UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

Evolução da Biblioteca para Replicação Tolerante a Faltas Bizantinas BFT-SMaRt

Bruno Filipe Cabo Verde de Branco e Brito

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

2011

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

Evolução da Biblioteca para Replicação Tolerante a Faltas Bizantinas BFT-SMaRt

Bruno Filipe Cabo Verde de Branco e Brito

DISSERTAÇÃO

Projecto orientado pelo Prof. Doutor Alysson Neves Bessani
e co-orientado pelo Prof. Doutor Marcelo Pasin

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

2011

Agradecimentos

Quero antes de mais agradecer ao Professor Doutor Alysson Bessani e ao Professor Doutor Marcelo Pasin, por toda a ajuda, apoio e por tudo o que me ensinaram ao longo deste ano. Agradeço também ao João Sousa, que me ajudou várias vezes ao longo do estágio.

Quero agradecer ao Bruno Correia, David Silva, Rui Almeida, Hugo Silva, João Casais, Pedro Martins, Rui Ferreira e à Sara Patrocínio pela amizade e pelas inúmeras gargalhadas e momentos inesquecíveis ao longo da Licenciatura e do Mestrado (pelo meio lá íamos fazendo uns projectos...).

Por fim, não sendo de todo o menos importante, agradeço à minha mãe e à minha avó pelo amor, apoio e paciência ao longo de todos estes anos.

À minha mãe e à minha avó.

Resumo

Numa época em que os sistemas informáticos são largamente utilizados, é natural que exista uma preocupação em garantir que um sistema informático seja resistente, tanto em termos de segurança como de tolerância a faltas. Devido a estes motivos, são criadas bibliotecas para tolerância a faltas bizantinas (ou arbitrárias), tal como a biblioteca BFT-SMaRt desenvolvida no Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa.

Um dos principais objectivos da biblioteca BFT-SMaRt é o alto desempenho, logo é importante que exista um modo simples e eficiente de obter medidas sobre o desempenho da biblioteca, tal como um mecanismo de monitorização eficiente.

Esta dissertação apresenta um módulo de monitorização e configuração para a biblioteca BFT-SMaRt, tal como um processo de avaliação de desempenho automatizada de modo a ser simples e eficiente. Serão também apresentados os resultados das sequências de testes efectuadas à biblioteca, de modo a avaliar o seu desempenho.

Palavras-chave: Tolerância a faltas bizantinas, Java Management eXtensions (JMX), monitorização, configuração, avaliação de desempenho

Abstract

In a time where computer-based information systems are widely used, it's natural that there is a concern in assuring that a given information system is resilient in terms of security and fault tolerance. It is due to these reasons that Byzantine fault tolerant replication libraries are designed and created, like the BFT-SMaRt library developed in the Department of Informatics of Lisbon's Faculdade de Ciências da Universidade de Lisboa.

One of the main objectives in the design of the BFT-SMaRt library is high performance, so it is important that there is a simple and efficient method of obtaining measurements regarding the library's performance, including an efficient monitorization mechanism.

This dissertation presents a monitorization and configuration module added to the BFT-SMaRt library, and also an automated performance evaluation process, designed to be simple and efficient. Finally, it presents the results of a set of tests to which we subjected the library as to evaluate its performance.

Keywords: Byzantine fault tolerance, Java Management eXtensions (JMX), monitorization, configuration, performance evaluation

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Vivemos numa época em que a utilização de sistemas informáticos é comum para praticamente todos os aspectos da nossa sociedade. Como tal, ao aumentar a sua utilização, também aumentam os riscos de ataques[1] a estes mesmos sistemas e aumenta a gravidade das consequências de ataques ou erros de *software* e *hardware*. Paralelamente, a ocorrência de erros de *software* também está a aumentar devido à crescente complexidade das aplicações. Por outro lado, para sistemas considerados críticos, a tolerância a este tipo de erros deveria ser automática, já que sistemas deste tipo usualmente não podem estar indisponíveis, nem funcionar incorrectamente.

É pelas razões indicadas anteriormente que são investigados mecanismos de tolerância a faltas bizantinas (termo derivado do problema de acordo denominado Problema dos Generais Bizantinos [10]). Uma falta bizantina (também chamada de falta arbitrária) poderá ser uma falta por omissão, ou seja, por exemplo, uma máquina que sofreu um *crash*, uma mensagem que não foi entregue pela rede, ou poderá também ser uma falta por valor, ou seja, o processamento de uma mensagem devolve um valor errado ou algo corrompeu um ficheiro no disco, que irá resultar na leitura de dados incorrectos.

Na última década, existiu (e continua a existir) muita investigação relacionada com tolerância a faltas bizantinas e, consequentemente, foram propostos muitos protocolos para replicação tolerante a este tipo de faltas (BFT[2]). Estes protocolos habitualmente consistem em máquinas de estados replicadas, em que a consistência do estado das réplicas é mantida através de algoritmos de acordo BFT.

1.1 Motivação

A biblioteca para concretização de máquinas de estados replicadas tolerantes a faltas bizantinas BFT-SMaRt [3], desenvolvida no âmbito do grupo Navigators [11] do Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa, é uma

[1] Um ataque informático é um método pelo qual um indivíduo, utilizando um sistema informático, tenta controlar ou afectar o funcionamento de outro sistema informático.

[2] BFT - *Byzantine Fault Tolerant*.

biblioteca BFT desenhada com ênfase na simplicidade, robustez e alto desempenho. Embora a biblioteca ainda se encontre em evolução, acreditamos que já tem um desempenho bastante competitivo quando comparada com outras concretizações BFT de referência.

Visto que um dos principais focos da biblioteca BFT-SMaRt é o desempenho, é importante ter acesso a um modo simples e eficiente de medir o desempenho da biblioteca. Um processo de testes que não esteja estruturado e optimizado pode resultar em rondas de testes demasiadamente repetitivas e lentas, tornando-se extremamente difícil obter uma variedade de resultados suficiente para daí se poderem retirar conclusões, em tempo útil. Outro aspecto importante é a facilidade de monitorização e configuração das réplicas no sistema. A biblioteca BFT-SMaRt não possui um mecanismo que permita alterar as configurações do sistema em tempo real, o que implica terminar a execução de todas as máquinas (clientes e servidores) no sistema, sempre que é necessária uma alteração nos seus parâmetros. Num sistema informático crítico isto é extremamente indesejável, devido à indisponibilidade introduzida.

1.2 Objectivos

Os objectivos propostos para este estágio foram a realização de sequências de testes que permitissem avaliar o desempenho da biblioteca BFT-SMaRt, a automatização do processo de testes para a biblioteca, juntamente com a concretização de um módulo que possibilitasse monitorizar e configurar a biblioteca.

Inicialmente, estava planeada a concretização de um mecanismo que adaptasse dinamicamente os parâmetros de configuração da biblioteca. No entanto, vimos que o mesmo não era necessário, sendo substituído pelo módulo de monitorização e configuração.

1.2.1 Planeamento Inicial

O planeamento inicial para este estágio foi o seguinte:

- Inicialmente, estudar a biblioteca BFT-SMaRt e artigos relacionados.
- Iniciar um processo de testes de desempenho, de modo a medir o desempenho da biblioteca BFT-SMaRt na sua versão existente.
- Analisar e concretizar um modo de automatização dos testes para o BFT-SMaRt, de modo a facilitar o processo de testes.
- Elaborar um relatório preliminar.
- Recolher medidas, identificando e analisando os parâmetros mais importantes em relação ao desempenho do BFT-SMaRt.

- Concretizar um módulo de monitorização e configuração para a biblioteca.
- Elaborar esta dissertação final e redigir um artigo [14] sobre este trabalho.

1.3 Contribuições

Ao longo do estágio descrito nesta dissertação, foram desenvolvidos *scripts* de modo a auxiliar e automatizar tanto quanto possível o processo de testes da biblioteca BFT-SMaRt. Utilizando este mecanismo de automatização, foram realizados diversos testes de modo a medir o desempenho da biblioteca e verificar quais os parâmetros que mais afectam o seu funcionamento. Também foram realizados testes de comparação com outras bibliotecas do mesmo tipo, de modo a confirmar a competitividade do BFT-SMaRt em relação a outras bibliotecas de referência neste campo.

Foi também concretizado um módulo de monitorização e configuração da biblioteca, recorrendo ao uso da tecnologia JMX [12] existente no Java, linguagem em que foi desenvolvida a biblioteca BFT-SMaRt. Este módulo permite a recolha de variadas medidas (utilizadas no processo de testes) e a alteração de determinados parâmetros de configuração dos servidores em tempo real, utilizando uma consola padronizada, também existente no Java, a *JConsole*.

Foi também redigido um artigo [14], intitulado *Desempenho e Escalabilidade de uma Biblioteca de Replicação de Máquina de Estados Tolerante a Faltas Bizantinas*, no qual as medidas recolhidas ao longo de todo este processo tiveram um papel importante.

1.4 Estrutura do documento

Este documento está estruturado do seguinte modo:

- Capítulo 2: Replicação de Máquinas de Estados - Neste capítulo é apresentada a biblioteca de replicação BFT-SMaRt, o seu modo de funcionamento e os seus componentes.
- Capítulo 3: Monitorização e Configuração em Tempo de Execução das Máquinas de Estados Replicadas - Neste capítulo é apresentado o módulo de monitorização e configuração que foi concretizado para a biblioteca BFT-SMaRt.
- Capítulo 4: Ambiente de Testes para a Biblioteca BFT-SMaRt - Neste capítulo é detalhado o ambiente no qual foram realizados os testes à biblioteca de replicação BFT-SMaRt e o processo de automatização que foi necessário desenvolver.
- Capítulo 5: Estudo do Desempenho da Biblioteca BFT-SMaRt - Neste capítulo são apresentados os resultados dos testes efectuados à biblioteca de replicação BFT-SMaRt.

Capítulo 2

Replicação de Máquinas de Estados: BFT-SMaRt

Um dos protocolos de referência no campo da tolerância a faltas bizantinas é o PBFT [4], devido a ter sido o primeiro protocolo BFT publicado que oferecia alto desempenho, passando posteriormente a existir vários protocolos baseados nele. Os protocolos deste tipo trabalham sobre algumas premissas, tais como: a existência de um sistema eventualmente síncrono [6], ou seja, o sistema é assíncrono mas existem momentos no tempo em que o sistema se comporta como um sistema síncrono; considera-se para o sistema n o número de réplicas (servidores) existentes e f o número máximo de máquinas maliciosas, assumindo também que $n \geq 3f + 1$.

A biblioteca para concretização de máquinas de estados replicadas tolerantes a faltas bizantinas BFT-SMaRt [3] permite, de um modo simples, a um utilizador acrescentar tolerância a faltas bizantinas a uma aplicação baseada no paradigma cliente-servidor. A biblioteca possibilita também ao utilizador focar a sua atenção no desenvolvimento das funcionalidades da aplicação, sabendo que o mecanismo de tolerância a faltas bizantinas é da responsabilidade da biblioteca.

A biblioteca BFT-SMaRt foi criada com os seguintes princípios de desenho em mente:

- Implementação em Java: A linguagem de programação Java foi escolhida por razões de segurança, portabilidade, facilidade de programação e manutenção de código. Foi tido em atenção o desafio de criar uma biblioteca BFT (*Byzantine Fault Tolerant*) em Java que tivesse alto desempenho, já que o Java é considerado uma linguagem bastante menos eficiente que o C (que é utilizado para implementar outros algoritmos BFT, como por exemplo o PBFT [4]).

- Modularidade: Existem algoritmos de máquinas de estados replicadas BFT de alto desempenho implementados com uma estrutura monolítica (tal como o PBFT), de modo a integrar os diversos mecanismos utilizados para concretizar o algoritmo, tais como consenso, difusão de ordem total, *checkpointing*, transferência de estados, eleição de líder, entre outros. O BFT-SMaRt foi desenhado de modo a garantir

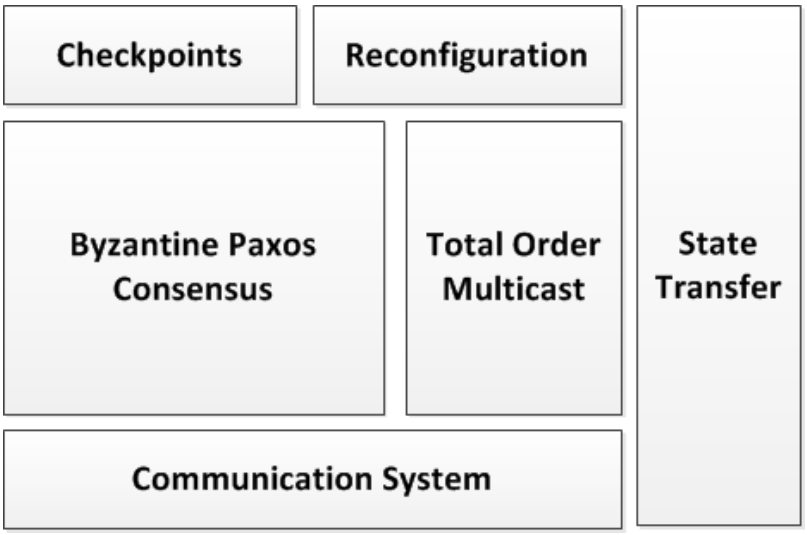


Figura 2.1: Módulos da biblioteca BFT-SMaRt

separação entre os vários módulos, conseguindo modularizar, entre outros, o consenso *Paxos* [9] tolerante a faltas bizantinas (*Paxos at War* [15]), a difusão com ordem total e o sistema de transferência de estados e *checkpointing*.

- Não adicionar complexidade: O BFT-SMaRt foi concretizado evitando utilizar optimizações que adicionem complexidade, que são habitualmente utilizadas noutras concretizações de algoritmos BFT e, ainda assim, mantém um bom nível de desempenho.

