UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

TYPHON: UM SERVIÇO DE AUTENTICAÇÃO E AUTORIZAÇÃO TOLERANTE A INTRUSÕES

João Catarino de Sousa

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2010

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

TYPHON: UM SERVIÇO DE AUTENTICAÇÃO E AUTORIZAÇÃO TOLERANTE A INTRUSÕES

João Catarino de Sousa

DISSERTAÇÃO

Dissertação orientada pelo Prof. Doutor Paulo Jorge Paiva de Sousa
e co-orientada pelo Prof. Doutor Alysson Neves Bessani

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2010

Agradecimentos

As primeiras pessoas a quem quero agradecer são aos meus orientadores, Professor Paulo Sousa e Professor Alysson Bessani. No decurso de todo este trabalho sempre estiveram disponíveis para me aconselhar, ensinar e ajudar no que precisei. Mesmo que não soubessem a solução para algum problema, conseguiam sempre orientar-me na direcção certa para o resolver. Sempre me deixaram à vontade para fazer as tarefas que me delegavam e nunca me pressionaram com nada. Acho sinceramente que tive muita sorte em tê-los como orientadores. Também gostei muito de trabalhar com eles, e espero continuar a poder fazê-lo.

Também quero agradecer ao Bruno Quaresma, que também foi aluno do PEI no departamento de informática e também foi orientado pelos mesmos orientadores que eu. Apesar de não termos trabalhado no mesmo projecto, ele esteve na mesma sala que eu, e chegou-me a aconselhar em algumas partes do meu trabalho. Também foi boa companhia tanto na sala como nas habituais pausas de trabalho, e acho que também tive sorte em ficar perto dele.

Também quero agradecer ao meu compadre Fred, que embora ainda não tenha chegado ao Mestrado em Engenharia Informática, também esteve presente durante este período da minha vida, e sempre teve paciência para ouvir os meus devaneios.

Por último mas bastante importante, quero agradecer à minha mãe, que também sempre me apoiou, e também nunca me pressionou com nada - e se existe pessoa que tem o direito de o fazer, é ela. Embora não lhe fosse possível ajudar-me directamente com a dissertação, ajudou-me no resto, tanto agora como sempre. Sem ela este documento nunca iria existir, e eu não teria oportunidade de chegar onde cheguei.

Dedico este trabalho à minha avó, que não viveu anos suficientes para me ver a tornar Engenheiro.

Resumo

A norma Kerberos v5 especifica como é que clientes e serviços de um sistema distribuído podem autenticar-se mutuamente usando um serviço de autenticação centralizado. Se este serviço falhar, por paragem ou de forma arbitrária (e.g., bug de software, problema de hardware, intrusão), os clientes e serviços que dependem dele deixam de poder autenticar-se.

Este trabalho apresenta um serviço de autenticação e autorização que respeita a especificação do Kerberos v5 tal como é descrita no RFC 4120, fazendo uso da técnica da replicação da máquina de estados e de componentes seguros para tornar o serviço mais resiliente. A técnica da replicação da máquina de estados utilizada oferece tolerância a faltas arbitrárias, enquanto os componentes seguros garantem que as chaves dos clientes e dos serviços são mantidas secretas mesmo na presença de intrusões.

Neste trabalho foi usada a biblioteca BFT-SMaRt, para concretizar a técnica de replicação da máquina de estados. Este trabalho também foi dedicado a acrescentar uma nova funcionalidade na biblioteca, que consiste de um protocolo de transferência de estado.

Os resultados de avaliação mostram que a latência e débito do serviço proposto são similares aos de uma concretização Kerberos bem conhecida, e os resultados da avaliação do protocolo de transferência de estado revelam valores esperados para estados de diferentes tamanhos.

Finalmente, tanto quanto é do nosso conhecimento, este trabalho é o primeiro a apresentar um serviço de autenticação e autorização tolerante a intrusões que cumpre a norma Kerberos v5.

Palavras-chave: Tolerância a intrusões, autenticação, Kerberos, tolerância a faltas bizantinas, segurança.

Abstract

The Kerberos v5 standard specifies how the clients and services of a distributed system may mutually authenticate through the use of a centralized authentication service. If this service fails, by crash or in an arbitrary way (e.g., software bug, hardware problem, intrusion), the clients and services that depend on it are not able to authenticate between themselves.

This work presents an authentication and authorization service that complies with RFC 4120 (Kerberos v5), and that uses Byzantine-fault-tolerant state machine replication and secure components to make the service more resilient. These secure components guarantee that clients' and services' secret keys are kept private even in the presence of intrusions.

During the course of this work it was used the BFT-SMaRt library in order to implement the state machine replication. This work was also dedicated in part to introduce a new functionality in the library, which is a state transfer protocol.

The evaluation results show that the proposed service has similar latency and throughput values to the ones of a well known Kerberos implementation, and also show expected values of latency in the state transfer protocol for states diferent sizes.

Finally, as far as we know, this work is the first to present an autententication and autorization service which is intrusion tolerant while still respecting the Kerberos v5 specification.

Keywords: Intrusion tolerance, authentication, Kerberos, Byzantine fault tolerance, security.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este relatório descreve o trabalho desenvolvido no âmbito do Projecto em Engenharia Informática (PEI) do Mestrado em Engenharia Informática da Faculdade de Ciências da Universidade de Lisboa.

Neste capítulo introdutório é apresentada a motivação por trás deste trabalho, os objectivos desejados no início, quais as contribuições resultantes deste trabalho, o planeamento previsto face aquele que realmente aconteceu e a estrutura deste documento.

1.1 Motivação

Hoje em dia, os sistemas informáticos da maioria das organizações utilizam algum tipo de serviço de autenticação e autorização de forma a impôr politicas de controlo de acesso a diferentes tipos de dados e/ou serviços. A norma Kerberos v5 [14, 21] propõe uma especificação para um serviço de autenticação (Kerberos) com duas características interessantes: faz uso de autenticação mediada, sendo portanto escalável do ponto de vista do número de chaves que cada entidade do sistema tem de armazenar, e apenas utiliza criptografia simétrica que é, como se sabe, mais rápida do que criptografia assimétrica.

No entanto, o serviço de autenticação Kerberos é centralizado e concretizações comuns deste serviço tendem a residir apenas em um processo e em uma máquina. Basta que esse processo ou máquina falhe para que o serviço de autenticação se torne indisponível (falha por paragem) ou errático (falha arbitrária). Também basta a um intruso comprometer esse mesmo processo ou máquina para obter todas as chaves de todos os clientes e serviços do sistema. A falha do serviço de autenticação pode forçar todo o sistema informático que depende dele a parar, ou até mesmo permitir a utilização do sistema sem autenticação, dependendo de como o sistema foi concretizado e do tipo de falha do serviço de autenticação. Para além disto, a norma Kerberos v5 só especifica um serviço de autenticação, não fazendo qualquer referência à forma como o controlo de acesso/autorização é efectuado. Na prática, os serviços que concretizam esta norma (e.g., Apache DS, Microsoft Active Directory) combinam o serviço de autenticação Kerberos com um serviço

de nomes e directorias (e.g., LDAP) onde é armazenada a política de controlo de acesso, i.e., quem é que pode aceder ao quê.

1.2 Objectivos

O principal objectivo deste trabalho foi conceber um serviço de autenticação que obedecesse à especificação Kerberos v5, mas que fosse construído de forma a tolerar intrusões. A abordagem prevista era construir esse serviço recorrendo à técnica de replicação de máquina de estados - para tolerar faltas arbitrárias - e a um componente seguro - para assegurar a confidencialidade das chaves no caso de acontecerem intrusões.

Um serviço de autenticação Kerberos tem a particularidade de ser um ponto único de falha: se este serviço falhar, qualquer sistema que dependa dele para realizar autenticação dos seus utilizadores, ficará impossibilitado de o fazer. E é precisamente por causa deste problema, que se pretende concretizar um serviço Kerberos v5 recorrendo à técnica de replicação da máquina de estados [17]. Ao replicar o serviço por diversas máquinas, é mais difícil para um atacante corromper o serviço, pois tem que comprometer mais do que um limite conhecido de réplicas. Se se conseguir assegurar que esse limite nunca é excedido, o serviço mantém o seu funcionamento correcto.

Adicionalmente aos objectivos já descritos, este trabalho consiste também na realização de algumas melhorias à biblioteca de replicação BFT-SMaRt [2], nomeadamente adicionar-lhe um protocolo de transferência de estado. Esse melhoramento será discutido em mais detalhe no capítulo 4.

1.3 Contribuições

Este trabalho apresenta o serviço de autenticação e autorização TYPHON[1], um serviço que obedece à especificação Kerberos v5, mas que é construído de forma a tolerar faltas arbitrárias, fazendo uso da técnica de replicação da máquina de estados. De forma a tolerar intrusões, é também usado um componente seguro.

Adicionalmente à concepção e concretização do TYPHON, também foi dedicado esforço a estudar a biblioteca BFT-SMaRt, para ser possível fazer melhoramentos a esta. Os primeiros meses de trabalho foram dedicados a analisar o seu código, assim como a ler os trabalhos científicos que motivaram a sua criação [20, 17]. Durante esse período de tempo foram adicionados comentários ao código fonte da biblioteca, de forma a facilitar a sua análise e/ou modificação por terceiros, pois o BFT-SMaRt trata-se de um projecto open-source que pode ser utilizado e modificado por qualquer pessoa.

Após o período de análise da biblioteca, foi iniciada a fase em que se concretizou melhoramentos à biblioteca. O maior esforço desta fase girou em torno da concretização

[1] Na mitologia grega, TYPHON é o pai de Cerberos (ou Kerberos).

do protocolo de transferência de estado, cujo objectivo é enviar o estado da aplicação a réplicas que tenham deixado de executar ou que se atrasem.

Para além do protocolo mencionado acima, também foram realizados outros melhoramentos, nomeadamente a adição de um novo programa de demonstração do BFT-SMaRt. Originalmente o BFT-SMaRt apenas possuía uma demo que tinha um contador do lado das réplicas, e os clientes enviavam valores para serem adicionados ao contador. Apesar de simples, esta demo era insuficiente para demonstrar a ordenação total das mensagens, porque a soma é uma operação comutativa (e.g., se existissem dois clientes a enviar valores A e B respectivamente, era indiferente que A fosse entregue primeiro que B, pois A + B = B + A). Por isso, foi adicionado mais uma demo em que os clientes geravam aleatoriamente valores e operações a serem aplicadas ao valor guardado no servidor (e.g., operações de soma, subtracção, divisão e multiplicação), em vez de apenas enviarem valores para serem somados. Como as operações de subtracção e divisão não são comutativas, a ordem pela qual são entregues a cada réplica passa a ser relevante e observável. Após o protocolo de transferência de estado ter sido concretizado com sucesso, foi possível lançar uma nova versão do BFT-SMaRt, que disponibilizava essa funcionalidade.

Deu-se então inicio à fase dedicada a concretizar uma especificação parcial do serviço de autenticação TYPHON. Como este serviço obedece à norma Kerberos v5, que tem uma especificação longa, previu-se ser inexequível concretizar toda a especificação, face ao tempo disponível e planeamento estabelecido. Mas o objectivo fundamental deste trabalho foi realizar um serviço tolerante a faltas e intrusões, e não uma concretização completa de uma norma. Ou seja, o que se pretende é mostrar um conceito, ao invés de se oferecer um produto final pronto a ser usado por terceiros. Também foi concretizado o código do componente seguro que lida com operações relacionadas com as chaves de cada entidade presente no sistema.

Foram realizadas avaliações a cada uma das concretizações elaboradas durante este trabalho. Foram feitos testes de latência ao protocolo de transferência de estado realizado ao BFT-SMaRt. No caso do TYPHON, foram realizados testes de latência e de débito. Para tornar a avaliação mais credível, também se fez avaliações de latência e débito ao serviço Kerberos oferecido pelo ApacheDS [1], um servidor LDAP mantido pela Apache. No caso particular dos testes de débito, foi necessário alterar o código do ApacheDS para os realizar.

Finalmente, tanto quanto é do nosso conhecimento, este trabalho é o primeiro a apresentar um serviço de autenticação e autorização tolerante a intrusões que cumpre a norma Kerberos v5.

1.3.1 Publicações

O trabalho descrito neste relatório deu origem à seguinte publicação:

Título Typhon: Um Serviço de Autenticação e Autorização Tolerante a Intrusões

Autores João Sousa, Alysson Bessani, Paulo Sousa

Em Actas do INForum - Simpósio de Informática 2010, Lisboa, Portugal, Setembro de 2010 (aceite).

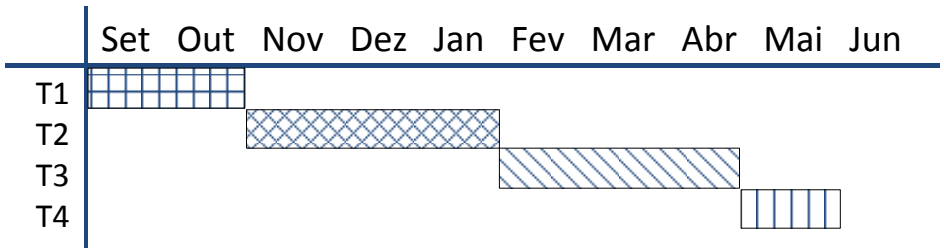
1.4 Planeamento

O planeamento inicial do trabalho era composto pelas seguintes tarefas (também representadas na figura 1.1):

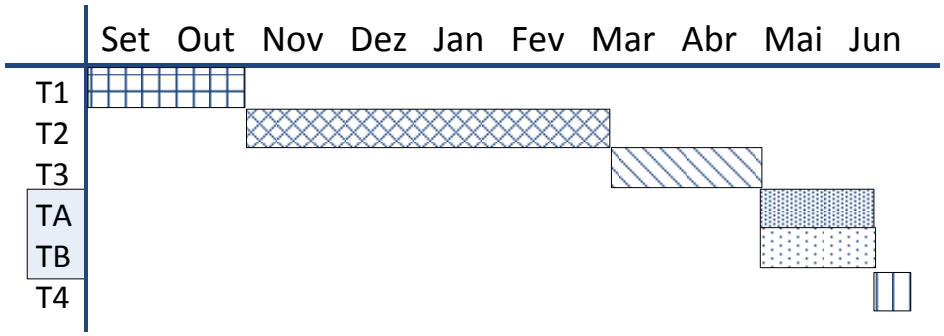
- T1: Setembro 2009 a Outubro 2009 - Estudo do algoritmo usado no BFT-SMaRt, assim como a sua concretização. Como complemento, também seria estudado o algoritmo PBFT [6].
- T2: Novembro de 2009 a Janeiro de 2010 - Durante este período, seriam feitas optimizações ao BFT-SMaRt. O principal objectivo desta tarefa seria estender a biblioteca de forma a concretizar um protocolo de transferência de estado, para que réplicas que falhassem por paragem, recuperassem, mas ficassem fora do contexto, pudessem resumir a execução. Outras optimizações, como correcção de possíveis bugs e melhoramento de desempenho também estavam previstos.
- T3: Fevereiro de 2010 a Abril 2010 - Concretização do TYPHON, de acordo com o RFC 4120 [21]. Esta concretização seria realizada sobre a biblioteca BFT-SMaRt, de forma a garantir a integridade e disponibilidade. Adicionalmente, ao nível da concretização do Kerberos v5, procurar-se-ia garantir a propriedade de confidencialidade com um componente seguro. Todos estes objectivos deveriam ser concretizados sem subverter a especificação do RFC 4120.
- T4: Maio de 2010 - Redacção deste documento.

Todas as tarefas foram realizadas com sucesso, mas foi necessário fazer ajustes à duração de todas, com excepção da T1. A tarefa T2 acabou por durar mais um mês que aquilo que era previsto, porque a concretização do protocolo de transferência de estado revelou-se um desafio mais trabalhoso que aquilo que se julgava inicialmente. Isto porque o código do BFT-SMaRt é um código complexo, e a mínima alteração em alguma parte dele pode comprometer o funcionamento inteiro da biblioteca. Foi esse tipo de situações que foram encontradas durante esta fase, em conjunto com alguma inexperiência do autor em lidar com código grande que foi produzido por várias pessoas.

Como consequência da extensão de tempo da tarefa T2, T3 foi iniciada mais tarde. Mas felizmente foi possível completá-la em menos tempo, pois já existia código de uma concretização parcial do Kerberos v5 que usava a biblioteca BFT-SMaRt. Foi necessário fazer uma separação do código, entre aquele que iria para o componente seguro, e o resto



(a) Planeamento proposto.



(b) Planeamento realizado.

Figura 1.1: Planeamento das tarefas efectuadas durante a realização deste trabalho.

que iria executar na aplicação. Também foi necessário adaptar esse código à nova versão do BFT-SMaRt que resultou da tarefa T2. Com isto tudo, a concretização do TYPHON foi concluída em início de Maio, como tinha sido inicialmente previsto.

No entanto acabaram por surgir duas novas tarefas no planeamento (tarefas TA e TB da figura 1.1b), que diziam respeito à avaliação do trabalho realizado e escrita do artigo científico. Essas tarefas foram executadas concorrentemente durante o mês de Maio, até meio de Junho.

Finalmente, a tarefa T4 foi realizada durante a segunda metade do mês de Junho, e parte do conteúdo do artigo científico produzido foi utilizado na redacção deste documento.

