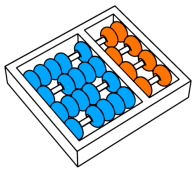
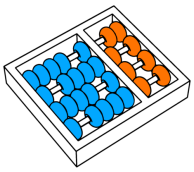


Daniel Cason

“Protocolo de Difusão Síncrona Totalmente Ordenada para Aglomerados de Alto Desempenho”

CAMPINAS
2013



Universidade Estadual de Campinas
Instituto de Computação

Daniel Cason

"Protocolo de Difusão Síncrona Totalmente Ordenada para Aglomerados de Alto Desempenho"

Orientador(a): **Prof. Dr. Luiz Eduardo Buzato**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Daniel Cason, sob orientação de Prof. Dr. Luiz Eduardo Buzato.

Assinatura do Orientador(a)

CAMPINAS
2013

FICHA CATALOGRÁFICA ELABORADA POR
MARIA FABIANA BEZERRA MULLER - CRB8/6162
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Cason, Daniel, 1987-
C27p Protocolo de difusão síncrona totalmente ordenada para aglomerados de alto desempenho / Daniel Cason. – Campinas, SP : [s.n.], 2013.

Orientador: Luiz Eduardo Buzato.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Algoritmos de ordenação total. 2. Tolerância a falha (Computação). 3. Sincronização. 4. Ethernet (Sistema de rede local de computação). 5. Computação em nuvem. I. Buzato, Luiz Eduardo,1961-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: Synchronous total order broadcast protocol for high performance clusters
Palavras-chave em inglês:
Total-order broadcast algorithms
Fault-tolerant computing
Synchronization
Ethernet (Local area network system)
Cloud computing
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Luiz Eduardo Buzato [Orientador]
Lásaro Jonas Camargos
Edmundo Roberto Mauro Madeira
Data de defesa: 28-02-2013
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 28 de Fevereiro de 2013, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Lásaro Jonas Camargos
FACOM / UFU

Prof. Dr. Edmundo Roberto Mauro Madeira
IC / UNICAMP

Prof. Dr. Luiz Eduardo Buzato
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Protocolo de Difusão Síncrona Totalmente Ordenada para Aglomerados de Alto Desempenho

Daniel Cason[1]

28 de Fevereiro de 2013

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (Orientador)
- Prof. Dr. Lásaro Jonas Camargos
  Faculdade de Computação — UFU
- Prof. Dr. Edmundo Roberto Mauro Madeira
  Instituto de Computação — UNICAMP
- Prof. Dr. Gustavo Maciel Dias Vieira (Suplente)
  Departamento de Computação — UFSCar
- Profa. Dra. Islene Calciolari Garcia (Suplente)
  Instituto de Computação — UNICAMP

[1]O desenvolvimento desta dissertação contou suporte financeiro do CNPq (Processos 132172/2010-0, 2010–2011 e 473340/2009-7, 2009–2012) e da FAPESP (Processo 2010/14555-6, 2011–2012).

Abstract

Total order broadcast algorithms are at the core of several toolkits for the construction of fault-tolerant applications. The importance and the difficulty of finding efficient total order broadcast (TOB) algorithms is attested by the long period that such algorithms have been the object of intense research and by the large number of algorithms already proposed. This work presents a new algorithm for total order broadcast that takes advantage of the inherent reliability and timeliness of high performance clusters in its design. Experimental results show that the performance of this very simple TOB is on a par with the performance of TOBs designed for asynchronous computing models.

The proposed protocol has been designed for the timed asynchronous computing model, enhanced with a simple pulse-based mechanism that is used to synchronize the processes' execution. The assumption behind the pulse-based synchronization is that modern clusters, given some workload conditioning, can maintain reasonably long failure-free execution periods in which they behave very much as synchronous system. This assumption allows the processes that engage in total order broadcasts to build a global view of their joint computation and this global view, in its turn, allows them to solve total order broadcast in a straightforward way. The protocol tolerates an unbounded number of timing failures, that can prevent its progress but have no impact on its safety, it is also safe in the in the presence of asynchrony, and processes failures. The protocol has been implemented in Java and tested on a Ethernet-based cluster. A comparison of the results obtained in the experiments with results published for other well-known TOBs allow us to conclude that our solution represents an interesting trade-off between performance and simplicity of design and implementation for total order broadcasts protocols. Beyond performance, this research seems to indicate that there is still room for the practical exploration of the interplay between synchronicity and asynchronicity in the engineering of distributed protocols.

Resumo

Protocolos de Difusão Totalmente Ordenada (DTO) constituem o núcleo de diversas soluções que dão suporte ao desenvolvimento de aplicações distribuídas tolerantes a falhas. O longo período no qual este problema vem sendo objeto de pesquisa e a quantidade de algoritmos que foram para ele propostos atestam, não só a sua importância, mas também a dificuldade de se obter soluções eficientes para DTO. Este trabalho apresenta um novo algoritmo de DTO, que explora a sincronia e a confiabilidade inerentes ao ambiente dos aglomerados ou *clusters* de alto desempenho para construir uma solução bastante simples de Difusão Totalmente Ordenada, cujo desempenho experimental mostrou-se comparável ao obtido por soluções de DTO projetadas para modelos assíncronos de computação.

O protocolo proposto destina-se ao modelo assíncrono temporizado de computação, aumentado com um mecanismo simples, baseado na difusão de pulsos, para a sincronizar a execução dos processos. A hipótese que sustenta este mecanismo de sincronização é que os aglomerados modernos, dado que se controle a carga a eles aplicada, podem executar por períodos razoavelmente longos sem que ocorram falhas de processos e apresentando um comportamento bastante similar ao de sistema síncronos. Dada esta hipótese, os processos que realizam Difusão Totalmente Ordenada tornam-se capazes de construir visões globais da computação distribuída, e a construção de visões globais, por sua vez, torna trivial a resolução de Difusão Totalmente Ordenada. O protocolo proposto tolera uma quantidade ilimitada de falhas de desempenho, que previnem o progresso da solução de DTO, mas que não levam à violação de suas propriedades de segurança, que são asseguradas na presença de assincronia e de falhas de processos. O protocolo foi implementado em Java e o seu desempenho foi avaliado em um aglomerado com máquinas interconectadas via Ethernet. A comparação dos resultados obtidos com os resultados de desempenho publicados para as principais soluções de DTO existentes nos permite afirmar que nossa solução representa um interessante compromisso entre desempenho experimental e simplicidade de projeto e implementação de soluções de Difusão Totalmente Ordenada. Além dos resultados de desempenho, esta pesquisa também revela que ainda há espaço para a exploração prática da interação entre sincronia e assincronia na engenharia de protocolos distribuídos.

Que trama é esta
do será, do é e do foi?
Que rio é este
pelo qual flui o Ganges?
Que rio é este cuja fonte é inconcebível?
Que rio é este
que arrasta mitologias e espadas?
É inútil que durma.
Corre no sonho, no deserto, num porão.
O rio me arrebata e sou esse rio.
De matéria perecível fui feito, de misterioso tempo.
Talvez o manancial esteja em mim.
Talvez de minha sombra,
fatais e ilusórios, surjam os dias.

Jorge Luís Borges

Sumário

Abstract **vii**

Resumo **viii**

Epígrafe **ix**

1 Introdução **1**

2 Fundamentação Teórica **5**
2.1 Modelos de Computação Distribuída 6
2.2 Modelo Assíncrono Temporizado de Computação 10
2.3 Computações Distribuídas baseadas em Rodadas 12
2.4 Difusão Totalmente Ordenada 15

3 Mecanismo de Sincronização e Geração de Rodadas **19**
3.1 Descrição do Mecanismo de Sincronização 19
3.2 Implementação do Mecanismo de Sincronização 22
3.3 Avaliação do Mecanismo de Sincronização 24
3.3.1 Avaliação do Sincronizador 25
3.3.2 Duração das Rodadas 28
3.3.3 Latências de Comunicação 31
3.4 Eficiência das Rodadas de Comunicação 34
3.5 Discussão dos Resultados 38

4 Protocolo de Difusão Síncrona Totalmente Ordenada **39**
4.1 Resolução de Difusão Totalmente Ordenada 39
4.2 Visão Geral do Protocolo 44
4.3 Protocolo para o Modelo de Rodadas Síncronas 46
4.4 Protocolo Tolerante a Falhas de Desempenho 50
4.4.1 Versão Final do Protocolo 53

4.4.2 Discussão do Protocolo 55
4.5 Trabalhos Relacionados 57

5 Avaliação do Protocolo de Difusão Síncrona Totalmente Ordenada 60
5.1 Métricas de Avaliação de Desempenho 60
5.2 Desempenho Experimental do Protocolo 61
5.3 Vazão Efetiva do Protocolo 64
5.4 Latência para a Entrega de Mensagens 67
5.5 Discussão do Desempenho do Protocolo 69
5.6 Protocolos para DTO de Alto Desempenho 71
5.7 Estudo Comparativo de Protocolos de DTO 74

6 Conclusões 77

Referências Bibliográficas 79

Capítulo 1

Introdução

Protocolos de Difusão Totalmente Ordenada (DTO) de mensagens constituem o núcleo de diversas soluções que dão suporte ao desenvolvimento de aplicações com requisitos de alta disponibilidade. Por aplicações de alta disponibilidade entendem-se, especificamente, aplicações que mantém seu funcionamento correto mesmo na ocorrência de falhas de parte dos componentes computacionais que hospedam a sua execução. O que, em particular, é uma funcionalidade que só pode ser obtida quando a aplicação encontra-se *distribuída* em um conjunto de servidores, de modo que a falha ou o comportamento assíncrono de parte deles não impeçam o funcionamento da aplicação como um todo. A cada instância da aplicação que executa em um servidor dá-se o nome de processo e o grupo de processos que constituem a aplicação se comunicam e se sincronizam através da troca de mensagens. É neste ponto que surge a necessidade de protocolos que gerenciem a comunicação entre os processos de um grupo e que confiram propriedades, como confiabilidade e ordenação, às trocas de mensagens realizadas entre eles, como é o caso dos protocolos de DTO.

Em linhas gerais, um protocolo de Difusão Totalmente Ordenada deve garantir que as mensagens enviadas para um conjunto de processos sejam recebidas por todos os processos do conjunto em uma mesma ordem total. Neste trabalho considera-se que o conjunto de processos aos quais se destinam as mensagens enviadas é composto por todos os processos do sistema, de modo que o padrão de comunicação estabelecido é do tipo um-para-todos, ou seja, as mensagens são *difundidas* pelos processos. O papel de um protocolo de DTO, então, é fazer com que o mesmo subconjunto das mensagens difundidas seja *entregue* a todos os processos corretos e, em adição, garantir que a entrega de tais mensagens se dê em uma mesma ordem total em todos eles. Ou, descritos de outra forma, os protocolos de DTO são responsáveis por, no decorrer da computação e a despeito da ocorrência de falhas e da assincronia que podem acometer parte dos processos, construir e entregar a todos os processos *visões consistentes* do conjunto de mensagens por eles difundidas.

As propriedades de Difusão Totalmente Ordenada fazem com que protocolos para a

sua resolução sejam peças fundamentais na implementação de uma das estratégias mais conhecidas e aceitas de a tolerância de falhas, chamada de *replicação ativa*. A estratégia de replicação ativa, proposta inicialmente por Lamport [41] e detalhadamente descrita por Schneider [52], consiste em modelar uma aplicação como uma máquina de estados determinista e distribuir cópias desta máquina de estados em vários servidores. Cada cópia desta máquina de estados corresponde a uma instância ou um *réplica* da aplicação, que interage com seus clientes, ao receber, processar e responder às suas requisições. As requisições dos clientes são modeladas como operações ou transições da máquina de estados, que antes de serem aplicadas pela instância da aplicação que as recebeu, são difundidas para todas as réplicas, através de um protocolo de Difusão Totalmente Ordenada. Cada réplica, por sua vez, aplica à sua cópia da máquina de estados todas as operações recebidas, na ordem em que elas lhe são entregues pelo protocolo de DTO. De modo que, dado que todas as réplicas tenham um mesmo estado inicial, como cada uma delas aplica a mesma *sequência* de operações à sua cópia da máquina de estados, os estados de todas réplicas serão normalmente idênticos e sempre consistentes. Este é o princípio da estratégia de replicação ativa e nota-se que sua corretude é determinada pela semântica na entrega das mensagens garantida pelo protocolo de DTO empregado.

A importância de se desenvolver soluções eficientes de Difusão Totalmente Ordenada é atestada pelo longo período no qual este problema vem sendo objeto de pesquisa e pelo número de soluções que foram para ele propostas. Como referência, no levantamento e classificação de algoritmos de DTO realizado por Defago et al [27], foram identificados mais de 60 protocolos para Difusão Totalmente Ordenada, desenvolvidos entre a década de 1970 e 2004, ano de publicação do trabalho. Porém, a despeito de todo o esforço de pesquisa e de desenvolvimento de soluções para este problema, até onde foi averiguado, em nenhum trabalho verificou-se o desempenho de um protocolo simples para DTO, que se aproveita das propriedades de rodadas síncronas [48] para construir uma ordenação total trivial para as mensagens. O mecanismo de ordenação empregado por este protocolo trivial, que a rigor seria destinado ao modelo síncrono de computação, é a intuição para o novo protocolo de DTO proposto neste trabalho. O nosso protocolo, porém, não se destina a sistemas (estritamente) síncronos: ele considera um modelo computacional muito menos restritivo — o modelo assíncrono temporizado [24] — e foi projetado a fim de explorar as características do ambiente dos aglomerados de alto desempenho.

O protocolo de Difusão Totalmente Ordenada proposto neste trabalho considera que as computações sejam organizadas em rodadas, com semântica síncrona, e destina-se aos períodos da computação de um sistema em que não há falhas de processos. Para garantir o cumprimento das propriedades de segurança de DTO —isto é, que a entrega de mensagens se dê em uma mesma ordem total em todos os processos — o protocolo requer que uma maioria dos processos seja correta. Porém, para que a solução obtenha progresso — isto

é, para que novas mensagens sejam difundidas e entregues pelos processos — o protocolo requer que todos os processos estejam ativos e enviem novas mensagens no início das rodadas. Em adição, o protocolo progride apenas nas rodadas em que o sistema apresenta um comportamento síncrono, de modo que as mensagens enviadas pelos processos no início da rodada sejam recebidas por todos processos no decorrer da mesma rodada. Assim, é a partir da *visão global* do estado dos demais processos obtida nestas rodadas síncronas e de um mecanismo trivial para a ordenação das mensagens nelas recebidas que o protocolo constroi a sequência de mensagens que deve ser entregue a cada processo. Já nas rodadas em que a ocorrência de *falhas de desempenho* impede a obtenção de informação global, o protocolo busca, através do reenvio sistemático de mensagens, *ressincronizar* os processos e, desta forma, voltar a progredir de forma segura na solução de DTO.