2.0.1 Arquitectura

Iremos, em seguida, apresentar a arquitectura da biblioteca BFT-SMaRt e indicar a função de cada um dos módulos que a compõem. Esta é apenas uma breve introdução ao funcionamento destes módulos, já que a maioria deles foge ao escopo desta dissertação. A figura 2.1 ilustra os diferentes módulos existentes na biblioteca BFT-SMaRt.

Communication System. Este módulo é responsável pela comunicação entre as réplicas e os clientes, para além da comunicação entre as réplicas. A comunicação cliente - réplica é feita utilizando a biblioteca Netty [5], enquanto que a comunicação entre réplicas é feita utilizando a API de *sockets* disponibilizada no Java. Para mais detalhes, veja a secção 2.4.

State Transfer. Este módulo é responsável pela transferência de estado entre as réplicas, caso uma dada réplica se atrase demasiado na sua execução ou recupere de uma falha. Para mais detalhes sobre este mecanismo, veja a secção 2.3.

Byzantine Paxos Consensus. Este módulo consiste numa implementação do algoritmo *Paxos at War* [15], que irá decidir uma ordenação das mensagens a executar, sendo portanto um mecanismo crucial para a replicação de máquinas de estados e para a primitiva de difusão de ordem total. Este módulo encontra-se detalhado nas secções 2.1 e 2.4.

Total Order Multicast. A primitiva de difusão de ordem total deriva da ordenação das mensagens por parte do consenso tolerante a faltas bizantinas, e permite que as mensagens tenham uma ordem global partilhada por todas as máquinas do sistema. Para mais detalhes, veja as secções 2.2 e 2.4.

Reconfiguration. O módulo de reconfiguração contém mecanismos que permitem adicionar ou remover réplicas do sistema. As réplicas são adicionadas e removidas por um cliente com permissões para reconfigurar o sistema (chamado TTP[1]). Isto faz com que, embora o número de réplicas no sistema tenha que obedecer à regra $n \geq 3f + 1$, os números identificadores das réplicas não têm que ser necessariamente sequenciais.[2]

Checkpoints. Este módulo contém o mecanismo de *checkpointing* do estado das réplicas, que permite que as réplicas reconheçam a evolução do seu estado e possibilita a ordenação dessa mesma evolução, sendo utilizada no mecanismo de transferência de estado. Para mais detalhes sobre este mecanismo, veja a secção 2.3.

2.1 Consenso tolerante a faltas arbitrárias

O BFT-SMaRt utiliza o protocolo de consenso tolerante a faltas bizantinas *Paxos at War* [15] de modo a implementar a replicação de máquinas de estados. Esta variante do consenso *Paxos* [9] foi escolhida devido a, na altura, ser a versão mais documentada e a mais eficiente do consenso tolerante a faltas arbitrárias.

Como indicámos no início do capítulo, o algoritmo implementado assume as seguintes condições: a existência de um sistema eventualmente síncrono [6], ou seja, o sistema é assíncrono mas existem momentos no tempo em que o sistema se comporta como um sistema síncrono; considera-se para o sistema *n* o número de réplicas (servidores) existentes no sistema e f o número máximo de máquinas maliciosas no sistema. Também é assumido que $n \geq 3f + 1$.

[1] TTP - *Trusted Third Party.*

[2] É habitual, especialmente no início da execução, os números identificadores das réplicas serem os números de 0 até $n - 1$.

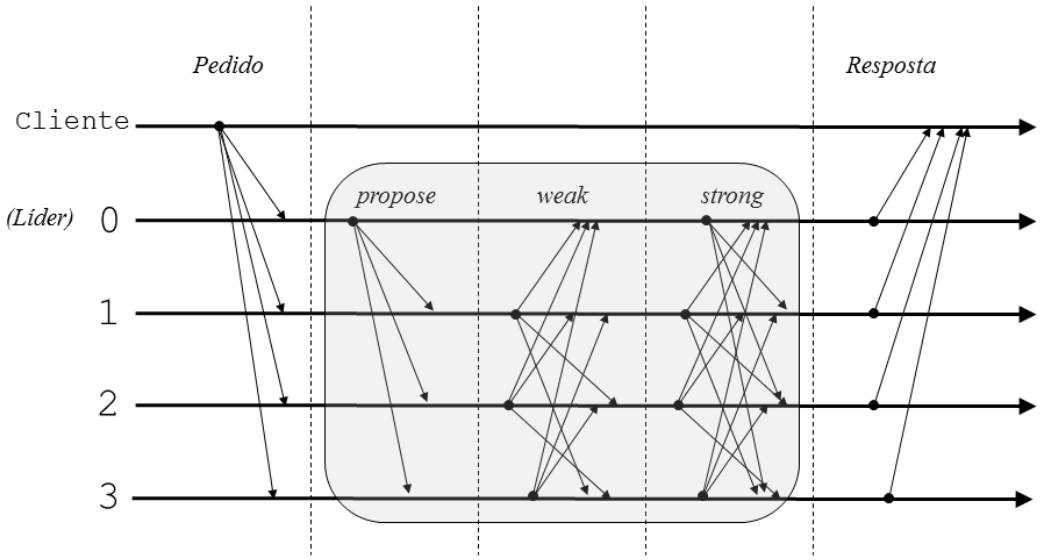


Figura 2.2: Execução de um *round* do consenso *Paxos at War* na secção sombreada.

2.1.1 *Paxos at War*

O *Paxos at War* foi desenvolvido por Piotr Zieliński de modo a melhorar a eficiência dos consensos tolerantes a faltas arbitrárias existentes, ou seja, minimizar o número de mensagens trocadas até se chegar a uma decisão. Até então, o consenso BFT mais eficiente era o utilizado no PBFT [4], em que eram trocadas 3 mensagens em cada *round*. O *Paxos at War* consegue optimizar este número para 2 mensagens nos casos em que não ocorrem faltas, continuando com 3 mensagens caso contrário.

2.1.2 Execução de um *round*

A figura 2.2 ilustra a execução de um *round* do consenso *Paxos at War*. O líder para o *round* começa por propôr um valor para o consenso utilizando uma mensagem *PROPOSE*, que contém um lote[3] de mensagens escolhidas de modo justo[4] para ordenar. Todas as réplicas recebem esta mensagem, verificam se quem a enviou é de facto o líder e se o valor proposto é válido.[5] Se tudo isto se verificar, cada réplica enviará uma mensagem *WEAK* a todas as outras, indicando que esta proposta tem um nível de aceitação fraca. Ao

[3]É um conjunto de mensagens proposto a ordenação. As mensagens são ordenadas por lotes, ao invés de individualmente, por razões de eficiência (por exemplo, é mais eficiente executar uma instância do consenso a ordenar 100 mensagens do que 100 instâncias do consenso com uma mensagem cada).

[4]É retirada uma mensagem de cada fila de mensagens dos clientes e este processo é repetido até preencher o lote de mensagens.

[5]No primeiro *round*, o valor proposto é automaticamente válido. Nos *rounds* seguintes, terá que ser enviada uma prova de validade juntamente com o valor proposto.

receber mais de $\frac{n+f}{2}$ mensagens *WEAK* iguais, ou seja, uma maioria das réplicas correctas aceitou fracamente o valor proposto, a réplica irá enviar uma mensagem *STRONG*, o que indica que a proposta tem um nível de aceitação forte. De novo, ao receber mais de $\frac{n+f}{2}$ mensagens *STRONG* iguais, indicando que uma maioria das réplicas aceitou fortemente o valor proposto, o valor proposto é utilizado como a decisão do consenso. Cada réplica constrói então uma lista ordenada de mensagens, construída a partir do lote de mensagens recebido no *PROPOSE*, e entrega essa mesma lista à aplicação.

É importante salientar que a mensagem *PROPOSE* contém o lote de mensagens a ordenar, mas as mensagens *WEAK* e *STRONG* contêm apenas um resumo criptográfico (*hash*) desse mesmo lote. Isto é feito para diminuir o tráfego de dados gerado pelo algoritmo.

2.1.3 Ocorrendo uma falha

Se um *round* não progredir (um exemplo disso seria o líder ser malicioso e propôr valores diferentes a réplicas diferentes ou se uma réplica simplesmente permanecer num *round* durante mais do que um determinado prazo pré-estabelecido), o *round* é congelado[6] e a réplica envia uma mensagem *FREEZE* às outras réplicas. Todas as réplicas que receberem f mensagens *FREEZE* irão congelar o round (se já não o tiverem feito) e vão enviar uma mensagem assinada *COLLECT* ao líder do próximo *round*, que todos conhecem, visto que é calculado de forma determinística.

A mensagem *COLLECT* especifica o estado da réplica e permite ao próximo líder escolher um valor a propôr no próximo *round* que não entre em contradição com valores decididos previamente.

2.2 Difusão com Ordem Total

Já vimos que a primitiva de Difusão com Ordem Total deriva da ordenação das mensagens feita pelo consenso tolerante a faltas arbitrárias. Foram, no entanto, tidos em atenção os seguintes factores que tornam não-trivial a concretização desta primitiva:

- Um líder pode propôr qualquer valor para um consenso: Foi necessário desenvolver um mecanismo que determine mensagens adulteradas, de modo a elas não serem aceites como valores possíveis. Este mecanismo consiste simplesmente em fazer com que os clientes assinem (utilizando criptografia assimétrica[7]) ou autentiquem (utilizando MACs[8]) as mensagens.

[6] Um *round* é considerado congelado se for congelado por todas as réplicas correctas.

[7] Criptografia que utiliza uma chave privada e uma chave pública.

[8] MAC: *Message Authentication Code*, um código gerado por uma função *hash* (por exemplo, MD5) a partir de uma dada mensagem.

- O líder é malicioso e não propõe (ou altera) mensagens de determinados clientes: Para esta situação, foram criados prazos associados às mensagens pendentes (ou seja, que estão à espera de ser ordenadas) recebidas dos clientes. As réplicas não-líder atribuem um prazo para o líder propôr a ordenação das mensagens. Caso este prazo não seja atingido, o consenso é congelado, forçando uma mudança de líder.

2.3 Transferência de Estado

De modo a permitir que as réplicas possam voltar ao sistema caso falhem, sem ser necessário reiniciar todo o sistema, foi implementado um mecanismo de transferência de estado entre réplicas. Quando uma réplica reinicia (ou verifica que está atrasada em relação às outras, possivelmente devido a desconexão ou a uma ligação de rede lenta), é desencadeada uma transferência de estado. Isto consiste em a réplica enviar uma mensagem *STATEREQUEST* às outras réplicas a pedir o estado resultante da execução das mensagens decididas nos consensos desde 0 até u-1 (sendo u o número da primeira decisão de consenso recebida pela réplica desde que está a recuperar). A réplica então espera por f+1 (sendo f o número máximo de falhas suportadas pelo sistema) mensagens *STATE* iguais, que é então utilizada para recuperar o estado da réplica.

De modo a que as réplicas possam responder às mensagens de *STATEREQUEST*, as operações executadas para cada consenso (nomeadamente, o valor da decisão do consenso) é guardado num registo estável e, de modo a limitar o tamanho desse mesmo registo, são efectuados *checkpoints* do estado a cada c execuções do algoritmo de consenso.

2.4 Funcionamento

Nesta secção, o objectivo é dar uma visão de alto nível sobre o funcionamento do BFT-SMaRt. Será dado o exemplo de uma mensagem enviada por um cliente chegar a uma réplica a correr o BFT-SMaRt e veremos as várias etapas gerais de execução ao longo do algoritmo. A figura 2.3 ilustra a arquitectura geral de uma réplica.

Chegada da mensagem às réplicas. As mensagens enviadas pelos clientes são recebidas pelas *threads* existentes numa *thread pool* criada para atender pedidos dos clientes. A comunicação entre clientes e servidores foi concretizada recorrendo à biblioteca Netty [5], enquanto que a comunicação entre réplicas é feita utilizando a API de *sockets* disponibilizada no Java. Foram implementados estes dois modos de comunicação distintos por razões de optimização, já que a biblioteca Netty está optimizada para a utilização de um grande número de ligações, o que a torna adequada para a comunicação feita pelos clientes, enquanto que a comunicação entre réplicas foi concretizada de modo a ser optimizada para um número reduzido de ligações. O líder das réplicas vai então preparar um

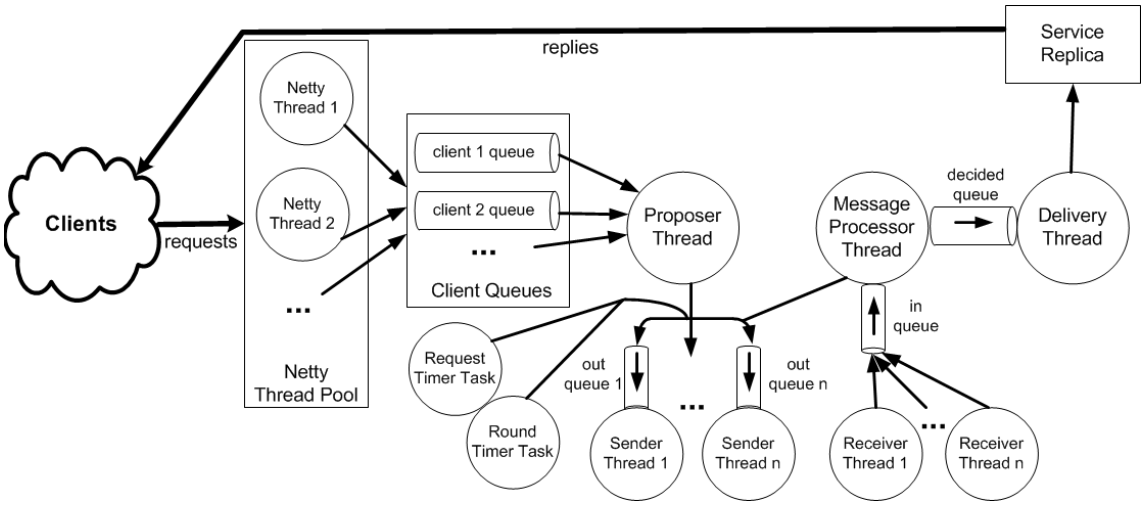


Figura 2.3: Arquitectura de uma réplica.

lote de mensagens para ordenar e propôr um consenso para decidir essa ordem. Existe, no entanto, um mecanismo que permite a uma mensagem ser executada apenas localmente e não ser incluída nos lotes de mensagens, o que torna o seu processamento bastante mais rápido. Isto poderá ser utilizado se o pedido não causar alterações no estado do sistema, por exemplo, uma operação de leitura local. Neste caso, o cliente vai esperar $2f + 1$ respostas iguais. Se não as receber, quer então dizer que existe um problema de consistência do estado do sistema no momento em que este pedido foi processado. Assim, o cliente irá reenviar o pedido, que será obrigatoriamente incluido no lote de mensagens, de modo a ser possível obter um número de respostas suficiente de modo a resolver o problema de consistência.

