Suportando Execuções Paralelas no BFT-SMaRt*

Eduardo Adilio Pelinson Alchieri

[1] Departamento de Ciência da Computação
Universidade de Brasília

Resumo. *A replicação Máquina de Estados é uma das abordagens mais usadas na implementação de sistemas tolerantes a falhas, tanto por parada quanto bizantinas. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados. Para isso, as requisições dos clientes devem ser ordenadas e executadas seguindo esta ordem em todas as réplicas. Este requisito fez com que a maioria dos trabalhos utilizassem uma única thread de execução em cada réplica. Com o objetivo de melhorar o desempenho do sistema, novas abordagens foram introduzidas para suportar várias threads de execução por réplica. Dando seguimento a estes trabalhos, este artigo descreve como um protocolo que possibilita o emprego de várias threads de execução nas réplicas foi adaptado e implementado no* BFT-SMaRt*, além de analisar uma série de experimentos realizados.*

Abstract. *State Machine Replication is an approach widely used to implement fault-tolerant systems. The idea behind this approach is to replicate the servers and to coordinate the interactions among clients and servers replicas, making all of these replicas present the same state evolution. Consequently, client requests must be ordered and executed following this order by every server replica. Due to this requirement, most of works used a single-threaded replica. In order to improve system performance, some approaches were introduced to allow multiple execution threads per replica. In this sense, this work describes as a protocol that allows multiple execution threads per replica was adapted and implemented in the Bft-SMaRt. Moreover, some experiments executed with this implementation are analysed.*

1. Introdução

A replicação Máquina de Estados (RME) [Schneider 1990] é a abordagem mais abrangente e também uma das mais utilizadas na implementação de sistemas tolerantes a falhas, tanto por parada [Schneider 1990] quanto bizantinas [Castro and Liskov 2002]. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados.

Desta forma, o ponto fundamental de uma RME é a necessidade de ordenação das requisições dos clientes para serem executadas pelas réplicas (servidores) que implementam o sistema, a fim de manter o determinismo de réplicas. Para isso, é necessário o emprego de um protocolo de difusão atômica, que geralmente é implementado através de um algoritmo de consenso, onde todas as réplicas entram em acordo sobre a ordem de execução das requisições, as quais são então organizadas em uma sequência para ser executada no sistema.

Para que esta ordem seja mantida durante a execução das requisições, a maioria dos protocolos para RME consideram apenas uma *thread* de execução [Castro and Liskov 2002,

*Este trabalho recebeu apoio da CAPES através do projeto Scalable Dependability (88881.062190/2014-01) e do CNPq através do projeto número 457272/2014-7.

Bessani et al. 2014]. Esta abordagem, embora adequada, pode limitar o desempenho do sistema quando consideramos servidores atuais que possuem um *hardware* com múltiplos núcleos, i.e., tendo apenas uma *thread* de execução, a quantidade de núcleos deixa de ser um fator que influência no desempenho do sistema uma vez que apenas um núcleo é utilizado.

Com o objetivo de obter proveito de arquiteturas com múltiplos núcleos (*multi-cores*) e melhorar o desempenho do sistema, recentemente surgiram abordagens para RME que empregam protocolos que suportam múltiplas *threads* de execução [Marandi et al. 2014, Marandi and Pedone 2014, Mendizabal et al. 2014, Kotla and Dahlin 2004, Zbierski 2015], de forma que várias requisições de clientes são executadas em paralelo nas réplicas. A ideia básica destas abordagens, chamadas de RME paralelas (ou semi-paralelas quando as ordenações são sequenciais), é classificar as requisições em dependentes (ou conflitantes) e independentes, sendo que requisições independentes são executadas em paralelo nas réplicas. Duas requisições são independentes quando acessam diferentes variáveis ou quando apenas leem o valor de uma mesma variável. Por outro lado, duas requisições são dependentes quando acessam pelo menos uma mesma variável e pelo menos uma das requisições altera o valor desta variável.

Mesmo com todos estes trabalhos sobre RME paralelas, ainda falta uma implementação que possa ser usada no desenvolvimento de aplicações. Este trabalho visa preencher esta lacuna e apresenta os esforços para adicionar suporte a execuções paralelas no sistema BFT-SMART [Bessani et al. 2014], que é uma concretização de uma RME tolerante a falhas tanto por parada quanto bizantinas. A abordagem utilizada nesta implementação baseou-se na adaptação do protocolo proposto em [Marandi et al. 2014] para emprego no BFT-SMART através do uso de uma espécie de paralelizador semelhante ao proposto em [Kotla and Dahlin 2004].

Resumidamente, as contribuições deste trabalho são as seguintes: *(i)* descrição de como o protocolo para RME paralela proposto em [Marandi et al. 2014] foi adaptado e implementado no BFT-SMART e *(ii)* apresentação e análise de uma série de experimentos com as implementações realizadas, trazendo uma melhor compreensão a respeito do funcionamento de uma RME com execuções paralelas.

2. Replicação Máquina de Estados

Em uma replicação Máquina de Estados [Schneider 1990], as réplicas devem apresentar a mesma evolução em seus estados: *(i)* partindo de um mesmo estado e *(ii)* executando o mesmo conjunto de requisições na mesma ordem, *(iii)* todas as réplicas devem chegar ao mesmo estado final, definindo o determinismo de réplicas. Os ítens *(i)* e *(iii)* são facilmente garantidos. Para prover o item *(i)*, basta iniciar todas as réplicas com o mesmo estado (i.e., iniciar todas as variáveis que representam o estado com os mesmos valores nas diversas réplicas). Para prover o item *(iii)*, é necessário que as operações executadas pelas réplicas sejam deterministas, i.e., que a execução de uma mesma operação (com os mesmos parâmetros) produza o mesmo resultado nas diversas réplicas e, além disso, o estado resultante (a mudança no estado) da execução desta operação deve ser o mesmo nas várias réplicas do sistema.

