pedidos recebidos até ao momento e que ainda não foram aceites para execução. No entanto, caso se deseje, pode-se usar esta função para se realizarem políticas mais interessantes, impedindo por exemplo que um cliente possa monopolizar o serviço de replicação. Neste caso, a função organizaria os pedidos pendentes por ordem (e.g., usando o *cid*), e só devolveria no máximo *M* pedidos por cliente.

5 O Wormhole

A próxima secção descreve brevemente como é que um wormhole poderia ser construído. Este conceito já foi concretizado no passado, numa versão em software, que pode ser descarregada na Internet. As propriedades de sincronia do wormhole, no entanto, são mais fortes do que é exigido por este trabalho, encontrando-se por isso em desenvolvimento outros wormholes mais fracos. A secção apresenta ainda um protocolo que materializa o serviço de votação do wormhole.

5.1 Formas de Concretizar o Wormhole

Um wormhole pode ser construído de diversas maneiras, recorrendo-se em maior ou menor grau a hardware e/ou software especializado. Essencialmente, onde estas concretizações vão diferir é na taxa de cobertura das hipóteses de segurança e de sincronia do wormhole. Por exemplo, será normalmente mais fácil garantir que um wormhole local não pode ser comprometido, se ele se encontrar numa máquina protegida fisicamente, e todos os acessos tiverem que ser feitos remotamente através da rede.

No desenvolvimento de um wormhole, devem-se observar um conjunto de princípios básicos, para que se consigam assegurar as suas propriedades com uma alta taxa de cobertura:

- **Interposição:** o wormhole deve estar localizado de forma a que consiga interceptar quaisquer acessos aos recursos vitais ao seu correcto funcionamento
- **Blindagem:** o wormhole deve ser construído de modo a que esteja protegido de faltas que ponham em causa a sua segurança e/ou sincronia
- **Validação:** a funcionalidade do wormhole deve ser simples para que possa ser verificável

Os “recursos vitais” mencionados na interposição são os recursos do wormhole indispensáveis a que ele se comporte de acordo com a sua especificação. A blindagem substancia a hipótese de que é possível fazer a prevenção de ataques que resultem em intrusões no wormhole. O princípio da validação implica que a concepção e realização do wormhole deve ser simples para que seja verificável, garantindo-se assim as suas propriedades de uma forma fundamentada. Por exemplo, o número de linhas de código necessita de ser limitado e a estrutura interna dos procedimentos e protocolos precisa de ser facilmente perceptível.

Um wormhole com uma cobertura elevada deve empregar na sua construção módulos de hardware dedicados (e.g., placas PC/104 ou PCI/104), para que haja uma separação física entre

os seus recursos e os do resto da máquina. Nesta arquitectura, todas as interacções com o wormhole são efectuadas através de uma interface reduzida e bem definida, o que possibilita um controlo efectivo sobre os fluxos de informação. Na interface podem-se incluir diversos mecanismos que permitem melhorar a segurança, tais como módulos para a validação cuidada dos dados trocados e para a realização de controlo de admissão de pedidos. Dependendo da localização das máquinas, se estão próximas ou distribuídas geograficamente, a rede de controlo do wormhole pode ser uma rede própria (e.g., Ethernet) ou numa rede partilhada com algumas garantias de qualidade de serviço (e.g., VPNs sobre ligações ISDN). O uso de operações criptográficas é necessário para proteger as mensagens do wormhole, conforme se consiga ou não uma protecção física das mesmas.

O protótipo de wormhole que actualmente existe foi desenvolvido em software, o que naturalmente leva a que tenha uma taxa de cobertura mais baixa. No entanto, esta aproximação tem a vantagem de simplificar a distribuição do protótipo pela comunidade científica, possibilitando a outros o seu uso em trabalhos de investigação. Este wormhole corre em PCs correntes. Cada PC executa uma versão tempo real do kernel Linux, RTAI [CL00], e encontra-se interligado por duas redes Ethernet, uma para a rede normal e outra para a de controlo. A maior parte do wormhole é realizada directamente no kernel, num módulo de carregamento dinâmico (do inglês *Loadable Kernel Module*) e em algumas tarefas que são geridas com garantias de tempo real. Estas tarefas têm como principal missão a gestão das comunicações na rede de controlo. Os serviços do wormhole são disponibilizados às aplicações através de uma biblioteca – existe nesta altura uma versão para a linguagem C e outra para Java.