Consenso e Difusão com Ordem Total. O líder envia o lote de mensagens que preparou às outras réplicas. As réplicas executam um *round* do consenso para determinar a ordem das mensagens, em que o líder propõe a ordem do lote. Como foi referido anteriormente (ver a secção 2.1.2), um *round* executa uma série de passos de modo a avaliar uma aceitação forte da proposta. Se cada réplica receber um número suficiente de mensagens a indicar a aceitação forte então a proposta foi aceite, o *round* do consenso acaba para ser iniciado um novo *round* com um novo líder, que será a próxima réplica na sequência de IDs existentes (por exemplo, tendo uma identificação numérica e sequencial para os servidores, se o líder actual tiver o identificador 1, o próximo será a réplica com o identificador 2). O acordo acerca da ordem das mensagens faz com que exista uma primitiva de difusão com ordem total [8] entre as réplicas, já que todas as réplicas do BFT-SMaRt vão executar o mesmo conjunto de mensagens, pela mesma ordem.

Processamento de mensagens. Enquanto o algoritmo descrito anteriormente está a ser executado, as réplicas do BFT-SMaRt continuam a receber mensagens enviadas pelos clientes. Essas mensagens são armazenadas em filas (uma para cada cliente) e é dessas filas que o líder retira mensagens para preparar o lote de mensagens.

Para as mensagens recebidas das outras réplicas, existe uma *thread*, chamada de *Message Processor Thread* (ver a figura 2.3), que trata dessas mensagens. Esta *thread* descarta mensagens que já são irrelevantes para a execução (por exemplo, uma mensagem de um consenso já decidido que, por alguma razão, demorou tempo demais a chegar) e armazena mensagens relevantes a consensos que ainda não foram executados.

Entrega da mensagem à aplicação. Quando um *round* do consenso chega ao fim e as réplicas chegam a uma decisão, a mesma é colocada numa fila para valores decididos. Cabe então a uma *thread* (chamada *Delivery Thread*, ver a figura 2.3) retirar o valor decidido da fila, recolher todas as mensagens existentes no lote de mensagens (removendo-as das filas de mensagens dos respectivos clientes) e entregar as mensagens ordenadas à aplicação. É também nesta altura que o estado da réplica é armazenado, e, se necessário, é criado um *checkpoint* do estado da réplica.

2.4.1 Replicação de Máquinas de Estados

Do ponto de vista do cliente, a biblioteca envia a mensagem da aplicação utilizando a primitiva da difusão com ordem total. Após o processamento dessa mensagem por parte das réplicas, cada réplica envia uma mensagem *REPLY* com o resultado ao cliente. Assim, quando um cliente recebe $2f + 1$ mensagens *REPLY*, tem a garantia que a mensagem foi ordenada e que uma maioria das réplicas correctas processou a mensagem com o mesmo resultado. Como vimos anteriormente, existe a possibilidade de executar uma operação *read-only*, bastando para tal invocar a primitiva de envio de mensagens com esse parâmetro.

2.5 Considerações Finais

Neste capítulo falámos da biblioteca BFT-SMaRt, nomeadamente do seu funcionamento e dos seus componentes mais importantes:

- Consenso tolerante a faltas arbitrárias
- Difusão com Ordem Total
- Transferência de Estado

No próximo capítulo iremos falar do desenho e concretização de um mecanismo de monitorização para esta biblioteca.

Capítulo 3

Monitorização e Configuração das Máquinas de Estados Replicadas

Um dos objectivos para este estágio curricular era a concretização de um módulo de monitorização para a biblioteca BFT-SMaRt, de modo a ser possível medir o desempenho da biblioteca, e, se necessário, alterar os parâmetros de configuração que regulam o funcionamento do BFT-SMaRt, tudo em tempo de execução.

De modo a alcançar este objectivo, decidimos utilizar a tecnologia JMX (*Java Management eXtensions*) [12] de modo a criar *MBeans*[1] de estatísticas e de configuração. Visto que a máquina virtual do Java reúne informação automaticamente sobre a sua utilização de memória, número e funcionamento de *threads*, entre outros e esta informação é visível na *JConsole*, decidimos utilizar este cliente de monitorização para visualizar as informações agregadas pelos *MBeans*, aumentando assim a utilidade da *JConsole* em relação à biblioteca BFT-SMaRt. É também uma grande vantagem na utilização do JMX o facto de, devido a ser uma tecnologia padronizada, a máquina virtual do Java conter nativamente todos os serviços necessários para utilizar os *MBeans* concretizados no BFT-SMaRt.

3.1 JMX (*Java Management Extensions*)

A especificação JMX faz parte da plataforma Java e define um modo padronizado de gerir recursos, sendo inclusivamente utilizada para monitorizar a JVM.[2] Esta especificação permite criar objectos Java denominados *MBeans*, que são utilizados para gerir recursos, sendo possível que um recurso possa ter vários *MBeans* a geri-lo.

A arquitectura genérica do JMX, tal como a podemos ver na figura 3.1, consiste em: um *MBean* concretizado numa determinada aplicação, que se liga a um servidor *MBean*; um servidor *MBean* (que poderá, ou não, encontrar-se na JVM onde está a correr a aplicação); e um conector, um módulo que serve de interface entre o servidor *MBean* e

[1] *MBean - Managed Bean*, o objecto Java que implementa as especificações definidas pela API do JMX.
[2] JVM - *Java Virtual Machine*, a máquina virtual do Java.

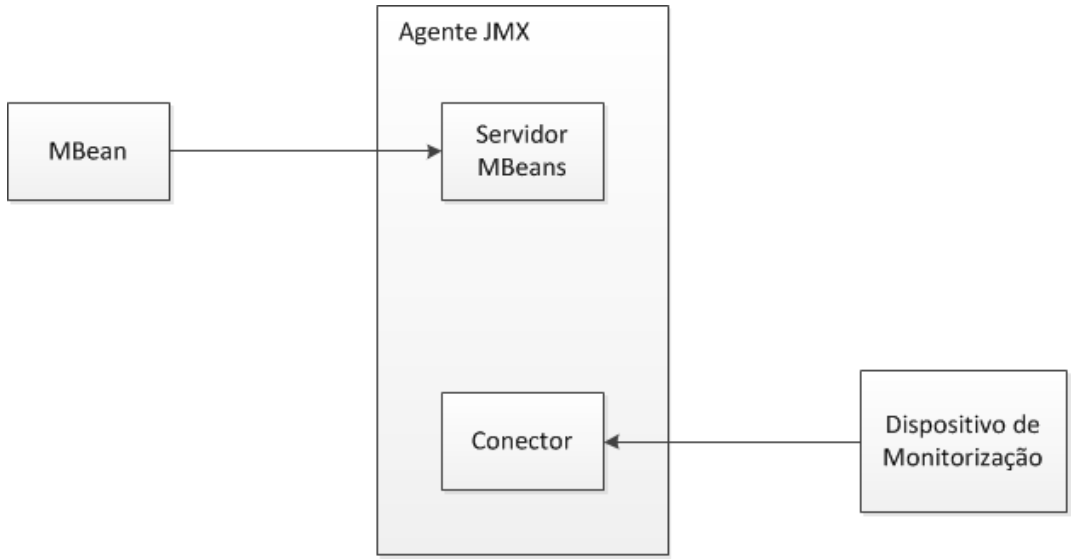


Figura 3.1: A arquitectura genérica do JMX.

o dispositivo de monitorização. Em conjunto, o servidor *MBean* e o conector formam um agente JMX. O dispositivo de monitorização poderá ser a *JConsole* já existente no Java ou uma aplicação desenvolvida de acordo com necessidades específicas.

É de salientar que, embora o conector para a *JConsole* utilize a tecnologia Java RMI[3], poderão ser utilizados conectores que sirvam de interface para outras tecnologias, tais como o CORBA[4] ou o LDAP.[5] Isto torna o JMX bastante versátil, caso existam ferramentas de monitorização previamente desenvolvidas, não sendo necessário refazer essas mesmas ferramentas ao adoptar a tecnologia JMX.

3.2 *MBeans* concretizados no BFT-SMaRt

A arquitectura do JMX tal como é utilizada no BFT-SMaRt é bastante simples, consistindo apenas em dois *MBeans* concretizados na biblioteca, um de estatísticas e um de configuração; um servidor *MBean*, no qual são registados os *MBeans* da biblioteca; um conector, neste caso utilizando Java RMI, para comunicar com a *JConsole* e, finalmente, a *JConsole* que utiliza as informações agregadas pelos *MBeans*. Tanto o servidor *MBean* como o conector utilizado para a ligação com a *JConsole* (no seu conjunto são denominados um agente JMX) já existem por omissão na máquina virtual do Java. A figura 3.2 ilustra a arquitectura JMX, do modo que é utilizada no BFT-SMaRt e iremos em seguida apresentar detalhes sobre os *MBeans* concretizados na biblioteca BFT-SMaRt.

[3]Java RMI - *Remote Method Invocation*, um mecanismo do Java para execução de métodos em diferentes máquinas virtuais, semelhante aos RPC (*Remote Procedure Calls*).

[4]CORBA - *Common Object Request Broker Architecture*

[5]LDAP - *Lightweight Directory Access Protocol*

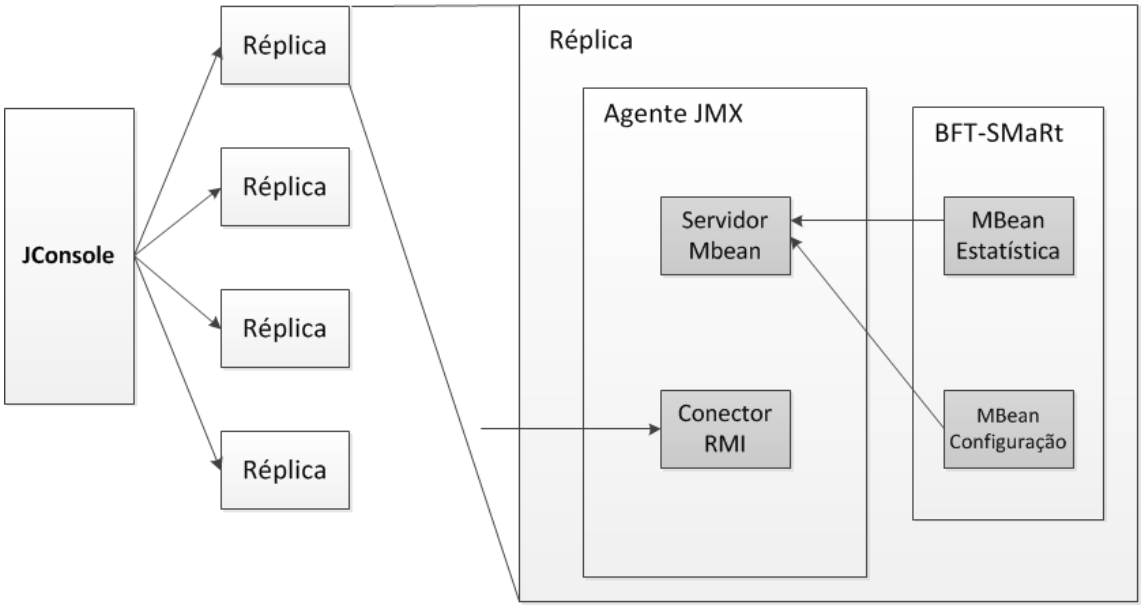


Figura 3.2: Arquitectura JMX utilizada na biblioteca BFT-SMaRt

3.2.1 *MBean* de Estatísticas

Os *Standard MBeans* definidos pela API do JMX são compostos por uma interface, permitindo ao dispositivo de monitorização determinar as operações e atributos disponíveis e por uma classe que implementa os métodos definidos nessa mesma interface. Sendo um *Standard MBean*, o *MBean* de estatísticas concretizado na biblioteca BFT-SMaRt é composto pela interface Java *StatisticsMBean* que define as medidas visíveis pela *JConsole* e por uma classe Java *Statistics* que implementa os métodos estipulados pela interface. Este *MBean* é então registado no servidor *MBean* da JVM pela classe *TOMConfiguration* do BFT-SMaRt.