O protocolo de Difusão Totalmente Ordenada proposto foi projetado para operar em pequenos aglomerados[1] (ou *clusters*) de alto desempenho, que são atualmente os ambientes empregados na maioria dos centros de processamento de dados espalhados pelo mundo, tanto em pequenas, como em grandes organizações [1]. Os aglomerados são compostos por um conjunto de servidores de processamento, normalmente homogêneos e muitas vezes construídos a partir de componentes de prateleira[2], conectados por uma rede local com alta capacidade de transmissão e baixa latência. A tecnologia de rede atualmente adotada na maioria destes ambientes é a Ethernet, que organiza a rede em uma topologia estrela, cujo nó central é um comutador de pacotes (ou *switch*) de rede, conectado a cada servidor através de enlaces *full-duplex*, que permitem a transmissão simultânea de dados nos dois sentidos [35]. Uma característica interessante deste ambiente de rede é que ele fornece suporte nativo à difusão de mensagens, visto que o comutador de rede é capaz de receber uma mensagem por uma porta de entrada, replicá-la e encaminhar cópias desta mensagem a parte ou a todas as portas de saída, quase sem custos extra associados. O potencial desta funcionalidade na implementação de soluções de DTO de alto desempenho já foi verificado [49] e também é explorado pelo protocolo proposto neste trabalho.

A principal hipótese que se faz neste trabalho é que no ambiente dos aglomerados de alto desempenho, dado que seja possível controlar a carga a ele aplicada, o comportamento do sistema será predominantemente síncrono. A partir desta hipótese, desenvolveu-se um mecanismo simples para a sincronização dos processos, que emprega a funcionalidade de difusão fornecida pela Ethernet para organizar as computações do sistema em rodadas. Este mecanismo foi implementado em um aglomerado de alto desempenho e mostrou-se capaz de controlar a carga que o protocolo aplica ao sistema — em particular, à rede — e de, em adição, induzir a execução de rodadas de comunicação com uma semântica

[1] Por *pequenos aglomerados* entende-se um conjunto de até uma dezena de servidores.

[2] Do termo inglês *commercial off-the-shelf*, que designa componentes genéricos, não especializados, e normalmente de baixo custo, que podem, portanto, ser encontrados em prateleiras de lojas.

síncrona. Assim, ao se *aumentar* o sistema, que apresenta um comportamento assíncrono temporizado, com este mecanismo de sincronização, obteve-se um ambiente computacional propício à implementação do protocolo de Difusão Totalmente Ordenada proposto.

O protocolo proposto, chamado de Difusão Síncrona Totalmente Ordenada (DSTO), foi implementado e seu desempenho foi avaliado experimentalmente. A taxa de bytes entregues à aplicação pelo protocolo de DSTO é inferior, porém comparável, à obtida por outras soluções de DTO, cujos resultados de desempenho, obtidos em ambiente computacional similar, estão disponíveis na literatura. Por outro lado, a latência para a entrega de mensagens — o tempo decorrido entre a sua difusão e a sua entrega para a aplicação — do protocolo de DSTO mostrou-se inferior — portanto, melhor — à obtida pelas demais soluções de DTO, nas escalas em que os protocolos foram avaliados. Assim, ao se considerar também a complexidade para a implementação desta solução em relação às demais, o protocolo de Difusão Síncrona Totalmente Ordenada apresenta um interessante compromisso entre desempenho e simplicidade de projeto e implementação. E, em adição, os resultados obtidos validam a hipótese de que é possível aproveitar a sincronia inerente ao ambiente dos aglomerados para desenvolver soluções de DTO de alto desempenho.

O restante deste trabalho é organizado da seguinte forma. O Capítulo 2 apresenta uma série de conceitos básicos de computação distribuída necessários à compreensão do restante do trabalho, como a organização de computações em rodadas e a modelagem de sistemas distribuídos, além de apresentar a definição formal do problema de Difusão Totalmente Ordenada. No Capítulo 3 descreve-se o mecanismo de sincronização empregado para organizar as computações do sistema em rodadas. A sua implementação nos aglomerados de alto desempenho também é descrita e avalia-se uma série de propriedades das rodadas por ele induzidas, o que constitui também uma avaliação das propriedades temporais do ambiente computacional empregado neste trabalho. No Capítulo 4 são discutidos aspectos gerais da solução de DTO e descreve-se com detalhes o protocolo de DTO proposto neste trabalho, chamado de protocolo de Difusão Síncrona Totalmente Ordenada. O comportamento do protocolo é discutido, em particular em relação aos requisitos para o seu progresso na ausência de falhas e aos procedimentos empregados para tratar a ocorrência de falhas de processo. Em adição, argumenta-se a sua corretude e apresentam-se os principais trabalhos relacionados. No Capítulo 5, então, apresenta-se os experimentos que foram realizados para avaliar o desempenho do protocolo de DSTO. Os resultados obtidos nestes experimentos são discutidos e comparados com resultados de desempenho encontrados na literatura, referentes a outras soluções de DTO, que também foram projetadas ou otimizadas para o ambiente dos aglomerados de alto desempenho. Finalmente, o Capítulo 6 conclui este trabalho com a discussão dos resultados obtidos e de suas principais contribuições, além de apresentar as perspectivas para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Um sistema distribuído pode ser definido como um conjunto de processos autônomos, interconectados via rede, que se comunicam através da troca de mensagens. Cada um dos processos que compõem o sistema executa — diz-se que ele está hospedado — em um processador e detém, em memória local, apenas uma quantidade limitada de informação. Considera-se que os processos não compartilham memória e que os processadores que hospedam sua execução estejam distribuídos em computadores ou unidades de processamento distintas. De forma que a transferência de informações, a interação e a sincronização de ações entre os processos é realizada exclusivamente através da troca de mensagens.

Estas características dos sistemas distribuídos potencializam seu uso para o desenvolvimento de uma vasta gama de soluções computacionais. Dentre as funcionalidades que são por elas disponibilizadas, destacam-se: o compartilhamento de recursos entre vários computadores, a coordenação de sistemas geograficamente distribuídos, a execução concorrente e cooperativa de tarefas e, principalmente, a tolerância a falhas de componentes computacionais. Neste trabalho estamos particularmente interessados na implementação de soluções para fornecer esta última funcionalidade, de tolerância a falhas, a aplicações distribuídas. Uma das estratégias para implementar esta funcionalidade, considerada neste trabalho, é chamada de replicação ativa e foi brevemente descrita no Capítulo 1. A implementação desta estratégia, por sua vez, depende da resolução de um problema de computação distribuída, denominado Difusão Totalmente Ordenada, cuja implementação em um ambiente computacional específico é o objeto principal deste trabalho.

Neste capítulo apresentam-se conceitos fundamentais de computação distribuída, que são necessários para a compreensão do restante do trabalho. Na Seção 2.1 apresenta-se a forma como os sistemas distribuídos são modelados, em termos do funcionamento dos processos, dos canais e da ocorrência de falhas. Na Seção 2.2 descrevem-se os conceitos gerais do modelo de computação distribuída que serviu como base para a elaboração deste trabalho, o modelo assíncrono temporizado. Na Seção 2.3 apresenta-se o conceito

de computações distribuídas baseadas em rodadas, que é implementado e empregado no restante do trabalho. Finalmente, a Seção 2.4 encerra o capítulo com a definição formal do problema de computação distribuída acima citado, a Difusão Totalmente Ordenada.

2.1 Modelos de Computação Distribuída

Algoritmos distribuídos são aqueles executados pelos processos de um sistema distribuído. É através deles que se implementa primitivas e funcionalidades que dão suporte à execução de aplicações distribuídas confiáveis. Algoritmos distribuídos, porém, não são projetados para um sistema distribuído específico, mas são projetados para executar em sistemas onde sejam válidas as hipóteses definidas por um *modelo computacional*.

O conjunto de hipóteses que definem um modelo computacional abstraem os aspectos específicos de um certo sistema computacional e descrevem, em alto nível, as características que devem ser levadas em consideração na elaboração de algoritmos distribuídos. Um modelo de computação também inclui as primitivas e funcionalidades, nativamente implementadas pelo sistema, que podem ser empregadas pelos os algoritmos em execução. De forma geral, as hipóteses sobre o comportamento do sistema distribuído podem ser classificadas em três grandes categorias, apresentadas separadamente a seguir: comunicação entre os processos, comportamento temporal e ocorrência de falhas.

Modelo de comunicação A troca de mensagens entre os processos é abstraída pelo conceito de canais de comunicação. Na maioria dos sistemas, supõe-se que todos os processos podem se comunicar diretamente entre si, isto é, que há um canal de comunicação que conecta cada par de processos. Esta forma de comunicação é chamada de ponto-a-ponto (em *unicast*). É possível também que os processos possam se comunicar em grupo, isto é, difundir uma única mensagem para vários processos (em *multicast*) ou para todos os processos do sistema (em *broadcast*). Estas primitivas de comunicação em grupo são, em certos ambientes, nativamente oferecidas pelas camadas de rede, mas em vários sistemas elas devem ser implementadas usando mensagens e canais ponto-a-ponto.

A hipótese de existência de canais de comunicação que interconectam todos os processos engloba a suposição de que há mecanismos de endereçamento, roteamento e transporte das mensagens entre cada par de processos, com a colaboração ou não de dispositivos intermediários. Em alguns casos, os passos individuais de comunicação são levados em consideração, isto é, considera-se que nem todos os processos se comunicam diretamente entre si. Nestes casos, o modelo de comunicação inclui a topologia da rede, que é um grafo de comunicação, que tem os processos como vértices e os canais de comunicação como arestas. Quando a topologia é omitida do modelo, considera-se que o grafo de comunicação é completo, com todos os vértices conectados por uma aresta.

Outras características que podem ser definidas pelo modelo de comunicação são a distribuição geográfica dos processos e as tecnologias empregadas para a conexão dos processos ou para o transporte das mensagens — informações que, de modo geral, detalham ou estendem a topologia da rede. Assim, por exemplo, é possível projetar algoritmos específicos para o ambiente das redes locais, caracterizado por apresentar poucos passos de comunicação entre cada par de processos, por altas taxas de transmissão e baixas latências de comunicação. Algoritmos que supõem um modelo computacional com estas hipóteses, não serão corretos ou eficientes, por exemplo, na internet, que possui características de comunicação muito diferentes. Existe também a possibilidade de se projetar algoritmos para ambientes de rede específicos, com semânticas de comunicação peculiares, como é o caso das redes sem fio. Ou então, projetar algoritmos que supõem a existência de primitivas ou tecnologias específicas de transporte de mensagens. Assim, pode-se supor que se tenha (ou que não se tenha) determinadas garantias na entrega das mensagens aos seus destinatários, tal como ordenação, detecção e/ou recuperação de perdas, sincronia etc. Ou ainda, especificar quais são os protocolos de comunicação disponíveis no sistema, como o TCP (ponto-a-ponto, com detecção e recuperação de perdas, ordenação, controle de fluxo etc.), ou UDP (sem garantias de entrega ou de ordenação, suporte à difusão etc.). Nota-se, porém, que hipóteses mais específicas quanto à comunicação muitas vezes se refletem também em hipóteses de cunho temporal ou quanto à ocorrência de falhas.

Modelo temporal Diversas hipóteses podem ser feitas sobre as propriedades temporais respeitadas pelos eventos do sistema que hospeda a execução de um algoritmo distribuído. Estas propriedades determinam o comportamento esperado para o sistema em relação a três fatores: (i) o tempo que os processos precisam para executar um passo de processamento; (ii) o tempo necessário para que os canais entreguem as mensagens aos seus destinatários; e (iii) a percepção que os processos têm da passagem do tempo.

Em um extremo dos modelos temporais, têm-se os sistemas completamente síncronos, nos quais se supõe que processos e canais de comunicação são capazes de realizar tarefas de processamento e de envio de mensagens dentro de prazos pré-estabelecidos. Ou seja, tem-se a hipótese de que há limitantes inferiores e superiores para o tempo necessário para que ações sejam realizadas pelos componentes do sistema e que estes limitantes são conhecidos. Em adição, no *modelo síncrono* de computação, supõe-se que os processos têm acesso a uma fonte global de tempo, cuja acurácia (isto é, a diferença entre leituras de tempo feitas simultaneamente por diferentes processos) e precisão também são limitadas.

No outro extremo dos modelos temporais tem-se o *modelo assíncrono*, que considera que os componentes do sistema realizam as tarefas de processamento e de comunicação a eles requisitadas em ordens e com prazos arbitrários. Ou seja, a hipótese é que não se tem limitantes para o tempo necessário para que uma ação se conclua, apesar deste tempo

ser sempre finito, quando a ação é realizada por componentes corretos. Além disso (ou por conta disso), considera-se que os processos não têm acesso a nenhuma fonte de tempo sincronizada e, portanto, que eles não são capazes de verificar a passagem do tempo.

Vários outros modelos temporais foram propostos — que, de certa forma, vagam entre estes dois extremos, ao fortalecer (ou enfraquecer) certas hipóteses do modelo assíncrono (ou do síncrono) —, aos quais se dá o nome genérico de *parcialmente síncronos*.

Modelo de falhas Os componentes de hardware sobre os quais executa um algoritmo distribuído podem, em um hipotético sistema computacional perfeito, ser considerados completamente confiáveis. Mas no caso normal, ao se considerar que os componentes do sistema não são confiáveis, o algoritmo deve ser projetado para tolerar falhas[1]. Assim, a especificação de como e se processos e canais de comunicação podem falhar também faz parte do modelo computacional para o qual um algoritmo distribuído é projetado.