5.2 O Protocolo do Serviço de Votação

Esta secção apresenta um protocolo que realiza o serviço de votação descrito na Secção 2.5, num wormhole constituído por N componentes locais $W = \{w_1, w_2, ..., w_N\}$. É assumido o modelo de sistema apresentado na Secção 2, que é caracterizado por ser parcialmente síncrono e onde apenas ocorrem faltas do tipo paragem. A correcção do protocolo exige que o invariante $N \geq 2f + 1$ seja sempre mantido, em que f é o número máximo de componentes locais que podem ter uma paragem. Para simplificar a descrição do protocolo assume-se que os canais de comunicação são fiáveis (e.g., através de retransmissões) e que o processo correspondente à entrada *elist[i]* se encontra na máquina com o wormhole local w_i.

O protocolo tem uma estrutura relativamente simples e o seu funcionamento divide-se basicamente em duas fases (ver Figura 7). Na primeira fase, cada wormhole local w_i recolhe *quorum* valores diferentes e armazena-os no conjunto *V_Set* (Linhas 5-9). Para atingir este objectivo, o w_i difunde o seu valor numa mensagem **W-value** e depois espera que lhe cheguem valores suficientes de outros componentes. Como as mensagens sofrem atrasos distintos conforme os seus destinos, é natural que cada wormhole local acabe por criar um conjunto diferente.

Na segunda fase, os componentes escolhem um dos conjuntos (Linhas 17-40), e depois utilizam-no para gerar a resposta (Linhas 12-15). A forma como é conseguida esta selecção tem algumas semelhanças com um mecanismo iterativo que tem sido empregue por outros autores [DW88], [MO01]. Tal como aconteceu com os outros dois protocolos descritos anteriormente, existem duas tarefas que realizam esta fase. A tarefa T1 é responsável por receber as mensagens relevantes, e a outra itera sobre vários passos até que se conclua a escolha do conjunto.

```
1   function voter(vid, quorum, value_i)
2   round ← 0
3   V_Set ← ∅                                          {o meu conjunto de valores}
4   delay ← (default timeout, ..., default timeout)    {atrasos por omissão para cada w_k}
5   broadcast W-value = < i, vid, value_i >
6   repeat
7       receive W-value = < k, vid, value_k >
8       V_Set ← V_Set ∪ { (k, value_k) }                {guardar o valor de w_k}
9   until (V_Set has quorum values from different local wormholes)
10  activate task(T1, T2)                              {iniciar duas tarefas concorrentes}

11  Task T1:
12  when receive W-decide = <k, vid, coord-set > do
13      v ← most voted value in coord-set
14      voted-ok ← create a mask with the wormholes that voted v
15      return (v, voted-ok)

16  Task T2:
17  while (true) do
18      coord ← (round mod N) + 1
19      if (i = coord) then                            {vai coordenar esta iteração}
20          coord-set ← V_Set
21          broadcast W-vset = < i, vid, round, coord-set > to W \ {w_i}
22      else
23          initiate timer with delay[coord]
24          wait until ((receive W-vset = <coord, vid, round, s >) or (timer expires))
25          if (received W-vset) then
26              coord-set ← s
27          else
28              coord-set ← ∅
29              delay[coord] ← delay[coord] + 1

30      C ← ∅
31      broadcast W-confirm-vset = < i, vid, round, coord-set >
32      repeat
33          receive W-confirm-vset = < *, vid, round, * >
34          C ← C ∪ {W-confirm-vset}
35      until (C has (N – f) entries)
36      if (∃ W-confirm-vset = <*, *, *, s > ∈ C : s ≠ ∅) do
37          V_Set ← s
38      if (there are (f + 1) W-confirm-vset = < *, *, *, s > ∈ C : s ≠ ∅) do
39          broadcast W-decide = <i, vid, V_Set >
40          break ()
```

Figura 7 : Protocolo de votação no wormhole (executado por cada $w_i \in W$)

Cada iteração de T2 é coordenada por um wormhole local diferente (até que se dê a volta, e recomece tudo outra vez), que é indicado pelo valor actual da variável *round* (Linha 18). No primeiro passo, o coordenador envia o seu conjunto *V_Set* para os outros wormholes locais (Linhas 19-29, e a mensagem **W-vset**). No passo seguinte, os wormholes locais difundem uma confirmação (ou não confirmação, colocando *coord-set* a $\varnothing$ na Linha 28) da recepção do *V_Set*, e depois esperam pela chegada de *N-f* confirmações[5] (Linhas 30-35, e mensagem **W-confirm-vset)**. Conforme as mensagens que recebem, cada um dos wormholes locais toma uma das seguintes três decisões:

a) Condições falsas nas Linhas 36 e 38: vai para a próxima iteração, porque não recebeu qualquer confirmação
b) Condição verdadeira na Linha 36 e falsa na 38: antes de ir para a próxima iteração, actualiza o seu *V_Set* com o conjunto do coordenador (que chegou directamente do coordenador ou por intermédio de um dos outros wormholes)
c) Condições verdadeiras nas Linhas 36 e 38: actualiza o seu conjunto, e difunde uma mensagem que autoriza a conclusão do protocolo (nas Linhas 12-15, e mensagem **W-decide**); depois, interrompe a tarefa T2