Já garantir o item *(ii)* envolve a utilização de um protocolo de difusão atômica (Figura 1). O problema da *difusão atômica* [Hadzilacos and Toueg 1994], também conhecido como difusão com ordem total, consiste em fazer com que todos os processos corretos de um grupo entreguem todas as mensagens difundidas neste grupo na mesma ordem. A Figura 1 mostra um exemplo onde dois clientes estão concorrentemente acessando quatro servidores. Neste caso, através de um protocolo de difusão atômica, suas requisições são entregues para serem executadas na mesma ordem pelos servidores, i.e., caso um servidor execute primeiro a requisição $op1$

(requisição do cliente 1) e depois a requisição $op2$ (requisição do cliente 2), então todos os outros servidores também executarão primeiramente $op1$ e depois $op2$, mantendo a ordem de entrega na execução.

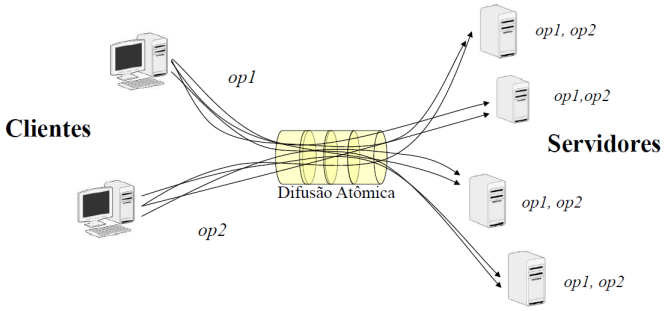


Figura 1. Replicação Máquina de Estados.

A execução do protocolo de difusão atômica envolve a execução de um protocolo de consenso [Castro and Liskov 2002, Bessani et al. 2014], de forma que os processos entrem em acordo (propriedade fundamental do consenso [Hadzilacos and Toueg 1994]) acerca da ordem de entrega das mensagens (requisições). Consequentemente, a complexidade de trocas de mensagens deste protocolo é $O(n^2)$, i.e., todas as réplicas precisam enviar e receber mensagens de todas as outras réplicas, fazendo com que este protocolo geralmente seja tratado como o gargalo de uma RME. De fato, muitos trabalhos foram propostos com o objetivo de diminuir os custos associados a este protocolo, como por exemplo [Lamport 2006, Marandi et al. 2010, Abd-El-Malek et al. 2005, Cowling et al. 2006, Kotla et al. 2009, Guerraoui et al. 2010].

Um aspecto que recentemente começou a ser explorado é a otimização dos procedimentos necessários para a execução das requisições depois de ordenadas [Marandi et al. 2014, Marandi and Pedone 2014, Mendizabal et al. 2014, Kotla and Dahlin 2004]. Como veremos na Seção 3, o objetivo principal deste trabalho é exatamente explorar a semântica das requisições para tentar melhorar o desempenho do sistema na etapa de execução das requisições, fazendo com que mais de uma requisição seja executada em paralelo nas réplicas.

2.1. BFT-SMaRt: Implementação de Replicação Máquina de Estados

O BFT-SMaRt [Bessani et al. 2014] representa a concretização de uma replicação Máquina de Estados [Schneider 1990] tolerante tanto a falhas por parada quanto bizantinas. Esta biblioteca *open-souce* de replicação foi desenvolvida na linguagem de programação Java e implementa um protocolo similar aos outros protocolos para tolerância a falhas (ex.: [Castro and Liskov 2002] para tolerar falhas bizantinas ou [Oki and Liskov 1988, Liskov and Cowling 2012] para tolerar apenas *crashes*). Atualmente, o BFT-SMaRt implementa protocolos para reconfiguração, *checkpoints* e transferência de estados, tornando-se assim uma biblioteca completa para RME, a qual foi desenvolvida seguindo os seguintes princípios [Bessani et al. 2014]:

1 *Modelo de falhas configurável* – é possível configurar o sistema para tolerar falhas bizantinas ou apenas *crashes*;
2 *Simplicidade* – não usa otimizações que aumentem a complexidade da implementação;
3 *Modularidade* – o BFT-SMaRt foi projetado de forma modular, apresentando uma notável separação entre os protocolos implementados (consenso, difusão atômica, *checkpoints*, transferência de estado e reconfiguração);
4 *API simples e extensível* –toda a complexidade de uma RME é encapsulada em uma API simples que pode ser estendida para implementação de uma aplicação mais complexa; e

5 *Proveito de arquiteturas multi-core* – como veremos adiante, a arquitetura do BFT-SMART já se aproveita de arquiteturas com múltiplos núcleos para otimizar a ordenação sequencial de requisições, neste sentido este trabalho melhora o BFT-SMART trazendo estes benefícios também para os procedimentos de execução das requisições.

O BFT-SMART assume um modelo de sistema usual para replicação Máquina de Estados [Castro and Liskov 2002, Bessani et al. 2014]: $n \geq 3f + 1$ (resp. $n \geq 2f + 1$) servidores para tolerar f falhas bizantinas (resp. *crashes*); um número ilimitado de clientes que podem falhar; e um sistema parcialmente síncrono para garantir terminação. Através de reconfigurações [Alchieri et al. 2013], tanto n quanto f podem sofrer alterações durante a execução. O sistema ainda necessita de canais ponto-a-ponto confiáveis para comunicação, que são implementados usando MACs (*message authentication codes*) sobre o TCP/IP. Além disso, pode-se configurar os clientes para assinar suas requisições, garantindo-se autenticação.

2.1.1. Arquitetura Básica do BFT-SMART

Esta seção discute os aspectos principais da arquitetura do BFT-SMART, uma discussão mais detalhada pode ser encontrada em [Bessani et al. 2014]. A arquitetura de cada réplica do sistema é mostrada na Figura 2 (que já apresenta as alterações introduzidas para execuções paralelas que serão discutidas adiante). De uma forma geral, podemos dividir esta arquitetura em três grandes partes: recebimento, ordenamento e execução de requisições.