1.5 Estrutura do Documento

Os demais capítulos deste relatório estão organizados da seguinte forma:

- (Capítulo 2) Trabalho Relacionado - Apresenta a especificação Kerberos v5, o conceito de componente seguro e outros trabalhos dedicados a serviços de segurança tolerantes a intrusões.

- (Capítulo 3) BFT-SMaRt - Este capítulo descreve a biblioteca BFT-SMaRt e o seu

algoritmo de replicação.

- (Capítulo 4) Protocolo de Transferência de Estado - Descreve o funcionamento do protocolo de transferência de estado que foi concretizado.
- (Capítulo 5) Serviço TYPHON - Apresenta a descrição, desenho e concretização do serviço TYPHON.
- (Capítulo 6) Avaliação - Apresenta os resultados das avaliações feitas ao protocolo de transferência de estado e ao TYPHON.
- (Capítulo 7) Conclusões e Trabalho Futuro - Aqui são discutidas as conclusões que se tiram de todo o trabalho realizado e o trabalho que poderá resultar daquilo que foi alcançado neste.

Capítulo 2

Trabalho relacionado

Neste capítulo são apresentadas a especificação do Kerberos v5 e a ideia de componente seguro. Adicionalmente também são apresentados alguns serviços de segurança tolerantes a intrusões, como a autoridade de certificação COCA [22], o serviço de gestão de chaves Ω [16], e a firewall aplicacional CIS [5].

2.1 Kerberos v5

Nesta secção é apresentada a norma Kerberos v5 e as interacções que cada entidade envolvida na norma realizada para conseguirem autenticar-se mutuamente.

2.1.1 Descrição Geral

O Kerberos v5 [14] é uma norma especificada no RFC 4120 [21]. Esta por sua vez é concretizada por várias aplicações, como o ApacheDS e o Microsoft Active Directory[1].

Esta norma permite comunicações seguras e identificadas entre duas entidades por cima de uma rede insegura - como é o caso da Internet - e possibilita que duas entidades (tipicamente cliente e serviço) que há partida não têm nada que prove que podem confiar uma na outra, se consigam autenticar mutuamente.

A norma previne ataques de *eavesdropping*[2] e repetição[3] e ainda garante a integridade dos dados. É concebida de acordo com a arquitectura cliente-servidor e ainda é possível a autenticação mútua entre duas entidades (normalmente entre um cliente e um serviço).

Para assegurar a confidencialidade dos dados e a autenticidade de clientes e serviços, é usada criptografia simétrica. Todos as entidades partilham uma chave secreta com o Kerberos e este faz de mediador entre as duas entidades que se pretendem autenticar.

[1] De notar que o ApacheDS e o Microsoft Active Directory são servidores LDAP que disponibilizam igualmente um serviço de autenticação Kerberos v5.

[2] Ataques de *eavesdropping* consistem em escutar as mensagens que estão a ser transmitidas pela rede

[3] Ataques de repetição consistem em capturar mensagens trocadas entre as entidades de um sistema, e mais tarde injectá-las de novo na rede para efeitos maliciosos

A autenticação de uma entidade perante outra é conseguida por intermédio de estruturas de dados produzidas pelo Kerberos, denominadas de *tickets*. Os *tickets* provam que o Kerberos autenticou a entidade que pretende comunicar com outrem.

O Kerberos está dividido em dois componentes lógicos: o *Authentication Service* (AS) e o *Ticket Granting Service* (TGS). O primeiro permite a autenticação de uma entidade perante o Kerberos. O segundo destina-se a mediar a autenticação entre duas entidades após ambas estarem autenticadas perante o Kerberos (AS). De notar que apesar de o AS e o TGS serem componentes lógicas diferentes, não implica que tenham necessariamente de residir em máquinas distintas. Normalmente são concretizados dentro da mesma aplicação.

2.1.2 Interacções

A figura 2.1 ilustra as interacções entre um cliente C, os componentes AS e TGS do Kerberos, e um serviço S[4].

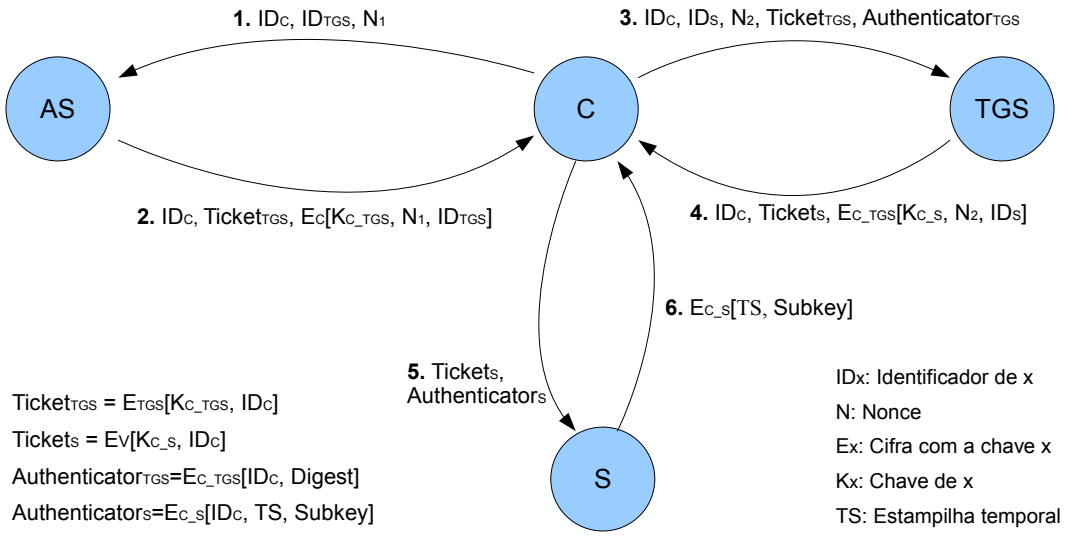


Figura 2.1: Ilustração simplificada das interacções das entidades presentes no Kerberos v5.

O objectivo das interacções ilustradas na figura 2.1 é autenticar o cliente C perante o serviço S. Para isso o cliente C precisa primeiro conseguir um *ticket granting ticket* (TGT) gerado pelo AS.

A obtenção do TGT começa no passo 1, que é um pedido de C dirigido ao AS. Este pedido é composto por:

- O seu próprio identificador (ID_C);

[4]Para facilitar a compreensão do protocolo, foram omitidos detalhes como o uso de *realms*, *flags* e o tempo de validade e renovação dos *tickets*.

- O ID do TGT (ID_{TGS});
- Um nonce (N_1).

Se o AS encontrar ID_C na sua base de dados, gera o TGT e envia a C uma resposta representada no passo 2, constituída por:

- O identificador de C (ID_C);
- O TGT que este requisitou ($Ticket_{TGS}$);
- Um tuplo cifrado com a chave de C ($E_C\ [K_{C\_TGS},\ N_1,\ ID_{TGS}]$), que contém a chave de sessão para ser usada com o TGT (K_{C_TGS}), o nonce enviado por C no pedido (N_1), e o ID do ticket granting server (ID_{TGS}).

No fim do passo 2 é garantido que apenas C e o TGS têm acesso à chave de sessão que foi gerada. Se quem requisitar o TGT for de facto C (i.e., não se tratando de um impostor) este conseguirá decifrar $E_C\ [K_{C\_TGS},\ N_1,\ ID_{TGS}]$ e obter a chave de sessão para interagir com o TGS (pois esse tuplo foi cifrado pelo AS com a chave de C). Caso contrário, quem pedir o TGT não conseguirá fazer nada com ele, porque não tem a chave de C para obter essa chave de sessão[5]. Por outro lado, o TGT foi cifrado com a chave do TGS, o que implica que só este conseguirá decifrar o TGT e obter a chave de sessão gerada.

Se os passos 1 e 2 foram realizados com sucesso, $E_C\ [K_{C\_TGS},\ N_1,\ ID_{TGS}]$ é decifrado por C e este obtém o nonce e a chave de sessão com o TGS. Também se certifica que o nonce recebido é igual ao nonce enviado no passo 1. O re-envio deste nonce por parte do servidor garante que C recebeu uma resposta para o pedido que enviou.

Se a decifra da resposta e a verificação do nonce forem realizadas com sucesso, C irá enviar um novo pedido, desta vez dirigido ao TGS e com o objectivo de receber um *ticket* específico para S. Esse pedido está representado no passo 3 e é constituído por:

- O identificador de C (ID_C);
- O identificador de S (ID_S);
- Um novo nonce (N_2);
- O TGT obtido na troca de mensagens anterior ($Ticket_{TGS}$);
- Um tuplo cifrado com a chave de sessão obtida anteriormente, designado de *authenticator* ($Authenticator_{TGS}$), que contém ID_C e uma estampilha temporal do tempo actual do cliente.

[5] A não ser ataques de força bruta ao TGT para obter a chave do cliente pelo qual se fez passar. Mas o RFC 4120 também prevê uma funcionalidade opcional de pré-autenticação para previr a geração de TGTs que não sejam requisitados legitimamente pelos clientes

Ao receber este pedido, o TGS decifra o TGT com a sua chave secreta, e o *authenticator* com a chave de sessão contida dentro desse TGT (K_{C_TGS}). Também calcula o resumo criptográfico do pedido que recebeu.

Se o TGS conseguir decifrar o *authenticator* com K_{C_TGS} e conseguir verificar que o resumo criptográfico contido neste corresponde ao resumo criptográfico que calculou, o TGS concluí que C realmente pediu esse TGT - porque cifrou o *authenticator* com a chave contida nesse TGT, e isso só seria possível se o AS tivesse enviado a C essa mesma chave - e que o pedido para gerar um *ticket* de C para S foi de facto enviado por C - pois C incluiu dentro do *authenticator* um resumo criptográfico que corresponde ao do pedido processado pelo TGS.

Uma vez passada essa verificação, o TGS irá gerar um *ticket* específico para S e envia uma resposta, tal como representado no passo 4, constituída por:

- O identificador de C (ID_C);
- O *ticket* específico para S que gerou ($Ticket_S$);
- Um tuplo cifrado com a chave de sessão entre C e o TGS ($E_{C\_TGS}\,[K_{C\_S},\ N_2,\ ID_S]$), que contém o identificador de S (ID_S), uma chave de sessão para ser usada entre C e S (K_{C_S}), e o nonce enviado no pedido ao TGS (N_2).

Uma vez recebida a resposta do TGS, C envia a S o *ticket* recebido do TGS ($Ticket_V$) e um novo *authenticator* ($Authenticator_S$), tal como ilustrado no passo 5. O *authenticator* é cifrado com a chave de sessão entre C e S dada pelo TGS (K_{C_S}) e contém o identificador de C (ID_C), uma estampilha temporal (TS), e uma sub-chave derivada de K_{C_S} ($Subkey$). A estampilha temporal é usada para prevenir ataques de repetição, e para que isso funcione C e S precisam de ter os seus relógios locais sincronizados dentro de uma taxa de desvio limitada e conhecida.

Finalmente no passo 6, S responde com um tuplo cifrado com a chave de sessão de ambos ($E_{C\_S}\,[TS,\ Subkey]$). Esse tuplo contém a estampilha temporal e sub-chave de sessão ($Subkey$) contidas no authenticador.

2.2 Componente Seguro

No contexto de segurança informática, um componente seguro é a parte de hardware, firmware, e software de um sistema de comunicação que concretiza os procedimentos básicos de segurança que acedem a recursos críticos.

Este ideia tem origem no conceito de *security kernel* [4], que é uma parte do sistema operativo onde todas as funções relacionadas com segurança foram reunidas, formando um pequeno núcleo seguro que está logicamente separado do resto do sistema. Como tal núcleo é pequeno, é também exequível verificar a sua correcção, e pode-se assegurar

que oferece um controlo de segurança eficiente contra a maior parte dos ataques feitos ao sistema operativo, porque tais ataques não conseguem obter acesso não autorizado a informação confidencial guardada dentro desse núcleo.

Mas o conceito original de *security kernel* está limitado a uma componente lógica local, sem considerar o hardware como parte dele. Essa lacuna veio a ser preenchida pelo conceito de *trusted computing base* (TCB) [13]. Este conceito vai mais longe que o de *security kernel* e considera que o núcleo seguro é composto também por hardware.

No entanto, uma TCB clássica não está directamente relacionada com computações distribuídas e não impõe restrições de tempo às computações que faz. Para oferecer essas propriedades, foi proposto o conceito de *trusted-timely computing base* (TTCB) [7]. Este conceito propõe a realização de uma TCB distribuída que executa no modelo síncrono, que almeja a possibilitar a concepção de protocolos à prova de ataques baseados no tempo que outrora eram dados como vulneráveis a esses mesmos ataques.

O conceito de TTCB é agora mais conhecido como *wormhole*, e o resto do sistema é agora referido como *payload* [15], que tipicamente executa no modelo assíncrono. Sabe-se também que o uso de *wormholes* pode melhorar o desempenho de protocolos, aumentar a resiliência dos mesmos, e oferecer independência do resultado da impossibilidade FLP [8].

2.3 Outros Trabalhos

Nesta secção são apresentados brevemente projectos semelhantes ao apresentado neste documento. Este projectos consistem em serviços de segurança tolerantes a intrusões.

2.3.1 COCA

O COCA (*Cornell OnLine Certification Authority*) [22] é uma autoridade de certificação tolerante a intrusões. Foi concebido para conseguir executar com assumpções o mais fracas possíveis acerca do ambiente em que está a trabalhar, i.e., não existem limites de tempo de execução nem de entrega de mensagens, e é esperado que os canais de comunicação se mostrem pouco fiáveis. Vários ataques de negação de serviço são bem sucedidos porque quebram as assumpções mais fortes que as apresentadas. Como o COCA foi construído com assumpções mais fracas em mente, também é menos vulnerável a estes ataques.

O COCA foi o primeiro trabalho a integrar um sistema de quóruns bizantinos - para conseguir obter disponibilidade - com recuperação proactiva [18] - para se conseguir defender de ataques e intrusões. Adicionalmente, também usa criptografia de limiar para assinar os certificados.

Este serviço também tem a particularidade de, ao contrário de autoridades de certificação comuns, manter-se sempre online. Isto possibilita que as entidades do serviço

façam a validação dos certificados que têm antes de os usarem. Desta maneira as entidades podem saber se esses certificados foram revogados.

2.3.2 Serviço de Gestão de Chaves Ω

O Ω é um serviço de gestão de chaves públicas distribuído [16]. Este serviço também é tolerante a falhas arbitrárias e intrusões, fazendo também uso da técnica de replicação de máquina de estados para tolerar essas falhas. Tal como o COCA, também se trata de um serviço que se mantém sempre on-line. E ao contrário do Kerberos v5, suporta controlo de acesso perante as funcionalidades que disponibiliza.

As funcionalidades disponibilizadas por este serviço são:

- Registar uma chave pública
- Obter uma chave pública previamente registada
- Revogar uma chave pública previamente registada
- Depositar uma chave privada
- Decifrar mensagens com a chave privada previamente depositada
- Recuperar uma chave privada que tenha sido previamente depositada

Finalmente, este serviço foi pioneiro na introdução de funcionalidades dedicadas a depositar nele chaves privadas, com o objectivo de permitir aos clientes fazerem *backup* dessas chaves, e também disponibilizar uma forma de as conseguir obter em contextos críticos. Para assegurar que um intruso que comprometa até um certo número de réplicas não consegue obter o valor das chaves privadas armazenadas, é usada criptografia de limiar.

2.3.3 Firewall Aplicacional CIS

A firewall aplicacional CIS [5] faz parte da arquitectura CRUTIAL, que almeja a conceber uma maneira de proteger infra-estruturas críticas de ataques oriundos da rede - caso essas infra-estruturas necessitem de estar ligadas à internet. Nesta arquitectura, toda a infra-estrutura é vista como uma WAN de LANs, e a firewall CIS é inserida no ponto de ligação entre uma LAN e a WAN. Esta abordagem permite que os sistemas de legado presentes nessas infra-estruturas críticas possam continuar a existir, sem alterar o seu comportamento nem eficiência.

Adicionalmente, a CIS não se trata de uma firewall comum, ma sim de um mecanismo de protecção distribuído com um modelo de controlo de acesso sofisticado. Tal como o COCA e o Ω, a CIS é tolerante a intrusões, e ainda é capaz de se auto-regenerar (através da recuperação proactiva e reactiva), mantendo um funcionamento perpétuo.

2.4 Considerações Finais

Neste capítulo foi apresentada a norma Kerberos v5 e descritas as interacções necessárias para que um cliente se autentique perante um serviço usando essa norma. Também foi apresentada a história e o conceito de componente seguro. Finalmente, fez-se referência a três trabalhos semelhantes a este, na medida em que também propõem serviços de segurança tolerantes a intrusões.

No próximo capítulo será apresentada a biblioteca usada pelo serviço TYPHON para realizar a técnica de replicação de máquina de estados.

Capítulo 3

BFT-SMaRt

Este capítulo descreve a biblioteca BFT-SMaRt [2, 3], que foi usada na construção do serviço TYPHON. Esta biblioteca possibilita a difusão atómica de mensagens enviadas por clientes, para um conjunto de réplicas. Para se compreender como esta biblioteca funciona, será descrito o algoritmo utilizado. De seguida, será descrita a sua concretização. Por fim, também serão discutidas as limitações desta biblioteca, inerentes ao algoritmo usado pela mesma.