Estas condições acabam por impor um invariante que é imprescindível para que a propriedade da concordância seja assegurada (ver Secção 2.5) – *se um wormhole local tomar a decisão c) então nenhum outro toma a decisão a)*. O invariante é importante porque garante que se um w_i finalizar o protocolo com um conjunto *s*, então todos os outros que o não fizerem nessa iteração, começam a próxima com os seus *V_Set* iguais *s*. Logo, ainda que a mensagem **W-decide** sofra um grande atraso, os wormholes que continuarem a iterar apenas poderão terminar com um conjunto idêntico. A razão por que este invariante existe tem a ver com a seguinte ideia: para que a decisão c) seja tomada em algum w_i é preciso que o seu *C* tenha *f+1* confirmações não vazias (ou seja, com $s \neq \varnothing$); por outro lado, o conjunto *C* em cada wormhole tem *N-f* confirmações; consequentemente, como $(N-f) + (f+1) > N$, cada conjunto *C* tem pelo menos uma confirmação não vazia.

A propriedade da terminação é assegurada através do uso de temporizadores. Quando um w_i começa, associa um atraso de omissão a cada um dos outros wormholes (Linha 4). Depois, sempre que se bloqueia à espera da mensagem **W-vset** do coordenador, inicia um temporizador com o valor de espera (Linhas 23-24). Se o temporizador expira antes da mensagem ser recebida, então uma de duas situações ocorreram: o coordenador teve uma paragem, e por isso a mensagem nunca irá chegar; ou, o valor corrente da espera não é suficientemente elevado para as condições actuais da rede, devendo assim o atraso ser aumentado (Linha 29). Este procedimento garante que depois de se atingir o TEG (ver Secção 2.2), os valores dos atrasos mais cedo ou mais tarde irão ser suficientemente elevados, para que os wormholes esperem pelas mensagens **W-vset** e o protocolo possa ser finalizado.

Quando um conjunto de valores é seleccionado, pode-se então calcular as respostas a serem retornadas (Linhas 13-15). Em caso de empate no valor mais votado, escolhe-se o menor dos valores mais votados, garantindo-se assim que todos os wormholes locais decidem de forma idêntica.

[5] Um * num campo de uma mensagem significa um valor válido qualquer.

6 Avaliação de Desempenho

Esta secção avalia o desempenho dos três protocolos em termos de complexidade de mensagens e temporal. O estudo baseia-se no melhor cenário de execução dos protocolos, ou seja, aquele que resulta num menor dispêndio de mensagens ou tempo. Ao longo da avaliação vai-se assumir que os protocolos se executam num ambiente onde não ocorrem faltas e onde os atrasos na transmissão das mensagens são constantes.

6.1 Complexidade de Mensagens

Os vários protocolos têm a mesma complexidade de mensagens no melhor cenário de execução. Esta complexidade é *O(N)* se a rede comportar directamente uma primitiva de difusão. Se esta primitiva não existir, então é necessário simulá-la com o envio de *N* mensagens ponto a ponto, resultando uma complexidade de $O(N^2)$.

6.2 Complexidade Temporal

Os factores mais importantes que influenciam a complexidade temporal dum protocolo tolerante a intrusões são os atrasos com a comunicação e os cálculos das operações criptográficas. Os valores reais destes atrasos, no entanto, estão muito dependentes do tipo de rede que é empregue e da capacidade de processamento das máquinas. Por exemplo, numa rede local com PCs correntes, a computação de uma assinatura digital acaba por retardar mais o protocolo do que a transmissão de uma mensagem. No entanto, numa rede de larga escala como a Internet, pode observar-se um comportamento oposto porque os interlocutores potencialmente encontram-se a uma grande distância, o que irá causar uma maior morosidade no envio dos dados. As operações criptográficas também não introduzem todas os mesmos atrasos – um MAC é calculado muito eficientemente, enquanto que as assinaturas digitais criam demoras significativas. Por estas razões, e com o objectivo de tornar a avaliação o mais independente da tecnologia que é possível, vai-se usar um conjunto de métricas que estimam o número de vezes que certas operações são executadas. Estas operações foram seleccionadas porque são aquelas que causam a maioria dos atrasos (ver Tabela 2).