Recebimento de Requisições. Os clientes acessam as réplicas através de um conjunto de *threads* (*Netty thread pool* – uma *thread* é iniciada para cada cliente), sendo que as requisições de cada cliente são adicionadas em uma fila separada. A autenticidade das requisições é garantida por meio de assinaturas digitais, i.e., os clientes assinam suas requisições (lembrando que o sistema pode ser configurado para não utilizar este mecanismo). Desta forma, qualquer réplica é capaz de verificar a autenticidade das requisições e uma proposta para ordenação, a qual contém a requisição a ser ordenada, somente é aceita por uma réplica correta após a autenticidade desta requisição ser verificada.

Ordenamento de Requisições. Sempre que existirem requisições para serem executadas, uma *thread* (*proposer*) iniciará uma instância do consenso para definir uma ordem de entrega para um lote de requisições. Durante este processo, a réplica se comunica com as outras através de um outro conjunto de *threads*, sendo que cada uma destas *threads* é responsável pela conexão com uma das outras n réplicas. Além disso, existe um conjunto de *threads* (também uma para cada réplica) responsável por receber as mensagens enviadas pelas outras réplicas. Todo o processamento necessário para garantir a autenticidade destas mensagens é executado nestas *threads*. No BFT-SMART a ordenação é sequencial, i.e., uma nova instância do consenso só é inicializada após a instância anterior ter terminado. Porém, a ordenação de requisições em lotes visa aumentar o desempenho do sistema. Além disso, a utilização destes conjuntos de *threads* faz com que o BFT-SMART aproveite arquiteturas *multi-cores* para melhorar o desempenho do processo de ordenação de requisições.

Caso uma requisição não seja ordenada dentro de um determinado tempo, o sistema troca a réplica líder. Um tempo limite para ordenação é associado a cada requisição r recebida em cada réplica i. Caso este tempo se esgotar, i envia r para todas as réplicas e define um novo tempo para sua ordenação. Isto garante que todas as réplicas recebem r, pois um cliente malicioso pode enviar r apenas para alguma(s) réplica(s), tentando forçar uma troca de líder.

Caso este tempo se esgotar novamente, i solicita a troca de líder, que apenas é executada após $f + 1$ réplicas solicitarem esta troca, impedindo que uma réplica maliciosa force trocas de líder.

Uma abordagem que visa aumentar o desempenho do sistema é a execução em paralelo de várias instâncias do algoritmo de consenso [Marandi et al. 2014], de modo que várias requisições (ou lotes de requisições) sejam ordenadas simultaneamente. No entanto, esta abordagem traz mais complexidade aos protocolos e possibilita que líderes faltosos degradem o desempenho do sistema, uma vez que é necessário executar instâncias do consenso (inicializadas por estes processos maliciosos) mesmo quando as propostas são para definir a ordem de entrega de requisições forjadas. Neste caso, a instância do consenso define a ordem para uma requisição *nop* (*no operation*), o que é necessário para não "travar" a entrega das mensagens [Castro and Liskov 2002]. Conforme já comentado, o BFT-SMART opta por executar as instâncias do consenso de forma sequencial e preserva o desempenho do sistema através da ordenação em lotes [Bessani et al. 2014], de forma que o custo da execução fica diluído entre as requisições do lote.

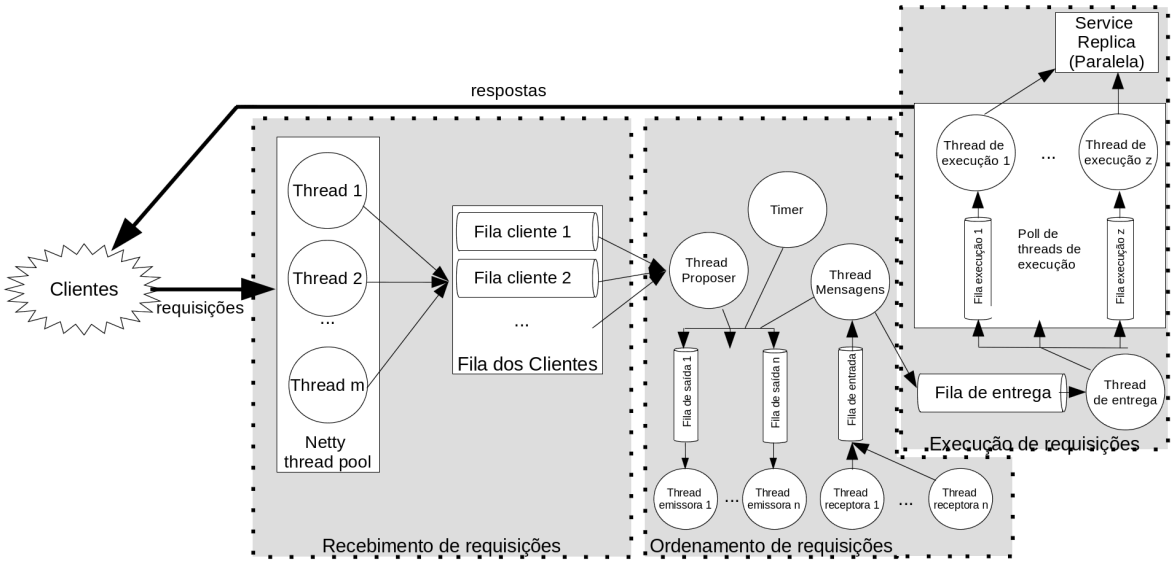


Figura 2. Arquitetura de uma Réplica do BFT-SMART.

Execução de Requisições. Quando a ordem de execução de um lote de requisições é definida, este lote é adicionado em uma fila para então ser entregue à aplicação por uma outra *thread* (*Thread de entrega*). Após o processamento da cada requisição, uma resposta é enviada ao cliente que solicitou tal requisição. O cliente, por sua vez, determina que uma resposta para sua requisição é válida assim que o mesmo receber pelo mesmo $f + 1$ respostas iguais (ou a primeira resposta caso o sistema esteja configurado para tolerar apenas *crashes*), garantindo que pelo menos uma réplica correta obteve tal resposta.

Note que a Figura 2 já apresenta a nova arquitetura do BFT-SMART, com as múltiplas (*pool*) *threads* de execução. Na Seção 3 é dicutido como podemos coordenar a execução destas várias *threads* de forma que mais do que uma requisições seja executada em paralelo.