3.1 Algoritmo

Nesta secção, é descrito o algoritmo usado pela biblioteca BFT-SMaRt. Este algoritmo corresponde no fundo ao algoritmo de consenso Paxos at War (PaW) [23], modificado e estendido de forma a concretizar difusão atómica [10]. Esta propriedade é crucial para que a técnica de replicação da máquina de estados funcione correctamente. Sendo assim, faz sentido começar por descrever o algoritmo PaW original, seguido do conceito de difusão atómica e da técnica de replicação da máquina de estados.

3.1.1 Paxos at War

O Paxos At War (PaW) é uma das variantes do algoritmo de consenso Paxos, proposto inicialmente por Leslie Lamport [12]. Este algoritmo oferece as seguintes propriedades:

- Terminação: Todos os processos correctos irão decidir um valor;
- Validade: Se um processo decide v, então v foi proposto por algum processo;
- Integridade: Nenhum processo decide duas vezes;
- Acordo: Todos os processos correctos decidem exactamente o mesmo valor.

A figura 3.1 ilustra o resultado da execução de um consenso identificado por i para um valor v, entre 4 processos.

Figura 3.1: Consenso em execução entre 4 processos.

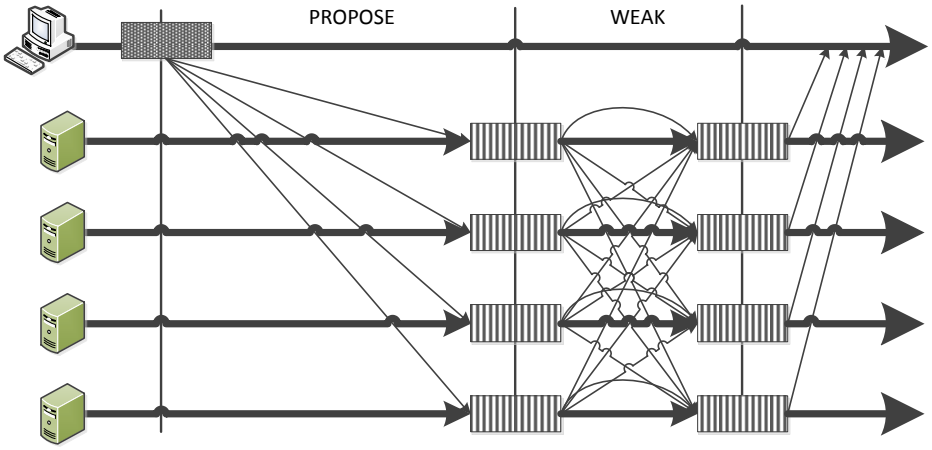
De todas as variantes do algoritmo de consenso Paxos existentes, o PaW foi escolhido porque permite tolerar faltas arbitrárias [11] e foi concebido para o modelo assíncrono de computação distribuída, onde não existem limites conhecidos para o envio/recepção de mensagens [9]. Este algoritmo tem a particularidade de, caso todos os processos sejam correctos, necessitar de apenas dois passos de execução - menos um passo que a generalidade dos algoritmos de consenso tolerantes a faltas bizantinas.

O número de processos *n* necessários para assegurar o funcionamento correcto do algoritmo depende do número máximo *f* de processos faltosos que possam existir. Esse número *n* terá que ser maior ou igual que *3f + 1* no caso do algoritmo PaW e na generalidade dos algoritmos de consenso sob o modelo assíncrono. Nesta explicação em particular, assume-se que $n = 3f + 1$.

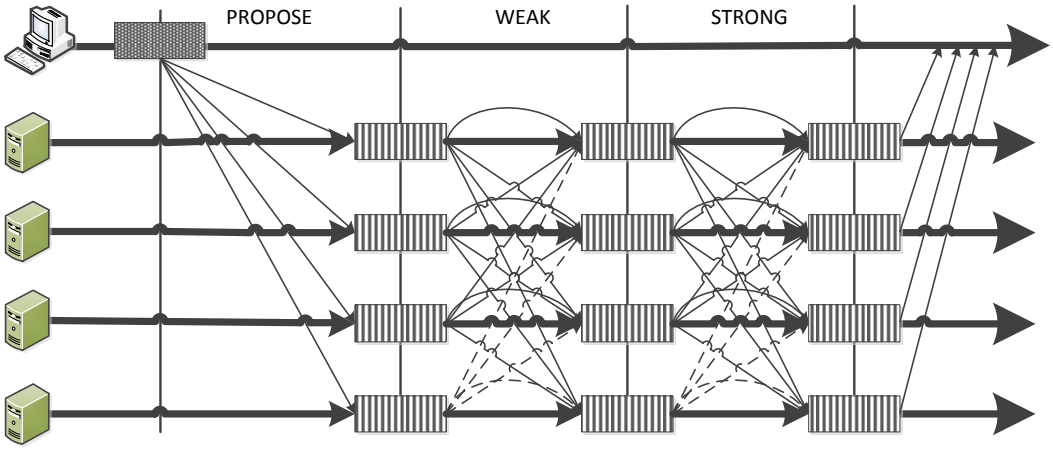
Para dar início ao consenso, cada processo invoca a operação *propose(i,v)*, onde *i* serve como um identificador da execução do consenso que está a decorrer, e *v* representa o valor proposto. Cada execução de um consenso é composta por no mínimo um *round*. Um *round* engloba todos os passos de execução do algoritmo. Irão ocorrer tantos *rounds* quantos forem necessários para se chegar a uma decisão.

Um dos processos que participa em cada execução de um consenso, é considerado o líder por todos os processos. Esse processo líder, aquando da invocação de *propose(i,v)*, irá enviar a todos os processos (incluindo para si) uma mensagem do tipo PROPOSE, contendo *i* e *v*. Quando cada processo recebe esta mensagem, guarda esse valor, e envia para os outros uma mensagem do tipo WEAK, contendo o valor *v* recebido na mensagem PROPOSE, assim como o respectivo identificador *i* da execução. Quando um processo recebe dos outros *2f + 1* mensagens WEAK para um determinado valor *v* de uma execução *i*, envia para todos os processos uma mensagem do tipo STRONG, contendo o valor para o qual foram recebidas essas mensagens WEAK (se verificar que obteve *3f + 1* mensagens, irá invocar imediatamente a operação *decide(i,v)*, poupando assim um passo de execução). De seguida, quando um processo recebe *2f + 1* mensagens STRONG

para um determinado valor, invoca a operação *decide(i,v)*. A figura 3.2 ilustra a troca de mensagens entre processos, tanto no caso normal, como em execuções favoráveis.



(a) Execução favorável, onde foi possível saltar um passo de execução



(b) Execução não favorável, onde todos os três passos tiveram que ser executados.

Figura 3.2: Troca de mensagens do algoritmo PaW.

Como já foi dito anteriormente, este algoritmo é concebido para funcionar sobre o modelo de computação distribuída assíncrono, e assume-se que existe um número máximo conhecido de processos maliciosos. Este algoritmo está preparado para funcionar correctamente se no máximo f réplicas forem maliciosas. Caso nenhuma dessas réplicas seja o líder, não é necessário despoletar qualquer passo de execução adicional aos apresentados. Caso contrário, será necessário iniciar a fase de troca de líder. Esta fase consiste em parar o processamento de mensagens relacionadas com a execução normal, determinar o novo líder, e enviar todos os valores recebidos para esse novo líder. Este deverá escolher um desses valores, e enviar uma nova mensagem do tipo PROPOSE com esse valor, dando assim início a um novo *round*.

Como já foi referido nesta secção, este algoritmo foi concebido para executar no modelo assíncrono. Isto significa que as mensagens trocadas entre processos não têm um limite de tempo conhecido para chegarem aos seus destinatários, e podem chegar mensagens do algoritmo de consenso que estejam fora do contexto, i.e., um processo pode estar a processar o consenso *i*, mas receber uma mensagem referente ao consenso *j*, em que *j* pode ser um consenso anterior ou posterior a *i*. Caso *j* seja posterior a *i*, a mensagem deverá ser guardada em memória para ser processada mais tarde, quando o processo em questão entrar no consenso *j*.

Finalmente, cada execução do consenso só termina quando ocorrer um *round* síncrono, i.e., todos os passos da execução forem realizados dentro de um limite de tempo definido em cada processo. Se esse limite de tempo não for respeitado, irá ser iniciado a fase de troca de líder, para de seguida se iniciar um novo *round*.

3.1.2 Difusão Atómica

A difusão atómica é uma primitiva fundamental para a concretização da replicação da máquina de estados. A difusão atómica deve garantir que todos os processos correctos membros de um grupo, entreguem todas as mensagens difundidas nesse grupo na mesma ordem à aplicação. Este problema pode ser resumido ao problema do consenso, i.e., um protocolo de consenso pode ser utilizado na resolução do problema da difusão atómica. Neste caso, é utilizado o PaW. A ideia consiste em ter um grupo aberto de processos, que recebem mensagens de vários clientes. Os clientes enviam cada mensagem para todos os processos. Sempre que os mesmos receberem uma mensagem de clientes, executam um consenso. No fim da execução do consenso, todos os processos entregam à aplicação uma mensagem, que será a mesma em todos os processos, tal como garante o PaW. A figura 3.3 ilustra este mecanismo.

Também é necessário observar que só pode haver uma instância do algoritmo a ser executada de cada vez. Isto deve-se ao facto de o identificador de cada consenso (o parâmetro 'i' usado tanto no método `propose` como `decide`) ser usado para determinar o número de sequência da mensagem a ser ordenada. Isto significa que o PaW é usado, não para determinar que número de sequência é atribuído a uma mensagem, mas sim para determinar que mensagem é atribuída a um número de sequência. Finalmente, para melhorar o desempenho deste mecanismo, cada processo propõe várias mensagens para serem ordenadas de uma vez. No fim, o mesmo conjunto de mensagens é entregue por cada processo à aplicação. Porém, é necessário que a aplicação ordene este conjunto de mensagens de forma determinista, para que assim todas sejam ordenadas da mesma maneira em cada máquina, preservando a ordem total.

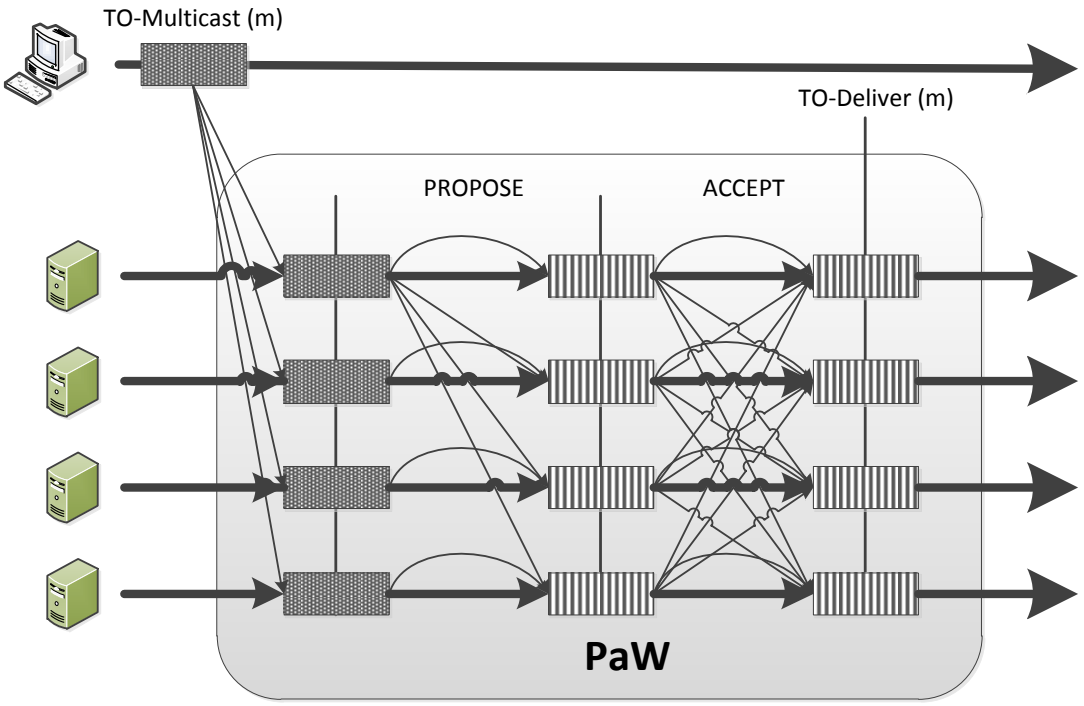


Figura 3.3: Ordenação total de uma mensagem de um cliente, recorrendo ao algoritmo PaW (em dois passos de execução).

3.1.3 Técnica de Replicação de Máquina de Estados

Consideremos uma aplicação cliente/servidor, onde a componente de servidor possui um estado, que evolui consoante os comandos enviados pelo cliente. A técnica da replicação da máquina de estados tem por objectivo replicar esse estado em vários servidores, chamados réplicas. Os clientes enviam os seus comandos para essas réplicas, e o estado das mesmas deverá evoluir exactamente da mesma forma. Para que isto seja possível, cada réplica necessita de preencher os seguintes requisitos [17]:

- Todas as réplicas começam a execução no mesmo estado.
- As réplicas têm que ser deterministas, ou seja, a execução do mesmo comando, com os mesmos argumentos, sobre o mesmo estado, em réplicas distintas, tem que modificar o estado exactamente da mesma maneira.
- Os comandos enviados por clientes, deverão ser todos entregues na mesma ordem às réplicas.

O primeiro requisito consegue-se concretizar trivialmente. O segundo já necessita de algum cuidado, principalmente quando as réplicas necessitam de gerar bytes aleatórios e usar estampilhas temporais (mais à frente neste capitulo, será descrito como o BFT-

SMaRt resolve este problema). Finalmente, para realizar o ultimo requisito, é necessário uma primitiva de difusão atómica, tal como descrito anteriormente.

É desta forma que a replicação da máquina de estados é conseguida pelo BFT-SMaRt: usa-se difusão atómica para entregar os pedidos de clientes à aplicação, que por sua vez usa a concretização do algoritmo PaW para determinar que comandos entregar em cada execução. Vale a pena mencionar que já existe um algoritmo semelhante ao apresentado nesta secção, tipicamente designado por PBFT [6]. Porém, enquanto o PBFT aborda a replicação de forma monolítica, o algoritmo apresentado tem uma abordagem modular. É relativamente fácil ver onde é efectuado o consenso, como é usado para realizar difusão atómica e, por sua vez, como é conseguida a replicação da máquina de estados. Adicionalmente, tem ainda a vantagem de, em execuções favoráveis, necessitar de menos um passo de execução.

3.2 Concretização

A biblioteca BFT-SMaRt está escrita na linguagem de programação Java, e é compatível com a versão 1.6 ou superior da respectiva máquina virtual (JVM). Desta maneira, é possível usá-la em qualquer máquina, independentemente da sua arquitectura e sistema operativo, desde que a máquina virtual do Java esteja presente. De seguida, será discutida a arquitectura da biblioteca BFT-SMaRt, assim como o seu modo de uso.

3.2.1 Arquitectura

O código fonte do BFT-SMaRt está dividido em três *packages* principais:

- `navigators.smart.communication`
- `navigators.smart.tom`
- `navigators.smart.paxosatwar`

O *package* `navigators.smart.communication` contém código relacionado com a comunicação entre réplicas, assim como entre clientes e réplicas. O *package* `navigators.smart.tom` (*total order multicast*) contém o código que concretiza a primitiva de ordenação total. O *package* `navigators.smart.paxosatwar` contém código que concretiza o algoritmo PaW. A figura 3.4 ilustra a arquitectura do BFT-SMaRt.

A componente *Communication System* possibilita que o envio de comandos de clientes para as réplicas seja feito de forma transparente, i.e., a aplicação no lado do cliente envia os comandos para a aplicação na parte do servidor, sem estar ciente das réplicas, nem de como entregar os comandos. A ideia é realizar um tipo de comunicação semelhante a um *remote procedure call* (RPC). Como tem que funcionar tanto do lado do cliente como do

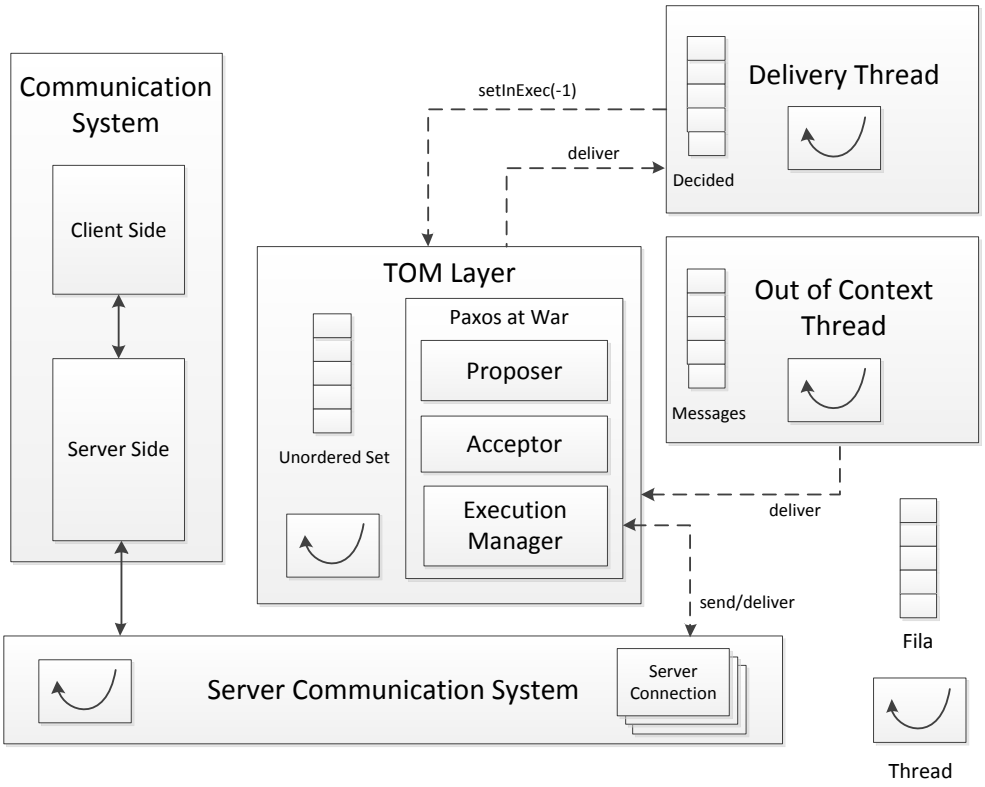


Figura 3.4: Arquitectura do BFT-SMaRt.

servidor, está dividida numa parte lógica que reside no lado do cliente (*Cliente side*), e outra que reside no lado das réplicas (*Server Side*).