A complexidade temporal que corresponde ao melhor cenário de execução dos protocolos é apresentada em seguida. Ao longo da descrição dos vários passos, é mostrado entre parênteses, o atraso acumulado pelos protocolos.

- **Protocolo de votação do wormhole:** o protocolo é iniciado nos vários wormholes locais na mesma altura; todos os w_i difundem os seus valores, e esperam pela chegada de *quorum* mensagens **W-value** (T_{difus}); o coordenador difunde o seu conjunto *V_Set*, enquanto que os demais se bloqueiam para o receberem ($2T_{difus}$); os w_i difundem uma confirmação da recepção do conjunto, e depois aguardam pela chegada de *N-f* mensagens **W-confirm-vset** ($3T_{difus}$); todos os wormholes terminam o protocolo difundindo a mensagem **W-decide**; como a recepção de uma mensagem de difusão é imediata para o seu emissor, a complexidade temporal do protocolo é: $\boldsymbol{T_{votação} = 3T_{difus}}$

Tabela 2: Operações tomadas em consideração na avaliação da complexidade temporal.

T_{difus}	Tempo para a difusão de uma mensagem de uma máquina para as outras
$T_{trans-cl}$	Tempo de transmissão de uma mensagem entre um cliente e um servidor
T_{ass}	Tempo para assinar um valor
T_{verf}	Tempo para verificar a assinatura de um valor
T_{ass-cl}	Tempo para criar uma assinatura de um pedido do cliente
$T_{verf-cl}$	Tempo para verificar a assinatura do pedido do cliente
$T_{ass-srv}$	Tempo para um servidor assinar uma resposta
$T_{verf-srv}$	Tempo para verificar a assinatura da resposta do servidor
T_{op}	Tempo para a execução do serviço no servidor

- **Protocolo de acordo vectorial:** os processos chamam a função *consensus()* no mesmo instante; todos os eles assinam os seus valores (T_{ass}); depois, difundem os valores e esperam pela chegada de *2f+1* mensagens **B-value** ($T_{ass} + T_{difus}$); à medida que constroem os vectores, verificam as assinaturas dos valores ($T_{ass} + T_{difus} + T_{verf}\ (2f+1)$); os processos difundem os vectores, e esperam pela chegada do vector do primeiro processo ($T_{ass} + 2\ T_{difus} + T_{verf}\ (2f+1)$); todos eles validam o vector do primeiro processo, e chamam o serviço do wormhole ($T_{ass} + 2T_{difus} + 2T_{verf}\ (2f+1)$); quando o resultado da votação é devolvido, todos os processos terminam o protocolo; a complexidade temporal do protocolo é: $\boldsymbol{T_{acordo} = T_{ass} + 2T_{difus} + 2T_{verf}\ (2f+1) + T_{votação}}$

- **Protocolo de coordenação dos servidores:** um servidor recebe o pedido do cliente, e valida a sua assinatura ($T_{verf-cl}$); depois, difunde o pedido pelos outros servidores ($T_{verf-cl} + T_{difus}$); os servidores verificam também a assinatura ($2T_{verf-cl} + T_{difus}$); todos eles chamam o protocolo de acordo para decidirem a ordem de execução da operação; a complexidade temporal do protocolo é: $\boldsymbol{T_{coord} = 2T_{verf-cl} + T_{difus} + T_{acordo}}$

- **Optimização do protocolo de coordenação dos servidores:** se o cliente enviar o seu pedido por difusão para os servidores, é possível evitar o passo em que o servidor reenvia a mensagem; neste caso, a complexidade temporal é: $\boldsymbol{T_{coord\text{-}optim} = T_{verf-cl} + T_{acordo}}$

- **Protocolo de execução de um serviço por um cliente:** o cliente assina o seu pedido e envia-o para um servidor ($T_{ass-cl} + T_{trans-cl}$); os servidores executam o protocolo de coordenação, e processam o serviço ($T_{ass-cl} + T_{trans-cl} + T_{coord} + T_{op}$); todos eles assinam uma resposta, e devolvem-na para o cliente ($T_{ass-cl} + 2T_{trans-cl} + T_{coord} + T_{op} + T_{ass-srv}$); o cliente recebe *f+1* respostas e verifica as suas assinaturas; a complexidade temporal do protocolo é $\boldsymbol{T_{srv} = T_{ass-cl} + 2T_{trans-cl} + T_{coord} + T_{op} + T_{ass-srv} + T_{verf-srv}\ (f+1)}$

7 Conclusões

O trabalho apresenta protocolos que possibilitam o desenvolvimento de aplicações distribuídas que toleram automaticamente as intrusões. Estes protocolos baseiam-se num modelo de sistema inovador, que postula a existência de um componente com propriedades mais fortes do que o resto do sistema, e que fornece um número reduzido de serviços. Os serviços devem ser usados raramente, mas em pontos críticos, para a realização de operações simples. Desta maneira, os protocolos podem ser projectados para um ambiente com hipóteses muito fracas, quer na sincronia quer nas faltas, tornando-se muito resistentes a ataques perpetrados por um qualquer adversário. Do ponto de vista prático, este é um aspecto importante da solução proposta porque possibilita a concepção de sistemas consideravelmente mais robustos do que é conseguido com as tecnologias correntes.