Listing 3.1: Código que regista os *MBeans* no servidor *MBean* da máquina virtual do Java. Este código encontra-se na classe *TOMConfiguration*.

```
//inicializar MBeans
Statistics statBean = new Statistics(queueSize, verbose);
Configuration confBean = new Configuration(this);

//inicializar servidor MBean
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

ObjectName name1 = new ObjectName("BFT-SMaRt:type=Statistics" + id);
ObjectName name2 = new ObjectName("BFT-SMaRt:type=Configuration" + id);

//registar MBean
mbs.registerMBean(statBean, name1);
mbs.registerMBean(confBean, name2);
```

O excerto de código 3.1 mostra como se registam os *MBeans* concretizados na biblioteca BFT-SMaRt. Quando a classe *Statistics* é inicializada, recebe dois parâmetros: um inteiro a indicar o tamanho das listas de valores guardados para calcular as medidas (por exemplo, se o inteiro for 500, serão guardados no máximo 500 valores de tamanho do lote de mensagens, de débito máximo, e por aí em diante) e um valor booleano a indicar se o *MBean* imprimirá os valores das medidas no *standard output* do Java, para além de os mostrar na *JConsole* (o que poderá ser útil para *logs*).

O excerto de código 3.2, parte do código da classe *Statistics*, demonstra como é calculada a medida do tamanho médio dos lotes de mensagens. As outras medidas do *MBean* de estatísticas são calculadas de modo semelhante.

Listing 3.2: Código que calcula o tamanho médio do lote de mensagens.

```
public int getBatchSize() {
  int size = batchSizeMetrics.size();
  int sumBatchSize = 0;

  //se a lista estiver vazia...
  if(size == 0)
    return 0;

  for (Integer i : batchSizeMetrics) {
    sumBatchSize += i;
  }

  if (verbose)
    System.out.println("#B Media de batch dos ultimos " + size + "
        consensos: " + (sumBatchSize / size));

  return (sumBatchSize / size);
}
```

São recolhidas as seguintes medidas pelo *MBean* de estatísticas, caso esteja activado:

- O tamanho médio dos lotes de mensagens propostos a ordenação (ver a secção 2.4).
- O tempo médio que um consenso demora a ser processado (ver a secção 2.4).
- O tamanho médio das filas de mensagens dos clientes, à espera de serem processadas (ver a secção 2.4).
- O tempo médio que uma mensagem fica na fila à espera de ser ordenada.
- O tempo médio que uma mensagem demora a ser processada pela aplicação (esta medida não depende do desempenho do BFT-SMaRt mas é, sem dúvida, uma medida importante para avaliar o desempenho do sistema no geral).
- O número médio de assinaturas verificadas por segundo (apenas actualizado se a biblioteca estiver a utilizar assinaturas com criptografia assimétrica - ver a secção 2.2).

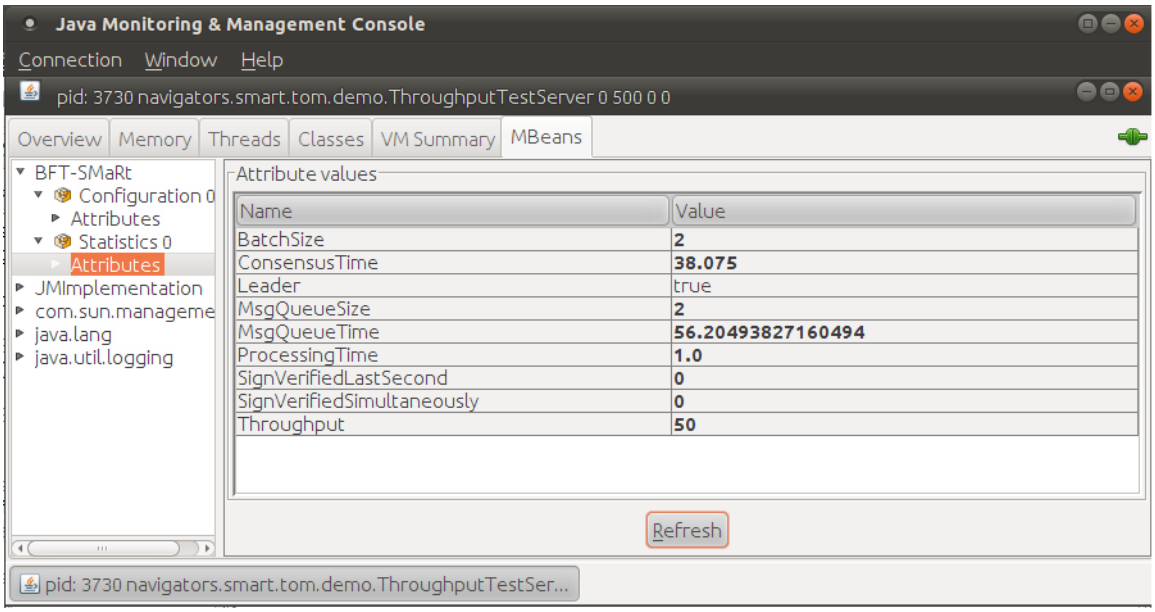


Figura 3.3: Janela da *JConsole* com as informações do *MBean* de estatísticas.

- O número médio de assinaturas verificadas simultaneamente (apenas actualizado se a biblioteca estiver a utilizar assinaturas com criptografia assimétrica, esta medida é importante para constatar um dos resultados da paralelização no desempenho da biblioteca).

- O débito[6] do BFT-SMaRt num intervalo de medida, por omissão é medido a cada 20000 mensagens tratadas.

- Um valor booleano a indicar se a réplica é a líder.

A figura 3.3 ilustra a representação da *JConsole* das informações agregadas pelo *MBean* de estatísticas.

3.2.2 *MBean* de Configuração

O *MBean* de configuração concretizado na biblioteca BFT-SMaRt é um *Dynamic MBean*, que é diferente de um *Standard MBean* como o de estatística, por apenas determinar os seus atributos e métodos em tempo de execução, que é indicado para este *MBean* de configuração, visto que os parâmetros de configuração podem, e provavelmente irão ser alterados frequentemente. Ele é composto por uma classe *ConfigurationMBean* que implementa directamente a interface *DynamicMBean*, contendo os métodos genéricos para obter e definir atributos. Tal como o *MBean* de estatísticas, é registado no servidor *MBean*

[6]Débito - É a quantidade de mensagens entregues com sucesso por medida de tempo.

da JVM, como demonstrado no excerto de código 3.1. Quando a classe *Configuration-MBean* é inicializada, recebe uma instância da *TOMConfiguration* como parâmetro, de modo a conseguir aceder aos parâmetros de configuração do BFT-SMaRt. O excerto de código 3.3 demonstra como foi implementado o método que altera o valor de um atributo contido no *MBean* de configuração.

Listing 3.3: Código que altera o valor de um atributo no *MBean* de configuração.

```
public synchronized void setAttribute(Attribute atrbt) throws
    AttributeNotFoundException, InvalidAttributeValueException {
  String name = atrbt.getName();

  if (this.props.getProperty(name) == null)
    throw new AttributeNotFoundException(name);

  Object value = atrbt.getValue();

  if(isWritableParameter(name)){
    if(isGlobalChange(name))
      propagateAttributeChange(name, value);
    else
      setTOMConfiguration(name, (String) value);
  }
  else
    throw new InvalidAttributeValueException("Property " + name + " is
        read-only!");
}
```

A concretização do *MBean* de configuração exigiu algumas alterações à biblioteca BFT-SMaRt. Foram adicionados métodos à classe *TOMConfiguration* de modo a ser possível alterar os parâmetros de configuração, o que anteriormente não era possível, já que estes parâmetros eram lidos de um ficheiro e permaneciam imutáveis. Também foi necessária uma alteração um pouco mais complexa, de modo a ser possível propagar alterações pelas réplicas (no excerto de código, aparece como o método *propagateAttributeChange(name, value)*). Isto é necessário porque existem parâmetros que podem ser alterados localmente em cada réplica, ou seja, réplicas diferentes poderão ter valores diferentes, sem afectar negativamente o comportamento da biblioteca. Um exemplo de um destes parâmetros é o *system.debug*, o parâmetro que controla se são ou não exibidas as mensagens de *debug*. No entanto, existem parâmetros que apenas podem ser alterados de modo global, ou seja, por todas as réplicas. Um exemplo destes parâmetros é o *system.totalordermulticast.state_transfer*, o parâmetro que activa ou desactiva o mecanismo de transferência de estado da biblioteca (ver a secção 2.3).

Os parâmetros que regulam o funcionamento da biblioteca BFT-SMaRt encontram-se na tabela 3.1, juntamente com a indicação se podem ser alterados localmente ou apenas globalmente. De seguida, indicamos a função de cada um destes parâmetros:

- *system.authentication*: Este parâmetro define se os canais de comunicação utilizam autenticação.

Nome do Parâmetro	Alterável
system.authentication	Não
system.communication.useSenderThread	Não
system.servers.num	Não
system.servers.f	Não
system.totalordermulticast.timeout	Local
system.totalordermulticast.highMark	Local
system.totalordermulticast.revival_highMark	Local
system.totalordermulticast.state_transfer	Global
system.totalordermulticast.checkpoint_period	Global
system.totalordermulticast.maxbatchsize	Local
system.totalordermulticast.nonces	Local
system.totalordermulticast.replayVerificationTime	Global
system.totalordermulticast.verifyTimestamps	Não
system.communication.inQueueSize	Local
system.communication.useControlFlow	Local
system.communication.outQueueSize	Local
system.communication.useSignatures	Não
system.communication.useMACs	Não
system.debug	Local
system.initial.view	Não
system.ttp.id	Não

Tabela 3.1: Tabela contendo os parâmetros do BFT-SMaRt e informação sobre a possibilidade de alteração.

- *system.communication.useSenderThread*: Este parâmetro indica se, quando uma réplica enviar uma mensagem, as mensagens deverão ser entregues à *Sender Thread* ou enviadas directamente utilizando um *socket*.

- *system.servers.num*: O número de réplicas existentes no sistema.

- *system.servers.f*: O número máximo de réplicas faltosas possível.

- *system.totalordermulticast.timeout*: Este parâmetro indica o período de tempo que uma réplica espera pela ordenação de uma mensagem para, caso contrário, iniciar o protocolo de troca de líder.

- *system.totalordermulticast.highMark*: Este parâmetro permite colocar um número máximo de mensagens em que uma mensagem não é descartada por pertencer a um *round* de um consenso superior ao que está a ser executado nesse momento.

- *system.totalordermulticast.revival_highMark*: Este parâmetro é semelhante ao anterior, mas é considerado apenas se a réplica se encontrar a executar o seu primeiro consenso.

- *system.totalordermulticast.state_transfer*: Este parâmetro regula a activação do mecanismo de transferência de estado entre réplicas (ver a secção 2.0.1).

- *system.totalordermulticast.checkpoint_period*: Este parâmetro indica o período entre criações de *checkpoints*, utilizado no mecanismo de transferência de estado.

- *system.totalordermulticast.maxbatchsize*: O número máximo de mensagens em cada lote.

- *system.totalordermulticast.nonces*: O número de *nonces*[7] gerados.

- *system.totalordermulticast.replayVerificationTime*: Este parâmetro permite definir um período após o qual deixam de ser feitas verificações contra ataques de *replay*[8] a um determinado cliente.

- *system.totalordermulticast.verifyTimestamps*: Este parâmetro indica se o *timestamp* existente na mensagem *PROPOSE* deve ser ou não verificado.

- *system.communication.inQueueSize*: Este parâmetro controla a quantidade de mensagens que podem ser armazenadas na fila de recepção de cada réplica, fazendo parte do módulo *Communication System* (ver a secção 2.0.1).

- *system.communication.useControlFlow*: O número máximo de mensagens contidas nas filas de mensagens dos clientes.

- *system.communication.outQueueSize*: Este parâmetro regula a quantidade de mensagens que podem ser armazenadas na fila de envio de cada réplica.

- *system.communication.useSignatures*: Este parâmetro, se tiver o valor booleano *true*, define que as mensagens enviadas pelos clientes estão a ser autenticadas por assinaturas digitais.

- *system.communication.useMACs*: De modo semelhante ao parâmetro anterior, se colocado no valor booleano *true*, as mensagens estão a ser autenticadas por MACs.

- *system.debug*: Este parâmetro permite colocar a réplica a imprimir informação de *debug* no *standard output*.

- *system.initial.view*: Este parâmetro contém uma lista composta pelos IDs das réplicas existentes no início da execução do sistema.

- *system.ttp.id*: Este parâmetro indica o ID do cliente com permissões para reconfigurar o sistema (ver a secção 2.0.1).

[7] *Nonce* criptográfico - *Number used once*, é um número (ou sequência de bits) gerado aleatoriamente e utilizado apenas uma vez.

[8] Ataque de *replay* - É uma forma de ataque à rede, em que uma transmissão de dados válida é repetida ou atrasada maliciosamente.

De modo a propagar as alterações enunciadas anteriormente por todas as réplicas, a alteração global não é efectuada imediatamente, como nas alterações locais. A alteração global, contendo o parâmetro e valor a serem alterados, é adicionada no final do próximo lote de mensagens a ser ordenado. Quando o lote é ordenado e as mensagens começam a ser processadas, todas as réplicas irão ter acesso à alteração global no final do lote, e irão efectuar a mudança de configuração simultâneamente. Esta alteração global terá obrigatoriamente que ser efectuada na réplica líder, já que é a réplica líder que propõe as mensagens a serem ordenadas, logo, é a réplica que constrói o lote de mensagens (ver a secção 2.4). Este mecanismo garante que as faltas são tratadas pelos algoritmos existentes de tolerância a faltas bizantinas da biblioteca, já que a alteração global é tratada como uma mensagem proveniente de um cliente.

No entanto, é importante termos em atenção um detalhe: o caso em que o líder das réplicas é malicioso. Neste caso, o líder poderá forçar mudanças de parâmetros globais para valores que comprometem o comportamento da biblioteca BFT-SMaRt. Este caso não se encontra resolvido na iteração corrente do módulo de monitorização. Porém, podemos adiantar desde já dois métodos possíveis de tratar o problema: simplesmente assumir que a utilização do módulo de monitorização enfraquece o algoritmo BFT da biblioteca, embora isto seja uma solução indesejável, já que diminui a funcionalidade chave da biblioteca; ou podemos implementar valores fronteira para cada um dos parâmetros. Nesta solução, caso o líder tente colocar um valor indevido num determinado parâmetro, a alteração é recusada e o líder é marcado como malicioso, forçando uma troca de líder.