A componente *Server Communication System* representa a mesma ideia, mas destina-se apenas à comunicação entre réplicas.

A componente *TOM Layer* realiza a ordenação de mensagens propriamente dita. Faz uso do *Comunication System* para receber comandos de clientes e enviar as respostas correspondentes, e do *Server Comunication System* para comunicar entre as réplicas. Sempre que recebe uma mensagem vinda de um cliente, guarda-a num conjunto de mensagens por ordenar (*Unordered Set*).

Adicionalmente, faz uso do componente *Paxos at War*, que concretiza o próprio algoritmo descrito na secção 3.1.1. Tem dois sub-componentes: o *Proposer*, que realiza o papel de líder conforme especificado para o PaW, e o *Acceptor*, que executa a restante especificação do algoritmo. Este componente também faz uso do *Server Comunication System*, mas como só comunica entre réplicas, não precisa do outro componente de comunicação. Assim que este componente decida um valor (valor esse que é um conjunto de mensagens a ser entregue à aplicação), entrega-o ao *TOM layer*.

A componente *Paxos at War* também tem um sub-componente denominado *Execution Manager* que faz a gestão de execuções de consensos. O *Proposer* e o *Acceptor* consultam

o *Execution Manager* para obterem informação dessas execuções e para saberem se a mensagem que lhes foi entregue pelo *Comunication System* é do consenso que está a ser executado no momento. Caso seja, a mensagem é processada pelo *Proposer* e pelo *Acceptor*. Caso contrário, o *Execution Manager* verifica se a mensagem é de um consenso futuro. Se o for, é guardada em memória como sendo uma mensagem fora do contexto, para só ser processada quando a réplica chegar ao consenso a que a mensagem se refere - isto desde que o número desse consenso não seja maior que o último que foi decidido somando com o valor de um parâmetro do BFT-SMaRt designado de *paxos_highmark* (i.e., se o número do consenso da mensagem for c e o último consenso decidido pela réplica for d, a mensagem só é guardada se $d \leq c \leq d$ + *paxos_highmark*).

Por sua vez, o *TOM Layer* entrega esse valor à componente *Delivery Thread*, que é responsável por entregar os comandos à aplicação. À medida que vai recebendo comandos, guarda-os num conjunto de mensagens ordenadas. Sempre que existir comandos nesse conjunto, irá entregar parte deles à aplicação.

É necessário referir que o BFT-SMaRt não envia um comando à réplicas sempre que um cliente lhe entrega algum comando. Em vez disso vai acumulando esses comandos, e envia vários de uma vez para as réplicas. Esse conjunto de comandos são designados de *batches* e são tratados pelo algoritmo de consenso como um único valor. Depois desses valores terem sido decididos é que a componente *TOM Layer* irá entregar cada comando à aplicação em cada réplica e pela mesma ordem. Esta sequência de acções está ilustrada na figura 3.5

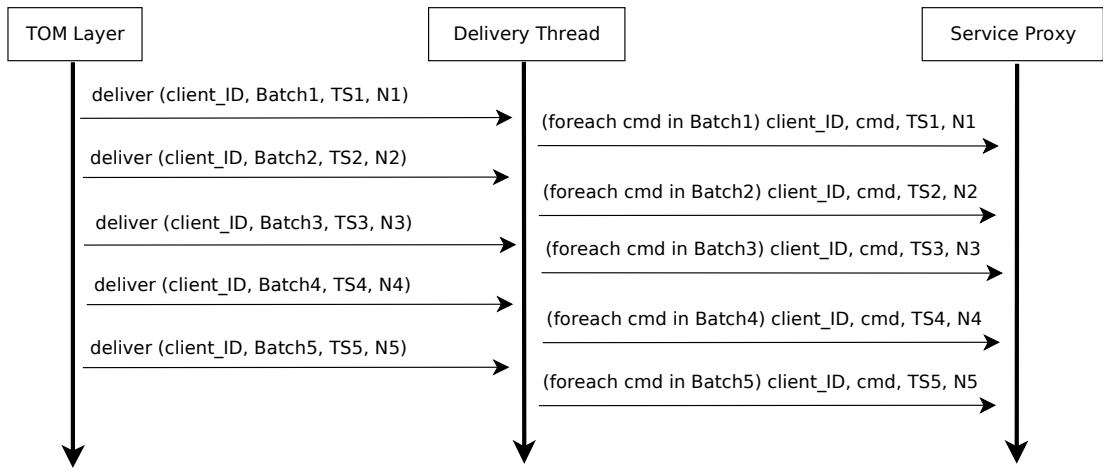


Figura 3.5: Entrega de *batches* de comandos à aplicação. Cada comando em cada batch é entregue em separado.

Tal como foi referido na secção 3.1.1, mensagens de consenso fora de contexto devem ser guardadas em memória, caso sejam referentes a um consenso posterior ao que estiver a ser executado num determinado processo. A componente que realiza a gestão dessas

mensagens é a *Out of Context Thread*. Sempre que um consenso terminar, a *Out of Context Thread* irá entregar ao componente *TOMLayer* as mensagens referentes ao consenso que está à espera de executar, caso alguma tenha sido recebida durante um consenso anterior.

Finalmente, existem dois tipos de mensagens que são trocados entre as réplicas: mensagens do tipo *PaxosMessage*, que são mensagens usadas ao nível da execução do consenso, e do tipo *TOMMessage*, que são usadas pelo *TOMLayer* para os clientes enviarem comandos às réplicas, assim como as réplicas enviarem respostas aos clientes.

3.2.2 Modelo de Uso

Para uma aplicação usar a biblioteca BFT-SMaRt , necessita de no lado do cliente:

1. Criar um objecto da classe `ServiceProxy` passando um ficheiro de configuração que define a lista de réplicas

2. Invocar o método `public byte[] invoke(byte[] request)` dessa classe

A classe *ServiceProxy* irá servir de intermediário entre o cliente e as réplicas. O seu método `invoke` é um RPC que irá enviar o comando para as réplicas, e devolverá a resposta correspondente.

Do lado do servidor, é necessário:

1. Codificar uma sub-classe de `ServiceReplica`

2. Concretizar o método `public abstract byte[] executeCommand(...)` dessa sub-classe

O programador deverá ter atenção ao concretizar este método, porque é aqui que deve ser garantido o determinismo. Isto significa que o programador não deverá gerar nada que o quebre (e.g., números aleatórios e estampilhas temporais) porque é impossível garantir que todas as réplicas gerem os mesmos números aleatórios - pelo simples facto de ser aleatório - ou estampilhas temporais - pois os relógios das réplicas não são sincronizados entre si. Para resolver este problema, é o próprio BFT-SMaRt que gera bytes aleatórios e estampilhas temporais, e entrega esses valores a todas as réplicas. Esses valores são gerados pela réplica líder da execução do consenso. São recebidos no método `executeCommand`, e estão nos parâmetros *nonces* e *timestamp*, respectivamente. Sempre que o programador necessitar deste tipo de dados, deverá usar os que são fornecidos neste método de forma a garantir o determinismo.

3.3 Limitações

O algoritmo do BFT-SMaRt na sua concretização actual, consegue oferecer uma primitiva de difusão atómica que entrega mensagens de vários clientes para as réplicas que constituem a aplicação, assim como tolerar comportamento faltoso de um número máximo de réplicas, sendo que para tolerar f réplicas faltosas, é necessário um total de pelo menos $3f + 1$ réplicas.

No entanto o algoritmo, só por si, não garante que esse número f nunca será excedido. Apenas é garantido que a difusão atómica será feita correctamente se no máximo f réplicas forem comprometidas. Garantir que nunca são comprometidas mais do que f não faz parte deste trabalho, e já existe investigação dedicada a resolver esse problema [18]. Logo, durante a realização deste trabalho, assumiu-se que nunca seriam comprometidas mais do que f réplicas.

A autenticação entre clientes e réplicas é feita com cifra assimétrica. Isto permite uma grande simplificação do algoritmo, e consequentemente, do código da biblioteca. Apesar de por si só não ser uma limitação, a operação de verificar assinaturas é relativamente lenta, e no caso de existirem muitos clientes, atrasa as computações. A resolução deste problema depende das futuras arquitecturas de computadores, nomeadamente na quantidade de núcleos dos processadores. A ideia é paralelizar a verificação das assinaturas, afectando cada thread responsável por as verificar, a um núcleo diferente. Desta forma consegue-se um melhoramento de desempenho do BFT-SMaRt.

Esta biblioteca também não suporta o conceito de vistas, i. e., parte do pressuposto que durante toda a execução do sistema o número de réplicas presentes é estático. Não é possível indicar que se querem tirar nem acrescentar réplicas durante a execução deste. Isto implica que para alterar o número de réplicas presentes é preciso parar a execução do sistema todo e começá-la de início com um novo número de réplicas.

3.4 Considerações Finais

Neste capítulo foi apresentada a biblioteca BFT-SMaRt e explicado o algoritmo usado por esta para realizar a técnica de replicação de máquina de estados. Foi também descrita a sua arquitectura e modo de uso do ponto de vista do programador. Por último, foram descritas as limitações da biblioteca.

No próximo capítulo é apresentada a funcionalidade introduzida nesta biblioteca durante a realização deste trabalho.

Capítulo 4

Protocolo de Transferência de Estado

Este capítulo destina-se a descrever o trabalho realizado na biblioteca BFT-SMaRt, que consistiu na realização de um protocolo de transferência de estado. Este protocolo tem como objectivo restaurar o estado em réplicas que o tenham perdido, por algum dos motivos descritos na secção 4.1.

4.1 Motivação

O protocolo de transferência de estado foi projectado para solucionar um conjunto de problemas que o BFT-SMaRt possuía antes do início deste trabalho. Este protocolo foi feito para ser desencadeado nas situações descritas de seguida.

4.1.1 Falha e Recuperação de Réplicas

Antes da realização deste trabalho, a biblioteca BFT-SMaRt possuía outra limitação para além das mencionadas na secção 3.3. Essa limitação estava relacionada com réplicas que falhassem - fosse por paragem, arbitrariamente ou intrusão - e recuperassem mais tarde: essas réplicas ficavam fora de contexto em relação ao resto do grupo. Isto devia-se ao facto de, durante o período de tempo compreendido entre a paragem e a recuperação, as réplicas perderem parte dos comandos enviados pelos clientes, assim como mensagens de consenso geradas pelas outras réplicas. Logo, como nunca recebiam esses comandos, não podiam actualizar o seu estado de forma a que este se tornasse igual aos das outra réplicas que continuavam a executar normalmente.

4.1.2 Atraso de Réplicas

Um outro problema existente na versão original do BFT-SMaRt estava relacionado com réplicas que se deixassem ficar atrasadas em relação a outras. Isto acontecia porque o algoritmo PaW só precisa no máximo de $n - f$ mensagens vindas de processos diferentes para avançar para o próximo passo do algoritmo. Consequentemente, são esses $n - f$

réplicas que avançam primeiro para o próximo consenso. Ou seja, $n - f$ processos ditam o ritmo de execução do BFT-SMaRt, enquanto as restantes f réplicas ficam para trás.

Os f processos atrasados acabavam por receber mensagens dos $n - f$ processos adiantados cujo conteúdo se refere ao consenso que estavam a executar. Mas essas mensagens vão ser interpretadas pelo *Execution Manager* como sendo fora do contexto. Este fenómeno em conjunto com a latência da rede cria um efeito "bola de neve", e cada processo acaba por atingir o limite *paxos_highmark* mencionado na secção 3.2.1 e param de guardar em memória essas mensagens. Acabam por se perder comandos que deveriam ser entregues à aplicação para o estado dela continuar a evoluir da mesma maneira que as restantes réplicas.

4.1.3 Novas Funcionalidades

A biblioteca BFT-SMaRt encontra-se em desenvolvimento e existe trabalho futuro para ser realizado sob ela que precisará de um protocolo de transferência de estado.

Como já foi dito no fim da secção 3.3, o BFT-SMaRt não suporta o conceito de vistas. Esta funcionalidade faz parte do trabalho futuro a ser realizado nela. Mas no caso de instalar uma nova vista com mais uma réplicas no grupo, será preciso passar o estado a essa réplica - porque ela irá começar a sua execução sob o estado inicial, que não corresponderá ao estado das réplicas que já estavam a executar na vista anterior.

Outro trabalho futuro será integrar o BFT-SMaRt no contexto do rejuvenescimento de réplicas tal como descrito em [18]. Para que as réplicas possam retomar a sua execução a partir do ponto em que estavam antes de serem rejuvenescidas, precisam também de um protocolo que lhes permita obter o estado de outras réplicas.

4.2 Checkpoints

Esta secção apresenta a funcionalidade de *checkpoints* que foi adicionada ao BFT-SMaRt, assim como o seu funcionamento.

4.2.1 Descrição Geral

O protocolo de transferência de estado executa ao nível do BFT-SMaRt. A aplicação nunca se apercebe de quando é que o protocolo é desencadeado nem como funciona. Por outro lado, é na aplicação que o estado reside e o BFT-SMaRt não participa na sua evolução.

Quando uma réplica recebe um pedido para enviar o estado a outra, é necessário que ela o tenha consigo para responder. Mas pedir o estado à aplicação sempre que surge um pedido é um passo que pode levar tempo, dependendo do tamanho e complexidade deste.

E também é inexequível guardar em memória todos as versões do estados resultantes de lhe aplicar cada valor decidido no consenso[1].

Adicionalmente, o BFT-SMaRt não sabe como o estado está estruturado na aplicação (i.e., não sabe se é apenas um inteiro, ou uma estrutura de dados complexa) nem sabe o que o ele representa (i.e., não sabe se é um contador ou uma imagem). O BFT-SMaRt é responsável pela replicação mas não é responsável por gerir o estado.

Para resolver este problema foi criada a funcionalidade de *checkpoints*. A ideia desta funcionalidade é requisitar o estado à aplicação periodicamente, em vez de somente quando chega um pedido de outra réplica. Essa requisição é efectuada de X em X execuções de consensos decididos pelo algoritmo PaW usado pelo BFT-SMaRt. Esse período é definido pelo parâmetro de configuração $checkpoint\_period$.

Mas cada réplica não se limita a pedir estado à aplicação em cada *checkpoint*. Também precisa de guarda todos os valores decididos nas execuções de consensos entre cada *checkpoint*. Isto porque lhe pode ser requisitado um estado que não tenha sido guardado num checkpoint (e.g., pedirem-lhe o estado da execução até o consenso 15 (inclusive), mas o último *checkpoint* ter sido realizado após a execução 10). A figura 4.1 ilustra a evolução da informação que é guardada em cada *checkpoint* ao longo das execuções de um consenso numa réplica a funcionar correctamente.

A ideia é devolver à replica que iniciar o protocolo dois tipos de informação:

- O estado que foi pedido no último *checkpoint*;
- Todos os $batches$ de mensagens que foram guardados desde esse *checkpoint* até ao consenso especificado por essa réplica.

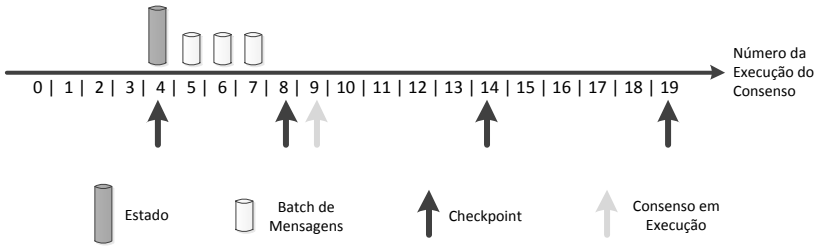
Sempre que é realizado um novo *checkpoint*, o estado e os $batches$ que estavam a ser guardados no anterior são descartados para o consumo de memória não crescer em função do tempo em que o BFT-SMaRt é deixado a executar. Mas isto implica que é possível que a réplica recuperada peça um estado que já não está guardado em memória pelo BFT-SMaRt. Se isso acontecer, as outras respondem com um estado vazio, e o protocolo será re-iniciado com a réplica recuperada a analisar mensagens de um próximo consenso. Este ciclo continua até que as restantes réplicas devolvam um estado à que o pediu.