Neste trabalho é descrito um protocolo que resolve uma variante dum problema essencial na área dos sistemas distribuídos, o acordo. O protocolo assegura que os processos correctos conseguem escolher um mesmo vector, que é calculado com os valores iniciais dos processos, ainda que um adversário ataque a sua execução. Os ataques podem ser os mais variados, incluído a corrupção de mensagens, atrasos na realização das operações, ou acções maliciosas por parte de um subconjunto dos processos. O protocolo resultante tem algumas características interessantes, das quais se salienta a capacidade de contornar a impossibilidade FLP sem qualquer hipótese temporal explícita, e um bom desempenho em termos de complexidade temporal.

Com base no protocolo de acordo vectorial, é ainda proposta uma solução genérica para a construção de serviços replicados tolerantes a intrusões (e.g. na Internet). Esta solução suporta um número indeterminado de clientes, podendo muitos deles ser maliciosos, que enviam pedidos, e esperam respostas, dum grupo de servidores. Ao longo do tempo, os servidores conseguem actualizar o seu estado de forma semelhante, com auxílio do acordo vectorial, que assegura o processamento dos mesmos pedidos por uma ordem idêntica. O protocolo de coordenação dos servidores tem algumas propriedades de realce, entre elas a independência de hipóteses temporais e o bom comportamento face ao aumento da carga.

Para a concretização do wormhole, foi apresentado um protocolo de votação para um modelo parcialmente síncrono relativamente fraco. Neste modelo assume-se que o sistema só se começa a comportar de uma maneira favorável após um período de instabilidade desconhecida, e mesmo nessa altura, não se sabe quais são os limites máximos para duração das operações no sistema (apenas se sabe que eles existem).

8 Referências

[BA00] R. Baldoni, J. Helary, M. Raynal, and L. Tanguy. Consensus in Byzantine asynchronous systems. *Proceedings of the International Colloquium on Structural Information and Communication Complexity*, pp. 1–16, June 2000.

[BE83] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pp. 27–30, August 1983.

[CA02] C. Cachin, J. Poritz. Secure intrusion-tolerant replication on the Internet. *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 167–176, June 2002.

[CH96] T. Chandra, S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CL00] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, K. Yaghmour. DIAPM-RTAI position paper. *Real-Time Linux Workshop*, November 2000.

[CS02] M. Castro, B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[DE01] H. Debar, A. Wespi. Aggregation and correlation of intrusion detection alerts. *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, Lecture Notes in Computer Science, W. Lee, L. M, and A. Wespi, Eds. Springer-Verlag, vol. 2212, pp. 85–103, 2001.

[DL87] D. Dolev, C. Dwork, L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DO97] A. Doudou, A. Schiper. Muteness failure detectors for consensus with Byzantine processes. EPFL, Tech. Rep. 97/30, 1997.

[DW88] C. Dwork, N. Lynch, L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[FI85] M. Fischer, N. Lynch, M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382, April 1985.

[GU96] R. Guerraoui, A. Schiper. Consensus service: A modular approach for building fault-tolerant agreement protocols in distributed systems. *Proceedings of the 26th IEEE International Symposium on Fault-Tolerant Computing Systems*, pp. 168–177, June 1996.

[LA82] L. Lamport, R. Shostak, M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LY96] N. Lynch. *Distributed Algorithms*, Morgan Kaufmann, 1996.

[MA04] M. Marsh, F. Schneider. CODEX: A Robust and Secure Secret Distribution System. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, 2004.

[ME97] A. Menezes, P. Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[MO01] A. Mostefaoui, M. Raynal. Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.

[RA83] M. Rabin. Randomized Byzantine Generals. *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pp. 403–409, November 1983.

[RE95] M. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, Lecture Notes in Computer Science, Springer-Verlag, vol. 938, pp. 99–110, 1995.

[ZH02] L. Zhou, F. Schneider, R. van Renesse. COCA: A Secure Distributed On-line Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329-368, November 2002.