Existem também parâmetros que não podem ser alterados no *MBean* de configuração, devido a serem alterações que não são possíveis de efectuar em tempo de execução. Um exemplo destes parâmetros é o *system.server.num*, que indica o número de réplicas existentes no sistema.

A figura 3.4 ilustra a representação da *JConsole* das informações agregadas pelo *MBean* de configuração. É possível verificar (se a imagem for visualizada a cores) que os parâmetros que são possíveis alterar aparecem numa tonalidade diferente.

3.2.3 Detalhes de Utilização

É possível activar ou desactivar a utilização dos *MBeans* pelo ficheiro de configuração da biblioteca BFT-SMaRt, de modo a que, quando não é necessário monitorizar o sistema, não estejam a ser usados recursos desnecessariamente. O excerto de código 3.4 contém os parâmetros de configuração adicionados ao BFT-SMaRt para os *MBeans* de estatísticas e configuração.

Caso seja adicionado um parâmetro de configuração à biblioteca (adicionando-o ao ficheiro *system.config* que faz parte da biblioteca), para esse parâmetro poder ser gerido pelo *MBean* de configuração, será necessário adicionar um método para alterar o seu valor à classe *TOMConfiguration*, semelhante aos já existentes, e modificar o *MBean* (especifi-

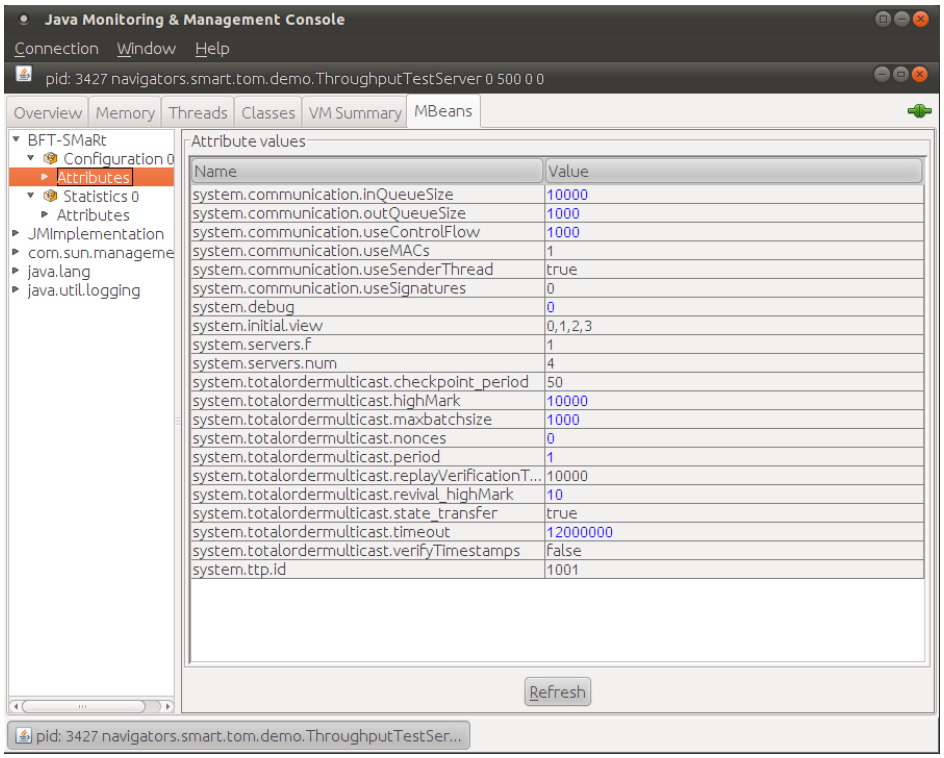


Figura 3.4: Janela da *JConsole* com as informações do *MBean* de configuração.

camente a classe *ConfigurationMBean*), adicionando o nome do parâmetro aos métodos que verificam se um parâmetro é alterável e, se for, se pode ser alterado localmente ou apenas globalmente (respectivamente, *isWritableParameter()* e *isGlobalChange()*).

Listing 3.4: Parâmetros de configuração do BFT-SMaRt relacionados com os *MBeans* de estatísticas e configuração.

```
#Set to true if you want to run the management and
# statistics MBeans, set to false otherwise
system.management = true

#The number of values stored in the statistics MBean
# queues (default is 500)
system.management.sampleSize = 500

#Set to true if you want the statistics MBean to send
# metrics to the standard output, set to false otherwise
system.management.verbose = true
```

3.3 Considerações Finais

Neste capítulo falámos da tecnologia JMX do Java e dos *MBeans* de estatísticas e configuração concretizados na biblioteca BFT-SMaRt. No próximo capítulo iremos falar do ambiente e automatização de testes para a biblioteca.

Capítulo 4

Ambiente de Testes para a Biblioteca BFT-SMaRt

Um dos objectivos deste estágio curricular era avaliar o desempenho da biblioteca BFT-SMaRt. Para isso, foi efectuada uma série de testes, alterando parâmetros que considerámos relevantes. Este capítulo serve como uma introdução ao processo de testes, detalhando em seguida o ambiente em que os testes foram efectuados e o processo de automatização que foi necessário.

4.1 Metodologia de Teste

Antes de ser possível utilizar as classes de teste já existentes no BFT-SMaRt, foi necessário fazer versões *shell* dos ficheiros utilizados para correr os testes, já que apenas existiam em versão Microsoft Windows (ficheiros .bat). Isto foi necessário devido ao facto que as versões *shell* para o Linux tornam o tratamento dos *logs* criados pelo BFT-SMaRt muito mais eficiente e flexível, graças a ferramentas como o *cat*, *awk* e *wc*. Os testes foram efectuados no *cluster* dos Navigators [11], a Quinta, com uma imagem contendo uma versão do Ubuntu Linux.

4.1.1 Parâmetros das Classes de Teste

O BFT-SMaRt possuia de origem algumas classes de teste de desempenho, que foram sendo alteradas ao longo do estágio de modo a permitir toda a variedade de testes que foram efectuados. O teste de débito máximo[1] é composto por um servidor de teste e um cliente de teste, que possuem os seguintes parâmetros de configuração:

Para o servidor:

[1] Relembramos que o débito é a quantidade de mensagens entregues com sucesso por medida de tempo. No caso dos nossos testes, será o número médio de operações efectuadas por unidade de tempo, e será indicado em milhar de operações por segundo (k op/s).

- *id* - ID da réplica, já que todos os servidores têm que ter um ID único e sequencial. No caso de *n* servidores, os IDs serão habitualmente 0...*n*-1.
- *measurement interval* - O intervalo de medida de débito. O resultado do teste no caso do servidor é a média de operações por segundo (em k op/s) para cada intervalo definido. Tem sido utilizado o valor 20000 para este parâmetro, o que significa que é feita uma medida do débito a cada 20000 operações efectuadas pelo servidor, sendo essa medida depois indicada numa informação "As últimas 20000 operações foram executadas a um ritmo de 45332 operações por segundo", por exemplo.
- *processing delay* - Um tempo de processamento adicionado ao receber uma mensagem, para simular a aplicação a efectuar computações.
- *reply size* - O tamanho da resposta do servidor a uma mensagem do cliente.

Para o cliente:

- *port* - Porto inicial do cliente (também utilizado como número do par de chaves para criptografia assimétrica). Visto que o cliente lança várias *threads* para enviar mensagens paralelamente, cada *thread* irá ter um porto, sendo atribuídos sequencialmente. Por exemplo, se colocarmos o porto inicial 1110 e o cliente correr com 10 *threads*, os portos utilizados serão 1110 a 1119 (inclusive). Este número indica também o número *n* do par chave pública e chave privada (que se encontram guardados em disco, com os nomes publickey*n* e privatekey*n*) que o cliente irá utilizar, caso seja utilizado algum mecanismo de criptografia assimétrica.
- *n.msg* - Número total de mensagens a enviar.
- *size.msg* - Tamanho de cada mensagem a ser enviada. As mensagens enviadas são dados aleatórios e não simbolizam nenhuma operação no servidor, logo o servidor simplesmente descarta a mensagem. Isto permite que saibamos o débito máximo possível para o BFT-SMaRt.
- *send.interval* - O intervalo entre envios de mensagens, este parâmetro foi colocado a 0, significando envio contínuo de mensagens.
- *n.threads* - Número de *threads* que o cliente vai criar. Cada *thread* irá enviar o número supracitado de mensagens, logo, este é, efectivamente, o número de clientes que se irá conectar ao servidor.

4.1.2 *Hardware* Utilizado nos Testes

Para os testes, os servidores e os clientes são iniciados em máquinas com identificação S que estejam disponíveis na Quinta, tendo apenas o servidor a correr nas máquinas designadas como servidores, enquanto que são iniciadas várias execuções simultâneas do

cliente nas máquinas que correm os clientes de teste. Existem máquinas com identificação R, no entanto não são potentes o suficiente para permitir avaliar o desempenho máximo do BFT-SMaRt.

As especificações das máquinas existentes na Quinta são as seguintes:

- Máquinas S: DELL PowerEdge R410, com dois procesadores Intel Xeon E5520 (2.27GHz), 32GB de memória RAM, disco SCSI de 146GB de capacidade e interfaces de rede Gigabit

- Máquinas R: DELL PowerEdge 850, com processador Intel Pentium 4 (2.80GHz), 2GB de memória RAM, disco ATA de 80GB de capacidade e interfaces de rede Gigabit

4.2 Primeira Fase de Testes

Para a primeira fase de testes, foi medido o débito máximo do sistema, utilizando 4 réplicas e tendo assim o número máximo de máquinas maliciosas f=1 (ver secção 2.1), aumentando gradualmente o número de clientes. Outro parâmetro essencial para o desempenho do BFT-SMaRt é o método de autenticação de mensagens (MAC ou com criptografia assimétrica), sendo a criptografia assimétrica bastante mais custosa computacionalmente. Para esta fase, testámos apenas com MAC.

Durante a primeira fase de testes, foi possível identificar dois parâmetros importantes para o desempenho do BFT-SMaRt: o tamanho máximo dos lotes de mensagens (ver secção 2.4) e o tamanho máximo da fila de mensagens para cada cliente (ver secção 2.4). Este segundo parâmetro, por algum motivo, não tinha um valor atribuído no ficheiro de configuração da versão utilizada do BFT-SMaRt, fazendo com que, para um número elevado de clientes e de mensagens, a máquina virtual do Java ficasse sem memória. Isto caracterizava-se com um erro *OutOfMemoryException: GC overhead limit exceeded*, que quer dizer que o mecanismo de *garbage collection* do Java estava a utilizar mais de 98% do CPU e a recuperar menos de 2% de memória.

No entanto, não foi possível obter valores consistentes do débito máximo do BFT-SMaRt devido à própria metodologia de teste. O modo de comportamento dos clientes era semelhante a um ataque de negação de serviço (DoS[2]), já que os clientes enviavam todas as mensagens sem esperar resposta, não modelando um comportamento realista. Porém, vendo os resultados obtidos, esperámos valores de débito máximo na ordem de pelo menos 100k op/s, para este caso (f=1 e utilizando MAC).

Com esta informação, e tendo já uma amostra dos *logs* produzidos pela biblioteca BFT-SMaRt ao longo de um teste, o passo seguinte foi alterar o funcionamento do cliente de teste e automatizar o processo de teste, que será explicado na próxima secção.

[2]DoS - *Denial of Service*.

4.3 Automatização

Para conseguir automatizar o processo de teste, o primeiro passo foi criar chaves SSH[3] de modo aos *scripts* não necessitarem de autenticação explícita pelo utilizador. Foram depois criados *scripts* para as seguintes operações:

- Colocação de todos os ficheiros necessários para o teste numa determinada máquina.
- Actualização da versão do BFT-SMaRt utilizada e actualização dos respectivos ficheiros de configuração.
- Inicialização de servidores e clientes de teste (incluindo a respectiva desactivação), com a opção de criarem ou não ficheiros de registo.
- Recuperação dos *logs* existentes numa determinada máquina, incluindo criação de um registo específico para geração de gráficos.
- Cálculo de métricas (débito máximo, médio e o desvio padrão) a partir dos valores dos *logs*.

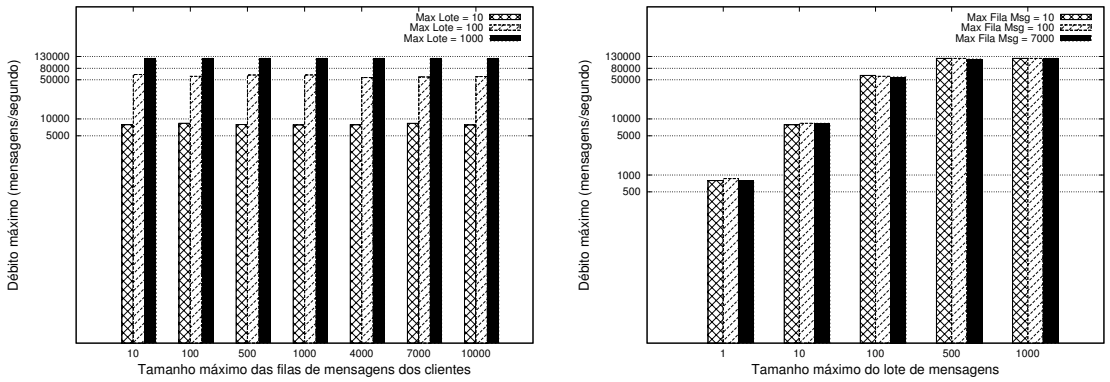
Foi também alterado o cliente de testes, passando a receber os seguintes parâmetros:

- *port* - Porto inicial do cliente e número do par de chaves utilizado para criptografia assimétrica.
- *n.msg.burst* - Número de mensagens a enviar por rajada. Cada *thread* irá enviar um número definido de mensagens por intervalo de tempo.
- *size.msg* - Tamanho de cada mensagem a ser enviada.
- *send.interval* - O intervalo de tempo entre envios de rajadas de mensagens.
- *n.threads* - Número de *threads* que o cliente vai criar.