2.1.2. Usando o BFT-SMART

A forma de utilização do BFT-SMART, para programação de uma aplicação tolerante a falhas através de replicação Máquina de Estados, é bastante simples. O Algoritmo 1 apresenta a API

básica para clientes e servidores, mostrando a classe que deve ser instanciada pelo clientes para acessar o sistema, bem como a interface que deve ser estendida pelos servidores para implementar o serviço replicado.

Algoritmo 1 API do BFT-SMART para clientes e servidores.

```
1  //API do Cliente
2  public class ServiceProxy {
3    public ServiceProxy(int id){
4       ...
5    }
6
7    public byte[] invokeOrdered(byte[] request){
8      ...
9    }
10 }
11
12 //API do Servidor
13 public class MyServer extends Executable {
14
15   public MyServer(int id){
16      new ServiceReplica(id, this, ...);
17   }
18
19   public byte[] executeOrdered(byte[] request, MsgContext ctx){
20      //CÓDIGO DA APLICAÇÃO
21   }
22 }
```

Para acessar o serviço replicado, um cliente do BFT-SMART apenas deve instanciar uma classe *ServiceProxy* fornecendo seu identificador (inteiro) e um arquivo de configuração contendo o endereço (IP e porta) de cada um dos servidores. Após isso, sempre que o cliente desejar enviar alguma requisição para as réplicas (servidores), o mesmo deve invocar o método *invokeOrdered* especificando a requisição (serializada em um *array* de *bytes*).

Por outro lado, para implementar o servidor, cada réplica deve estender a interface *Executable* e implementar o método abstrato *executeOrdered* que é invocado quando uma requisição deve ser executada. Além disso, é necessário instanciar uma *ServiceReplica* que representa propriamente a réplica, fornecendo o identificador (inteiro) da réplica que é mapeado para uma porta e endereço IP através de um arquivo de configuração.

Conforme já comentado, esta é a API básica do BFT-SMART que já é suficiente para implementar uma aplicação tolerante a falhas. No entanto, esta API é muito mais rica, apresentando também métodos para o gerenciamento do estado das réplicas (caso deseja-se empregar recuperação de réplicas). Uma descrição mais aprofundada sobre a API do BFT-SMART pode ser encontrada em [Bessani et al. 2014].

3. Execuções Paralelas no BFT-SMART

Este seção descreve como o protocolo para RME paralela proposto em [Marandi et al. 2014] foi adaptado e implementado no BFT-SMART. De uma forma geral, em [Marandi et al. 2014] as *threads* das réplicas formam grupos, que são definidos de acordo com as dependências, e as requisições são ordenadas e executadas em paralelo nestes grupos. Uma requisição pode ter conflito com mais de um grupo e a requisição é ordenada e "executada" em todos estes grupos (uma única *thread* executa a requisição e as outras apenas a aguardam).

Como no BFT-SMART as instâncias de consenso são executadas de forma sequencial e todos os processos participam da ordenação formando um único grupo de ordenação, o cliente

não precisa definir os grupos de *threads* de execução para enviar a requisição a ser ordenada e executada. No entanto, o cliente deve fornecer o identificador do grupo para que cada réplica faça um mapeamento da requisição para as *threads* correspondentes ao grupo. Para isso, a nova API do BFT-SMART (Algoritmo 2) introduz o método *invokeParallel* no cliente. Já no lado do servidor (réplicas), para execuções paralelas basta criar uma *ParallelServiceReplica* fornecendo o número $numThreads$ de *threads* de execução (ao invés de uma *ServiceReplica*, que deve ser utilizada para execução sequencial).

Algoritmo 2 Atualizações na API do BFT-SMART para clientes e servidores.

```
1  //API do Cliente
2  public class ServiceProxy {
3
4      public byte[] invokeParallel(byte[] request, int groupId){
5
6  }
7
8  //API do Servidor
9  public class MyServer extends Executable {
10
11     public MyServer(int id, int numThreads){
12         new ParallelServiceReplica(id, numThreads, this, ...);
13         //Definir grupos de conflito caso necessário
14     }
15
16 }
```

Cada *thread* criada recebe um identificador de 0 até $numThreads - 1$ e possui associada uma fila de requisições para serem executadas (ver Figura 2). Por padrão, cada *thread* faz parte de um grupo representado pelo seu identificador: por exemplo, a *thread* 0 faz parte do grupo com identificador 0 e este grupo não possui outras *threads*. Também são criados um grupo para requisições que são conflitantes com todas as outras requisições (`CONFLICT_ALL`) e um grupo para requisições que não são conflitantes com nenhuma outra requisição (`CONFLICT_NONE`), totalizando $numThreads + 2$ grupos. Ainda é possível definir novos grupos que executarão requisições conflitantes (método $addExecutionConflictGroup$ da *ParallelServiceReplica* – Algoritmo 3). Note que a requisição é conflitante dentro do grupo, mas não interfere nos outros grupos. Os servidores armazenam os seguintes dados para cada grupo: identificador do grupo, identificadores das *threads* pertencentes ao grupo e uma barreira para sincronização.

O Algoritmo 3 (método *decided* da *ParallelServiceReplica*, que é executado pela *Thread de entrega* – Figura 2) representa uma espécie de paralelizador [Kotla and Dahlin 2004] que distribui as requisições para as *threads* de execução de acordo com as dependências. Quando um lote de requisições (`TOMMessage`) é decidido, cada requisição de clientes r é adicionada na fila das *threads* segundo quatro critérios:

- Caso r não seja confitante com nenhuma outra requisição (linhas 24-26), então uma única *thread* é escolhida para executar r. Atualmente, esta escolha segue o algoritmo *round-robin*, mas é possível implementar mecanismos mais sofisticados para balanceamento de carga.
- Caso r seja confitante com qualquer outra requisição (linhas 27-30), então r é adicionada em todas as filas.
- Caso r seja destinada a um grupo padrão composto por uma determinada *thread* t (linhas 31-32), então r é adicionada na fila de t.
- Caso r seja destina a algum outro grupo (linhas 33-38), então r é adicionada nas filas de cada *thread* deste grupo.