4.2.2 Guardar o Estado

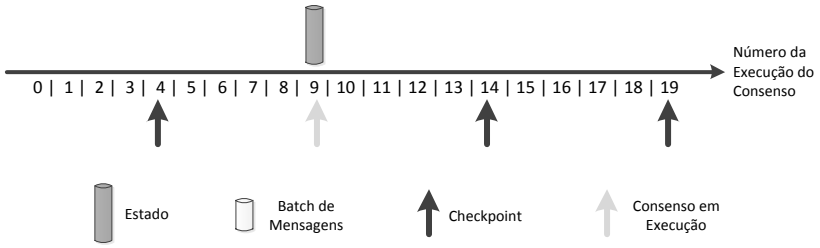
Em cada *checkpoint*, depois do consenso ser decidido e se obter o estado da aplicação, o BFT-SMaRt guarda os seguintes dados em memória:

- O estado obtido da aplicação;

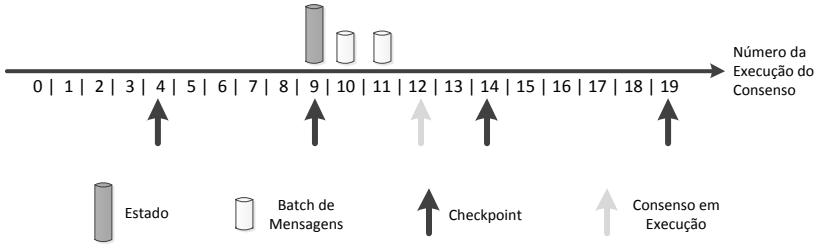
[1] No contexto do BFT-SMaRt, os valores do consenso entregues à aplicação correspondem aos $batches$ de mensagens ordenados



(a) Aqui a réplica termina a execução do consenso nº4, pede o estado à aplicação e guarda-o. Entre o consenso nº5 e nº8 vai guardando os $batches$ de mensagens que recebe do consenso.



(b) Aqui a réplica chega ao consenso nº9 e pede de novo o estado. Esse estado é sobreposto ao antigo, e os $batches$ que tinham sido guardados até aqui são descartados.



(c) Aqui a réplica já avançou para o consenso nº12, tendo guardado os $batches$ decididos no consenso nº10 e nº11.

Figura 4.1: Ilustração da evolução dos *checkpoints* numa réplica, considerando que o $checkpoint\_period$ é igual a 5.

- O resumo criptográfico deste estado;
- O número do consenso após o qual o estado foi guardado;
- O *round* deste consenso em que houve uma decisão;
- A réplica que foi o líder no *round* em que se deu a decisão.

É responsabilidade da aplicação codificar o estado num vector de bytes já que o BFT-SMaRt não sabe como ela representa o estado, como já foi dito anteriormente. O papel do resumo criptográfico será explicado na secção 4.3.2. O número do consenso é necessário para conseguir associar o estado a um consenso. Finalmente, o *round* e o líder são

necessários porque a réplica que pedir o estado terá que saltar a execução de todos os consensos entre aquele em que iniciou o protocolo de transferência de estado e aquele que estiver associado ao estado que pediu. Como a informação do *round* e do líder vão sendo actualizados durante a execução do algoritmo PaW, é necessário actualizar os dados do algoritmo quando a réplica instalar o novo estado e indicar ao algoritmo qual a execução do consenso actual.

4.2.3 Guardar os $Batches$ de Mensagens

Entre cada *checkpoint* e depois do consenso ser decidido, o BFT-SMaRt guarda os seguinte dados em memória:

- O $batch$ de mensagens que foi decidido nesse consenso;
- O número do consenso em que o estado foi guardado;
- O *round* do consenso em que houve uma decisão;
- A réplica que foi o líder no *round* em que se deu a decisão.

Como já foi dito, os $batches$ de mensagens precisam de ser enviados juntamente com o estado. E também estes precisam de ter associado o número do consenso, *round* de decisão e líder desse *round* para actualizar o algoritmo PaW.

4.3 Restauro do Estado

A secção 4.2 descreve como uma réplica correcta do BFT-SMaRt prepara o estado na eventualidade de uma das réplicas necessitar dele. Aqui irá ser descrito como uma réplica do BFT-SMaRt que falhe e se recupere se apercebe que isso aconteceu e como pede o estado.

4.3.1 Desencadeamento

O protocolo de transferência de estado é desencadeado sempre pela réplica que precisa do estado, quando ocorre qualquer uma das situações descritas na secção 4.1. A mesma metodologia pode ser usada para todos os casos mencionados, porque a réplica em questão irá sempre capturar mensagens de consensos mais adiantados que aquele que ela está à espera de executar. A réplica irá guardar essas mensagens como sendo fora de contexto, e acabará por atingir o limite $paxos\_highmark$. Quando esse limite é ultrapassado a réplica dá início ao protocolo.

Apesar da metodologia descrita servir para todos os casos da secção 4.1, foi feita uma optimização para o caso de réplicas que tenham re-iniciado a sua execução. Nesses

casos é usado o limite $revival\_highmark$, que é mais curto que o $paxos\_highmark$. Uma réplica sabe que deve considerar o $revival\_highmark$ se verificar que ainda está à espera do primeiro consenso de todos, mas as restantes já estão a executar um consenso mais adiantado (e.g., a réplica recuperada está à espera que comece a ser executado o consenso 0, mas observa que o consenso 10 já está a ser executado pelas outras).

Quando a réplica se apercebe que tem que pedir o estado, espera por receber $f + 1$ mensagens vindas de réplicas diferentes que se refiram ao mesmo consenso[2]. Após receber essas mensagens a réplica tem a garantia que pelo menos uma réplica correcta está a executar esse consenso. Após ter obtido $f + 1$ mensagens para um consenso c, irá dar inicio ao protocolo para obter o estado correspondente ao consenso $c - 1$. A réplica pede o estado para o consenso $c - 1$ porque se estiver a ser executado o consenso c, significa que $c - 1$ já foi decidido e por isso os comandos entregues à aplicação durante esse consenso já foram aplicados ao estado correspondente a $c - 1$.

4.3.2 Descrição do Protocolo

Quando o protocolo entrar em execução, todas as mensagens que digam respeito ao algoritmo de consenso irão deixar de ser processadas pela réplica que pede o estado e serão guardadas em memória como sendo fora de contexto. Os passos do protocolo estão representados de uma forma geral na figura 4.2

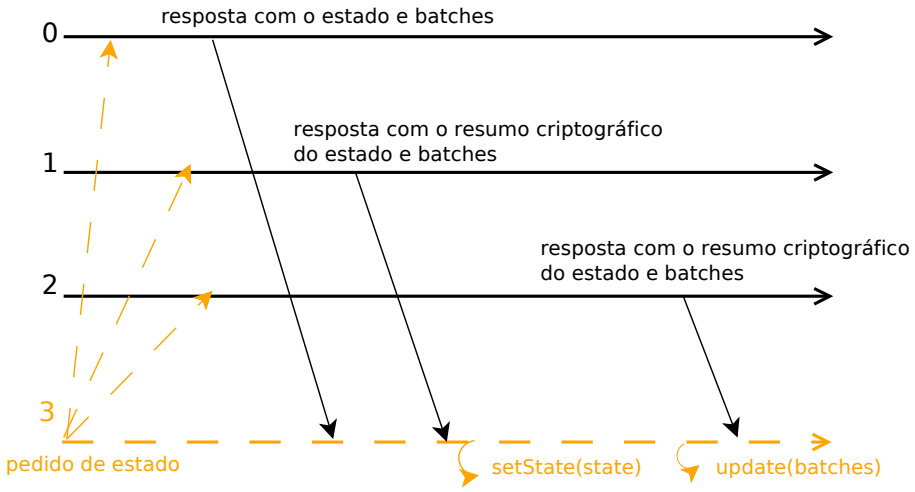


Figura 4.2: Ilustração geral da troca de mensagens do protocolo de transferência de estado. A réplica que inicia o protocolo está desenhada a tracejado. Nesta figura assume-se $f = 1$.

A réplica que precisa pedir o estado analisa as mensagens que recebe das outras para

[2]Devido à latência da rede, as mensagens que recebe podem ser de consensos antigos, mas essas mensagens também podem ser usadas. O que importa para desencadear o protocolo é descobrir um consenso que já tenha sido decidido.

descobrir um consenso que já tenha sido decidido, tal como descrito na secção anterior. Após obter essa informação, envia a todas as réplicas do grupo um pedido para estas lhe enviarem o estado associado à execução desse consenso. Após receber a mesma resposta vinda de $f+1$ réplicas diferentes para a instância do consenso requisitada, a réplica actualiza o estado da aplicação, entrega-lhe os $batches$ pela mesma ordem que foram entregues nas outras réplicas, e retoma a sua execução normal. São precisas $f+1$ resposta de réplicas diferentes para existir a certeza de que pelo menos uma réplica correcta respondeu e que o estado recebido é verdadeiro. Quando todos os comandos contidos em cada $batch$ forem executados sobre o estado recebido, a réplica terá o mesmo estado que as outras tinham na execução do consenso que foi pedido no protocolo. A partir daqui, a réplica irá processar as mensagens de consenso que guardou como sendo fora do contexto que sejam posteriores ao consenso para o qual se pediu o estado.

Finalmente, não são todas as réplicas a enviar o estado. A única que o envia é aquela que for especificada no pedido. As restantes mandam apenas o resumo criptográfico do estado. Isto é uma optimização que serve para reduzir a carga que o protocolo injecta na rede. É a réplica que pede o estado quem escolhe qual das outras réplicas deverá enviar o estado inteiro. Note que se o estado recebido estiver errado, é escolhida outra réplica para enviar o estado e o protocolo é reiniciado.

Interacções

As interacções detalhadas entre as réplicas estão representado na figura 4.3. Tendo em conta a figura, considera-se o seguinte:

- A réplica que pede o estado é a R_3;
- O estado pedido é o correspondente ao consenso número N;
- O estado que cada réplica obteve da aplicação no último *checkpoint* é o correspondente ao consenso número M.

Ao receber $f+1$ mensagens vindas de réplicas diferentes para um consenso cujo número ultrapasse o limite $revival\_highmark$ (ou $paxos\_highmark$), a réplica irá enviar um pedido às restantes constituído por:

- O número do consenso após o qual quer obter o estado (C_N);
- O identificador da réplica que lhe deverá enviar o estado inteiro (ID_{R0}).

O identificador do consenso C_N é usado por R_3 para conseguir associar as respostas das outras réplicas ao pedido que lhes enviou. Por outro lado, também é usado pelas outras réplicas para determinarem quantos $batches$ deverão enviar (e.g., podem ter $batches$ guardados desde o consenso 21 até ao 25, mas se for pedido o estado para o consenso 23,

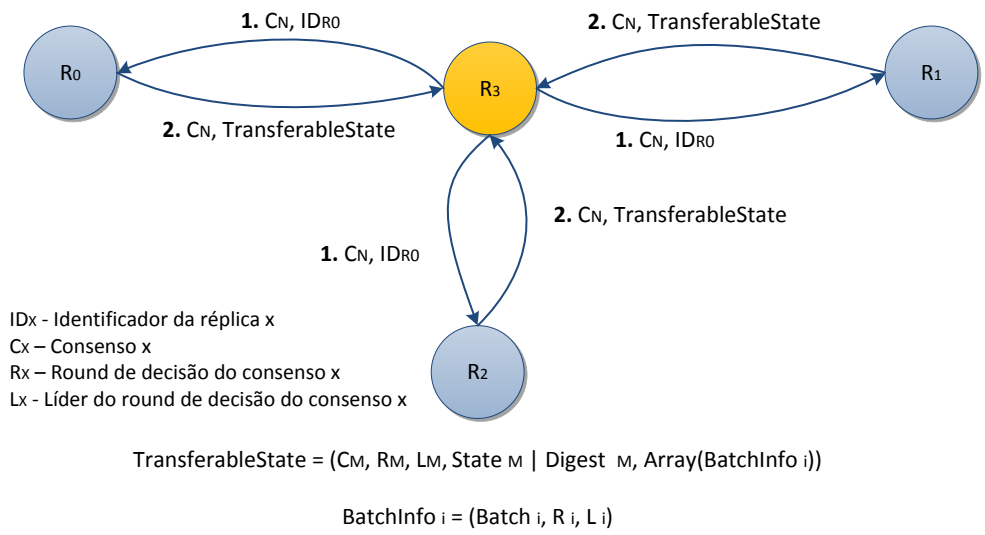


Figura 4.3: Interacções detalhadas do protocolo de transferência de estado. A réplica que pede o estado é a réplica R_3

será devolvido o estado do *checkpoint* realizado no consenso 20 e os $batches$ do consenso 21 a 23).

Quando uma réplica recebe um pedido de estado, verifica se C_N está entre C_M e o último $batch$ de mensagens guardado. Se estiver, constrói uma estrutura de dados designada de $TransferableState$ que contém:

- O estado obtido da aplicação após o consenso C_N ou o resumo criptográfico desse estado ($State_M$ ou $Digest_M$);
- O número do consenso M (C_M);
- O *round* de C_M em que foi decidido um valor (R_M);
- O identificador do processo que era o líder em R_M (L_M);
- Um vector de $batches$ de mensagens ($Array(BatchInfo_i)$).

Conforme a réplica em questão for aquela que deverá devolver o estado é incluído no $TransferableState$ o estado associado ao consenso C_M, ou o seu respectivo resumo criptográfico - no caso da figura 4.3, é a réplica R_0 que foi escolhida por R_3 para enviar o estado.

Dentro da estrutura $TransferableState$ também é incluído um vector de tuplos designados de $BatchInfo$, que associam a cada $batch$ o *round* do consenso em que foi decidido o valor do $batch$ (R_i), assim como o identificador da réplica líder nesse *round* (L_i). Os $batches$ incluídos no vector são todos aqueles que foram entregues à aplicação

entre o C_M (exclusive) e o consenso C_N (inclusive), e estão ordenados pela ordem em que foram entregues à aplicação.

É então enviada a resposta a R_3 contendo:

- A estrutura $TransferableState$;
- O número do consenso N (C_N).

R_3 irá verificar se o número do consenso de cada resposta corresponde a C_N. Se corresponder, guarda essa resposta em memória. Quando receber a resposta de R_0 com o estado, faz a seguinte verificação:

- Se já tiver f respostas com o resumo criptográfico consistente com o estado que recebeu de R_0, irá usar esse estado para actualizar a aplicação.
- Se tiver menos de f respostas com o resumo criptográfico mas este for consistente com o estado que recebeu de R_0, espera por mais respostas.
- Se já tiver obtido $f + 1$ respostas com o resumo criptográfico, mas o estado que recebeu de R_0 não for consiste com o resumo das $f + 1$ respostas, re-inicia o protocolo especificando outra réplica - uma das que enviaram o resumo criptográfico - responsável por enviar o estado.

Se a réplica verificar que pode usar o estado, irá forçar a aplicação a sobrepôr o estado actual pelo que foi recebido de R_0. De seguida actualiza a informação do algoritmo PaW de forma a registar que no consenso C_M o *round* de decisão foi R_M e o identificador da réplica líder foi L_M.

Finalmente, entrega à aplicação cada comando contido nos $batches$ para serem aplicados no estado pela mesma ordem que foram aplicados nas restantes réplicas. O algoritmo PaW também é actualizado para cada $batch$, de maneira a registar o *round* e líder do consenso em que foram entregues.

4.4 Concretização

Nesta secção são descritas as adições que foram necessárias fazer ao BFT-SMaRt para este suportar o protocolo de transferência de estado e também o que fazer para a aplicação suportar esse protocolo.

4.4.1 Arquitectura

Para concretizar este protocolo, foi adicionado ao código do BFT-SMaRt um novo *package* designado de `navigators.smart.statemanagment`, que contém o código

responsável por realizar esta funcionalidade. Todas as classes que foram criadas para concretizar este protocolo estão neste *package*.

Na arquitectura do BFT-SMaRt, foi adicionado ao componente $TOMLayer$ o subcomponente $StateManager$, que faz a gestão da transferência de estado e também é responsável pela gestão da informação dos *checkpoints*, através de uma parte lógica especializada para isso, designada de $StateLog$. A figura 4.4 ilustra essa alteração.