As falhas de processos podem ser modeladas nas seguintes formas: por parada, por parada com recuperação, por omissão, falhas de desempenho e falhas arbitrárias. As falhas por parada (ou *crash-stop*) são as falhas consideradas na maior parte dos modelos computacionais. Nelas, o processo cessa de forma definitiva a sua participação no sistema, normalmente sem indicação prévia — caso haja indicação, tem-se um tipo menos desruptivo de falha, conhecido como *crash-fail*. As falhas por parada com recuperação (ou *crash-recovery*) têm semântica idêntica às falhas por parada, com a diferença que o processo pode, após um procedimento de recuperação, reingressar na computação. Já nas falhas por omissão, os processos deixam de enviar e/ou receber mensagens corretamente em certos períodos, normalmente arbitrários, da execução. Comportamento similar é observado nas falhas de desempenho, com a diferença que nestas últimas a causa da omissão é, especificamente, o não cumprimento dos prazos estabelecidos para a realização de uma tarefa. Todas as falhas apresentadas até aqui são consideradas falhas *benignas*, o que as difere da última classe, de falhas arbitrárias. Processos que incorrem nestas falhas se comportam de forma inesperada, omitem-se no envio e recepção e podem inclusive realizar ações indevidas, o que caracteriza um comportamento malicioso ou bizantino.

Em relação aos canais de comunicação é bastante comum considerar que eles não sofrem com falhas. Isto é o que normalmente ocorre quando no modelo de comunicação supõe-se a existência de camadas e protocolos de rede (por exemplo, o protocolo TCP), que mascaram e se recuperam das falhas que ocorrem na comunicação em nível físico. As falhas de comunicação, quando modeladas, referem-se às mensagens em trânsito pelos canais, que podem ser perdidas, reordenadas, duplicadas ou corrompidas. E se, em adição, o modelo temporal supõe que há prazos para a entrega de mensagens, as violações destes

[1]Adota-se neste trabalho os termos falha, erro e defeito como tradução para, respectivamente, os termos *fault*, *error* e *failure*, com o mesmo significado utilizado por Avizienis et al [8].

prazos são consideradas falhas de desempenho dos canais de comunicação.

Um modelo computacional é, portanto, definido por um subconjunto das hipóteses dos modelos de comunicação, temporal e de falhas acima apresentados. De modo geral, estes modelos definem o *grau de incerteza* e o *grau de independência* quanto aos eventos do sistema [48], com os quais os algoritmos distribuídos devem lidar. Quanto maiores forem estes graus, maior é a concorrência com a qual se supõe que os eventos do sistema ocorrem e mais complicado será o desenvolvimento de algoritmos para o modelo.

O modelo computacional mais simples para se raciocinar e para se empregar no projeto de algoritmos distribuídos é o *modelo síncrono* [48]. Nele considera-se que a comunicação é confiável e ordenada; que os processos e canais de comunicação realizam suas tarefas dentro de prazos pré-estabelecidos, isto é, são síncronos; que os processos falham por parada e que a sua falha é em algum momento detectada pelos demais processos. Assim, dado este conjunto de hipóteses, pode-se dizer que os graus de incerteza e de paralelismo do modelo síncrono são os mais baixos que se pode conceber, e que, portanto, a solução de problemas de computação distribuída seja relativamente simples neste modelo.

Porém, do ponto de vista do desenvolvimento de algoritmos genéricos, que possam ser empregados em uma vasta gama de sistemas de computação, os modelos computacionais mais interessantes são os que consideram conjuntos menos restritivos de hipóteses. Assim, pode-se considerar que os canais de comunicação não garantam a entrega das mensagens, isto é, que mensagens possam ser perdidas; que não haja nenhuma suposição em relação à velocidade relativa dos processos, às latências de comunicação ou à ordem em que os eventos ocorrem no sistema; e que os processos falhem por parada, sem aviso prévio, e que eles possam se recuperar e reingressar na computação. Como consequência, a corretude dos algoritmos projetados para esta variante do *modelo assíncrono* [32] de computação independe da sincronia efetivamente apresentada pelo sistema que hospeda sua execução e da semântica de seus canais de comunicação, o que os torna particularmente robustos. Por outro lado, as hipóteses consideradas por este modelo são insuficientes para que se distinga processos falhos de processos muito lentos, o que agrega muita incerteza à execução dos algoritmos, a ponto de tornar impossível a resolução de alguns problemas fundamentais de computação distribuída neste modelo, mesmo quando ele é fortalecido com a hipótese da comunicação ser confiável e de que os processos que falham não se recuperem [32].

Assim, se por um lado a solução de problemas é relativamente simples em modelos síncronos, mas estes modelos não representam fidedignamente a maioria dos sistemas computacionais existentes, por outro lado os modelos assíncronos, que podem representar razoavelmente quase qualquer sistema computacional, não têm hipóteses suficientes para permitir a resolução de vários problemas de computação distribuída. De modo que para a construção de algoritmos distribuídos que possam ser empregados na prática, normal-

mente recorre-se a modelos que são classificados como *parcialmente síncronos*, como é o caso do modelo computacional que é apresentado na próxima seção.

2.2 Modelo Assíncrono Temporizado de Computação

Dentre os modelos computacionais classificados como parcialmente síncronos, um teve particular influência na concepção deste trabalho: o modelo assíncrono temporizado de computação distribuída [25]. A origem deste modelo computacional remonta a um trabalho de Cristian [23], que discute protocolos para o modelo síncrono de computação e define o conceito de *falhas de desempenho*. Um processo ou um canal de comunicação incorre em uma falha de desempenho quando o tempo por eles empregado para a realização de uma tarefa de processamento ou para o consequente envio de uma mensagem excede os prazos que foram estipulados para a sua realização. Da mesma forma que ocorre com as falhas por omissão, a detecção das falhas de desempenho se dá através da hipótese da existência de relógios, que permitem aos processos detectar o não recebimento a tempo de uma mensagem esperada. Porém, ao contrário das falhas por omissão, o número de processos ou canais de comunicação que podem ser afetados por falhas de desempenho não é limitado, e as mensagens podem ser recebidas após o prazo estipulado para o seu recebimento, mas neste caso elas devem ser descartadas pelos processos.

Assim, o modelo assíncrono temporizado relaxa a hipótese do modelo síncrono de que processos e canais de comunicação corretos, durante toda a computação distribuída, realizam suas tarefas dentro de prazos pré-estabelecidos. De fato, em um outro trabalho, Fetzer e Cristian [31] definem o modelo como uma alternativa para implementar soluções síncronas em sistemas que apresentam um comportamento assíncrono, mas cujos processos têm acesso a relógios de *hardware*. Nota-se que a inexistência de relógios sincronizados ou com taxas de progressão similares é uma hipótese da definição mais frequente do modelo assíncrono [32]. Esta hipótese é sustentada pelo argumento que o desvio entre relógios (dos processadores) de diferentes processos — por conta de vários fatores, como a carga a eles aplicada e as suas temperaturas, que influenciam as suas frequências de operação e, portanto, alteram suas taxas de progressão — ser arbitrário, imprevisível e ilimitado [17]. Assim, para sustentar o seu argumento, os autores do modelo assíncrono temporizado citam relógios de *hardware* existentes na maioria dos computadores modernos, cujas taxas de progressão, determinadas pela vibração de um cristal de quartzo, não são influenciadas pela carga aplicada ao processadores ou pela sua frequência de operação [25, 31].

Na definição formal do modelo assíncrono temporizado [25], a hipótese de existência de relógios sincronizados é relaxada para a hipótese de que os relógios dos processos apresentam taxas de progressão idênticas, com desvios entre elas limitados por uma constante conhecida. Como referência, Cristian e Fetzer citam novamente os relógios de quartzo,

com desvios nas suas taxas de progressão muito reduzidos, da ordem de 10^{-4} a $10^{-6}s$. Com esta modificação, os relógios deixam de servir como referência para aferir instantes do tempo e passam a ser empregados apenas no agendamento de temporizadores, normalmente para intervalos curtos de tempo, de modo que os possíveis desvios nas taxas de progressão dos relógios dos vários processos tornem-se quase irrelevantes.

Além da suposição de existência de relógios que permitam o uso de temporizadores, o modelo assíncrono temporizado determina que os serviços oferecidos pelos processos e canais de comunicação sejam *temporizados*. Assim, ao contrário da definição do modelo assíncrono [32], a especificação do sistema nesse modelo inclui os atrasos esperados para a realização de tarefas pelos seus componentes. Estes atrasos, por sua vez, são normalmente condensados por uma constante, que representa o tempo total esperado entre o recebimento de um evento por um processo e a chegada, em seus destinatários, da mensagem enviada como decorrência de seu processamento. De forma que, dada a existência desta constante do sistema, os processos, a partir do uso de temporizadores, detectam a ocorrência de falhas de desempenho e são capazes de tratá-las [25, 31].

Uma outra característica interessante do modelo assíncrono temporizado é que ele, ao contrário do modelo assíncrono [32], considera que as mensagens podem ser perdidas pelos canais de comunicação. O que significa, em particular, que o sistema pode sofrer com o particionamento da rede, isto é, com a desconexão temporária de um conjunto de processos dos demais. O argumento dos autores desse modelo é que a implementação de canais de comunicação confiáveis — que não perdem mensagens — requer que sejam feitas considerações relativas ao comportamento temporal do sistema ou que se suponha que os canais de comunicação sejam perfeitos, o que não é realista [25].

De modo geral, o modelo assíncrono temporizado adiciona ao modelo assíncrono [32], chamado por seus autores modelo *time-free* ou livre de tempo [25], as seguintes hipóteses. A primeira hipótese é o acesso dos processos a relógios, não necessariamente sincronizados, mas que apresentem as características mínimas esperadas de um relógio. A segunda hipótese é que os serviços sejam temporizados e que existam prazos para a realização de tarefas, que podem ser descumpridos por qualquer processo ou canal de comunicação a qualquer momento e com qualquer frequência. A última hipótese, que é a única que fortalece o modelo assíncrono original, é considerar que mensagens podem ser perdidas pelos canais de comunicação, o que configura uma falha de desempenho.

Finalmente, os autores do modelo assíncrono temporizado se viram obrigados a definir uma hipótese de progresso para o modelo proposto. O progresso no modelo assíncrono, que não considera a existência de relógios, se embasa na existência de um limite superior de processos e na confiabilidade dos canais de comunicação. Como no modelo assíncrono temporizado estas duas hipóteses são removidas, o progresso precisa ser assegurado de outra forma. O que este modelo considera é que o sistema eventualmente atravessa períodos

de "bom" comportamento, nos quais a ocorrência de falhas de desempenho é limitada, de modo que os processos conseguem se comunicar e realizar tarefas de processamento dentro dos prazos estabelecidos [25, 31]. O que é uma definição idêntica à considerada pelos modelos parcialmente síncronos [29, 30], de que o sistema a partir de algum momento se comporta de maneira síncrona e, portanto, os algoritmos distribuídos obtém progresso.

2.3 Computações Distribuídas baseadas em Rodadas

A forma mais comum de modelar um processo de um sistema distribuído é representá-lo através de uma máquina de estados determinista [48]. Cada máquina de estados, que é um autômato determinista, é composta por um conjunto (não necessariamente finito) de estados, dos quais um subconjunto contém estados iniciais, e por uma função de transição, que mapeia, dois a dois, os estados. As transições de estados são desencadeadas por eventos que ocorrem no decorrer da execução do processo e, como resultado destas transições, o processo realiza ações. O estado para o qual o processo transita com a ocorrência de um evento e a ação correspondente por ele realizada são determinados pelo algoritmo distribuído em execução. Ou, de forma análoga, define-se o algoritmo distribuído como sendo a coleção de máquinas de estados executadas por cada processo [48].

Os eventos que desencadeiam transições de estados nos processos são a recepção de mensagens e alguns eventos locais, como o recebimento de valores de entrada ou, caso o modelo de computação permita, eventos de sincronização ou de caráter temporal (por exemplo, o estouro de temporizadores). As ações desencadeadas por transições de estados, por sua vez, são o envio de mensagens a um ou vários processos, a geração de valores de saída, o encerramento da computação ou o agendamento, caso o modelo de computação permita, de eventos temporais (de temporizadores, por exemplo).

Em relação às estratégias para se organizar a execução de um algoritmo distribuído, uma bastante conhecida e empregada é fazer com que ela proceda em *rodadas* [30, 48]. Esta estratégia pode ser definida da seguinte forma: em cada rodada da computação todos os processos executam, de forma coordenada, um passo de um algoritmo distribuído. Isto é, eles enviam mensagens para algum subconjunto dos processos, aguardam pelo recebimento (de parte) das mensagens para eles enviadas naquela mesma rodada e então processam as mensagens e demais eventos locais recebidos durante a rodada [30]. Como resultado deste processamento em bloco dos eventos recebidos em uma rodada, o estado do processo é atualizado e as ações decorrentes das transições de estado efetuadas são realizadas, também em bloco, no início da rodada seguinte. De modo que a computação obtida é caracterizada pela alternância entre períodos de bloqueio, nos quais os processos aguardam pela ocorrência de eventos, com passos coordenados de processamento, que são desencadeados pelo final de cada rodada e resultam na realização de ações [48].

A estratégia de organizar as computações em rodadas remonta o funcionamento dos primeiros sistemas distribuídos, cujas propriedades deram origem ao modelo síncrono de computação distribuída [48]. Em tais sistemas, denominados redes síncronas, os processos só podiam enviar mensagens em instantes inteiros ou *pulsos* de um relógio global, ao qual todo processo tinha acesso. Em adição, os canais de comunicação eram capazes de carregar apenas uma mensagem de cada vez e as latências de transmissão não excediam a duração de um pulso [9]. Ao *receber* um pulso, cada processo gerava e enviava, através dos canais de comunicação que dele partiam, uma mensagem para cada um de seus vizinhos. Estas mensagens — que chegavam aos seus destinatários antes do próximo pulso — eram coletadas dos canais de comunicação e processadas, a fim de que os processos atualizassem seus estados. Em conjunto, estas duas etapas — de geração e envio de mensagens e de seu recebimento e processamento — eram realizadas entre dois pulsos do relógio e constituíam uma rodada da computação [48]. E como estas etapas eram realizadas simultaneamente pelos processos, dizia-se que as computações progrediam em *rodadas síncronas*.

Apesar desta hipótese de que os componentes do sistema realizam ações de forma simultânea não ser muito realista — principalmente nos sistemas distribuídos modernos, muito mais velozes e complexos que aqueles que originaram este modelo —, a organização das computações em rodadas síncronas é uma estratégia interessante para a descrição e para a análise de algoritmos distribuídos. Assim, o *modelo de rodadas síncronas* [48] é largamente empregado, não só para descrever computações em sistemas síncronos, mas também para analisar e ilustrar o funcionamento de algoritmos distribuídos em geral. A razão para que ele seja tão empregado é, em grande parte, devido à sua propriedade fundamental de comunicação, que estabelece que todas as mensagens enviadas por processos corretos no início de uma rodada são recebidas por todos os processos corretos até o final da mesma rodada. O que, além de constituir um predicado muito forte de comunicação, determina que se um processo correto não recebe, até final de uma rodada, uma mensagem de um processo qualquer, então este outro processo falhou naquela rodada [19, 48].