Assim, cada *thread* enviará uma rajada de *n.msg.burst* mensagens a cada *send.interval* milisegundos, até ser terminada. Isto modela um padrão habitual para um cliente, que vai efectuando operações num determinado serviço, ao longo do tempo.

Existe uma funcionalidade que foi planeada, mas acabou por não ser concretizada na automatização, devido a não ter sido efectuado nenhum teste que a utilizasse: a utilização do *scheduler* do Linux crontab para criar cenários de utilização (por exemplo, ao iniciar o teste, marcar a falha de uma determinada réplica para 20 minutos após o início do teste e consequente recuperação para 30 minutos após o início do teste). Para demonstrar o resultado do processo de automatização, encontra-se no anexo A.1 o código do *script* utilizado para iniciar (e terminar) servidores e clientes de teste.

[3] SSH - *Secure Shell*.



(a) Tamanho do lote de mensagens

(b) Tamanho das filas de mensagens

Figura 4.1: Débito máximo do BFT-SMaRt, variando o tamanho máximo do lote de mensagens e o tamanho das filas de mensagens. Note que os valores do débito máximo são apresentados em escala logaritmica de base 10, já que, devido à grande disparidade entre os valores mínimo e máximo, numa escala linear os valores mais baixos se tornavam extremamente difíceis de distinguir.

4.4 Segunda Fase de Testes

Após a automatização dos processos necessários para testar a biblioteca BFT-SMaRt, foi iniciada uma segunda fase de testes, que produziu as medidas para os resultados mostrados no capítulo seguinte desta dissertação. Durante os testes iniciais, foram utilizados muitos valores para variar os parâmetros em relação ao o número máximo de mensagens num lote e o número máximo de mensagens nas filas de mensagens dos clientes. Isto fez com que, no início, mesmo uma sequência de testes que variasse apenas dois parâmetros fosse bastante demorada. Como exemplo, na primeira sequência de testes efectuada na segunda fase de testes, utilizando MACs para autenticar as mensagens e uma carga imposta de 120000 mensagens/segundo, os parâmetros testados foram: o número máximo de mensagens num lote nos valores 1, 10, 100, 500 e 1000; e o número máximo de mensagens nas filas de mensagens dos clientes nos valores 10, 100, 500, 1000, 4000, 7000 e 10000. Isto deu origem a 35 testes com valores diferentes, cada um deles repetido 3 vezes para assegurar valores consistentes, já que correr o teste apenas uma vez não nos permite assegurar um comportamento consistente da biblioteca e usar 5 ou mais repetições tornava os testes demasiado demorados.

A escolha destes parâmetros surgiu a partir dos parâmetros por omissão que já se encontravam definidos na biblioteca, em que tinhamos um número máximo de 500 mensagens num lote e um número máximo de 1000 mensagens nas filas de mensagens dos clientes. A partir destes valores, tentámos utilizar outros que nos permitissem constatar diferenças significativas no comportamento da biblioteca.

Depois desta primeira sequência de testes, verificámos que não era necessário variar os parâmetros com todos estes valores. Podemos ver a razão para isto na figura 4.1. Em-

bora se confirme que as variações no tamanho máximo do lote de mensagens produzem alterações significativas no desempenho da biblioteca, a variação do tamanho das filas de clientes causa alterações negligíveis, especialmente nos casos em que a biblioteca está a processar todas as mensagens. Este resultado era de esperar, já que, caso a fila de mensagens de um cliente esteja cheia, o BFT-SMaRt simplesmente descarta a mensagem, logo, este parâmetro apenas afecta a percentagem de mensagens de um cliente que são efectivamente processadas, que é uma métrica que não foi considerada em nenhum dos nossos testes (visto que o nosso objectivo era apenas medir o desempenho da biblioteca em termos de débito e da latência de alguns componentes principais do algoritmo BFT). Depois do cenário de testes discutido anteriormente, decidimos diminuir o número de valores com os quais iriamos variar o tamanho das filas de mensagens dos clientes, passando a variar apenas nos valores 10, 100, 500 e 1000.[4]

É importante salientar que esta primeira sequência de testes foi realizada com uma versão do BFT-SMaRt anterior à utilizada em todos os testes que iremos mostrar no capítulo seguinte. Nesta versão anterior ainda não tinha sido corrigido um *bug* que fazia com que fossem transmitidas três pacotes pela rede para cada mensagem do protocolo. Isto explica o aumento do débito máximo de 120k op/s na figura 4.1(b) para 135k op/s na figura 5.1(a), num teste com as mesmas condições.

4.5 Considerações Finais

Neste capítulo apresentámos o ambiente de testes e o processo de automatização necessários para os testes a que submetemos a biblioteca BFT-SMaRt. No próximo capítulo iremos apresentar os resultados destes mesmos testes.

[4]Como exemplo, caso continuássemos a utilizar todos os valores iniciais, ao variar outro parâmetro, tal como o tamanho da mensagem enviada pelo cliente (cujos resultados iremos ver no capítulo seguinte), este cenário de testes iria dar origem a 140 testes diferentes (sem tomar em conta as repetições necessárias)!

Capítulo 5

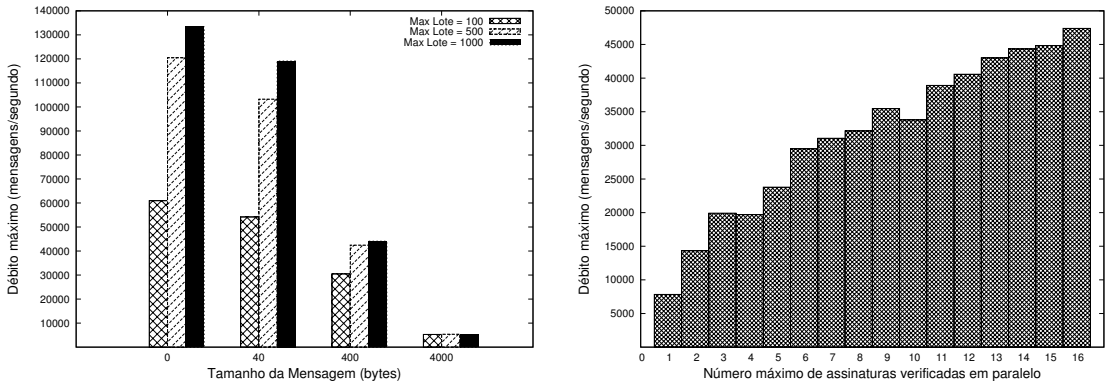
Estudo do Desempenho da Biblioteca BFT-SMaRt

Apresentamos neste capítulo os resultados de vários testes a que foi submetida a biblioteca BFT-SMaRt. De modo a avaliar a biblioteca, foram utilizadas quatro réplicas de forma a tolerar uma falha em qualquer uma dessas réplicas, ou seja, tendo n=4 e f=1 (excepto, claro, no teste que efectuámos para f=2, onde teremos n=7).

Foram consideradas apenas execuções sem falhas e existem duas razões para tal escolha: (1) este é o caso normal esperado da execução do sistema, (2) o desempenho com falhas é bastante previsível: em caso de falha nas réplicas que não o líder, o desempenho tende a melhorar (se a replica ficar silenciosa, existem menos mensagens a processar pelas outras) ou igual (caso a replica falhada envie mensagens irrelevantes ou maliciosas); em caso de falha do líder, o sistema pára de processar mensagens até que se dê um *timeout* (configurável) e proceda à eleição de um novo líder. Neste período, o débito do sistema será 0 e a latência terá acrescida o valor do *timeout* mais a operação de troca de líder (que foi determinado anteriormente a este estágio que demora aproximadamente 5 ms, a depender da quantidade de mensagens pendentes).

Os resultados apresentados nas secções seguintes são uma agregação de variados testes individuais. Existem medidas recolhidas durante os testes à biblioteca que não irão ser representadas neste capítulo, isto acontece devido a serem demasiado semelhantes a casos que já são representados pelos resultados mostrados neste capítulo.

À excepção do teste em que comparamos o BFT-SMaRt com outras bibliotecas de replicação, cada teste foi repetido variando o tamanho das operações enviadas pelos clientes. Os tamanhos escolhidos foram 0 bytes (comandos nulos, para testar apenas o *overhead* da biblioteca), 40 bytes (para os casos em que os dados enviados à aplicação são pequenos), 400 bytes (para o caso em que a aplicação é algo mais complexo, por exemplo, uma base de dados cujos pedidos são comandos SQL complexos) e 4000 bytes (uma mensagem de tamanho similar a blocos de dados NFS). Para estes testes, o serviço replicado não executa nenhuma operação útil, para evitar interferências nas nossas medidas do protocolo. A não ser que seja mencionado o contrário, os pedidos são autenticados



(a) Débito máximo com vectores de MAC.

(b) Débito máximo com assinaturas.

Figura 5.1: Débito máximo do BFT-SMaRt para diferentes tamanhos de pedidos e em execuções com carga máxima suportada.

com vectores de MAC, em vez de assinaturas digitais, como foi descrito na secção 2.2.

Nas medidas utilizadas nestes testes, todas as réplicas do BFT-SMaRt estavam configuradas com um tamanho de fila de entrada de mensagens que suportasse um máximo de 1000 mensagens por cliente. No entanto, à excepção do teste em que comparamos o BFT-SMaRt com outras bibliotecas, também variámos o tamanho máximo que os lotes de mensagens podem alcançar (para um máximo de 100, 500 ou 1000 mensagens).

5.1 Testes de Débito Máximo

Nestes testes, foi medido o débito máximo do sistema, conforme apresentado na figura 5.1. A primeira conclusão que podemos retirar destes resultados, é que o tamanho máximo dos lotes de mensagens é um factor importante para maximizar o desempenho do BFT-SMaRt, já que lotes reduzidos limitam o débito. Também se pode concluir que, à medida que se aumenta o tamanho dos pedidos, o débito máximo vai diminuindo assinalavelmente. Isto é explicado pelo tamanho da mensagem *PROPOSE* que o líder deve disseminar às outras réplicas, já que mensagens maiores demoram, como é de esperar, mais tempo a serem processadas (ver a secção 2.1.2). Pode-se observar que, para comandos de 4000 bytes, o impacto é elevado (cada *PROPOSE* tem aproximadamente 4MB de tamanho, para lotes de 1000 mensagens) e o débito máximo decresce bastante (à volta de 6000 mensagens por segundo, por oposição a 135000 mensagens por segundo com comandos de 0 bytes). É de notar ainda que, durante este teste, observámos uma utilização de CPU média de 13% em cada núcleo, com picos de 28% e minímos de 0,8%. Isto significa que o alto desempenho da biblioteca de replicação não implica o uso de todo o processador, deixando portanto bastantes recursos para o serviço replicado (que, como referimos, nestes testes não realiza processamento).

O uso de vectores de MACs para autenticar mensagens de clientes pode levar a proto-

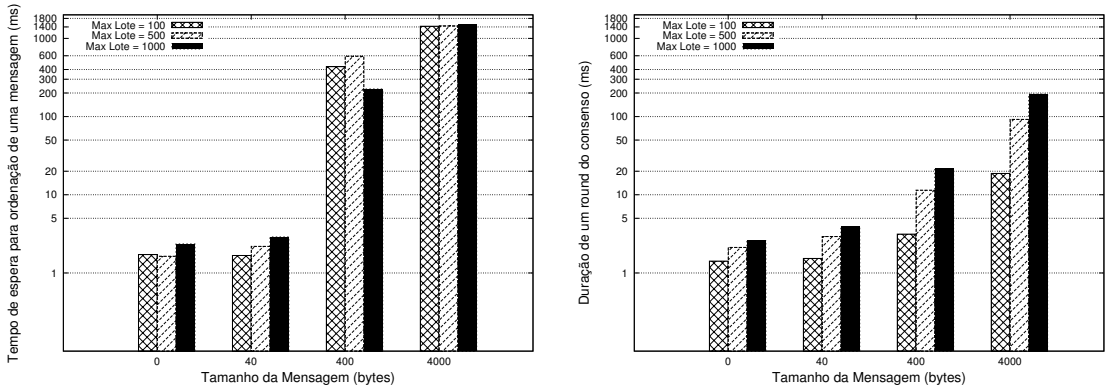
colos vulneráveis em que clientes maliciosos podem forçar a troca de líder através da criação de vectores de MACs parcialmente correctos [2]. Assim, uma concretização robusta de replicação de máquinas de estados BFT requer o uso de assinaturas digitais (utilizando criptografia assimétrica). Para verificar qual o impacto do uso de assinaturas digitais no desempenho do protocolo, configurámos o BFT-SMaRt para autenticar as mensagens com assinaturas e fizemos com que as réplicas executassem entre 1 a 16 verificações de assinaturas em paralelo (8 núcleos em cada máquina, cada um com *hyper-threading*, dando ao sistema operativo 16 núcleos virtuais), utilizando pedidos de 0 bytes de modo a conseguirmos medir o número máximo possível de verificações. A figura 5.1(b) apresenta os resultados obtidos. Podemos observar que o débito máximo alcançado vai aumentando, à medida que aumenta o número de verificações de assinaturas que são realizadas simultaneamente. Isto deve-se ao facto da verificação de assinaturas ser feita pelas *threads* do *Netty* (ver a secção 2.4), que recebem as requisições dos clientes, e portanto naturalmente podem ser feitas em paralelo, sem nenhuma necessidade de sincronização entre si. Observámos neste teste uma utilização dos processadores de aproximadamente 50% em média. Este teste demonstra a capacidade do BFT-SMaRt de tirar proveito das arquitecturas modernas para aumentar seu desempenho, preservando ainda uma quantidade considerável de processador para a aplicação executar o serviço replicado.