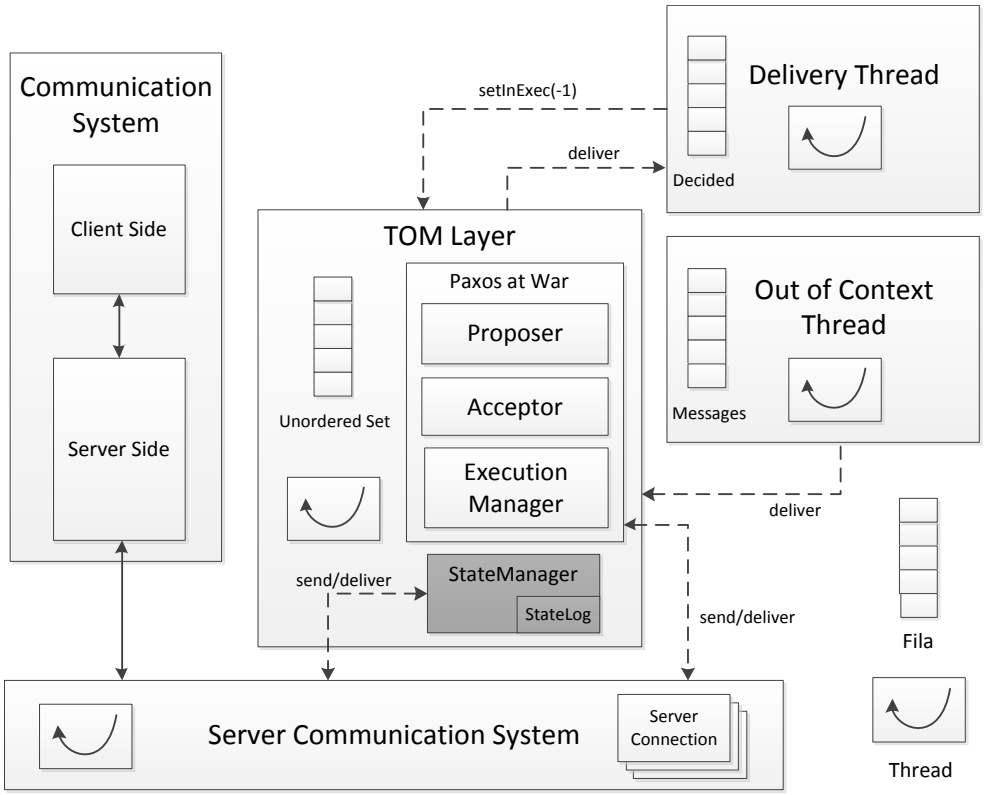


Figura 4.4: Arquitectura do actual do BFT-SMaRt, que incluí o módulo responsável pela transferência de estado, representado a cinzento.

Foram feitas algumas alterações ao comportamento da $DeliveryThread$, de forma a conseguir realizar os *checkpoints*. Essa alteração está ilustrada na figura 4.5. Após entregar cada *batch* à aplicação - que tem de ser concretizada numa classe que descenda de $ServiceProxy$ - a $DeliveryThread$ irá verificar se deve efectuar um *checkpoint* para o consenso que lhe foi entregue, i. e., dado um consenso número c, irá verificar se $c \,\%\, checkpoint\_period = 0$. Caso seja necessário realizar um *checkpoint* para c, irá pedir o estado à aplicação e guardá-lo no $StateLog$. Caso contrário, irá guardar no $StateLog$ o batch de mensagens.

A $DeliveryThread$ também foi alterada para impôr à aplicação o estado recebido do protocolo, assim como entregar à aplicação todos os comandos contidos em cada *batch* de mensagens que também chegaram do protocolo.

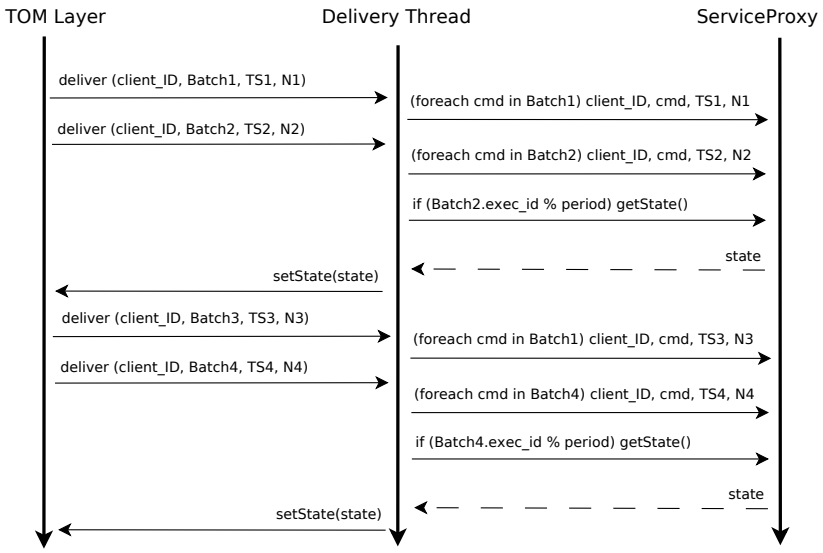


Figura 4.5: Ilustração das interacções entre o *TOM Layer* e a *Delivery Thread* aquando da realização de checkpoints.

Também foi introduzido um novo tipo de mensagens designado de `SMMessage`. Estas são as mensagens responsáveis por realizar este protocolo. Há duas variantes destas mensagens: A variante `SM_REQUEST` é enviada pela réplica que inicia o protocolo, e contém a informação apresentada no passo 1 da figura 4.3. A variante `SM_REPLY` é enviada pelas restantes réplicas para responderem à que pede o estado, e contém a informação apresentada no passo 2 da figura 4.3.

4.4.2 Modelo de Uso

Para uma aplicação usar a biblioteca BFT-SMaRt após a adição deste protocolo, é necessário codificá-la do lado do cliente da mesma maneira que foi descrito na secção 3.2.2. Do lado do servidor, para além do que foi dito na secção 3.2.2, passa a ser necessário concretizar os seguinte métodos na classe onde é concretizada a aplicação:

```
protected abstract byte[] serializeState();
protected abstract byte[] deserializeState(byte[] state);
```

O método `serializeState()` é usado pela $DeliveryThread$ para se obter o estado da aplicação. Como já foi dito na secção 4.2, o BFT-SMaRt não precisa de saber como nem o que é representado no estado, mas para ser possível guarda-lo no BFT-SMaRt, é necessário entregar à biblioteca algo que ela esteja à espera. O programador deverá assegurar que este método codifica o estado da aplicação para um vector de bytes de forma a que ao descodificá-lo, o resultado seja igual ao estado que se codificou.

O método `deserializeState(byte[] state)` é usado pela $DeliveryThread$ para impôr à aplicação o estado recebido do protocolo. O programador deverá assegurar que este método descodifica o vector de bytes e sobrepõe o estado obtido àquele que a aplicação têm no momento.

4.5 Considerações Finais

Neste capítulo foi descrito o protocolo de transferência de estado que foi adicionado ao BFT-SMaRt durante a realização deste trabalho e as motivações para a concretização do mesmo, que vão desde falhas de réplicas até funcionalidades que vão ser introduzidas no futuro e que necessitam de fazer uso deste protocolo. Também foi descrita a nova funcionalidade de *checkpoints*, que obtém o estado da aplicação e prepara-o para ser enviado na eventualidade deste protocolo ser desencadeado.

No próximo capítulo é apresentado o serviço TYPHON, que faz uso da versão do BFT-SMaRt resultante das adições descritas neste capítulo.

Capítulo 5

Serviço TYPHON

Neste capítulo é descrito em detalhe o serviço TYPHON, um serviço de autenticação e autorização tolerante a intrusões. Este serviço foi construído usando a primitiva de difusão atómica oferecida pela biblioteca BFT-SMaRt [2] e um conjunto de funcionalidades disponibilizadas por um componente seguro. De notar que é possível conceber este serviço sem subverter a especificação do Kerberos v5 tal como é apresentada no RFC 4120.

5.1 Limitações do Kerberos

O Kerberos v5 tal como descrito permite que várias entidades se autentiquem sem que cada uma precise de partilhar uma chave secreta com as outras - o que implicaria um crescimento exponencial de chaves em função do número de entidades - e também evita a necessidade de criptografia assimétrica - que implica o uso de computações geralmente mais lentas que as de cifra simétrica. Isto é possível graças ao uso de uma aplicação centralizada (composta pelo AS e TGS) que conhece as chaves secretas de todas as entidades do sistema.

Mas a norma Kerberos foi criada com um pressuposto muito forte: o processo que concretiza a norma e a máquina que o executa nunca são comprometidos. O único perigo está na rede, que não assegura por si só a confidencialidade das mensagens. Mas tal assumpção não pode ser dada como garantida na prática. Isso dá origem a um ponto fraco nesta arquitectura: se o processo que executa o AS/TGS falhar por paragem, torna-se impossível realizar autenticação a partir do momento em que a falha ocorre. Os clientes que já tenham adquirido *tickets* para serviços ainda conseguem dialogar com os serviços para os quais esses *tickets* foram gerados[1], mas não é possível fazer novas autenticações.

Também é possível que ocorram falhas arbitrárias (e.g., falhas de hardware e bugs) e também intrusões. No caso das intrusões, as chaves secretas podem ser obtidas, e quem se apoderar delas pode vir a personificar clientes e serviços.

[1] Mas não eternamente, porque os *tickets* têm um prazo de validade.

Ainda existe uma outra limitação na especificação do Kerberos: embora a norma defina em pormenor como garantir a autenticação de cada entidade presente no sistema, não define nada sobre a autorização que essas mesmas entidades têm. A parte de autorização é delegada aos serviços nos quais os clientes se autenticam.

O desafio do serviço TYPHON é tornar mais robustas as quatro propriedades de segurança de um serviço Kerberos v5:

- Autenticidade - Medida em que quem usa o sistema consegue provar a sua identidade;
- Integridade - Medida em que os dados do sistema estão protegidos da sua corrupção;
- Confidencialidade - Medida em que os dados do sistema estão protegidos de acesso não autorizado;
- Disponibilidade - Medida em que o sistema está protegido contra ataques que visem perturbar a sua execução.

A propriedade de autenticidade já é oferecida à partida pelo serviço Kerberos v5, uma vez que o seu objectivo é precisamente garantir a autenticidade dos clientes e serviços que o usam. Para garantir as três propriedades de segurança restantes, será usada a técnica de replicação de máquina de estados combinada com um componente seguro local em cada réplica.

5.1.1 Técnica de replicação de máquina de estados

Como já foi dito na secção 3.1.3, a técnica da replicação da máquina de estados tem por objectivo replicar esse estado em várias processos que são considerados réplicas. A ideia é replicar o serviço Kerberos por $n \geq 3f + 1$ réplicas de forma a que o serviço continue disponível mesmo que no máximo f réplicas falhem. Assim conseguesse reforçar as propriedades de integridade e disponibilidade

No entanto, é preciso notar que esta técnica de replicação, apesar de reforçar as propriedades de Integridade e Disponibilidade, não assegura Confidencialidade e Autenticidade. Pelo contrário, torna estas propriedades mais fracas. Isto porque ao tornarmos o sistema replicado, aumentamos a probabilidade de algum servidor ser corrompido. Como o estado está replicado em todos os servidores e, no caso do Kerberos o estado corresponde às chaves, basta uma intrusão para que estas sejam obtidas.

5.1.2 Protecção das Chaves

Como foi explicado na secção anterior, a técnica de replicação de máquina de estados é imprópria para garantir a propriedade de confidencialidade. Como tal, somente com essa

técnica, apenas é possível construir um serviço Kerberos tolerante a faltas arbitrárias - mas não tolerante a intrusões.

Para se evitar o problema acabado de descrever, é necessário garantir que as chaves armazenadas por cada réplica não ficam acessíveis mesmo que a réplica seja comprometida. Neste trabalho, a protecção das chaves é efectuada através da utilização de um componente seguro [7]. Um componente seguro trata-se de uma parte do sistema que deverá ter uma concretização suficientemente simples, de forma a ser exequível verificar as suas propriedades funcionais e não funcionais.

O componente seguro permite obter a propriedade de confidencialidade, uma vez que armazena todos os dados confidenciais, i.e., as chaves. Também se delegam a este as operações que processam esses dados. A ideia é guardar os dados confidenciais no componente e executar operações sobre eles sem que estes sejam expostos ao resto do sistema. Na secção 5.4 será explicado como é que este componente assegura a autenticidade.

5.2 Modelo de Sistema

Assumimos um modelo e arquitectura de sistema híbridos em que o sistema é composto por duas partes, com propriedades e pressupostos distintos [19]. Estas duas partes designam-se tipicamente por *payload* e *wormhole*. Assume-se que a parte payload é composta por um conjunto de $n \geq 3f + 1$ réplicas e que um máximo de f réplicas podem falhar de forma arbitrária. Na parte *wormhole* executa um componente seguro com as funcionalidades que serão descritas mais à frente. Assume-se que esta parte do sistema é segura por construção.

A biblioteca BFT-SMaRt executa na parte *payload*, assim como os componentes AS e TGS do serviço TYPHON. É assumido que esta parte executa sob um modelo de sistema eventualmente síncrono semelhante ao definido em [6] uma vez que a primitiva de difusão atómica oferecida pela biblioteca BFT-SMaRt necessita deste pressuposto. Também se assume sincronia local, i.e., o tempo máximo de processamento local e a taxa de desvio de cada relógio local são limitados e conhecidos. Estes pressupostos são requeridos pelo serviço de estampilhas temporais do BFT-SMaRt. Este serviço é usado pelo TYPHON tal como será explicado mais à frente.

5.3 Descrição Geral

O AS e o TGS apesar de serem componentes lógicos distintos são executados pelo mesmo processo. Por sua vez este processo é replicado em várias máquinas, e estas fazem uso da técnica da replicação da máquina de estados concretizada pelo BFT-SMaRt.

Cada réplica tem acesso a um componente seguro. Esse componente guarda dentro de si as chaves dos clientes e dos serviços e oferece todas as operações essenciais para

realizar operações com elas. Cada componente seguro possui ainda uma chave secreta que é usada para cifrar dados, gerar chaves de sessão e criar MACs para uma estrutura de dados especial que são os *ticket approvals* (TAs).

TAs são estruturas de dados geradas pelo componente seguro de cada réplica para garantir que a geração, renovação e validação de um *ticket* para um serviço é permitida segundo a politica de autorização definida no TYPHON[2]. Quando os componentes TGS das várias réplicas do TYPHON recebem um pedido de *ticket* de serviço, cada réplica pede ao seu componente seguro a geração de um TA e envia esse TA para todas outras as réplicas. Posto isto, cada réplica espera a recepção de *2f + 1* TAs, incluindo o TA dela própria. Se de entre *2f + 1* TAs existirem pelo menos *f + 1* válidos, significa que pelo menos uma réplica correcta está a autorizar a geração do *ticket* para esse serviço, logo o componente seguro de cada réplica pode também gerá-lo.

A figura 5.1 ilustra a estrutura de um TA. Para assegurar a sua autenticidade, os TAs incluem um MAC gerado com a chave secreta dos componentes seguros - que é a mesma para todos. Para se saber qual a réplica que gerou um TA em particular, é incluído nele o identificador da réplica que o gerou ($Replica\ ID$). Também incluem dentro de si um resumo criptográfico dos parâmetros a serem passados ao componente seguro para a geração do *ticket* de serviço ($Hash\ Request$), de forma a assegurar que um conjunto de $f+1$ TAs válidos só pode ser usado para gerar o *ticket* de serviço a que estão associados. Dentro de cada TA também está incluído uma estampilha temporal ($Timestamp$) que evita ataques por repetição. Esta estampilha em conjunto com a chave secreta do componente seguro são usados na geração das chaves de sessão, de forma a garantir que é impossível prever qual a próxima chave a ser gerada pelo componente. Adicionalmente, também contêm o nome do cliente que requisitou o *ticket* ($Client\ Name$) e o nome do serviço para o qual esse *ticket* é dirigido ($Server\ Name$).

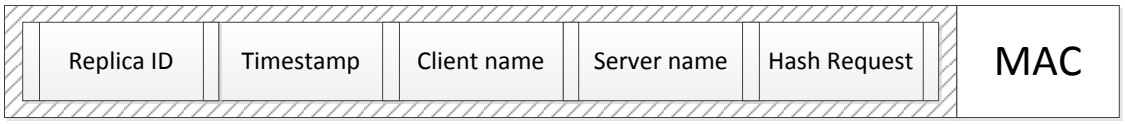


Figura 5.1: Ilustração da estrutura de um *ticket approval* (TA). A parte a tracejado corresponde ao corpo do TA e o MAC incluído neste é gerado com o resumo criptográfico desse corpo em conjunto com a chave secreta dos componentes seguros, para garantir a autenticidade do TA.

Todas as funcionalidades que não necessitem de realizar operações com/ou sobre as chaves são delegadas ao AS e TGS, como por exemplo definição de flags, verificação de realms, cálculo da validade dos *tickets* a serem emitidos, e outras operações semelhantes.

[2]Tal politica pode consistir simplesmente numa matriz de acesso que associa a cada cliente quais os serviços que este tem autorização para aceder.

5.4 Componente Seguro κ

Na concepção de um componente seguro há que ter em consideração o seguinte:

- A quantidade de código no componente (quanto menor for, mais fácil verificar a sua correcção)
- A interface do componente (quanto menor for, mais fácil usar e verificar que essa interface não pode ser usada para fazer acções erradas)
- A quantidade de vezes que o componente é acedido (porque como tipicamente reside numa máquina separada, é lento acedê-lo)

Tendo estas regras em consideração, foi concretizado o componente seguro κ, que o TGS deverá contactar sempre que precisar de fazer operações que envolvam as chaves dos clientes e/ou serviços[3]. κ foi concebido para oferecer as operações mínimas necessárias a executar sobre *tickets* que envolvam as chaves. κ também tem uma chave secreta que usa para cifrar chaves de sessão quando são devolvidas para o resto do sistema. Desta maneira não é necessário guardá-las em κ para preservar a sua confidencialidade. Os métodos de κ estão expostos na tabela 5.1.