Porém, se por um lado o modelo de rodadas síncronas mostra-se muito poderoso para a resolução de problemas de computação distribuída, por outro a implementação de ambientes computacionais que atendam à sua propriedade fundamental de comunicação é complexa, se é que ela é realmente possível em sistemas que não são de tempo real [2]. Por conta desta limitação na empregabilidade das soluções destinadas a este modelo — limitação que é inerente ao modelo síncrono, quer se organize a computação em rodadas, quer não [2] — foram propostos outros modelos de computação distribuída, que também possuem uma semântica síncrona, mas que enfraquecem algumas hipóteses consideradas pelo modelo síncrono. A estes modelos — que incluem o modelo assíncrono temporizado, descrito na Seção 2.2 — dá-se o nome genérico de parcialmente síncronos [29, 30].

Em um dos trabalhos que apresentam a classe de modelos parcialmente síncronos,

Dwork et al [30] também definem o *modelo básico de rodadas*. Este modelo considera que as computações são organizadas em rodadas síncronas e que cada rodada é subdividida em três subrodadas ou fases. Na fase de envio, executada no início de cada rodada, os processos enviam mensagens para algum subconjunto dos processos. Na fase seguinte, a de recepção, um subconjunto das mensagens enviadas na fase de envio correspondente é entregue ao processo. Na última fase, a de processamento, cada processo atualiza seu estado, com base no processamento das mensagens recebidas durante a rodada [30]. Em relação ao modelo de rodadas, a principal diferença deste modelo é que não se considera que todas as mensagens enviadas em uma rodada são recebidas no final da mesma rodada por seus destinatários. O que, segundo Dwork et al, pode ocorrer por conta de mensagens terem sido perdidas pelos canais de comunicação ou por elas não terem sido recebidas *a tempo* por seus destinatários, isto é, por elas terem sido recebidas após o final da rodada em que elas foram enviadas, o que faz com elas sejam desconsideradas [30].

Em dois trabalhos subsequentes [21, 33], a organização de computações em rodadas voltou a ser discutida, com o objetivo de apresentar versões mais simples e curtas para provas de algoritmos e de resultados importantes de computação distribuída, como por exemplo, algumas impossibilidades e limitantes inferiores para resolução de problemas. O primeiro trabalho [33] estende o modelo básico de rodadas para praticamente qualquer tipo de (modelagem de) sistema, através da análise de como o sistema evolui, rodada por rodada, e da construção de predicados de comunicação que permitam diferenciar as diversas modelagens de sistemas. Esta abordagem é estendida no segundo trabalho [21], que busca unificar os predicados de comunicação e a modelagem de falhas considerados por um modelo computacional através do conceito de falhas de transmissão, que decorrem no não recebimento de mensagens que eram esperadas em uma rodada da computação.

Finalmente, a menos de algumas especificidades, o conceito de rodadas empregado no restante deste trabalho é o definido pelo modelo básico de rodadas. A sua propriedade fundamental define que se uma mensagem é entregue a um processo no decorrer de uma rodada, então esta mensagem necessariamente foi enviada naquela mesma rodada [21, 30]. O que equivale a dizer que se considera que as rodadas sejam *fechadas para comunicação*, em termos de que uma mensagem enviada em uma certa rodada só é entregue aos processos que a receberem no decorrer da mesma rodada. Já as mensagens que forem recebidas após o término das rodadas nas quais elas foram enviadas — isto é, as mensagens que forem consideradas atrasadas — são definitivamente descartadas pelos processos [21, 30].

2.4 Difusão Totalmente Ordenada

O problema de Difusão Totalmente Ordenada (DTO) é formalmente definido através de duas primitivas de comunicação, uma destinada à *difusão* de mensagens e a outra

destinada à *entrega* de mensagens aos processos. A primitiva de difusão de mensagens é empregada — diz-se também que ela é invocada — por um processo para requisitar que uma mensagem m seja ordenada e entregue a todos os processos. Quando esta primitiva é invocada por um processo p, diz-se que p difundiu uma mensagem m. Supõe-se que as mensagens difundidas são distinguíveis entre si e que o processo p, chamado de o *emissor* de m, invoca a primitiva de difusão uma única vez para cada mensagem m. Os processos que podem invocar a primitiva de difusão são chamados de emissores e o conjunto de mensagens difundidas pelos emissores constitui a entrada de um algoritmo de DTO.

A segunda primitiva, destinada à *entrega* de mensagens, está relacionada à recepção pelos processos das mensagens difundidas e já totalmente ordenadas. Ela é invocada pelas instâncias do algoritmo DTO que executam em cada processo com intuito de disponibilizar às aplicações que delas são clientes as mensagens que foram difundidas. Assim, quando a instância de DTO associada ao processo p invoca a primitiva de entrega, relativa a uma mensagem m, diz-se que a mensagem m foi entregue a p. Em adição, quando não houver ambiguidade, diz-se que p — referindo-se à instância de DTO associada a p — entregou m. A sequência de mensagens que é entregue por cada processo p, em repetidas invocações desta primitiva pela aplicação, constitui a saída do algoritmo de DTO em p.

Com base nestas primitivas de difusão e de entrega de mensagens, o problema de Difusão Totalmente Ordenada é definido pelas seguintes propriedades [27, 36]:

Validade: Se um processo correto p difunde uma mensagem m, então m será finalmente[2] entregue pelo processo p.

Acordo Uniforme: Se uma mensagem m é entregue por um processo p qualquer, então m será finalmente entregue por todos os processos corretos.

Integridade Uniforme: Qualquer mensagem m é entregue por cada processo no máximo uma vez e se, e somente se, m tiver sido previamente difundida por algum processo.

Ordenação Total Uniforme: Sejam duas mensagens m e m' e dois processos p e q. Se ambas as mensagens são entregues pelos dois processos, então m é entregue antes de m' pelo processo p, se, e somente se, m for entregue antes de m' pelo processo q.

As propriedades que caracterizam algoritmos distribuídos costumam ser classificadas em duas classes: propriedades de segurança (ou *safety*) e de progresso (ou *liveness*) [46, 48]. As duas primeiras propriedades de DTO (Validade e Acordo Uniforme) são propriedades de progresso, pois determinam que algo ocorre finalmente — isto é, ocorrerá em algum momento futuro — como consequência de uma ação tomada por um processo [46]. Estas

[2]Por *finalmente* busca-se expressar o significado do advérbio *eventually* da língua inglesa, que denota a ideia de que um predicado se tornará válido, de forma definitiva, a partir de algum momento futuro.

propriedades não precisam ser válidas no momento em que sua condição é desencadeada, mas deverão se tornar válidas, de forma definitiva, a partir de algum instante de tempo, posterior ao desencadeamento da sua condição, mas não conhecido ou especificado a priori. Em particular, estas duas propriedades definem que se um processo p correto invoca a primitiva para a difusão de uma mensagem m, então p e os demais processos corretos, em algum momento após a difusão de m, invocam a primitiva para a entrega da mesma mensagem m. O que corresponde a dizer que todas as mensagens difundidas por processos corretos devem ser finalmente entregues por todos os processos corretos.

As duas propriedades restantes, Integridade Uniforme e Ordenação Total Uniforme, são classificadas como propriedades de segurança [46]. Ao contrário das propriedades de progresso, propriedades de segurança definem predicados que devem ser cumpridos pelo algoritmo distribuído durante toda a sua execução. Se algum destes predicados, em qualquer circunstância ou instante da execução, torna-se inválido, tem-se uma violação da segurança do algoritmo e, consequentemente, da corretude da solução. Dito de outra forma, propriedades de segurança são aquelas que estabelecem restrições para a execução de um algoritmo, isto é, que determinam o que o algoritmo não pode fazer. Este é o caso das duas propriedades de DTO acima citadas, que estabelecem, respectivamente, que soluções para DTO não podem gerar ou duplicar mensagens e que processos que entregam um mesmo conjunto de mensagens, não as podem entregar em ordens distintas. Assim, se as propriedades de progresso garantem a entrega das mensagens difundidas por processos corretos a todos os processos corretos, as propriedades de segurança definem qual é a semântica que deve ser respeitada na entrega de tais mensagens a cada processo.

É interessante notar que a definição de DTO apresentada emprega três propriedades classificadas como *Uniformes*. Uma propriedade é uniforme quando ela impõe restrições, não só quanto ao comportamento de processos corretos, mas também ao comportamento de processos falhos. No caso da propriedade de Acordo, ao defini-la como Uniforme, determina-se que entrega de uma mensagem por processo qualquer, seja ele correto ou falho, é condição suficiente para que ela deva ser finalmente entregue por todos os processos corretos. Em outras palavras, a propriedade de Acordo Uniforme proíbe que um processo falho entregue uma mensagem se não houver garantias que esta mensagem será finalmente entregue por pelo menos um processo correto — e, portanto, por todos os processos corretos, como requer a versão não uniforme desta propriedade. Em relação à propriedade de Ordenação Total, o fato de ela ser Uniforme proíbe a qualquer processo, mesmo que ele apresente comportamento falho, entregar mensagens em uma ordem que difira da que foi definida para os processos que se comportam corretamente. Assim, a menos dos predicados que asseguram que mensagens difundidas são finalmente entregues — não aplicáveis a processos falhos, visto que eles podem abandonar a execução a qualquer momento — as restrições definidas pelas propriedades de DTO são impostas, de forma indistinta, a

todos os processos. O que faz com que a caracterização empregada neste trabalho seja considerada a mais restritiva para o problema de Difusão Totalmente Ordenada [27, 57].

Uma definição alternativa para DTO, que incorpora as propriedades já definidas e que será empregada neste trabalho é a seguinte. Dado o conjunto de mensagens difundidas pelos processos, um algoritmo de DTO deve construir, de forma distribuída, progressiva e tolerante a falhas, uma *sequência de mensagens* que seja comum a todos os processos. Cabe, então, a cada instância do algoritmo de DTO entregar, como resultado da i-ésima invocação pela aplicação da primitiva que solicita a entrega de uma mensagem, a i-ésima mensagem desta sequência. Assim, supondo que a sequência construída não contenha mensagens duplicadas ou espúrias, como requer a propriedade de Integridade Uniforme, os requisitos na entrega de mensagens são sintetizados pela seguinte propriedade [7, 27]:

Prefixo Comum: Sejam dois processos p e q e sejam $seq(p)$ e $seq(q)$ as sequências de mensagens entregues, respectivamente, pelos processos p e q, então, sempre, ou $seq(p)$ é um prefixo de $seq(q)$, ou $seq(q)$ é um prefixo de $seq(p)$.

Uma sequência é um conjunto ordenado de elementos — neste caso específico, dois a dois disjuntos — e seu k-prefixo é também uma sequência, que compartilha os k primeiros elementos com a sequência original, preservando-se a mesma relação de ordem. Assim, ao respeitar a propriedade de Prefixo Comum, um algoritmo de DTO trivialmente não viola a propriedade de Ordenação Total Uniforme. Se o algoritmo também garantir progresso, a sequência de mensagens entregues a um processo irá crescer com o passar do tempo, até que não se tenha mais novas mensagens difundidas ou até que o processo abandone a execução. Além disto, se a cada mensagem entregue por um processo a propriedade de Prefixo Comum continuar sendo válida, garante-se também o Acordo Uniforme, pois os elementos de uma subsequência devem necessariamente pertencer à sequência original. O que se agrega à definição original com a propriedade de Prefixo Comum é que se a k-ésima mensagem da sequência que foi construída é entregue a qualquer processo, então a ele necessariamente já foram entregues todas as $k-1$ mensagens que a precedem[3]. Em outras palavras, esta propriedade determina que não se tenha *lacunas* na sequência de mensagens que é entregue a qualquer processo, mesmo que ele seja falho[4]. O problema gerado pelas lacunas formadas na sequência de mensagens entregues por um processo é que elas fazem com que ele atinja estados inconsistentes, difunda mensagens a partir de tais estados, que, por mais que o processo seja falho, podem ser aceitas pelos demais processos, gerando um fenômeno indesejável conhecido como *contaminação* [7, 27].

[3]Nota-se que este predicado não é garantido, no caso do processo ser falho, pela definição original de DTO, pois a propriedade de Acordo Uniforme assegura a entrega de mensagens somente aos processos corretos e a Ordenação Total se aplica apenas às mensagens que foram entregues a um processo.

[4]Uma definição alternativa para a propriedade de Ordenação Total Uniforme, que não permite a ocorrência de lacunas na sequência de mensagens a ser entregue, foi sugerida por Aguilera et al [2].

Capítulo 3

Mecanismo de Sincronização e Geração de Rodadas

A organização da computação em rodadas, cujos princípios foram discutidos na Seção 2.3, é um dos elementos essenciais para a implementação da solução de Difusão Totalmente Ordenada proposta neste trabalho (descrita no Capítulo 4). No modelo síncrono, onde a organização de computações em rodadas tem sua origem, considera-se a existência de— ou ao menos que é possível implementar [14, 42] — relógios sincronizados, acessíveis por todos os processos e que ditam o início das rodadas. Porém, no ambiente computacional considerado neste trabalho, os aglomerados de alto desempenho, esta hipótese de existência de relógios sincronizados não é válida [54]. Assim, este capítulo apresenta um mecanismo responsável por *sincronizar* a execução dos processos, que através da periódica geração e difusão de pulsos, organiza as computações em rodadas com uma semântica síncrona.

A descrição do mecanismo de sincronização proposto neste trabalho e da modelagem do sistema por ele considerada é realizada na Seção 3.1. A Seção 3.2 apresenta detalhes de sua implementação no ambiente dos aglomerados de alto desempenho. No restante do capítulo, a partir da Seção 3.3, apresenta-se a avaliação experimental do mecanismo de sincronização proposto e implementado, o que também constitui uma análise do comportamento temporal do ambiente computacional que hospeda sua execução.

3.1 Descrição do Mecanismo de Sincronização

A essência do mecanismo de sincronização proposto, que será descrito com mais detalhes no decorrer desta seção, é a seguinte. Um dos processos que participam da execução é eleito para ser o *sincronizador* do sistema. O sincronizador é responsável por ditar o ritmo da execução, através da periódica difusão de mensagens especiais chamadas de *tiques*. Os tiques difundidos pelo sincronizador têm o mesmo papel dos pulsos gerados

pelos antigos sistemas síncronos. Assim, sempre que um processo recebe um novo tique do sincronizador eleito, ele inicia uma nova rodada de sua execução.