Podemos concluir que é preferível que o tamanho do lote de mensagens seja elevado (já que um tamanho elevado para o número máximo de mensagens num lote não significa que os lotes sejam compostos pelo número de mensagens definido, apenas que os lotes não irão ultrapassar este número) de modo a não limitar o desempenho, desde que isto não cause um atraso significativo no processamento da mensagem *PROPOSE*. Na secção 5.5 iremos tentar extrapolar um tamanho óptimo para o lote de mensagens, baseando-nos nestas limitações.

5.2 Testes de Latência

Nestes testes, foi medido o tempo médio necessário para, uma vez que uma mensagem chegue a uma réplica, ser ordenada pelo sistema. Adicionalmente, também foi medido o tempo médio necessário para a execução normal do algoritmo ordenar um lote de mensagens. Estes dois componentes correspondem ao acréscimo de latência do protocolo de replicação num sistema cliente-servidor. A figura 5.2 apresenta os resultados observados para diferentes tamanhos de pedidos e tendo um número variável de clientes a impôr a carga máxima de trabalho suportada para cada tamanho de mensagem (ver figura 5.1(a)).

Considerando o tempo médio de cada pedido na fila de espera, podemos observar que para comandos nulos, o tempo de espera é negligível, já que todas as mensagens vão sendo processadas à medida que vão chegando. Para comandos de 40 bytes, confirmamos



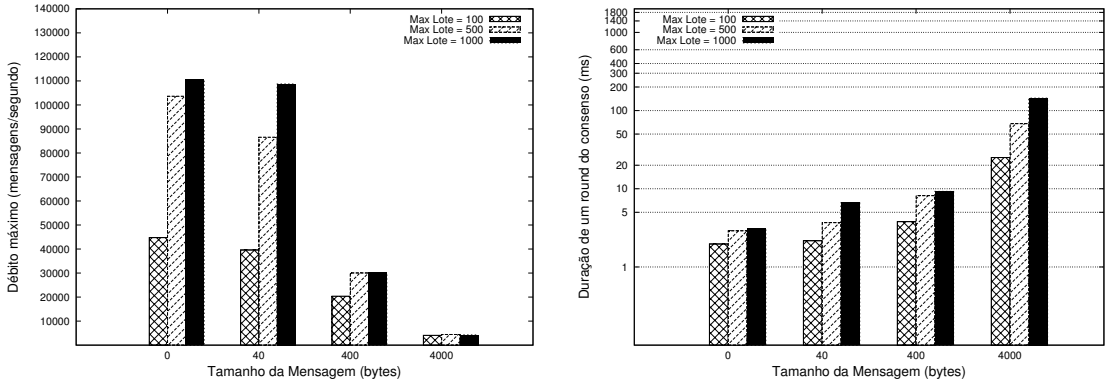
(a) Tempo de espera para ordenação.

(b) Latência da ordenação de mensagens.

Figura 5.2: Componentes da latência do BFT-SMaRt para diferentes tamanhos de pedidos e em execuções com carga máxima suportada. Note que os valores de latência são apresentados em escala logaritmica de base 10, já que, devido à grande disparidade entre os valores mínimo e máximo, numa escala linear os valores mais baixos se tornavam extremamente difíceis de distinguir.

assim que um tamanho máximo de 100 mensagens por lote é uma limitação, pois para os máximos de 500 e 1000 mensagens, o tempo de espera continua a ser negligível. Para comandos de 400 e 4000 bytes, pode-se observar que, à medida que o tamanho dos comandos vai aumentando, o tempo de espera também aumenta, passando a ser um factor com grande impacto no desempenho do sistema.

Considerando o tempo médio de duração da execução normal da biblioteca, pode-se constatar que, para comandos nulos e de 40 bytes, a duração dessa execução é bastante reduzida. Para 400 bytes, torna-se ligeiramente maior, mas ainda assim bastante próxima à das mensagens de tamanho reduzido. Com 4000 bytes, já se observa um grande crescimento na duração média, especialmente para lotes de 1000 mensagens, pelo tamanho alargado da mensagem *PROPOSE* enviada pelo líder (ver a secção 2.4). No geral, observamos que o aumento do tamanho da mensagem numa ordem de grandeza - 40 para 400 e 400 para 4000 bytes - implica o aumento da latência também numa ordem de grandeza - 3ms para 20ms e 20ms para 200ms, respectivamente. Esta mesma observação é válida também para o tamanho máximo do lote de mensagens quando estão a ser processadas mensagens grandes: quando modificado o batch máximo de 100 para 1000, temos um aumento de uma ordem de grandeza na latência do protocolo, que não se reflete no débito (os diferentes tamanhos de batch tem pouco efeito no débito observado para mensagens de 400 e 4000 bytes - ver a figura 5.1(a)). Isto mostra que, apesar do tamanho do lote de mensagens ser fundamental para obtenção de débitos altos para mensagens relativamente pequenas, ele não é vantajoso quando as requisições esperadas são grandes.



(a) Débito máximo com f=2. (b) Latência da ordenação de mensagens com f=2.

Figura 5.3: Débito máximo e latência da ordenação de mensagens do BFT-SMaRt para diferentes tamanhos de pedidos e em execuções com carga máxima suportada, suportando duas faltas (f=2). Note que os valores de latência são apresentados em escala logaritmica de base 10.

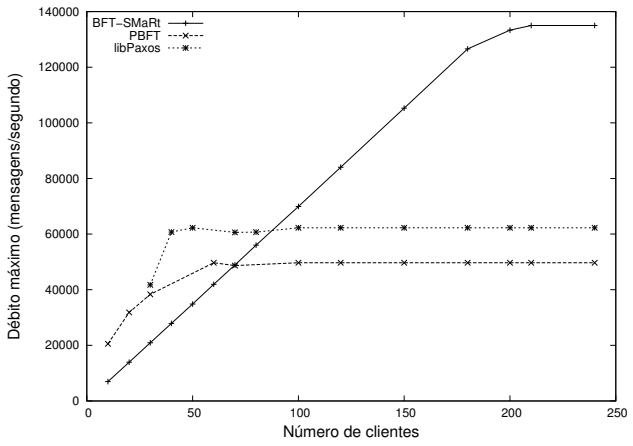
5.3 Testes de Débito Máximo e Latência com f=2

Neste teste, foi medido o impacto sobre a biblioteca BFT-SMaRt de suportar duas faltas (f=2) ao invés de uma, como no resto dos testes. Poderemos verificar esse impacto comparando a figura 5.3 com as figuras 5.1(a) e 5.2(b).

Podemos verificar que o desempenho da biblioteca segue o mesmo comportamento evidenciado nos testes com f=1, nas secções 5.1 e 5.2. Embora exista um impacto no débito máximo da biblioteca, que é justificável já que os *rounds* do consenso envolvem 7 máquinas ao invés de apenas 4 (ver a secção 2.1.2), o que significa um aumento das mensagens na rede, este impacto não é muito severo e a biblioteca continua a comparar-se muito favoravelmente em termos de desempenho com outras bibliotecas BFT (ver a secção 5.4). É de notar também que a latência do consenso neste cenário de testes é semelhante à latência do cenário testado com f=1. Isto é expectável, já que o processamento dos *rounds* do protocolo é largamente afectado pelo tamanho das mensagens, como vimos na secção anterior, e não pelo número de réplicas envolvidas no protocolo do consenso.

5.4 Comparação com Outras Bibliotecas BFT

De modo a comparar o BFT-SMaRt com outras bibliotecas de replicação, executámos alguns testes com o PBFT [4], considerado o algoritmo fundamental para avaliação de protocolos BFT, e o libPaxos [13], uma implementação recente do algoritmo *Paxos* para tolerância a faltas por paragem. É de salientar que ambas as bibliotecas foram concretizadas em C++, ao contrário do BFT-SMaRt, concretizado em Java. O nosso principal objectivo para este teste é comparar o débito máximo e latência do BFT-SMaRt em



Clientes	1	30	70
libPaxos	0.65	0.71	1.13
PBFT	0.48	0.68	0.85
BFT-SMaRt	1.71	2.75	3.12

Figura 5.4: Comparação do BFT-SMaRt com as bibliotecas de replicação libPaxos e PBFT em termos de débito e latência (milisegundos) para requisições de 0 bytes.

relação a estas duas outras bibliotecas de replicação.

No gráfico da figura 5.4 podemos observar que, até cerca de 70 ou 80 clientes, o débito do libPaxos e do PBFT é mais alto que o do BFT-SMaRt, sendo o débito máximo do primeiro maior que o do segundo (o que faz sentido, já que o libPaxos tolera apenas faltas por paragem e portanto não realiza verificações de MACs). Mas por essa altura, tanto o libPaxos como o PBFT atingem o seu limite, enquanto que o débito do BFT-SMaRt continua a crescer, até cerca dos 200 clientes. A partir daí o débito do BFT-SMaRt também estabiliza, mas consegue alcançar um débito máximo muito mais elevado (135000 ops/seg) do que tanto o PBFT (49000 ops/seg) ou o libPaxos (61000 ops/seg). Uma explicação destes resultados requer um entendimento completo da concretização tanto do PBFT quanto do libPaxos, o qual não possuímos no momento. No entanto, podemos adiantar que pelo menos dois factores contribuem para o melhor débito do BFT-SMaRt: (1) a sua arquitectura explora agressivamente o uso de multiplas *threads*, enquanto que as outras bibliotecas utilizam uma única *thread*; (2) o BFT-SMaRt foi desenhado desde o início para usar lotes de mensagens muito grandes, enquanto as outras bibliotecas limitam-se a usar lotes menores e, para além disso, executar várias ordenações de mensagens em paralelo (ao contrário do BFT-SMaRt, que executa uma ordenação de mensagens de cada vez, como foi visto na secção 2.1.2).

Na figura 5.4 também temos uma tabela com a latência das operações das três bibliotecas com variados números de clientes. Os resultados aqui mostram que o BFT-SMaRt impõe uma latência adicional maior que as outras bibliotecas. Isto deve-se parcialmente ao facto da arquitectura da biblioteca ter sido planeada de modo a obter a grandes débitos, podendo existir então um impacto na latência.

Além disso, não vemos como um problema prático termos um *overhead* de 2ms ao invés de 0,5ms, pois as dimensões aqui são de facto muito pequenas quando comparadas a latência observada em aplicações reais. É de notar que essa perda de desempenho

Tamanho msg (bytes)	Débito máximo (k op/s)	Latência consenso (ms)	Tamanho médio lote	Tamanho *PROPOSE* (bytes)
0	134	2.683	430	12470
20	134	3.677	586	28714
40	134	6.803	1072	73968
50	130	7.808	1180	93220
60	122	7.577	1074	95586
80	113	8.583	1102	120118
100	102	9.113	1051	135579
120	95	10.191	1080	160920
140	85	8.05	753	127257
160	80	9.019	788	148932
180	75	10.565	859	179531
200	70	9.806	741	169689
1000	20	95.907	1964	2020956

Tabela 5.1: Tabela contendo os resultados do teste ao número óptimo de mensagens num lote.

acontece apenas para operações que alteram o estado do sistema, já que para operações apenas de leitura (ou seja, pedidos que não requerem ordenação) todas as três bibliotecas de replicação apresentam uma latência de 0,2ms.

5.5 Tamanho Óptimo do Lote de Mensagens

Tendo visto como o tamanho dos lotes de mensagens têm um impacto significativo no desempenho do BFT-SMaRt, tentámos determinar se existiria um tamanho óptimo para o mesmo. Neste testes foi aumentado gradualmente o tamanho da mensagem até ao ponto que os consensos começam a demorar demasiado e a afectar o desempenho. A tabela 5.1 contém os resultados deste teste.

Podemos ver que as réplicas conseguem atingir o débito máximo até às mensagens de 40 bytes, que origina um *PROPOSE* de aproximadamente 74kB. Poderíamos então assumir um valor à volta dos 80 kB para tamanho máximo do *PROPOSE*, de modo a manter uma latência do consenso à volta dos 7ms.

O objectivo seguinte seria correr testes com tamanhos maiores de mensagens, limitando o tamanho das mensagens *PROPOSE* aos 80kB, mas, devido ao número máximo de mensagens num lote ser menor (o máximo seria um valor por volta das 620 mensagens), isto causou com que existissem mais mensagens à espera de ser processadas e as réplicas entraram em *thrashing*[1], causado por um *bug* ainda não resolvido. Foi também devido a isto que, à medida que o tamanho das mensagens era aumentado, era necessário diminuir a carga imposta ao sistema.

[1] *Thrashing* ou Rarejamento - Um fenómeno no qual um sistema gasta uma grande quantidade de recursos a efectuar uma quantidade mínima de trabalho.

5.6 Considerações Finais

Foram apresentados neste capítulo os resultados dos testes a que foi submetida a biblioteca para concretização de máquinas de estados replicadas tolerantes a faltas bizantinas BFT-SMaRt, e foi também discutida a relevância destes resultados e a sua interligação com o funcionamento da própria biblioteca.

Capítulo 6

Conclusão

Neste documento foram apresentados os resultados da sequência de testes a que a biblioteca BFT-SMaRt foi submetida, bem como os resultados do processo de automatização dos mesmos. Também foi apresentado o novo mecanismo concretizado para a biblioteca, um módulo que permite monitorizar e configurar as réplicas do sistema.