Algoritmo 3 Algoritmo de distribuição das requisições.

```
1  public class ParallelServiceReplica extends ServiceReplica {
2
3     private int nextThread = 0;
4     private Executor exec;
5     private LinkedBlockingQueue[] queues;
6
7     public ParallelServiceReplica (int id, int numThreads, Executor exec, ...){
8          this.exec = exec;
9          queues = new LinkedBlockingQueue[numThreads];
10         for (int i = 0; i < mapping.getNumThreads(); i++) {
11            queues[i] = new LinkedBlockingQueue();
12            new ParallelServiceReplicaWorker(queues[i], exec, i).start();
13         }
14         //cria as barreiras para requisições CONFLICT_ALL (caB) e para reconfigurações (recB)
15    }
16
17    public boolean addExecutionConflictGroup(int groupId, int[] threadsId){
18       ...
19    }
20
21    public void decided(TOMMessage[] requests) {
22       for (TOMMessage request : requests) {
23          if (request.getType() == ORDERED_REQUEST) {
24              if (request.getGroupId() == CONFLICT_NONE) {
25                queues[nextThread].put(request);
26                nextThread = (nextThread + 1) % getNumThreads();
27              }else if (request.getGroupId() == CONFLICT_ALL) {
28                for (LinkedBlockingQueue q : queues) {
29                   q.put(request);
30                }
31              }else if (request.getGroupId() < getNumThreads()) {
32                   queues[request.getGroupId()].put(request);
33               } else {
34                   LinkedBlockingQueue[] q = getQueues(request.getGroupId()); //filas do grupo
35                   for (LinkedBlockingQueue q1 : q) {
36                      q1.put(request);
37                   }
38               }
39         } else if (request.getType() == RECONFIG) {
40              enqueueUpdate(request); //for reconfiguration
41         }
42      }
43      if (hasUpdates()) {
44         TOMMessage reconfReq = new TOMMessage(..., RECONFIG);
45         for (LinkedBlockingQueue q : queues) {
46            q.put(reconfReq);
47         }
48      }
49    }
50 }
```

Por outro lado, caso a requisição seja para reconfiguração do sistema, a mesma é armazenada (linha 40) para ser processada juntamente com outras requisições de reconfiguração contidas no lote [Alchieri et al. 2013] e, no final, uma requisição é adicionada em todas as filas (linhas 43-48) para que as *threads* parem durante a execução da reconfiguração.

O Algoritmo 4 apresenta o código executado por cada *thread* de execução (ver Figura 2). Sempre que uma requisição r estiver disponível na fila de execução de uma determinada *thread* t, os seguintes casos são possíveis: *(1)* r não é confitante com nenhuma outra requisição ou foi destinada ao grupo padrão composto por t, então t executa r (linhas 17-19); *(2)* r representa uma requisição de reconfiguração, então t apenas pára sua execução durante a reconfiguração (linhas 20-22); e *(3)* r foi destinada a um grupo, então t determina se é a *thread* que irá exe-

cutar r (sempre a *thread* de menor identificador no grupo é escolhida) ou apenas aguardar pela execução de r por alguma outra *thread* do grupo (linhas 23-33).

Algoritmo 4 Algoritmo de execução de requisições.

```
1  public class ParallelServiceReplicaWorker extends Thread {
2
3        private LinkedBlockingQueue myQueue;
4        private int myId;
5        private Executor exec;
6
7        public ServiceReplicaWorker(LinkedBlockingQueue queue, Executor exec, int id) {
8           this.myQueue = queue;
9           this.exec = exec;
10          this.myId = id;
11       }
12
13       public void run() {
14          TOMMessage req = null;
15          while (true) {
16                req = myQueue.take(); //bloqueia até ter uma requisição para executar
17                if (req.getGroupId() == CONFLICT_NONE || req.getGroupId() == myId) {
18                   byte[] resp = exec.executeOrdered(req.getContent(), req.getContext());
19                   sendReply(resp, req.getContext());
20                }else if(req.getGroupId() == RECONFIG){
21                   recB.await(); // sinaliza que vai aguardar a reconfiguração
22                   recB.await(); //aguarda a execução da reconfiguração
23                }else { //requisição com conflito (entre um grupo ou todas as threads)
24                   if (myId == getExecutorThread(req.getGroupId())) { // é a executora
25                      getBarrier(req.getGroupId()).await(); //aguarda as outras threads
26                      byte[] resp = exec.executeOrdered(req.getContent(), req.getContext());
27                      sendReply(resp, req.getContext());
28                      getBarrier(req.getGroupId()).await(); //sinaliza fim da execução
29                   } else { // apenas deve parar e aguardar a execução
30                      getBarrier(req.getGroupId()).await(); //sinaliza que vai aguardar
31                      getBarrier(req.getGroupId()).await(); //aguarda o final da execução
32                   }
33                }
34          }
35       }
```

Veja que várias *threads* podem executar paralelamente requisições destinadas a diferentes grupos (ou `CONFLICT_NONE`). Porém, todas as *threads* de um mesmo grupo devem estar envolvidas (executando ou aguardando) na execução de uma requisição destinada a tal grupo.

4. Experimentos

Visando analisar o desempenho da implementação desenvolvida bem como o comportamento de uma RME com execuções paralelas, alguns experimentos foram realizados no Emulab [White et al. 2002], em um ambiente consistindo por 4 máquinas *d710* (2.4 GHz 64-*bit* Intel Quad Core Xeon E5530 com 2 *threads* por núcleo, 12GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gb. O BFT-SMART foi configurado com $n = 3$ servidores para tolerar até uma falha por parada (*crash*). Cada servidor executou em uma máquina separada, enquanto que 100 clientes executaram na outra máquina. O ambiente de *software* utilizado foi o sistema operacional Ubuntu 12.04 LTS 64-*bit* com *kernel* 3.2.0-56 e máquina virtual Java de 64 *bits* versão 1.7.0_75. Todos os experimentos realizados tiveram uma fase de *warm-up*.