A frequência com que os tiques são gerados é ditada pelo relógio do processo que hospeda o sincronizador. Supõe-se que todos os processos têm acesso a relógios locais e que estes relógios exibem, em sucessivas leituras, valores monotonicamente crescentes. Os relógios dos processos não são sincronizados, de modo que as diferenças entre os valores exibidos por relógios de diferentes processos podem ser arbitrariamente grandes. Entretanto, considera-se que os relógios dos processos apresentam taxas de progressão que, na maior parte do tempo, são estreitamente envolvidas pelo tempo real. Em outras palavras, espera-se que duas leituras ao relógio de um processo, com um intervalo de δ unidades de tempo entre elas, retornem dois valores t_1 e t_2, cuja diferença $t_2 - t_1$ seja de aproximadamente δ unidades. Porém, como o sistema que hospeda a execução não é de tempo real, não se tem garantias de que esta propriedade seja de fato válida.

Apoiando-se nestas suposições sobre o comportamento dos relógios, um processo, ao ser eleito como sincronizador, agenda a geração de uma sequência infinita de tiques para instantes $t_1, t_2, t_3, \ldots$ de seu relógio, distantes entre si Δ unidades — o período Δ é um parâmetro do sistema. Assim, supondo que o processo se torne o sincronizador no instante t_0 de seu relógio, a difusão do i-ésimo tique ocorrerá no instante $t_i = t_0 + i\Delta$, também aferido segundo seu relógio. Como resultado, se a taxa de progressão do relógio do sincronizador for estreitamente envolvida pelo tempo real, um tique será difundido a cada Δ unidades de tempo. Caso contrário, o intervalo entre as difusões será arbitrário, tanto maior, como menor que o valor esperado Δ, mas o sincronizador ainda irá difundir tiques com alguma frequência. O que se espera é que os relógios se comportem de forma próxima ao ideal em alguns períodos e de forma arbitrária em outros, mas que, em média — considerando-se períodos longos de execução, muito maiores que Δ — o intervalo entre a difusão dos tiques seja de aproximadamente Δ unidades de tempo.

Múltiplos sincronizadores podem coexistir no sistema, em períodos de assincronia ou no início das execuções, mas supõe-se que a partir de algum momento um único processo seja considerado o sincronizador. A eleição de que processo será o sincronizador é feita a partir de um identificador de fase, anexado aos tiques difundidos. Cada fase é associada a um processo sincronizador e os identificadores de fase são comparáveis. Assim, quando um processo que atua como sincronizador recebe um tique com número de fase superior ao por ele empregado, ele interrompe o envio de tiques. Já se um processo desconfia que o sincronizador eleito não esteja mais operante — pelo não recebimento de tiques por um longo período — ele passa a atuar como sincronizador, empregando para tal, um novo identificador de fase, maior que os empregados anteriormente. Da mesma forma, cada processo a cada instante participa de uma fase e só considera os tiques recebidos que pertençam àquela fase. Se um tique de fase superior é recebido, o processo ingressa na

nova fase e, portanto, passa a ignorar os tiques de fases anteriores.

Com o conceito de fase, os tiques se tornam unicamente identificáveis, através dos pares (f, i), onde f é seu identificador de fase e i é seu número de sequência na fase f. Um tique recebido é considerado *válido* por um processo quando ele pertence à fase de que o processo participa e for posterior (isto é, seu número de sequência for maior) ao último tique válido recebido. Somente tiques válidos levam ao início de novas rodadas e as rodadas são identificadas pelos tiques que as iniciam. Assim, se um processo que executa a rodada (f, j) recebe o tique (f', i), com $f' = f$ e $i > j$, este tique será considerado válido e levará o processo a finalizar sua j-ésima rodada da fase f, para dar início a sua i-ésima rodada da mesma fase. Caso contrário, o tique recebido será descartado — a menos do caso em que $f' > f$, já discutido, em que o processo ingressa na fase f'.

Rodadas de Comunicação Com a difusão periódica de tiques pelo sincronizador torna-se possível organizar a execução dos processos em uma sequência de rodadas de comunicação. Em cada rodada — isto é, com o recebimento de cada tique válido — os processos realizam a seguinte sequência de ações: (i) processam as mensagens recebidas durante a rodada anterior; (ii) difundem a mensagem referente à rodada que teve início; (iii) aguardam pelo recebimento das mensagens difundidas naquela rodada. À aplicação que emprega o mecanismo de sincronização para realizar sua comunicação são entregues, no início de cada rodada, as mensagens que foram recebidas na rodada anterior, para que elas sejam processadas. E, como resultado deste processamento, a aplicação gera a mensagem que deve ser difundida pelo processo na rodada que teve início.

A principal propriedade das computações baseadas em rodadas (descritas na Seção 2.3) é que as mensagens difundidas em uma rodada só podem ser entregues aos processos que as receberem no decorrer da mesma rodada. Para garantir que esta propriedade seja cumprida, toda mensagem difundida através do mecanismo de sincronização carrega o identificador da rodada à qual ela pertence e do processo que a difundiu. Somente as mensagens recebidas enquanto o processo executa a rodada em que elas foram difundidas são aceitas e somente uma mensagem proveniente de cada processo é aceita por rodada. As mensagens que foram difundidas em rodadas anteriores à executada pelo processo no instante do seu recebimento são sempre descartadas — o mesmo vale para mensagens que pertençam a fases diferentes da que o processo executa. Já as mensagens que pertencem a rodadas ainda não iniciadas pelo processo, são retidas até que tais rodadas sejam iniciadas ou até que elas recaiam no caso anterior e tenham que ser descartadas.

Com o uso do mecanismo de sincronização para gerar rodadas de comunicação, faz-se necessário agregar algumas suposições sobre o comportamento temporal do sistema. Dado que neste modo de execução as mensagens recebidas com atraso são definitivamente descartadas — e seu conteúdo é perdido pela aplicação — é necessário supor que as

latências para a entrega das mensagens difundidas sejam, ao menos na maior parte do tempo, controladas. O que não significa que se suponha a existência de limites superiores para as latências — o que caracterizaria uma comunicação síncrona — mas que seja possível estipular valores que, com certa probabilidade, limitem as latências para a entrega da maior parte das mensagens difundidas, o que é uma prerrogativa essencial do modelo assíncrono temporizado [25], descrito na Seção 2.2. Em adição, espera-se também que os instantes nos quais os processos iniciam uma mesma rodada não sejam arbitrários, mas que eles estejam normalmente concentrados em um período relativamente curto de tempo. Para que o mecanismo de sincronização apresente tal *precisão* é necessário que o sistema, na maior parte do tempo, apresente um comportamento *estável*. Isto é, que a taxa de tiques perdidos e as variabilidades nas latências para a entrega dos tiques e em seus atrasos de processamento sejam, na maior parte do tempo, reduzidas. Hipóteses que se espera que sejam válidas por conta da natureza do ambiente computacional empregado: um aglomerado de máquinas homogêneas, dedicadas à computação distribuída e conectadas por uma rede Ethernet de alto desempenho e com suporte nativo à difusão.

Assim, ao se estipular limitantes — isto é, valores máximos esperados, com validade probabilística — para (i) a precisão do relógio do sincronizador, (ii) a precisão obtida na sincronização dos processos e (iii) a latência das mensagens difundidas pelos processos, torna-se possível estipular o período Δ a ser empregado na difusão de tiques, de forma a se obter rodadas de comunicação com certa taxa máxima esperada de mensagens que serão descartadas por terem sido recebidas com atraso. Por um lado, a taxa esperada de mensagens descartadas será tão menor quanto maior for Δ, isto é, quanto menos restritivos forem os limitantes — e, portanto, mais flexíveis forem os prazos — adotados. Por outro lado, com menores frequências na geração de tiques, menor é o número de rodadas executadas e de mensagens difundidas por unidade de tempo e, portanto, maior é o tempo que os processos ficam (por vezes, desnecessariamente) bloqueados.

3.2 Implementação do Mecanismo de Sincronização

O mecanismo de sincronização foi implementado usando a linguagem Java e tem como base a plataforma desenvolvida para o Treplica [55]. A comunicação entre os processos se dá através do protocolo UDP de transporte não confiável de datagramas e da primitiva de comunicação em *multicast* do protocolo IP de rede. Acima destes protocolos nativos foi implementada uma abstração de transporte, que é responsável pelo endereçamento dos processos, pelo empacotamento e desempacotamento das mensagens e pela verificação de sua integridade. Com esta camada agrega-se às funcionalidades fornecidas pelo protocolo UDP/IP apenas a detecção de mensagens corrompidas, que são descartadas. Assim, a comunicação não é baseada em conexões e não é confiável: as mensagens difundidas

podem ser perdidas ou duplicadas, mas não podem ser corrompidas; e não há mecanismos de controle de fluxo, nem para a detecção ou recuperação de perdas. A comunicação é sempre realizada por difusão e os processos não precisam conhecer os endereços uns dos outros: as mensagens são enviadas para endereços IP-multicast pré-definidos e todos os processos se "cadastram" para receber as mensagens enviadas para tais endereços.

Cada processo mantém duas instâncias desta abstração de transporte, associadas a dois grupos IP-multicast (pares endereço e porta). A primeira instância é destinada às mensagens que carregam os tiques difundidos pelo sincronizador, que são mensagens pequenas, com cerca de 120 bytes. A segunda instância é empregada para a difusão e o recebimento das mensagens difundidas pelos processos em cada rodada, que possuem tamanhos variados. Nota-se que, apesar desta distinção lógica entre os transportes, as mensagens trocadas através deles compartilham um mesmo substrato de rede (camadas física, de enlace e de rede) e, portanto, competem pelos mesmos recursos.

A fonte de tempo empregada pelos processos encontra-se nos próprios processadores que hospedam as suas execuções, que mantém uma contagem de tempo baseada em sua frequência de funcionamento. Aos valores apresentados por estes contadores, o sistema aplica um deslocamento ajustável, a fim de que os valores retornados representem instantes do tempo real. As taxas com que estes relógios progridem dependem de vários fatores que afetam a frequência dos processadores (como a carga a ele aplicada e sua temperatura), o que faz com que não se possa considerá-los como fontes precisas de tempo. Não obstante, os valores retornados pela chamada de sistema para consultar estes relógios, cuja precisão é de nanossegundos ($1ns = 10^{-9}s$), são empregados pelos processos para aferir a passagem do tempo, pois as taxas de progressão por eles apresentadas são, na maior parte do tempo, estreitamente envolvidas pelo tempo real. Por outro lado, os valores exibidos por estes relógios apresentam precisões entre si e acurácias em relação ao tempo real insuficientes para que eles possam ser empregados pelos processos para sincronizar sua ações[1].

O sincronizador é implementado por uma rotina independente, iniciada quando o processo é eleito e executada enquanto ele se considerar o sincronizador do sistema. Para agendar a geração de tiques, o processo emprega temporizadores, cujas durações são computadas para que eles expirem nos instantes em que a geração de tiques foi agendada. Porém, por conta da complexidade do sistema — o custo associado às trocas de contexto, a imprecisão do escalonador de tarefas etc. — o controle da execução será normalmente passado à rotina que implementa o sincronizador com algum atraso. Assim, apesar da geração do tique ter sido agendada para um instante t_i, segundo o relógio do processador, ele será gerado de fato em um instante $t'_i \geq t_i$, segundo o mesmo relógio. Mesmo que este

[1] O deslocamento aplicado ao contador do processador é determinado pelo sistema operacional, a partir de leituras de relógios físicos presentes nas máquinas e de protocolos de sincronização remota, como o NTP [50]. Porém, a acurácia obtida por estes procedimentos, que é da ordem de frações de milisegundos ($10^{-3}s$), é ainda insuficiente para que se possa empregar tais relógios como se eles fossem sincronizados.

atraso seja considerável, para seguir a especificação apresentada na Seção 3.1, o instante no qual o próximo tique deve ser gerado continua sendo $t_{i+1} = t_i + \Delta$. Portanto, dado que o relógio do processador é considerado a referência de tempo, a magnitude e a variação dos atrasos na geração dos tiques determinam a imprecisão dos relógios locais dos processos em gerar tiques com o período esperado de Δ unidades do tempo real.

Além da rotina que implementa o sincronizador, cada processo mantém outras duas rotinas, para processamento das mensagens recebidas pelas duas instâncias de transporte. Estas rotinas compartilham uma variável, que mantém o identificador da rodada em execução. Quando um tique novo é recebido, ele é comparado com esta variável e, caso ele seja válido, a rodada é atualizada. As mensagens pertencentes à rodada finalizada, que foram coletadas pela outra rotina, são repassadas para processamento pela aplicação. Quando este processamento é finalizado, a mensagem gerada pela aplicação é difundida e a rotina principal bloqueia até a chegada do próximo tique. Nota-se que, por serem recebidas por uma camada de transporte própria e serem processadas por uma rotina independente e com maior prioridade, as mensagens que carregam os tiques não são sofrem com os atrasos de enfileiramento que atingem as demais mensagens difundidas.

3.3 Avaliação do Mecanismo de Sincronização

Esta seção apresenta a série de experimentos que foram realizados a fim de verificar a validade das condições de sincronia necessárias para a implementação do mecanismo de sincronização — e para a consequente organização das computações em rodadas — no ambiente computacional considerado neste trabalho: os aglomerados de alto desempenho com máquinas interconectadas através da tecnologia Ethernet de rede.

Ambiente experimental Os experimentos com o mecanismo de sincronização foram realizados em um aglomerado de máquinas Supermicro idênticas, equipadas com dois processadores Intel-Xeon Quadricore de 2.40 GHz e com 12 GB de memória RAM. O sistema operacional empregado é o GNU/Debian Linux 6.0 (Squeeze) e os experimentos são executados sobre uma máquina virtual Sun Java SE Runtime Environment 1.6. Cada máquina está equipada com duas interfaces de rede Ethernet Gigabit, sendo que uma delas é dedicada ao processamento distribuído. Os processos são interconectados por um Switch 3Com 4200G Ethernet Gigabit de 24 portas, também dedicado aos experimentos e com tempo de resposta, para transmissão em nível físico, de cerca de 0.2 ms.

Descrição dos experimentos Cada experimento descrito nesta seção é composto por um conjunto de execuções independentes. Em cada execução um dos processos é previamente escolhido para ser o sincronizador e assim permanece durante toda execução.