Foi possível verificar, ao termos as medidas resultantes dos testes efectuados, que o objectivo original de desenvolver um mecanismo que permitisse à biblioteca adaptar dinamicamente os seus parâmetros de configuração, de modo a optimizar o seu desempenho, não era necessário. Isto porque pudémos ver que os valores utilizados de origem pela biblioteca já eram os valores que permitiam o melhor funcionamento do BFT-SMaRt. Devido a esta razão, foi elaborado um novo objectivo, de acrescentar um módulo de monitorização ao BFT-SMaRt, que, como vimos, foi concretizado.

Também verificámos que, embora o BFT-SMaRt seja bastante competitivo em termos de desempenho quando comparado com outros algoritmos BFT de referência, este desempenho tem ainda margem para ser melhorado, agora que conhecemos os parâmetros que mais afectam o seu funcionamento. Isto é importante, já que a biblioteca BFT-SMaRt ainda se encontra em evolução, caso ilustrado pelo facto de, durante a sequência de testes, a versão utilizada da biblioteca ter sido alterada três vezes, para versões cada vez mais optimizadas.

Pelos motivos referidos anteriormente, consideramos que a principal contribuição deste trabalho foi a demonstração que os valores utilizados originalmente pela biblioteca eram os que melhor permitem um bom desempenho da biblioteca, juntamente com o módulo de monitorização e configuração da biblioteca e a automatização do processo de testes, de modo a torná-los mais rápidos e eficientes no futuro.

6.1 Trabalho Futuro

Uma das conclusões que podemos retirar dos testes realizados é que a utilização de lotes de mensagens demasiado grandes tem um efeito adverso no desempenho da bib-

lioteca. Na sequência disto, um modo de melhorar a biblioteca no futuro poderá passar por limitar os lotes de mensagens por tamanho em vez de por mensagens. Isto fará com que os lotes nunca se tornem tão grandes que provoquem uma latência dos consensos demasiado elevada. Este mesmo conceito poderá ser também utilizado para limitar as filas de mensagens dos clientes, em que podem deixar de ser limitadas por número de mensagens para passarem a ser limitadas consoante a capacidade de memória da máquina a ser utilizada e o número de clientes ligados à réplica. Isto não foi feito porque a modificação da biblioteca BFT-SMaRt não era um objectivo deste estágio.

Outro aspecto que precisa de ser melhorado num futuro próximo é a latência da biblioteca. Como foi possível verificar, a latência do BFT-SMaRt é superior à das outras bibliotecas testadas. No entanto, não existe um motivo óbvio para esse facto, pelo que é então um aspecto que deverá ser corrigido brevemente.

Apêndice A

Anexos

A.1 Listagem do código do *script* de gestão de servidores e clientes de teste

Listing A.1: Código do *script* de gestão de servidores e clientes de teste

```
#!/bin/bash
#@author Bruno Branco e Brito, Marcelo Pasin

SMARTDIR=/home/SMaRt
LOGDIR=$SMARTDIR/logs
#RUNNING=$LOGDIR/running-processes
EXPLOG=$LOGDIR/`hostname`-experiment.txt
OPTION=$1

PROCID=`date +%s`-$$

#########
#SERVERS
#########

launch_throughput_server()
{
# 4 arguments
# $1: replica.id
# $2: measure.interval
# $3: processing.delay
# $4: reply.size
  local RID=$1
  local MINT=$2
  local PDEL=$3
  local RSIZ=$4
  local LOGFILE=$LOGDIR/$PROCID-$RID-$MINT-$PDEL-$RSIZ.txt
  {
    #echo "ThroughputTestServer $$ $LOGFILE" >> $RUNNING
    echo "#ThroughputTestServer"
    echo "#measurement interval: $MINT"
    echo "#processing delay: $PDEL x 0.03 ms (aprox.)"
    echo "#reply size: $RSIZ"
    cd $SMARTDIR
```

```
    exec $SMARTDIR/runscripts/smartrun_server.sh navigators.smart.tom.
        demo.ThroughputTestServer $RID $MINT $PDEL $RSIZ
  } > $LOGFILE &
}

launch_verbose_throughput_server()
{
# 4 arguments
# $1: replica.id
# $2: measure.interval
# $3: processing.delay
# $4: reply.size
  local RID=$1
  local MINT=$2
  local PDEL=$3
  local RSIZ=$4
  {
    cd $SMARTDIR
    exec $SMARTDIR/runscripts/smartrun_server.sh navigators.smart.tom.
        demo.ThroughputTestServer $RID $MINT $PDEL $RSIZ
  } &
}

launch_latency_server()
{
# 1 argument
# $1: replica.id
# $2: measure.interval
  local RID=$1
  local MINT=$2
  local LOGFILE=$LOGDIR/$PROCID-$RID-$MINT.txt
  {
    #echo "LatencyTestServer $$ $LOGFILE" >> $RUNNING
    echo "#LatencyTestServer"
    echo "#measurement interval: $MINT"
    cd $SMARTDIR
    exec $SMARTDIR/runscripts/smartrun_server.sh navigators.smart.tom.
        demo.LatencyTestServer $RID $MINT
  } > $LOGFILE &
}

launch_verbose_latency_server()
{
# 1 argument
# $1: replica.id
# $2: measure.interval
  local RID=$1
  local MINT=$2
  {
    cd $SMARTDIR
    exec $SMARTDIR/runscripts/smartrun_server.sh navigators.smart.tom.
        demo.LatencyTestServer $RID $MINT
  } &
}

#########
```

```
#CLIENTS
#########

launch_throughput_client()
{
# 5 arguments
# $1: port
# $2: num.messages/burst
# $3: tamanho.msg
# $4: intervalo.envio
# $5: num.threads
  local PORT=$1
  local NMSG=$2
  local TMSG=$3
  local SINT=$4
  local NTHR=$5
  {
    cd $SMARTDIR
    exec $SMARTDIR/runscripts/smartrun.sh navigators.smart.tom.demo.
        ThroughputTestClient $PORT $NMSG $TMSG $SINT $NTHR
  } &
}


launch_latency_client()
{
# 5 arguments
# $1: port
# $2: num.messages
# $3: tamanho.msg
# $4: intervalo.envio
# $5: read-only?
  local PORT=$1
  local NMSG=$2
  local TMSG=$3
  local SINT=$4
  local READ=$5
  {
    cd $SMARTDIR
    exec $SMARTDIR/runscripts/smartrun.sh navigators.smart.tom.demo.
        LatencyTestClient $PORT $NMSG $TMSG $SINT $READ
  } &
}


######
#MISC
######

kill_procs()
{
# 0 arguments
  exec killall java
}

clean_logs()
```

```
{
# 0 arguments
  rm $LOGDIR/*
}




############
#THROUGHPUT
############

#-lts: launch throughput server
#example: -lts 1 20000 0 0
if [ "$OPTION" == "-lts" ]
then
  RID=$2
  MINT=$3
  PDEL=$4
  RSIZ=$5
  clear
  echo `date +%d.%m.%Y-%T` " - Launching server replica $RID (PID $$),
      measure interval $MINT, processing delay $PDEL, reply size $RSIZ
      ..." >> $EXPLOG
  launch_throughput_server $RID $MINT $PDEL $RSIZ
fi

#-lvts: launch verbose throughput server
#example: -lvts 1 20000 0 0
if [ "$OPTION" == "-lvts" ]
then
  RID=$2
  MINT=$3
  PDEL=$4
  RSIZ=$5
  clear
  launch_verbose_throughput_server $RID $MINT $PDEL $RSIZ
fi

#-ltc: launch throughput clients
#example: -ltc 3 1121 5 0 50 24
if [ "$OPTION" == "-ltc" ]
then
  NCLI=$2
  # 5 arguments
  # $3: port
  # $4: num.messages
  # $5: tamanho.msg
  # $6: intervalo.envio
  # $7: num.threads
  PORT=$3
  NMSG=$4
  TMSG=$5
  SINT=$6
```

```
  NTHR=$7
  LOGFILE=$LOGDIR/$PROCID-$PORT-$NMSG-$TMSG-$SINT-$NTHR.txt
  {
    #echo "time - <n.proc, n.threads, period, msg.burst, msg.size>" >>
        $EXPLOG
    echo `date +%d.%m.%Y-%T` " - <$NCLI, $NTHR, $SINT, $NMSG, $TMSG>"
        >> $EXPLOG
    for (( i=0; i<$NCLI; i++ ))
    do
      CID=$(( $PORT + ($i * $NTHR) ))
      #echo `date +%D-%T` " - Launching client $CID..." >> $EXPLOG
      launch_throughput_client $CID $NMSG $TMSG $SINT $NTHR
      ping -c 1 -n 127.0.0.1
    done
  } > $LOGFILE
fi

#-lvtc: launch verbose throughput clients
#example: -lvtc 3 1121 5 0 50 24
if [ "$OPTION" == "-lvtc" ]
then
  NCLI=$2
  # 5 arguments
  # $3: port
  # $4: num.messages
  # $5: tamanho.msg
  # $6: intervalo.envio
  # $7: num.threads
  PORT=$3
  NMSG=$4
  TMSG=$5
  SINT=$6
  NTHR=$7

  for (( i=0; i<$NCLI; i++ ))
  do
    CID=$(( $PORT + ($i * $NTHR) ))
    launch_throughput_client $CID $NMSG $TMSG $SINT $NTHR
    ping -c 1 -n 127.0.0.1
  done
fi


#########
#LATENCY
#########

#-lls: launch latency server
#example: -lls 1 20000
if [ "$OPTION" == "-lls" ]
then
  RID=$2
  MINT=$3
  clear
  echo `date +%d.%m.%Y-%T` " - Launching server replica $RID (PID $$)
      ..." >> $EXPLOG
```

```
  launch_latency_server $RID $MINT
fi

#-lvls: launch verbose latency server
#example: -lvls 1 20000
if [ "$OPTION" == "-lvls" ]
then
  RID=$2
  MINT=$3
  clear
  launch_verbose_latency_server $RID $MINT
fi

#-llc: launch latency clients
#example: -llc 30 1121 1000 0 1 false
if [ "$OPTION" == "-llc" ]
then
  NCLI=$2
  # 4 arguments
  # $3: port
  # $4: num.messages
  # $5: tamanho.msg
  # $6: intervalo.envio
  # $7: read-only?
  PORT=$3
  NMSG=$4
  TMSG=$5
  SINT=$6
  READ=$7
  LOGFILE=$LOGDIR/$PROCID-$PORT-$NMSG-$TMSG-$SINT-$READ.txt
  {
    #echo "time - <n.proc, period, msg.burst, msg.size>" >> $EXPLOG
    echo `date +%d.%m.%Y-%T` " - <$NCLI, $SINT, $NMSG, $TMSG>" >>
        $EXPLOG
    for (( i=0; i<$NCLI; i++ ))
    do
      CID=$(( $PORT + $i ))
      #echo `date +%D-%T` " - Launching client $CID..." >> $EXPLOG
      launch_latency_client $CID $NMSG $TMSG $SINT
      ping -c 1 -n 127.0.0.1
    done
  } > $LOGFILE
fi

#-lvlc: launch verbose latency clients
#example: -lvlc 30 1121 1000 0 1 false
if [ "$OPTION" == "-lvlc" ]
then
  NCLI=$2
  # 4 arguments
  # $3: port
  # $4: num.messages
  # $5: tamanho.msg
  # $6: intervalo.envio
  # $7: read-only?
  PORT=$3
```

```
  NMSG=$4
  TMSG=$5
  SINT=$6
  READ=$7

  for (( i=0; i<$NCLI; i++ ))
  do
    CID=$(( $PORT + $i ))
    launch_latency_client $CID $NMSG $TMSG $SINT $READ
    ping -c 1 -n 127.0.0.1
  done
fi


######
#MISC
######

#-kill: kill all java processes
if [ "$OPTION" == "-kill" ]
then
  echo `date +%d.%m.%Y-%T` " - Killing all java processes..." >>
      $EXPLOG
  kill_procs
fi

#-clean: clean all SMaRt logs
if [ "$OPTION" == "-clean" ]
then
  clean_logs
fi
```

Bibliografia

[1] Gnuplot - Portable command-line driven graphing utility. `http://www.gnuplot.info/`.

[2] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Byzantine Replication Under Attack. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'08)*, June 2008.

[3] A. Bessani, P. Sousa, and E. Alchieri. Experiences in the Design and Implementation of a BFT Replication Library. Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal.

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

[5] JBoss Community. Netty - the Java NIO Client-Server Socket Framework. `http://jboss.org/netty`.

[6] Cyntia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–322, 1988.

[7] Miguel Garcia, Alysson Bessani, Illir Gashi, Nuno Ferreira Neves, and Rafael Obelheiro. OS Diversity for Intrusion Tolerance: Myth or Reality? In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'11)*, Hong Kong, 2011.

[8] Vassos Hadzilacos and Sam Toueg. A Modular Approach to the Specification and Implementation of Fault-Tolerant Broadcasts. Technical Report TR 94-1425, Department of Computer Science, Cornell University, New York - USA, May 1994.

[9] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[11] LaSIGE-DI. Navigators Research Team. `http://www.navigators.di.fc.ul.pt/`.

[12] Oracle. Java Management Extensions (JMX). `http://download.oracle.com/javase/6/docs/technotes/guides/jmx/`.

[13] Marco Primi. libpaxos[2]. `http://libpaxos.sourceforge.net/`.

[14] João Sousa, Bruno Branco e Brito, Alysson Bessani, and Marcelo Pasin. Desempenho e Escalabilidade de uma Biblioteca de Replicação de Máquina de Estados Tolerante a Faltas Bizantinas. *INForum 2011*, 2011.

[15] P. Zieliński. Paxos at War. University of Cambridge Computer Laboratory, Cambridge, UK, June 2004. University of Cambridge.