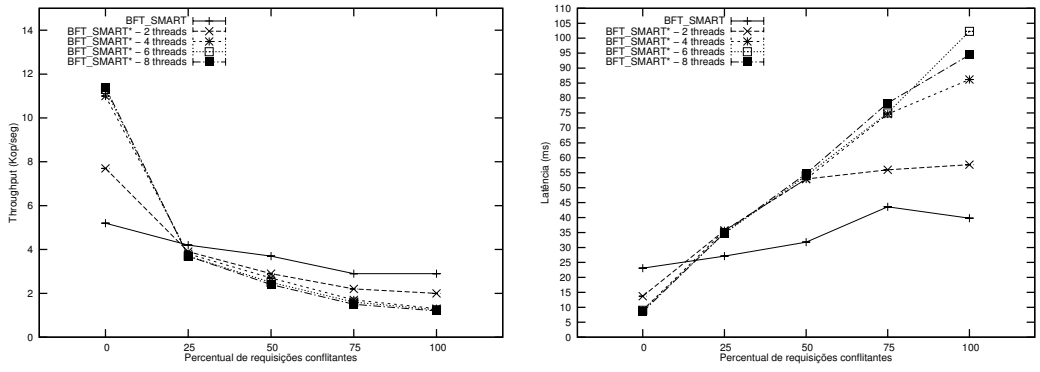
Duas aplicações foram desenvolvidas e tiveram seus desempenhos analisados. A latência e o *throughput* das operações para acesso a uma lista e a um espaço de tuplas foram determinados, com o objetivo de comparar o desempenho do sistema quando configurado para execuções sequenciais com quando configurado para execuções paralelas. Posteriormente,

diferentes estratégias para classificar as operações como dependentes foram analisadas para o espaço de tuplas, com o objetivo de verificar como as escolhas do desenvolvedor de uma aplicação pode afetar o seu desempenho. A latência foi medida em um dos clientes e os valores apresentados representam a média de 1000 execuções, excluindo-se 10% dos valores com maior devio. Já o *throughput* apresentado é o pico atingido pelos servidores, medido no servidor líder do consenso [Bessani et al. 2014] a cada 5000 requisições.

4.1. Lista Encadeada

Esta aplicação foi implementada no servidores através de uma lista encadeada (*LinkedList*), que é uma estrutura de dados não sincronizada, i.e., caso duas ou mais *threads* acessem esta estrutura concorrentemente e pelo menos uma delas modifique a sua estrutura, então estes acessos devem ser sincronizados. Neste experimento, a lista foi utilizada para armazenar inteiros (*Integer*) e as seguintes operações foram implementadas para seu acesso: *boolean add (Integer i)* – adiciona i no final da lista e retorna *true* caso i ainda não esteja na lista, retorna *false* caso contrário; *boolean remove(Integer i)* – remove i e returna *true* caso i esteja na lista, retorna *false* caso contrário; *Integer get(int index)* – retorna o elemento da posição indicada; e *boolean contains(Integer i)* – retorna *true* caso i esteja na lista, retorna *false* caso contrário. Note que todas estas operações possuem um custo de $O(n)$, mas a operação add tende a demorar mais pois primeiro percorre toda a lista para verificar se determinado dado já está armazenado. As seguintes dependências foram definidas para estas operações: add e $remove$ são dependentes de todas as outras operações (`CONFLICT_ALL`), enquanto que get e $contains$ não possuem dependências (`CONFLICT_NONE`), somente as já definidas com add e $remove$.

A lista foi inicializada com 100k entradas em cada réplica e os valores para as operações get, $remove$ e $contains$ foram selecionados aleatóriamente seguindo uma distribuição uniforme, enquando que para a operação add foram escolhidos valores que ainda não estavam na lista. Os *workloads* foram gerados "estaticamente" antes da execução dos experimentos, possibilitando que cada uma das abordagens executasse o mesmo conjunto de operações.



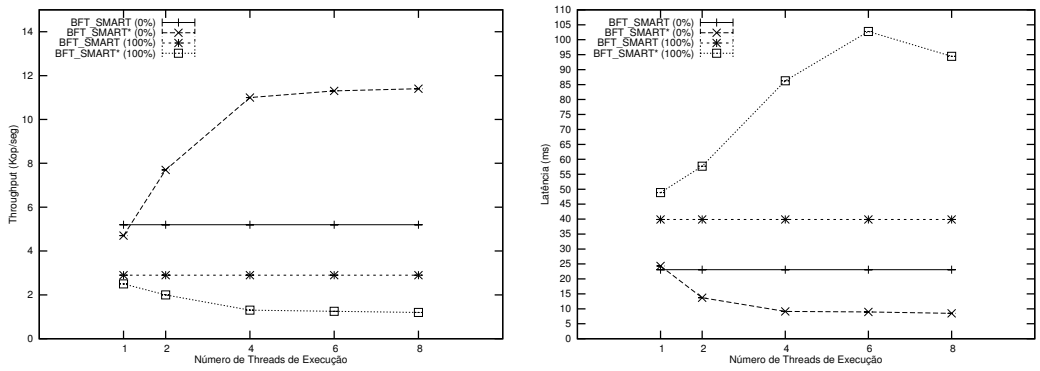
(a) *Throughput*. (b) Latência.

Figura 3. Desempenho para diferentes *workloads*.

A Figura 3 apresenta o *throughput* e a latência de acesso a esta lista considerando *workloads* com diferentes percentagens de operações conflitantes (add e $remove$). São apresentados os valores tanto para o sistema configurado para execuções sequenciais (BFT_SMART) quanto para execuções paralelas (BFT_SMART*). Nesta figura é possível perceber que quando não temos operações conflitantes (0%), execuções paralelas aumentaram o desempenho do sistema em aproximadamente 2.2 vezes com 8 *threads* de execução. Além disso, mostrou-se vantajoso utilizar execuções paralelas para *workloads* com até aproximadamente 25% das operações

conflitantes. Após isso, o tempo necessário para sincronizar as *threads* (para executar uma requisição conflitante) faz com que o desempenho seja melhor no modo sequencial.