Ou seja, o sincronizador é um processo fixo, determinado a priori, que não falha e não é suspeito pelo algoritmo de eleição do restante dos processos — que é desativado. Ao sincronizador é fornecido o período Δ a ser empregado na geração e difusão de tiques, que é dado em microssegundos ($1\mu s = 10^{-6}s$) e corresponde à duração esperada (ou de referência) para as rodadas geradas pelo mecanismo de sincronização.

No início de toda rodada, cada processo que a executa difunde uma nova mensagem. Estas mensagens são compostas por um cabeçalho com 252 bytes e por uma carga útil (ou *payload*) de S bytes aleatórios, sendo S um parâmetro do experimento. O tamanho das mensagens difundidas pelos processos é, portanto, fixa e determina a carga aplicada à rede durante as execuções do experimento — uma vez que o tráfego induzido pelo envio de tiques, que têm cerca de 120 bytes cada, é desprezível. Supondo que a frequência com que tiques são gerados e com que rodadas são iniciadas pelos processos são as previstas, cerca de $10^6/\Delta$ rodadas são iniciadas por segundo. Em adição, se todos os n processos participam de todas as rodadas e difundem uma mensagem que transporta S bytes de carga útil, aplica-se à rede durante as execuções do experimento, teoricamente, uma carga de $8n(S + 252)/\Delta$ Mb/s (a carga é dada em milhões de bits por segundo).

3.3.1 Avaliação do Sincronizador

O primeiro aspecto avaliado no mecanismo de sincronização foi a acurácia com a qual o processo sincronizador é capaz de agendar a geração e o envio de tiques. Conforme descrito na Seção 3.2, este agendamento é realizado através da escolha de uma sequência infinita de instantes $t_1, t_2, t_3, \ldots$, distantes entre si Δ unidades de tempo, nos quais o sincronizador solicita que o sistema o "acorde" para enviar um novo tique. Mas como o sistema que hospeda o processo sincronizador não é um sistema de tempo real, não se tem garantias quanto à acurácia com que os agendamentos são cumpridos. Assim, seja t_i o instante para que o envio do i-ésimo tique foi agendado, denota-se por ts_i o instante em que o mesmo tique foi de fato gerado, que é o seu *timestamp* ou seu "carimbo de tempo". Como a implementação do sincronizador garante que $ts_i \geq t_i$, para todo tique i, define-se como o *atraso de processamento* do i-ésimo tique a diferença $ts_i - t_i$, que pode ser facilmente mensurada pelo processo que atua como sincronizador.

O que se espera, dadas as características do ambiente experimental — um aglomerado de máquinas de alto desempenho, dedicadas à computação distribuída —, é que os atrasos de processamento observados sejam reduzidos na maior parte do tempo (para a maior parte dos tiques). Porém, mais importante do que os atrasos de processamento serem reduzidos, é que a sua variância seja controlada, de modo que, para a maior parte dos tiques, o intervalo entre as suas difusões seja de fato da ordem do valor esperado Δ. Assim, com objetivo de se avaliar a acurácia dos sincronizadores na geração de tiques

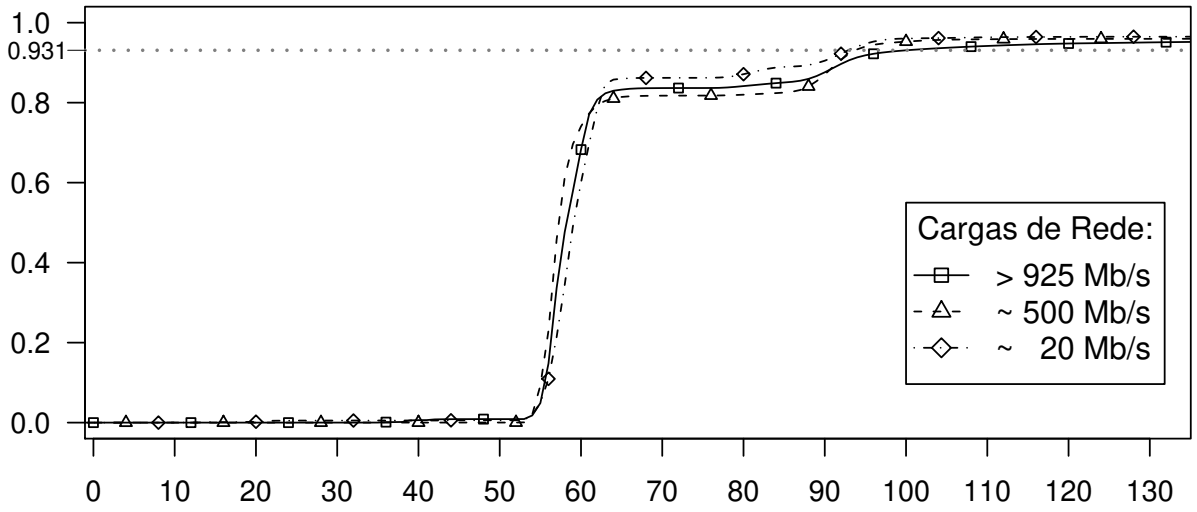


Figura 3.1: Função de distribuição cumulativa para os atrasos de processamento para os tiques em cinco execuções, aferidos em μs e gerados com período $\Delta = 500\mu s$.

realizou-se uma série de experimentos, nos quais se mensurou os atrasos de processamento aferidos por diferentes processos, segundo os relógios de seus processadores, na geração de tiques. Os resultados apresentados a seguir foram computados a partir de cinco execuções independentes, com cinco sincronizadores diferentes e 300 mil tiques difundidos em cada uma delas. O período empregado na geração de tiques foi $\Delta = 500\mu s$, o que equivale à taxa de cerca de 2 mil tiques gerados por segundo pelo sincronizador. Como se verá nas próximas seções, esta é uma duração consideravelmente curta para as rodadas, uma vez que o tempo de resposta do sistema é, na maior parte dos casos, superior a $500\mu s$. Porém, ao empregar este período foi possível explorar o comportamento do mecanismo de sincronização em situações limite, com altas cargas de rede e de processamento.

Na Figura 3.1 apresentam-se as funções de distribuição acumulada para os atrasos de processamento mensurados em três experimentos, nos quais se empregou um tamanho diferente para as mensagens, de forma a se obter três cenários de carga aplicada à rede em decorrência das mensagens difundidas pelos processos no início de cada rodada. No primeiro cenário os processos difundiram mensagens "ocas" — isto é, sem carga útil, com apenas 252 bytes — o que resultou em uma carga de 20 Mb/s aplicada à rede. No segundo cenário, o tamanho S da carga útil foi computado de forma que a carga aplicada à rede fosse de 500 Mb/s, isto é de metade da capacidade da rede, que é de 1 Gb/s. Mesmo com a variabilidade na carga de rede medida pelos processos, este valor foi atingido e se manteve em todas as execuções. No terceiro cenário, o objetivo foi avaliar o comportamento do mecanismo de sincronização na presença de uma rede completamente congestionada. Para tal, dobrou-se o tamanho das mensagens em relação ao cenário anterior, de forma a se

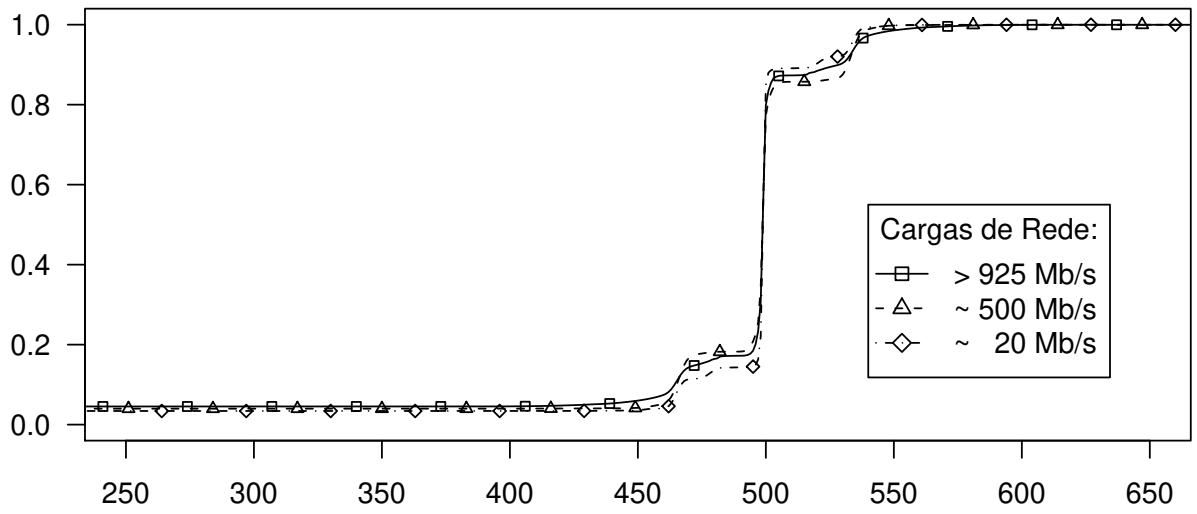


Figura 3.2: Função de distribuição cumulativa dos intervalos entre a geração de tiques consecutivos pelo sincronizador, aferidos em μs e com valor de referência $\Delta = 500\mu s$.

obter uma carga teórica de 1Gb/s com a difusão de mensagens. A carga de rede mensurada nestas execuções oscilou bastante, em poucos momentos atingiu 1 Gb/s, mas se manteve na maior parte do tempo acima de 925 Mb/s, que é o valor de referência apresentado.

Para os três cenários de carga avaliados, considerados em ordem crescente, 96.0%, 95.3% e 93.1% dos tiques gerados tiveram seus atrasos de processamento limitados por $100\mu s$ — a menor destas proporções, 93.1%, é representada no gráfico por uma linha pontilhada. Em adição, nos três experimentos, mais de 80% dos atrasos medidos estão concentrados em um intervalo curto, entre 50 e $65\mu s$. Estes dados nos permitem afirmar que, mesmo com períodos curtos como $500\mu s$, os processos se mostraram capazes de cumprir a maior parte dos agendamentos com uma boa acurácia. Como consequência, o sincronizador difunde a maior parte dos tiques com intervalos entre a difusão de dois deles muito próximos ao período Δ fornecido. Este comportamento é ilustrado pela distribuição apresentada na Figura 3.2, construída a partir destes mesmos experimentos, em que cerca de 92.6% dos intervalos entre a difusão de tiques foram entre 450 e $550\mu s$ — isto é, na faixa $\Delta \pm 50\mu s$ — mesmo para a maior carga aplicada à rede.

Por outro lado, para os três experimentos, respectivamente, em ordem crescente de carga, 3.0%, 3.5% e 4.5% dos atrasos de processamento mensurados foram superiores a $500\mu s$. O que significa que tiques foram gerados em instantes posteriores ao instante agendado para a geração dos tiques que os sucediam, o que fez com que a difusão deles ocorresse quase simultaneamente. O reflexo deste comportamento discrepante é observado na distribuição de intervalos entre a difusão de tiques, na qual, respectivamente, 3.36%, 3.91% e 4.36% dos valores são inferiores a $100\mu s$. As rodadas induzidas por tais tiques

tendem a ser inúteis para os processos, uma vez que a sua duração tende a ser muito baixa, o que é um comportamento indesejável. Porém, em todos os experimentos realizados, mesmo com valores superiores para Δ e cargas inferiores aplicadas à rede, este mesmo comportamento foi observado, mesmo que em menores proporções, o que indica que os atrasos de processamento provavelmente não são limitados superiormente.

3.3.2 Duração das Rodadas

O segundo aspecto avaliado no mecanismo de sincronização foi a duração das rodadas executadas pelos processos, como resultado do recebimento dos tiques difundidos pelo sincronizador. A duração de uma rodada i, segundo um processo p que a executa, é dada pela diferença entre os instantes em que os dois tiques que a delimitam — o tique i que a inicia e o tique $j > i$ que a encerra — são entregues a p. Idealmente, a duração de qualquer rodada deveria ser da ordem do parâmetro Δ fornecido ao mecanismo de sincronização. Porém, para que isto ocorra na prática, é importante que: (i) o tique que encerra a i-ésima rodada seja o tique $i+1$, isto é, as mensagens que carregam os tiques não sejam perdidas nem reordenadas; (ii) os atrasos de processamento do sincronizador para a difusão dos tiques i e $i+1$ sejam similares e, portanto, o intervalo entre a difusão dos tiques seja da ordem de Δ; e (iii) as latências de transmissão das mensagens que carregam os tiques i e $i+1$ e os seus atrasos de processamento no processo p também sejam similares.

Quando uma ou mais destas condições de sincronia não são cumpridas pelo sistema, têm-se rodadas cujas durações provavelmente irão diferir do valor Δ esperado. Dentre as situações ocasionadas pelo não cumprimento destas condições, destacam-se três que são particularmente indesejadas. A primeira situação ocorre quando o tique que deveria dar início à i-ésima rodada não é recebido por um processo ou quando ele é recebido muito tardiamente, isto é, após a chegada de um tique $j > i$ — o que torna o tique i inválido. Em ambos os casos, a rodada i não é executada pelo processo, que não difunde e nem recebe as mensagens daquela rodada, tornando-se uma rodada perdida. A segunda situação indesejada ocorre quando os tiques que delimitam uma rodada são recebidos dentro de um intervalo muito estreito de tempo, o que resulta em uma rodada de curta duração, com consequências similares a ter-se uma rodada perdida. A terceira situação ocorre quando o tique que encerra a rodada tarda a chegar, o que faz com que a duração da rodada seja muito superior ao valor esperado Δ. Apesar desta situação não ser particularmente prejudicial, ela se torna indesejada porque a sua ocorrência está normalmente conjugada à ocorrência das outras duas situações em rodadas anteriores ou sucessivas.

O objetivo dos experimentos apresentados nesta seção foi verificar a frequência com que estas três situações indesejáveis ocorrem e, consequentemente, estimar qual a proporção das rodadas geradas pelo sincronizador são de fato úteis aos processos.