Método	Argumentos	Retorno	Sumário
makeTA	timestamp, client_name, server_name, hash_request	ticket_approval	Gera ticket-approvals.
encryptParts	encrypted_sessionKey, ticket_data, reply_data, ticket_approvals	ticket, reply_encrypted_part	Gera o *ticket* e a parte cifrada da resposta do kerberos.
decryptParts	ticket, authenticator, server_name	ticket_data, authenticator_data, encrypted_sessionKey	Decifra o *ticket* e o *authenticator* enviados pelo cliente.
decryptPreAuth	PrincipalName, pa_data	byte_array	Decifra a pré-autenticação do cliente.

Tabela 5.1: Métodos disponibilizados por κ em cada réplica.

O método *makeTicketApproval* gera um ticket-approval (TA) contendo a estampilha temporal, nome de cliente que requisita o *ticket*, nome de serviço ao qual o *ticket* se

[3]Não é necessário que o AS use TAs, pois não faz sentido que os clientes não tenham autorização para se autenticarem perante o TYPHON.

destina, e o resumo criptográfico passados à função. Estas estruturas de dados foram explicadas na secção 5.3.

O método *encryptParts* é usado na geração de *tickets* e na renovação/validação dos mesmos. Também precisa de receber *2f + 1* TAs gerados pelos componentes κ das outras réplicas. Se de entre *2f + 1* TAs existirem pelo menos *f + 1* válidos, significa que pelo menos uma réplica correcta está a autorizar a geração do *ticket* para esse serviço, logo o componente κ da réplica em questão pode também gerá-lo.

O método *decryptParts* é essencial para decifrar *tickets* e *authenticators*. A chave de sessão contida no *ticket* passado como argumento é devolvida cifrada com a chave secreta de κ, de forma a evitar que seja guardada dentro dele.

Finalmente, o método *decryptPreAuth* serve para verificar a pré-autenticação do cliente. A pré-autenticação é opcional segundo o RFC 4120, logo este método não precisa de ser concretizado. Como tal, ao deixá-lo de fora mais facilmente se consegue validar a concretização de κ.

5.5 Interacções

O algoritmo do TYPHON cumpre a especificação do Kerberos v5 e define um conjunto de interacções adicionais com um componente seguro. Ou seja, as interacções entre clientes e serviços, assim como as interacções entre clientes e AS/TGS continuam iguais ao Kerberos v5. No entanto, AS e TGS têm de contactar κ sempre que efectuem operações relacionadas com as chaves de clientes e serviços. A figura 5.2 ilustra estas interacções.

Quando C envia um pedido ao AS para obter um TGT, este encaminha o pedido para κ, que por sua vez gera o TGT ($Ticket_{TGS}$) e a parte da resposta que é cifrada com a chave de C ($E_C\ [K_{C\_TGS},\ N_1,\ ID_{TGS}]$).

Após criar a chave de sessão, κ devolve ao AS o TGT ($Ticket_{TGS}$) e o tuplo cifrado ($E_C\ [K_{C\_TGS},\ N_1,\ ID_{TGS}]$). O AS por sua vez devolve a C o TGT e o tuplo.

Quando C envia um pedido ao TGS para obter um *ticket* para S, este re-envia para κ o ID de C (ID_C), o TGT ($Ticket_{TGS}$) e o *authenticator* ($Authenticator_{TGS}$). κ por sua vez decifra estas duas estruturas e devolve o seu conteúdo ao TGS, com o cuidado de cifrar a chave de sessão contida no TGT com a sua própria chave secreta - pois se o TGS for comprometido por um agente malicioso, este teria acesso a essa chave.

Após o TGS terminar as operações sobre os dados do TGT/*authenticator* e calcular novos dados de acordo com o RFC 4120, irá invocar κ para este criar um *ticket* para S. Esta interacção é semelhante àquela que cria o TGT, com a diferença de que é especificado o identificador de S em vez do identificador do TGS, e também é fornecida a chave de sessão entre C e o TGS (cifrada previamente com a chave do κ) de forma a que κ consiga criar $E_{C\_TGS}\ [K_{C\_V}, N_2, ID_S]$. Também é neste passo que todas as réplicas do sistema geram *ticket* approvals e enviam esse *ticket* para as outras, para provar a κ que C tem

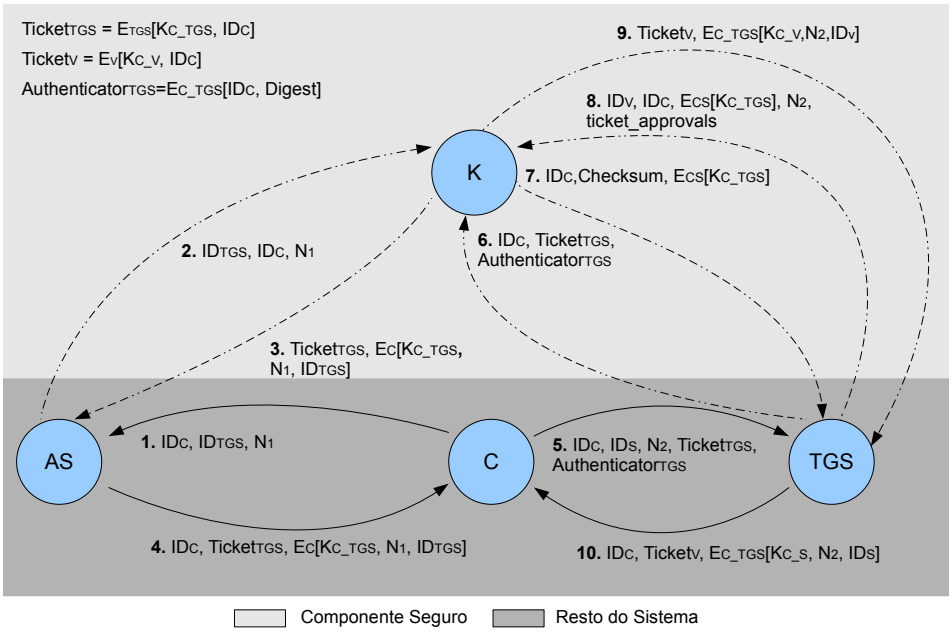


Figura 5.2: Ilustração simplificada das interacções das entidades presentes no TYPHON. A interacção com o serviço foi omitida desta figura porque é igual à da figura 2.1. A pré-autenticação também foi omitida por se tratar de um passo opcional.

autorização para interagir com S e por isso o *ticket* pode ser gerado.

Por sua vez, κ devolve ao TGS o *ticket* para S (ID_S) e o tuplo cifrado que contém a chave de sessão entre C e S ($E_{C\_TGS}\ [K_{C\_V}, N_2, ID_S]$). Estes são finalmente entregues pelo TGS a C. A partir daqui, a interacção entre C e S é igual à do Kerberos normal.

As interacções apresentadas aqui visam preservar a confidencialidade das chaves. Como κ é um componente seguro, as chaves nunca serão reveladas ao exterior, mesmo que no limite todas as réplicas sejam comprometidas. Isto porque κ não tem nenhuma operação que devolva as chaves. Os únicos dados confidenciais que o κ devolve são chaves de sessão, que vêm cifradas com a chave secreta deste. Se o tempo de utilidade dessas chaves (de sessão) for mais curto que o tempo que é necessário para quebrar a sua cifra através de um ataque de força bruta, também não é possível que intrusos consigam obter essas chaves em tempo útil. Também não se consegue gerar *tickets* para serviços arbitrariamente se o sistema for comprometido, devido ao uso de TAs, tal como explicado nas secções 5.3 e 5.4. Desta forma, conseguimos reforçar as propriedades de confidencialidade e autenticidade.

Note-se que os TAs não seriam necessários se fosse assumido que o TYPHON seria apenas um serviço de autenticação, deixando de lado a autorização. Os TAs servem para provar que existe pelo menos uma réplica correcta a fabricar esses *tickets*, o que significa que o cliente que requisitou esse *ticket* tem permissão para contactar o serviço que deseja. Mesmo que um intruso invadisse uma réplica e usasse κ para produzir *tickets* falsos em

nome de outro cliente, este não conseguiria obter a chave de sessão que deve ser usada para fabricar o *authenticator* que deverá ser apresentado ao serviço para o qual se quer autenticar. Como não teria um *authenticator* para apresentar, não conseguiria fazer-se passar pelo cliente legítimo. Também não conseguiria usar ataques de repetição, pois a própria especificação do Kerberos está feita para se defender disso.

5.6 Gestão das chaves secretas

A especificação do Kerberos v5 é bastante explícita acerca da maneira como clientes e serviços devem dialogar com o AS e TGS. O maior desafio deste trabalho foi perceber como incluir um componente seguro numa concretização do kerberos v5 sem alterar a especificação das mensagens e manter a transparência entre cliente e serviços que usem o Kerberos de forma padronizada.

Para realizar a gestão de chaves, poder-se-ia adicionar os seguintes elementos ao sistema:

- A função `setKeys(EncryptedKeys)` ao componente κ, onde o parâmetro `EncryptedKeys` é uma estrutura de dados cifrada com a chave de κ que contém todas as chaves secretas de clientes e serviços;
- Um tipo de pedido adicional à especificação do TYPHON especial para enviar a estrutura de dados mencionada anteriormente;
- Um cliente especial e seguro por construção, designado de α.

O cliente α conheceria todas as chaves contidas em κ, conheceria a chave secreta de κ, e só seria acessível pelo administrador do sistema. O administrador faria as alterações localmente em α, que ofereceria uma interface CRUD (*Create, Replace, Update and Delete*). Após terminar as alterações, faria *commit* às mesmas. As chaves seriam então cifradas com a chave de κ, gerando o argumento `EncryptedKeys`, e são enviadas através do BFT-SMaRt para as réplicas através do pedido descrito anteriormente. Após cada réplica receber esse pedido, iria invocar o método `setKeys` com as chaves que recebeu. Se κ conseguisse decifrar `EncryptedKeys`, iria descartar todas as chaves que tivesse, e guardar as que recebesse.

5.7 Transferência de Estado

A política de autorização do TYPHON reside no *payload* e faz parte do estado a aplicação. Mas as chaves também fazem parte do estado e residem no *wormhole*. Como o protocolo de transferência de estado oferecido pelo BFT-SMaRt executa no *payload*, é necessário introduzir métodos em κ que devolvam as chaves contidas dentro deste e também para

sobrepôr novas chaves. No entanto, essas chaves têm que ser cifradas com a chave de κ, para não circularem em claro pela rede.

A função que seria necessário adicionar seria designada de `getKeys()`, e iria devolver uma estrutura de dados com todas as chaves contidas em κ, cifradas com a sua chave secreta. Esta função seria usada para enviar o estado às restantes réplicas. Se uma réplica necessitar de recuperar o estado, pode usar a função que foi descrita na secção anterior (`setKeys(EncryptedKeys)`).

5.8 Concretização

O TYPHON está escrito na linguagem de programação Java, e é compatível com a versão 1.6 ou superior da respectiva máquina virtual (JVM). Desta maneira, é possível usá-lo em qualquer máquina, independentemente da sua arquitectura e sistema operativo, desde que a máquina virtual do Java esteja presente. De seguida é apresentada a arquitectura do TYPHON e também a forma como este deve ser usado.

5.8.1 Arquitectura

A arquitectura do TYPHON está ilustrada na figura 5.3. A componente *TyphonAccessor* reside nos clientes e serviços que precisem de se autenticar mutuamente e também perante o Kerberos. A componente *KDCServer* reside em cada réplica, e tratasse da concretização do TYPHON. Essas duas componentes comunicam entre si por intermédio do BFT-SMaRt. Finalmente, a componente K tratasse do componente seguro, e deverá comunicar com *KDCServer* através de uma ligação dedicada e segura por construção[4].

Adicionalmente, o código do TYPHON está dividido nos seguintes *packages*:

```
typhon
typhon.exceptions
typhon.messages
typhon.messages.structures
typhon.messages.trsutedComponent
```

O *package* `typhon` possuí as principais classes do programa, que contém os pontos de entrada da execução do cliente e do servidor. Essas classes são:

- `TyphonAccessor`: classe a ser invocada pelos clientes. Esta classe faz uso do BFT-SMaRt para comunicar com as réplicas do sistema.
- `KDCServer`: classe que concretiza o TYPHON. É o ponto de arranque de cada réplica do sistema.

[4]Durante estre trabalho não se chegou a instalar o componente seguro numa máquina separada.

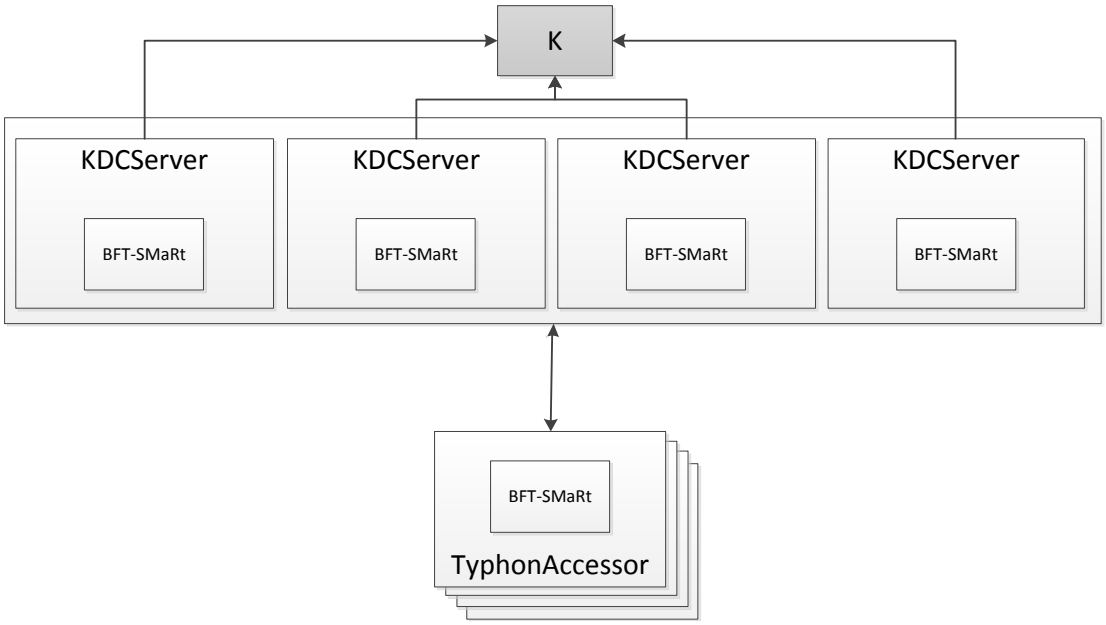


Figura 5.3: Arquitectura do TYPHON.

O *package* `typhon.exceptions` contém as excepções que podem ser retornadas pelos objectos da classe `TyphonAccessor`. O *package* `typhon.messages` contém as estruturas das mensagens trocadas entre um cliente e o KDC. Por sua vez, o *package* `typhon.messages.structures` contém as estruturas de dados que são incluídas nas mensagens. Finalmente, o *package* `typhon.messages.trustedComponent` contém todo o código que diz respeito ao componente seguro κ.

5.8.2 Modelo de uso

Para executar o TYPHON é necessário lançar tantos processos quantos forem necessários para o número f que se quiser considerar. Cada processo corresponde a uma réplica. É esperado que cada processo execute numa máquina diferente, mas é possível executá-los todos na mesma máquina.

Do lado do cliente não há nenhum processo para ser lançado em específico para realizar a autenticação. Em vez disso a ideia é concretizar uma aplicação que represente o cliente e faça os seguintes passos:

1. Instanciar um objecto da classe `TyphonAccessor`;
2. Use o método `public Ticket login(String ServiceName)` desse objecto para pedir um *ticket* para um serviço.

A partir do momento em que a aplicação do cliente obtêm o *ticket*, deverá ser ela realizar o resto da norma Kerberos por sua conta, tal como o serviço que quiser contac-

tar. Isto porque o TYPHON ainda não oferece uma concretização para esses passos de comunicação.

5.9 Considerações Finais

Neste capítulo foi apresentado o serviço TYPHON, um sistema de autenticação e autorização tolerante a intrusões que cumpre a norma do Kerberos v5 sem lhe fazer alterações. Foi descrito o modelo de sistema deste serviço e também o seu funcionamento, com alguma ênfase na parte do componente seguro. Também é explicado como o serviço suporta a funcionalidade de transferência de estado e como se faz a gestão das chaves.

O próximo capítulo é dedicado a discutir os resultados das avaliações feitas tanto ao TYPHON como ao protocolo de transferência de estado.

Capítulo 6

Avaliação

Nesta capítulo são apresentados os resultados de um conjunto de experiências efectuadas para avaliar o protocolo da transferência de estado descrito no capítulo 4 e o serviço TYPHON apresentado n capítulo 5.

É apresentada a latência do protocolo de transferência de estado e são comparados a latência e o débito de TYPHON e ApacheDS. O ApacheDS é um serviço de nomes e directorias, mantido pela Apache, que concretiza especificações como LDAP e Kerberos v5.