A Figura 4 apresenta os resultados para os *workloads* de 0% e 100% conflitantes, considerando diferentes quantidades de *threads* de execução. Em nossos experimentos utilizamos até 8 *threads* pois era o número de núcleos disponíveis. Podemos perceber que execuções paralelas são promissoras para aplicações onde o conjunto de operações não possua muitas dependências (conflitos). Comportamento similar a este também foi observado em [Marandi et al. 2014].



(a) *Throughput*. (b) Latência.

Figura 4. Desempenho variando a quantidade de *threads* de execução.

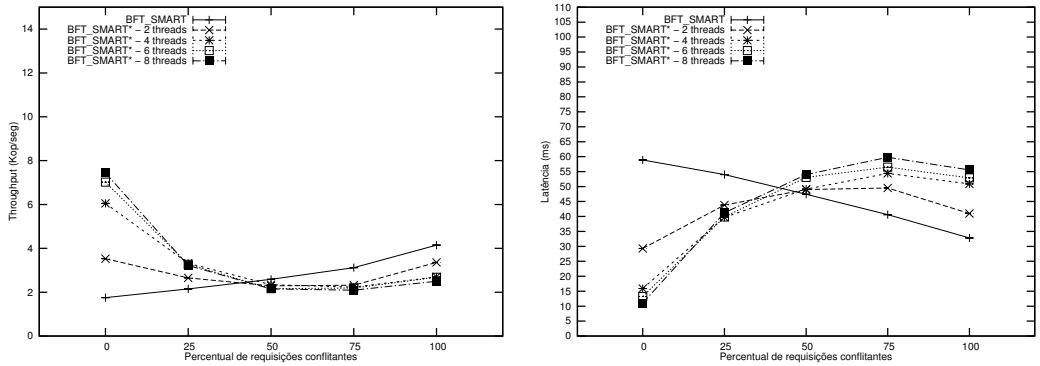
4.2. Espaço de Tuplas

Um espaço de tuplas (TS) [Gelernter 1985] pode ser visto como uma memória associativa que armazena um conjunto de tuplas, sendo que uma tupla é uma sequência ordenada de campos. O TS foi implementado nos servidores através do auxílio de uma série de listas (uma para cada conjunto de tuplas com a mesma quantidade de campos) e para seu acesso as seguintes operações foram implementadas: *out(Tuple t)* – adiciona a tupla t no TS; *Tuple rdp(Tuple t)* – retorna uma tupla que combine com o molde t[1] ou *null* caso nenhuma tupla combine com t; e *Tuple inp(Tuple t)* – mesma semântida do rdp mas neste caso a tupla é removida do TS. As operações inp e rdp têm custo $O(n)$ enquando que o custo do out é $O(1)$. As operações out e inp foram definidas como dependentes de todas as outras operações, enquanto que uma requisição rdp não é conflitante com outras operações de rdp.

Neste experimento, utilizou-se tuplas com 1 até 10 campos (para cada campo foi utilizado um *String* de tamanho 50) e o TS foi inicializado com 100k tuplas de 1 até 10 campos (10k para cada quantidade de campos). Os moldes ou tuplas de todas as operações foram selecionados aleatóriamente seguindo uma distribuição uniforme (tanto o número de campos quanto o seu conteúdo). Novamente, os *workloads* foram gerados antes da execução dos experimentos e todas as abordagens executaram o mesmo conjunto de operações.

A Figura 5 apresenta o *throughput* e a latência de acesso ao TS considerando *workloads* com diferentes percentagens de operações conflitantes (out e inp), tanto para execuções sequenciais (BFT_SMART) quanto paralelas (BFT_SMART*). Podemos perceber que sem operações conflitantes (0%), execuções paralelas aumentaram o desempenho do sistema em aproximadamente 4.25 vezes com 8 *threads* de execução, mantendo vantagem sobre execuções sequenciais para *workloads* com até aproximadamente 50% das operações conflitantes.

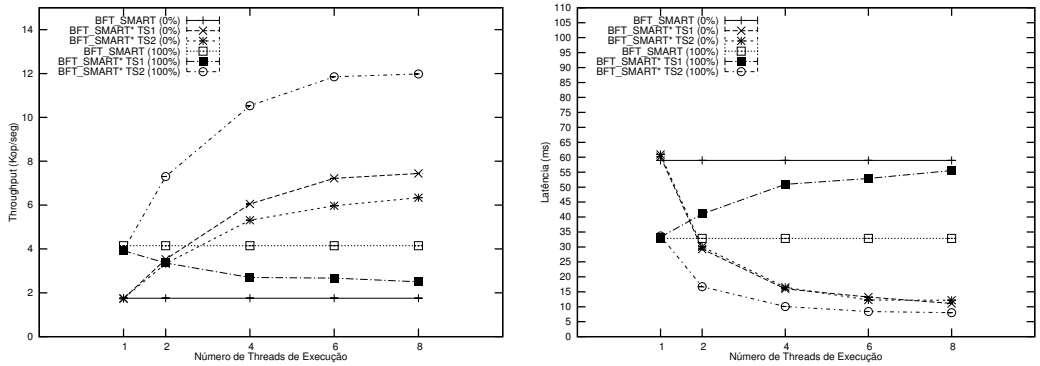
[1]Uma tupla e um molde combinam caso possuam a mesma quantidade de campos e todo campo definido no molde seja igual ao campo correspondente na tupla [Gelernter 1985].



(a) *Throughput*. (b) Latência.

Figura 5. Desempenho para diferentes *workloads*.

Para verificar como as escolhas podem afetar o desempenho, analisamos outra estratégia para determinar as dependências destas operações: uma operação *out(Tuple t)*, *rdp(Tuple t)* ou *inp(Tuple t)* somente é conflitante com qualquer outra operação cujo parâmetro seja uma tupla ou molde com a mesma quantidade de campos de t. Esta estratégia é possível pelo fato desta implementação usar uma lista para cada conjunto de tuplas com a mesma quantidade de campos. Assim, 10 grupos foram criados e distribuídos entre as *threads* de execução, de forma que as requisições endereçadas a um determinado grupo foram executadas pela mesma *thread*.



(a) *Throughput*. (b) Latência.