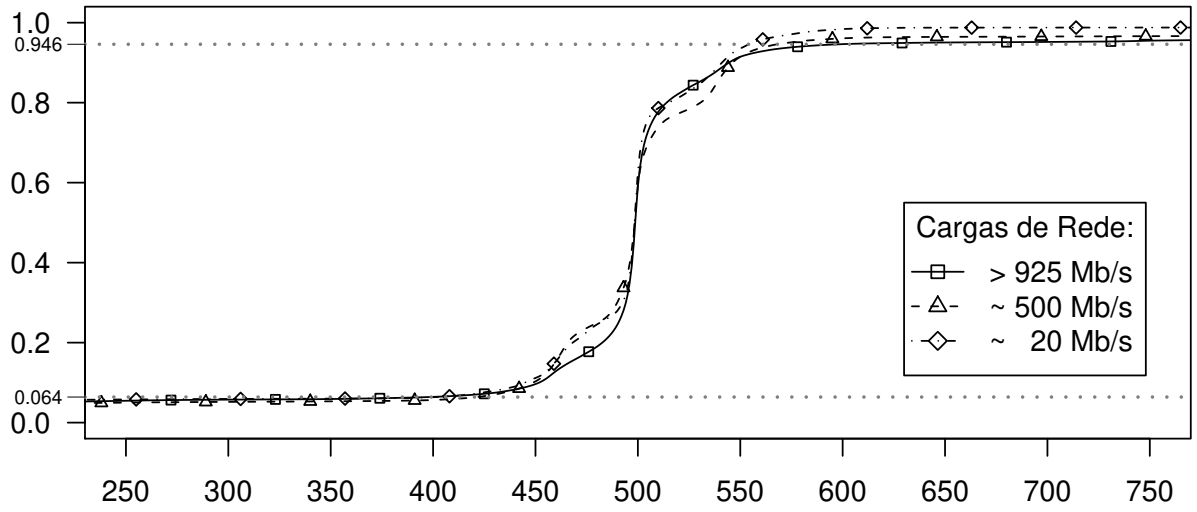


Figura 3.3: Função de distribuição cumulativa para as durações de 1.5 milhões de rodadas, aferidas por 5 processos, com valor de referência $\Delta = 500\mu s$, para três cenários de carga.

Variação de carga Os primeiros resultados, apresentados na Figura 3.3, referem-se aos mesmos experimentos descritos na seção anterior, isto é, execuções com $n = 5$ processos e duração esperada para as rodadas $\Delta = 500\mu s$. As funções de distribuição cumulativas apresentadas foram computadas a partir das durações aferidas pelos processos para 1.5 milhão de rodadas, em cinco execuções independentes, em que cada processo atuou como sincronizador — portanto, são computadas com base em 7.5 milhões de medições.

As duas linhas pontilhadas da Figura 3.3 delimitam a porção das rodadas que tiveram duração mensurada dentro de $\Delta \pm 100\mu s$, isto é, com desvios de até $100\mu s$ em relação ao valor esperado. Nota-se que a maior parte das rodadas está concentrada nesta faixa de valores, ou seja, que este é o comportamento preponderante nas execuções, mesmo quando a carga aplicada à rede é alta. Mais especificamente, no cenário de rede completamente congestionada, 88.2% das rodadas geradas tiveram durações aferidas pelos processos no intervalo $\Delta \pm 100\mu s$; com cargas de rede da ordem de 500 Mb/s, esta proporção sobe para 90.4% e com cargas de 20 Mb/s ela atinge 92.0%. Dentre o restante das rodadas, destaca-se as que se enquadram nas três situações indesejadas relacionadas. Para as cargas teóricas de 1 Gb/s, 500 Mb/s e 20 Mb/s, respectivamente, 4.00%, 4.19% e 5.34% das rodadas tiveram durações consideradas muito curtas, de até $100\mu s$. No outro extremo, as rodadas cujas durações aferidas foram superiores a $1000\mu s$ (isto é, a 2Δ) e aquelas cujo tique que deveria iniciá-las não foi recebido ou teve que ser descartado foram 4.1%, 3.2% e 1.22%. Em particular, as rodadas que foram perdidas pelos processos (com durações computadas como infinitas) foram, respectivamente, 3.6%, 2.5% e 1.19% do total de rodadas.

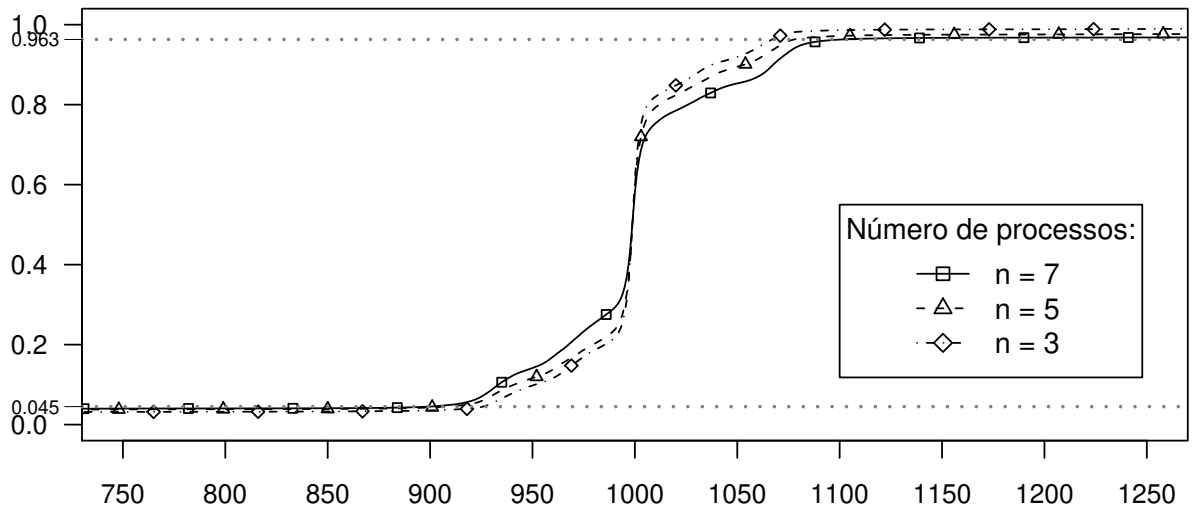


Figura 3.4: Funções de distribuição cumulativa para as durações de 1.5 milhões de rodadas, com valor de referência $\Delta = 1000\mu s$ e cargas de 500 Mb/s aplicadas à rede.

Variação de escala Uma segunda rodada de experimentos foi realizada com intuito de avaliar o impacto do número de processos que participam das execuções nas durações computadas para as rodadas. Para tal, empregou-se um período $\Delta = 1000\mu s$ para a geração de tiques e computou-se o tamanho das mensagens difundidas de modo a se obter uma carga aplicada à rede de 500 Mb/s em todos os experimentos. As distribuições para as durações computadas para rodadas, em cinco execuções com 1.5 milhões de tiques difundidos em cada uma delas, são apresentadas não Figura 3.4. Nota-se que o formato das três funções de distribuição é idêntico ao observado na Figura 3.3, com a diferença que a região central, delimitada pelas linhas pontilhadas, torna-se maior quando aumenta-se a duração esperada para as rodadas. Assim, com $n = 5$, tem-se 92.8% das rodadas com durações no intervalo delimitado por 900 e $1100\mu s$ ($\Delta \pm 100\mu s$), 2.2% delas são perdidas ou têm durações acima de $2000\mu s$ (2Δ) e 2.9% delas tem duração de até $100\mu s$[2]. Com $n = 3$ processos tem-se resultados melhores: 95.1% das durações em $\Delta \pm 100\mu s$, 1.3% abaixo de $100\mu s$ e 1.0% das rodadas foram perdidas ou tiveram durações acima de 2Δ. Já com $n = 7$ processos, a proporção de rodadas perdidas ou com altas durações é 3.1%, as com menos de $100\mu s$ são 3.3% e as dentro da faixa central são 91.8% do total de rodadas — proporção que, apesar de ser inferior aos demais casos, ainda é bastante interessante.

[2]Para efeito de comparação, os mesmos resultados para $\Delta = 500\mu s$ foram 90.4% das durações em $\Delta \pm 100\mu s$, 3.2% delas com durações acima de 2Δ ou perdidas e 4.19% com durações de até $100\mu s$.

3.3.3 Latências de Comunicação

Avaliadas a acurácia do sincronizador em gerar tiques em instantes pré-determinados e eficiência e escalabilidade das rodadas induzidas pelo recebimento dos tiques, analisa-se a partir desta seção o padrão de comunicação estabelecido pelo mecanismo de sincronização. O objetivo é verificar se, com a organização das execuções em rodadas, o sistema apresenta um comportamento temporal que permita classificá-lo como assíncrono temporizado, o que confirmaria uma das principais hipóteses que embasam este trabalho.

Nesta seção avaliam-se as latências observadas para a entrega das mensagens difundidas pelos processos, em função da carga útil que elas transportam. Estas mensagens são difundidas no início de cada rodada e transportam, como carga útil, uma sequência de S bytes aleatórios. O que se espera verificar é que, para diferentes valores de S, seja possível estipular limitantes superiores para as latências que sejam válidos na maior parte do tempo, isto é, que sejam respeitados pela maior parte das mensagens difundidas.

A primeira decisão que teve que ser tomada ao se avaliar latências de comunicação foi como mensurá-las. Ao contrário dos demais aspectos analisados, a latência para a entrega de uma mensagem não pode ser aferida diretamente por um processo através de seu relógio, a menos que ele esteja sincronizado com os relógios dos demais processos. Como esta suposição não é válida no ambiente experimental e nem é uma hipótese do modelo computacional para ele assumido, a medição das latências foi realizada de forma indireta. De modo que, ao invés de mensurar a latência para a entrega de cada mensagem, aferiu-se o seu Tempo Total de Resposta (TTR) em relação ao tique que desencadeou seu envio. Isto é, computou-se a diferença entre o instante de recebimento de uma mensagem — pertencente a uma rodada i — e o instante em que o tique correspondente — o tique i, que desencadeou o seu envio — foi difundido pelo sincronizador. Portanto, o TTR de uma mensagem m difundida por um processo p inclui, além da latência de m, a latência para a entrega de um tique a p e o atraso de processamento de um tique em p.

Além de considerar a imprecisão gerada pela medição indireta das latências, há uma outra questão que deve ser levada em consideração ao se avaliar os valores de TTRs aferidos. Dado o padrão de comunicação estabelecido pelo mecanismo de sincronização, espera-se que todos os processos difundam mensagens quase que simultaneamente. De modo que as mensagens difundidas (em particular as que pertencem a uma mesma rodada) competem pelos recursos da rede e se "enfileiram" nos canais de comunicação, o que faz com que suas latências não possam ser consideradas independentes. Esta dependência será tão maior quanto mais processos difundirem mensagens ao mesmo tempo e quão maiores forem as mensagens; e, em particular, ela é acentuada quando o intervalo entre as difusões é reduzido, pois passa a haver também interferência entre mensagens de rodadas diferentes. Por todos estes motivos, espera-se que os TTRs computados estejam distribuídos, de forma razoavelmente uniforme, em um intervalo de valores, e não concentrados ao redor

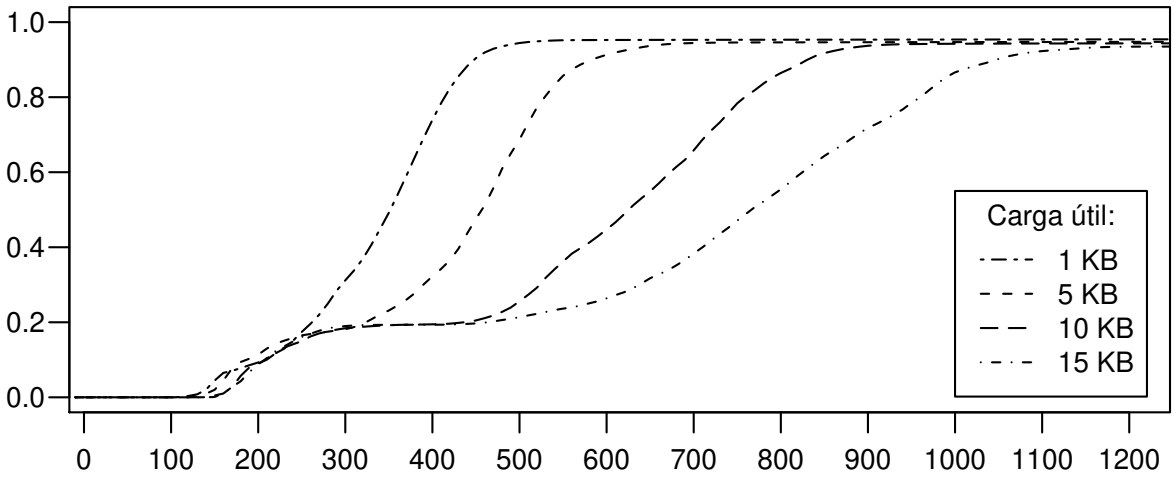


Figura 3.5: Funções de distribuição cumulativa do Tempo Total de Resposta (aferido em μs) das mensagens difundidas por $n = 5$ processos, em função de seus tamanhos.

de um valor médio, como foi observado nos aspectos anteriormente avaliados.

Este comportamento pode ser observado na Figura 3.5, que apresenta as funções de distribuição cumulativa para os TTRs mensurados em quatro experimentos, nos quais se variou o tamanho S da carga útil das mensagens. Cada experimento foi composto por cinco execuções independentes, nas quais um dos $n = 5$ processos foi o sincronizador e mensurou os TTRs das mensagens — visto que os instantes de envio dos tiques foram registrados segundo o seu relógio. Em cada execução, o sincronizador difundiu 300 mil tiques, com intervalo $\Delta = 1000\mu s$ entre eles. Porém, para obter as distribuições foram desconsideradas as 50 mil primeiras rodadas de cada execução, por conta da maior taxa de perdas de tiques observada nos segundos iniciais de cada execução. De forma que os resultados apresentados foram computados a partir de 750 mil rodadas, nas quais 3.75 milhões de mensagens deveriam ter sido difundidas. Para referência, as cargas de rede mensuradas nos experimentos foram, considerando as cargas úteis de 1 KB, 5 KB, 10 KB e 15 KB, respectivamente, de cerca de 50 Mb/s, 210 Mb/s, 400 Mb/s e 600 Mb/s.