6.1 Ambiente

O ambiente de testes consistiu em um conjunto de 5 máquinas interligadas por um switch gigabit. Cada máquina era constituída por um processador *2 x Intel Xeon E5520 2.27GHz*, 32GB de memória RAM e 136GB de espaço em disco rígido. Todas as máquinas executavam o sistema operativo *GNU/Linux Debian Lenny*, versão 5.0.4 para arquitecturas de 64 bits. A versão da máquina virtual do Java que se usou foi a 1.6.0_12-b04 também para arquitecturas de 64 bits.

6.2 Protocolo de Transferência de Estado

Os resultados dos testes de latência do protocolo de transferência de estado são exibidos na tabela 6.1. Foram efectuados 1000 pedidos de estado numa réplica, tendo-se calculado a média da latência dos últimos 100 pedidos. Os primeiros 900 pedidos foram utilizados para forçar a máquina virtual Java a fazer uso do JIT (*Just In Time compiler*) para compilar para código máquina as instruções do BFT-SMaRt que tratam do pedido de estado em todas as réplicas.

Foram efectuadas 3 sessões de testes tal como descrito no parágrafo anterior. Em cada sessão, variou-se o tamanho do estado da aplicação. Foi realizada uma sessão com um estado de 10 kilobytes, uma sessão com um estado de 100 kilobytes, e outra com 1

megabyte. Os resultados mostram que a latência cresce em função do tamanho do estado, tal como era de esperar, mas não na mesma proporção: um aumento do tamanho do estado por um factor de 10, causa um aumento de 16% a 47% na latência.

Média da Latência (em milisegundos)		
10 KB	100 KB	1 MB
381.35	442.48	652.01

Tabela 6.1: Resultados dos testes de latência para a transferência de estado.

6.3 TYPHON

Para avaliar o TYPHON, este foi replicado em quatro máquinas ($n = 4$) de forma a tolerar uma falta ($f = 1$) em alguma dessas 4 máquinas. A restante máquina executou todos os processos cliente, que estiveram constantemente a pedir um TGT seguido de um *ticket* de serviço. O ApacheDS foi avaliado de forma semelhante, com apenas uma máquina a executar o serviço e tendo vários processos clientes noutra máquina a enviar-lhe pedidos da mesma forma que no caso do TYPHON.

Foram efectuados dois tipos de experiências. Na primeira experiência foi avaliada a latência fim-a-fim da geração de um TGT seguido da geração de um *ticket* de serviço. Na segunda experiência foi medida a quantidade de TGTs e *tickets* de serviço que se conseguem gerar por segundo. Estas experiências foram efectuadas em ambos os sistemas e os resultados são descritos na próxima secção.

Como já foi referido, o TYPHON usa a biblioteca BFT-SMaRt, que também está escrita em Java. O TYPHON tem 2943 linhas de código (não contando as 7557 linhas de código da biblioteca BFT-SMaRt) e apenas 8,7% são executadas pelo componente seguro.

6.3.1 Latência

Os resultados dos testes de latência de TYPHON e ApacheDS são exibidos na tabela 6.2. Para cada sistema, foram efectuados 20.000 pedidos de TGTs e *tickets* de serviço, tendo-se calculado a média da latência dos últimos 10.000 pedidos. Os primeiros 10.000 pedidos foram utilizados para forçar a máquina virtual Java a fazer uso do JIT para compilar para código máquina as instruções do ApacheDS e TYPHON que tratam dos pedidos de TGTs e *tickets* de serviço.

Os resultados da latência dos *tickets* de serviço mostram que o TYPHON é mais lento a realizar essa operação do que o ApacheDS. Por outro lado, podemos observar que o ApacheDS é bastante mais lento do que o TYPHON na geração de TGTs. Este resultado é inesperado, mas sabemos que enquanto o TYPHON guarda todos os seus dados em memória depois de ser inicializado, o ApacheDS vai buscá-los ao disco. Isto explica

Média da Latência (em milisegundos)			
TYPHON		ApacheDS	
TGT	*Ticket* de serviço	TGT	*Ticket* de serviço
4.6	5.7	26.3	3.0

Tabela 6.2: Resultados dos testes de latência para TYPHON e ApacheDS.

que a latência do ApacheDS seja maior que a do TYPHON[1]. A diferença entre os valores de latência do TYPHON para TGTs e *tickets* de serviço é explicada pelo facto dos TGTs não requererem a utilização de *ticket approvals* (TAs), enquanto que os *tickets* de serviço requerem. A latência média do envio e recepção de TAs corresponde portanto a 1.1 milisegundos, a diferença entre a latência dos *tickets* de serviço (5.7 milisegundos) e a latência dos TGTs (4.6 milisegundos).

6.3.2 Débito

Em termos de avaliação de débito, calculou-se o número de pedidos por segundo processados por ambos os sistemas em intervalos de 1000 pedidos. Aumentámos progressivamente o número de clientes de forma perceber o nível de saturação de cada sistema. Os resultados são apresentados na figura 6.1.

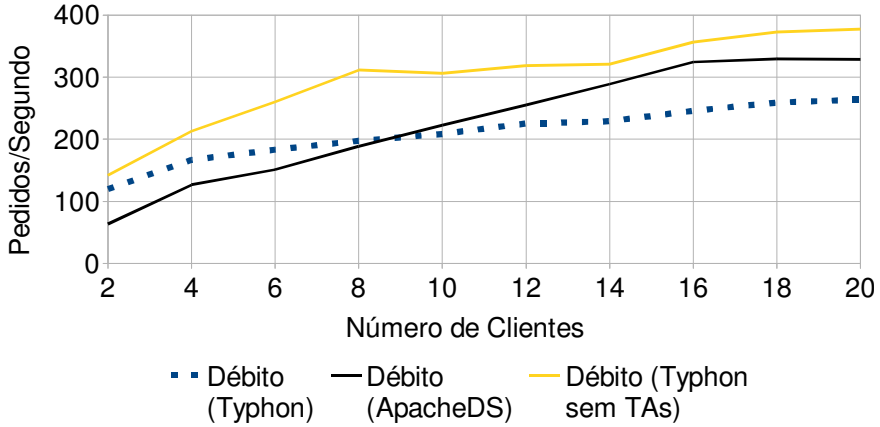
Os gráficos mostram um comportamento semelhante tanto no caso de TGTs como de *tickets* de serviço, e sugerem que quando se trata de poucos clientes a enviar em simultâneo, o TYPHON mostra melhor comportamento que o ApacheDS. Mas a partir de 10 clientes, o TYPHON estagna, e o ApacheDS consegue escalar melhor. No entanto, a partir de 20 clientes também o ApacheDS estagna, apesar de se manter mais eficiente que o TYPHON.

Finalmente, foi feito um último teste retirando o uso de TAs da concretização do TYPHON, tornando-o somente num serviço de autenticação. Foi observado que a remoção deste passo faz com que o sistema se torne mais eficiente a processar pedidos, chegando ao ponto de mostrar melhor desempenho que o ApacheDS.

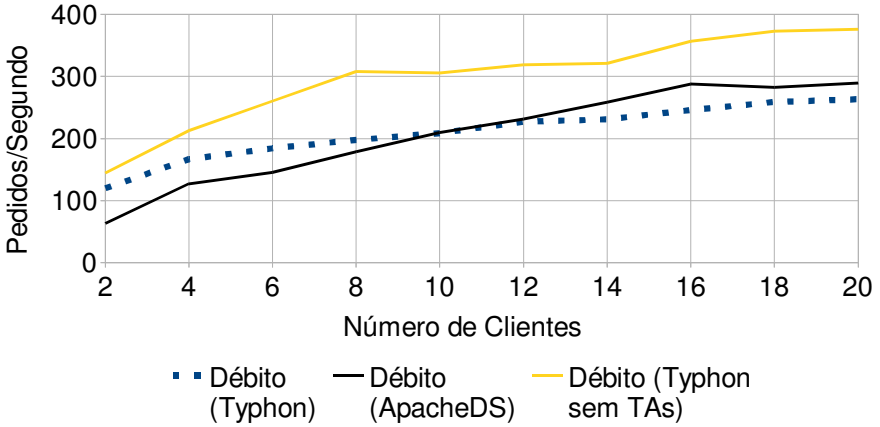
6.4 Considerações Finais

Neste capítulo foram discutidos os resultados da avaliação do protocolo de transferência de estado e do TYPHON. No caso do protocolo de transferência de estado foi avaliada a latência deste, e no caso do TYPHON foi avaliado para além da latência também o débito, ambos comparados com a mesma avaliação feita ao ApacheDS. Tanto o protocolo como o serviço mostram desempenhos aceitáveis.

[1]No entanto, após contactar a equipa de desenvolvimento do ApacheDS, sabe-se que este devia guardar essa informação em cache. Eles próprios fizeram os seus testes de latência e confirmaram estes resultados, mas ainda precisam de investigar o que está a acontecer para explicar este fenómeno.



(a) Débito da geração de TGTs.



(b) Débito da geração de *tickets* de serviço.

Figura 6.1: Resultados dos testes de débito para os serviços ApacheDS e TYPHON (com e sem *ticket approvals*).

Capítulo 7

Conclusões e Trabalho Futuro

7.1 Conclusões

Durante o percurso deste trabalho foi estudada a biblioteca BFT-SMaRt, que oferece uma concretização da técnica de replicação de máquina de estados. Durante este estudo, foram adicionados comentários ao seu código fonte. Neste período houve a oportunidade do autor deste relatório obter contacto com código fonte em que nenhuma parte tinha sido escrita por ele e observar como um projecto sério é feito.

Terminado o estudo do BFT-SMaRt, foi iniciada a fase de concretização do protocolo de transferência de estado. Esta fase também se tornou um período de aprendizagem do ponto de vista de modificar código consideravelmente mais complexo que aquilo que o autor estava habituado. O protocolo foi difícil de concretizar, mas com essa dificuldade também veio experiência a lidar com projectos complexos. Durante esta fase também foi adicionada um novo programa de demonstração da biblioteca, que oferece melhor percepção da ordenação de mensagens.

Terminada a fase de concretização do protocolo de transferência de estado, iniciou-se a concepção de um serviço de autenticação que concretiza a norma Kerberos v5, com a particularidade de ser tolerante a intrusões. O Kerberos v5 é uma norma que especifica como clientes e serviços se devem autenticar mutuamente por intermédio de uma entidade centralizada que guarda as chaves de todos os participantes. O problema reside na possibilidade dessa entidade centralizada falhar, quer seja por paragem, arbitrariamente ou até por intrusão. Se isso acontecer não é possível fazer mais autenticações de clientes ou serviços.

Durante esta fase foi necessário analisar com atenção a norma do Kerberos v5, em particular as interacções entre um cliente e o AS e o TGS, porque foi preciso perceber quais as alturas críticas em que era necessário o AS e TGS acederem às chaves e/ou fazerem operações com elas. Isto foi necessário para perceber que funções é que o componente seguro deveria oferecer. Quando este trabalho foi iniciado - todo o trabalho, não somente esta fase - não se tinha a certeza se o Kerberos seria apenas um "passador de bolas" para

o componente seguro, ou se era possível delegar apenas uma parte do código ao componente seguro. Veio-se a concluir que de facto é possível fazer um componente seguro minimalista e deixar a maior parte da concretização fora do componente.

Note que a norma Kerberos foi criada com um assumpção muito forte: a máquina e o processo que concretiza a norma nunca é comprometido,e o único perigo está na rede, que pode ser escutada. Mas tal assumpção não pode ser dada como garantida na prática.

Nesta fase de trabalho nasceu o serviço TYPHON tal como apresentado neste relatório. Este é um serviço de autenticação e autorização que segue a especificação do Kerberos v5 ao mesmo tempo que introduz na sua concretização mecanismos para tolerar intrusões. Por um lado, usa a técnica de replicação da máquina de estados para oferecer tolerância a faltas arbitrárias. Por outro lado, faz uso de componentes seguros para guardar as chaves dos clientes e dos serviços de forma a assegurar que estes não são expostos na eventualidade de intrusões.

Terminada a concepção do TYPHON, deu-se inicio à fase de avaliação do trabalho desenvolvido. Esta fase também foi proveitosa em aprendizagem. O autor aprendeu a importância de realizar testes de latência e de débito às aplicações, porque podem revelar se a aplicação tem um desempenho aceitável em situações reais. Não foi muito fácil realizar os testes ao ApacheDS, pois foi necessário perceber como alterar o seu código fonte e de seguida compilá-lo.

Os resultados da avaliação do serviço TYPHON mostram que o seu desempenho é similar ao do ApacheDS, um serviço de autenticação e autorização não replicado que concretiza a norma Kerberos v5. Os resultados da avaliação do protocolo de transferência de estado mostram que o protocolo também tem uma latência aceitável, face ao tamanho do estado usado nos testes.

Tendo em conta todo o trabalho realizado e os resultados obtidos, percebeu-se que existia material para publicar um artigo científico. Esse artigo foi escrito sob o título 'Typhon: Um Serviço de Autenticação e Autorização Tolerante a Intrusões' e aceite na conferência Inforum 2010. Com a escrita deste artigo também foi adquirida experiência a redigir textos científicos, mais concretamente na parte de produzir texto objectivo e sucinto.

Também a escrita do presente relatório implicou uma curva de aprendizagem, porque foi necessário adquirir competências no uso do processador de texto latex. Esta curva de aprendizagem de início fez com que a redacção deste relatório levasse mais tempo, mas mais tarde revelou-se essencial para gerir um documento deste tamanho.

Finalmente, tanto quanto é do conhecimento do autor deste relatório e dos seus orientadores, este trabalho é o primeiro a apresentar um serviço de autenticação e autorização tolerante a intrusões que cumpre a norma Kerberos v5. Mostrar que tal é possível, foi a contribuição essencial deste trabalho.

7.2 Trabalho Futuro

A biblioteca BFT-SMaRt continua em desenvolvimento, e serão feitas modificações à sua arquitectura. O protocolo de transferência de estado que foi concretizado neste trabalho irá ser aproveitado para a concretização do conceito de vistas. Também será aproveitado quando o BFT-SMaRt for integrado no rejuvenescimento de réplicas tal como descrito em [18].

O TYPHON também continuará a ser desenvolvido. O próximo passo será torná-lo numa concretização total da norma Kerberos, tal como descrita no RFC 4120. O componente seguro κ neste momento limita-se a código reunido numa classe, mas será portado para um processo separado, adequado a executar numa máquina especial, que supostamente será minimalista e segura por construção. Adicionalmente, também se pretende adicionar ao componente seguro funções para lidar com o protocolo de transferência de estado oferecido agora pelo BFT-SMaRt. Apesar de teoricamente ser possível integrar o TYPHON nesse protocolo, não houve necessidade de o fazer durante este trabalho. Finalmente, também será desenvolvida uma forma de fazer a gestão de chaves no TYPHON, para que seja possível adicionar, modificar e remover clientes e serviços da sua base de dados.

Referências

[1] ApacheDS, http://directory.apache.org/apacheds/1.5/.

[2] BFT-SMaRt, http://code.google.com/p/bft-smart/.

[3] E. Alchieri. *Uma Infra-Estrutura com Segurança de Funcionamento para Coordenação de Serviços Web Cooperantes*. PhD thesis, Universidade Federal de Santa Catarina, 2007. Ver capitulo 4: Difusão atómica baseada no Algoritmo Paxos Bizantino.

[4] S. R. Ames, M. Gasser, and R. R. Schell. Security kernel design and implementation: An introduction. *Computer*, 16(7), July 1983.

[5] Alysson Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. The CRUTIAL way of critical infrastructure protection. *IEEE Security & Privacy*, 6(6):44–51, Nov-Dec 2008.

[6] Miguel Castro and Barbara Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, November 2002.

[7] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Proc. of the Fourth European Dependable Computing Conference*, Toulouse, France, October 2002.

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

[9] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*, chapter Timing Assumptions. Springer, 2005.

[10] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, May 1994.

[11] Lamport, Shostak, and Pease. The byzantine generals problem. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 4, 1982.

[12] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst*, 16(2):133–169, 1998.

[13] Lampson, Abadi, Burrows, and Wobber. Authentication in distributed systems: Theory and practice. *ACMTCS: ACM Transactions on Computer Systems*, 10, 1992.

[14] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[15] N. F. Neves, M. Correia, and P. Veríssimo. Wormhole-aware Byzantine protocols. In *2nd Bertinoro Workshop on Future Directions in Distributed Computing: Survivability - Obstacles and Solutions*, June 2004.

[16] Michael K. Reiter, Matthew K. Franklin, John B. Lacy, and Rebecca N. Wright. The omega key management service. In *ACM Conference on Computer and Communications Security*, pages 38–47, 1996.

[17] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *CSURV: Computing Surveys*, 22, December 1990.

[18] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2010.

[19] Paulo Verissimo. Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

[20] Paulo Veríssimo, Miguel Correia, Nuno Ferreira Neves, and Paulo Sousa. Intrusion-resilient middleware design and validation. *Information Assurance, Security and Privacy Services*, 4:615–678, 2009.

[21] T. Yu, C. Neuman, S. Hartman, and K. Raeburn. The kerberos network authentication service (V5). RFC 4120, USC-ISI, July 2005.

[22] Lidong Zhou, Fred Schneider, and Robbert Van Rennesse. COCA: A secure distributed online certification authority. *ACM Transactions Computer Systems*, 20(4):329–368, November 2002.

[23] Piotr Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge, Computer Laboratory, June 2004.