Figura 6. Desempenho variando a quantidade de *threads* de execução.

A Figura 6 apresenta os resultados para os *workloads* de 0% e 100% conflitantes, considerando diferentes quantidades de *threads* de execução e as abordagens sequencial (BFT_SMART), paralela (BFT_SMART* TS1) e paralela com a novas definições de dependências (BFT_SMART* TS2). Podemos perceber que para a nova configuração de dependências o sistema apresenta melhor desempenho quando temos operações conflitantes e desempenho similar quando não temos operações conflitantes. Por exemplo, quando comparado com a abordagem sequencial e considerando o *workload* de 100% conflitantes, a abordagem anterior apresentou um decréscimo no desempenho para até aproximadamente 0.6 vezes, enquando que a nova abordagem aumentou o desempenho para aproximadamente até 2.9 vezes.

5. Lições Aprendidas

Esta seção discute alguns apectos observados durante o desenvolvimento deste trabalho. Primeiramente, como podemos perceber nos experimentos, utilizar RME paralela (ou semi-paralela) é

mais vantajoso quando o *workload* possui poucas requisições dependentes. De fato, sempre que uma requisição com dependências precisar ser executada, as *threads* envolvidas neste conflito devem ser sincronizadas, o que possui um custo associado (além de uma aguardar pelas outras, enquanto uma *thread* executa a requisição as outras ficam paradas).

Outro faltor que deve ser considerado é o tempo necessário para os servidores executarem uma requisição, a tendência é que quanto maior for este custo mais vantajoso seja utilizar execuções paralelas. Por exemplo, no experimento com a lista, caso a lista seja inicializada com apenas 10k inteiros, considerando o *workload* com 0% conflitantes e 8 *threads* de execução, o desempenho do sistema diminui para 0.40 vezes quando comparado com execuções sequenciais (inicializando com 100k inteiros teve um aumento de 2.2 vezes). Por outro lado, inicializando a lista com 1000k inteiros, considerando o *workload* com 100% conflitantes e 8 *threads* de execução, as duas abordagens apresentam o mesmo desempenho (inicializando com 100k inteiros o desempenho diminui para 0.42 vezes).

Ainda com relação ao desempenho, como podemos perceber no experimento com espaços de tuplas, o projeto das dependências entre as operações é fundamental para obter o máximo desempenho do sistema. Por tudo isso, é dificil determinar a configuração ideal para determinada aplicação, pois vários aspectos relacionados com o *workload* vão determinar qual é a melhor escolha. No caso do BFT-SMART, como as ordenações são sequenciais, o uso de execuções paralelas apenas será vantajoso quando uma única *thread* de execução não consegue executar todas as requisições já ordenadas (lote) antes do final da ordenação do próximo lote. No entanto, como o BFT-SMART utiliza uma arquitetura com múltiplas *threads* para ordenação (Figura 2), os efeitos da ordenação sequencial tendem a ser minimizados.

Dentre as aplicações que poderiam obter proveito de execuções paralelas para melhorar seus desempenhos, vale a pena destacar aquelas que acessam arquivos (banco de dados ou serviços duráveis [Bessani et al. 2013]), que utilizam funções criptográficas (por exemplo, o DepSpace [Bessani et al. 2008]) ou ainda que acessam outros serviços disponíveis na Internet.

6. Conclusões

Este artigo apresentou os esforços empregados na adição de suporte a execuções paralelas no BFT-SMART, bem como um conjunto de experimentos que ajudaram a elucidar o seu funcionamento. De acordo com a classificação dada em [Marandi et al. 2014], o BFT-SMART tornou-se uma biblioteca para RME semi-paralela, visto que a ordenação é sequencial (mesmo utilizando várias *threads* nesta parte dos protocolos – Figura 2). Por fim, esta é uma abordagem para RME relativamente nova e explorar todo o seu potêncial é uma tarefa ainda bastante desafiadora.

Referências

Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., and Wylie, J. (2005). Fault-scalable Byzantine fault-tolerant services. In *ACM Symposium on Operating Systems Principles*.

Alchieri, E. A. P., Bessani, A. N., and da Silva Fraga, J. (2013). Replicação máquina de estados dinâmica. In *Anais do XIV Workshop de Teste e Tolerância a Falhas- WTF 2013*.

Bessani, A., Santos, M., Felix, J. a., Neves, N., and Correia, M. (2013). On the efficiency of durable state machine replication. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 169–180. USENIX Association.

Bessani, A., Sousa, J., and Alchieri, E. (2014). State machine replication for the masses with BFT-SMaRt. In *Proc. of the International Conference on Dependable Systems and Networks*.

Bessani, A. N., Alchieri, E. A. P., Correia, M., and da Silva Fraga, J. (2008). Depspace: a byzantine fault-tolerant coordination service. *SIGOPS Oper. Syst. Rev.*, 42(4):163–176.

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

Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006). HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112.

Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. (2010). The next 700 BFT protocols. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2009). Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39.

Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks*.

Lamport, L. (2006). Fast paxos. *Distributed Computing*, 19(2).

Liskov, B. and Cowling, J. (2012). Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT.

Marandi, P. J., Bezerra, C. E., and Pedone, F. (2014). Rethinking state-machine replication for parallelism. In *Proceedings of the 34th Int. Conference on Distributed Computing Systems*.

Marandi, P. J. and Pedone, F. (2014). Optimistic parallel state-machine replication. In *Proceedings of the 33rd International Symposium on Reliable Distributed Systems*.

Marandi, P. J., Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

Mendizabal, O., Marandi, P. J., Dotti, F., and Pedone, F. (2014). Recovery in parallel state-machine replication. In *Proc. of Int. Conference on Principles of Distributed Systems*.

Oki, B. and Liskov, B. (1988). Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17.

Schneider, F. B. (1990). Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symp. on Operating Systems Design and Implementations*.

Zbierski, M. (2015). Parallel byzantine fault tolerance. In Wilinski, A., Fray, I. E., and Pejas, J., editors, *Soft Computing in Computer and Information Science*, volume 342 of *Advances in Intelligent Systems and Computing*, pages 321–333. Springer International Publishing.