As quatro distribuições apresentadas na Figura 3.5 têm em comum uma região inferior, que representa pouco menos que 20% das mensagens difundidas e concentra os valores de TTR mais reduzidos dentre os observados. A razão para estes TTRs serem reduzidos é que eles se referem às mensagens que foram difundidas pelo processo que hospeda o sincronizador da execução e que, portanto, não foram transmitidas via rede, mas via *loopback* (que é a interface de rede destinada à comunicação local). Assim, a latência de

transmissão de tais mensagens corresponde, na realidade, a um atraso de processamento[3]: é o custo das trocas de contexto e de se copiar as mensagens entre *buffers*. É interessante notar, que mesmo para estas mensagens "locais", os TTRs mensurados dependem do tamanho das mensagens difundidas: quando a carga útil é de 1 KB, os menores TTRs são da ordem de $130\mu s$, enquanto que com 15 KB, não são inferiores a $160\mu s$.

Após esta região inicial observa-se, principalmente para as mensagens maiores, um intervalo de TTRs em que praticamente nenhuma mensagem é recebida. De fato, para os quatro tamanhos de mensagem, do menor para o maior, quase nenhuma mensagem transmitida via rede teve TTR inferior a $250\mu s$, $310\mu s$, $440\mu s$ e $470\mu s$. A partir destes valores as funções de distribuição assumem um comportamento monotonicamente crescente, até atingir novamente um patamar de estabilidade. Para as mensagens com 1 KB de carga útil este patamar inicia-se com TTR = $550\mu s$, a partir de onde a função praticamente se estabiliza em cerca de 95.2%; para cargas de 5 KB, a função se estabiliza em 94.5% a partir de $700\mu s$; para cargas de 10 KB os valores são $950\mu s$ e 94.2%; e para cargas de 15 KB os valores são $1200\mu s$ e 93.5%. Nota-se que, como anteriormente referido, com o aumento da carga útil das mensagens os TTRs mensurados, além de serem mais elevados, estão menos concentrados: as taxas de crescimento de suas funções de distribuição são menores e eles estão distribuídos em intervalos com maior extensão. Em particular, para carga útil de 15 KB observa-se que uma porção das mensagens tem TTRs superiores ao intervalo $\Delta = 1000\mu s$ entre as difusões e, portanto, provavelmente estas mensagens sofreram com a interferência das mensagens que foram difundidas nas rodadas anteriores.

Em relação à porção das funções de distribuição que é não apresentada na Figura 3.5, ressalta-se elas continuam crescendo, mesmo que com taxa bastante reduzida, durante todo intervalo avaliado. Isto significa que há, mesmo em quantidades muito reduzidas, mensagens com TTRs muito superiores aos valores esperados — superiores, inclusive, a 60 segundos, nestes e em outros experimentos realizados — o que indica que de fato não se pode supor a existência de limitantes superiores (razoáveis) para os TTRs. O último aspecto analisado é a proporção de mensagens que não foram recebidas pelos sincronizadores — os processos que mensuraram os TTRs — e que, em termos de função de distribuição, tiveram seus TTRs computados como infinitos. Com 1 KB de carga útil, 1.68% das mensagens esperadas não foram recebidas; com 5 KB esta porcentagem sobe para 2.62%, sobe para 2.95% com 10 KB e chega a 4.08% com 15 KB. Apesar de estas taxas incluírem mensagens que não foram difundidas pelos processos — por eles não executarem certas rodadas — elas representam predominantemente mensagens difundidas que foram perdidas pela rede.

[3]A difusão dos tiques ocorre com o modo *loopback* desativado, de modo que o TTR destas mensagens "locais" também inclui a latência de transmissão (via rede) de um tique e seu atraso de processamento.

3.4 Eficiência das Rodadas de Comunicação

A principal finalidade do mecanismo de sincronização é organizar a execução do sistema em uma sequência de rodadas de comunicação, nas quais os processos difundem mensagens uns para os outros. Como descrito na Seção 2.3, a principal característica deste modo de execução é que as rodadas são fechadas para comunicação, isto é, que as mensagens difundidas em uma rodada só são entregues aos processos que as recebem no decorrer da mesma rodada. O que, em outras palavras, significa que uma mensagem só é entregue aos processos que a recebem *a tempo* (ou seja, antes do final da rodada na qual ela foi difundida) e será desconsiderada pelos demais, nos quais ela é recebida *com atraso*. Neste contexto, a *eficiência* das rodadas de comunicação geradas é dada pela proporção das mensagens difundidas em uma execução que é de fato entregue aos processos. Ou seja, é a proporção das mensagens que não foram perdidas pela rede e que não foram descartadas pelos processos (por terem sido recebidas com atraso) e que, portanto, são úteis a uma aplicação que empregue o mecanismo de sincronização para difundir suas mensagens.

Nesta seção avaliamos como a escolha dos parâmetros Δ, S e n impacta a eficiência das rodadas obtidas pelo mecanismo de sincronização, em termos das proporções de mensagens temporizadas, atrasadas e perdidas aferidas nos experimentos. Em uma definição mais precisa, uma mensagem é temporizada segundo um processo, se ela tiver sido recebida por ele no decorrer da rodada em que ela foi difundida. Em contrapartida, uma mensagem é classificada como atrasada por um processo quando ela é recebida por ele após o final da rodada na qual ela foi difundida. Nota-se que uma mensagem pode ser considerada atrasada mesmo quando a sua latência de entrega estiver dentro do esperado, pois a duração da rodada na qual ela foi difundida, segundo o processo que a classifica, pode ter sido inferior à duração esperada. Assim, a taxa de mensagens atrasadas representa de fato a proporção das mensagens difundidas que são descartadas pelos processos por conta da ocorrência qualquer tipo de falha de temporização. Finalmente, uma mensagem é considerada por um processo como perdida quando ela não for recebida, nem durante a rodada em que ela deveria ter sido difundida, nem em rodadas posteriores[4]. De modo que esta mensagem, ou não foi difundida (por seu emissor não ter participado da rodada em que ela deveria ter sido recebida), ou foi perdida pelos canais de comunicação.

Variação de S. O primeiro fator avaliado foi o tamanho S da carga útil transportada pelas mensagens. Para analisar este fator foram empregados os experimentos realizados na Seção 3.3.3, nos quais o sistema era composto por $n = 5$ processos e a duração das rodadas foi $\Delta = 1000\mu s$. Os resultados, apresentados na Tabela 3.1, demonstram que a eficiência

[4]As mensagens que são recebidas antes do início da rodada a que pertencem (isto é, mensagens adiantadas) são retidas e entregues ao processo, na rodada adequada, como mensagens temporizadas.

das rodadas — denotada pela proporção de mensagens classificadas como temporizadas — cai com o aumento do tamanho da carga útil transportada pelas mensagens. Esta relação já era esperada, pois mensagens maiores tendem a ter latências maiores e mais variáveis e, portanto, a probabilidade delas serem recebidas a tempo pelos processos será menor. Na tabela 3.1 é possível observar a associação entre a eficiência computada para as rodadas e a proporção das mensagens que tiveram TTRs aferidos de até $1000\mu s$. Uma exceção que merece destaque ocorre para a carga útil de 15 KB, em que estes dois valores divergem em quase 5%. O motivo desta divergência — que em parte se deve à variabilidade nas durações efetivas nas rodadas — é que os TTRs aferidos incluem as latências para a entrega e processamento de um tique. Esta latência extra torna-se evidente no caso das mensagens com 15 KB de carga, dentre as quais 5.67% têm TTRs entre 1000 e $1100\mu s$ — para as demais cargas, a proporção de mensagens nesta faixa de TTRs não chega a 0.1% — o que justifica esta maior diferença entre a eficiência e os TTRs limitados por Δ.

Carga útil das mensagens	1 KB	5 KB	10 KB	15 KB
Carga aplicada à rede	50 Mb/s	210 Mb/s	400 Mb/s	600 Mb/s
TTR de até $1000\mu s$	95.36%	94.68%	94.23%	86.61%
Mensagens temporizadas	95.32%	94.75%	94.32%	91.53%
Mensagens atrasadas	3.00%	2.65%	2.72%	4.39%
Mensagens perdidas	1.68%	2.62%	2.95%	4.08%

Tabela 3.1: Eficiência das rodadas de comunicação com $\Delta = 1000\mu s$ e $n = 5$ processos.

Outro aspecto observado na Tabela 3.1 é que, além da taxa de mensagens atrasadas, a taxa de mensagens perdidas cresce com o aumento do tamanho das mensagens difundidas. Como já foi dito, as mensagens classificadas como perdidas são aquelas que não foram recebidas pelo processo que afere a eficiência, pois (i) não foram difundidas ou (ii) foram perdidas pelos canais de comunicação. Em ambos os casos as perdas decorrem de falhas de comunicação — o primeiro caso decorre basicamente da perda de tiques pela rede (eles raramente são recebidos fora de ordem), que faz com que processos não participem de rodadas — que normalmente estão relacionadas com uma alta carga aplicada à rede. Assim, mais do que o tamanho das mensagens, é a carga aplicada à rede como resultado das mensagens difundidas (que é proporcional a S) que aumenta a taxa de mensagens perdidas — esta associação pode ser observada de forma mais clara na Tabela 3.2.

Outra questão importante na análise das taxas de perdas de mensagens é o tamanho dos *buffers* de recepção empregados. Nestes experimentos — e em todos os experimentos descritos neste capítulo — empregou-se *buffers* de recepção de 127 KB, que é o valor padrão da distribuição GNU/Linux utilizada. A razão para não se empregar *buffers*

maiores é que, se por um lado se reduziria a perda de mensagens, por outro se teria um aumento nas latências de comunicação, devido aos atrasos advindos do enfileiramento de mensagens nos *buffers*. Assim, como ao se empregar rodadas de comunicação se torna indiferente receber mensagens com atraso ou não recebê-las — ambas são, do ponto de vista da aplicação, mensagens perdidas — optou-se pelo não uso de *buffers* ou o uso de *buffers* mínimos. Em adição, ao se repetir os experimentos de onde foram extraídos os valores da Tabela 3.1 com *buffers* de recepção de 4 MiB (isto é, 32 vezes maiores) não se notou uma variação significativa da taxa de mensagens perdidas. Como exemplo, para as mensagens com 15 KB de carga útil (as com maiores perdas), a taxa caiu apenas de 4.08% para 4.05%.

Variação de Δ. O segundo fator — e, provavelmente, o mais óbvio — que tem impacto na eficiência das rodadas de comunicação é a duração Δ esperada para elas. Com o aumento a duração das rodadas se aumenta a probabilidade se receber as mensagens difundidas a tempo e, portanto, a probabilidade das rodadas terem alta eficiência. Por outro lado, com o aumento de Δ, a quantidade de mensagens que podem ser difundidas por unidade de tempo — a vazão do sistema — diminui. Assim, de modo geral, a duração Δ das rodadas é o parâmetro do sistema que precisa ser estipulado, a partir do número de processos n e do tamanho S das mensagens, tendo em vista a eficiência desejada para as rodadas. A Tabela 3.2 apresenta as eficiências obtidas em experimentos idênticos aos da Seção 3.3.3, mas nos quais a carga útil S transportada pelas mensagens foi fixada em 10 KB e a duração esperada para as rodadas foi variada, em torno do valor $\Delta = 1000\mu s$ de referência. As taxas de mensagens temporizadas e atrasadas obtidas confirmam o que se esperava: ao se aumentar Δ e se manter os demais parâmetros fixos, as rodadas têm maiores eficiências. Em relação às taxas de mensagens perdidas, nota-se que elas de fato estão associadas à carga aplicada à rede, uma vez que em todos os experimentos o tamanho das mensagens é o mesmo.

Duração das rodadas	$800\mu s$	$900\mu s$	$1000\mu s$	$1100\mu s$	$1200\mu s$
Carga aplicada à rede	495 Mb/s	445 Mb/s	400 Mb/s	365 Mb/s	338 Mb/s
Mensagens temporizadas	92.13%	94.11%	94.32%	94.80%	95.44%
Mensagens atrasadas	4.25%	2.84%	2.72%	2.50%	2.17%
Mensagens perdidas	3.62%	3.04%	2.95%	2.70%	2.38%

Tabela 3.2: Eficiência das rodadas de comunicação com $S = 10$ KB e $n = 5$ processos.

Variação de n. O último fator analisado foi a quantidade de processos que participam do sistema e que, portanto, difundem mensagens a cada rodada. Dois dos efeitos do aumento do número de processos no sistema já foram discutidos em seções anteriores. O primeiro efeito, discutido na Seção 3.3.2, é a redução da qualidade da sincronização obtida, como resultado do aumento das proporções de rodadas que não são executadas pelos processos ou que têm durações discrepantes. Este efeito é, de certa forma, esperado, pois a frequência com que as falhas de temporização — latências de comunicação ou atrasos de processamento discrepantes — ocorrem tende a ser maior com o aumento do número de processos no sistema e com a difusão de um número maior de mensagens. O segundo efeito, discutido na Seção 3.3.3, está relacionado às latências para a entrega das mensagens, cujos valores e variâncias tendem a aumentar quando um número maior de processos difundem mensagens (quase que) simultaneamente. Em conjunto, a consequência destes dois efeitos é o aumento das taxas de mensagens classificadas como atrasadas — e, portanto, uma menor eficiência das rodadas — quando se tem um aumento no número n de processos no sistema, não acompanhado por um aumento da duração Δ esperada para as rodadas.

Este efeito pode ser observado na Tabela 3.3, que compara a eficiência obtida para as rodadas em três experimentos, onde se variou o número n de processos no sistema. A duração esperada para as rodadas empregada foi $\Delta = 1000\mu s$ e o tamanho da carga útil transportada pelas mensagens difundidas foi computada de modo a se obter uma mesma carga de cerca de 400 Mb/s aplicada à rede em todos os experimentos. Nota-se que, a despeito do tamanho das mensagens difundidas em cada experimento, a proporção de mensagens atrasadas observadas com $n = 7$ é de fato superior à apresentada com $n = 5$ (que é superior a com $n = 3$), o que se reflete nas eficiências aferidas para as rodadas.

Número de processos	$n = 3$	$n = 5$	$n = 7$
Carga útil das mensagens	17 KB	10 KB	7 KB
TTRs limitados por $1000\mu s$	95.94%	94.23%	92.83%
Mensagens temporizadas	95.96%	94.32%	93.09%
Mensagens atrasadas	2.18%	2.72%	2.94%
Mensagens perdidas	1.87%	2.95%	3.97%

Tabela 3.3: Eficiência das rodadas de comunicação com carga teoricamente aplicada à rede de cerca de 400 Mb/s, rodadas de $\Delta = 1000\mu s$ e com $n = 3, 5$ e 7 processos.

O efeito mais notável, porém, do aumento do número de processos no sistema é o aumento da taxa de mensagens perdidas, que ocorre a despeito do tamanho das mensagens ser menor com o aumento de n e da carga aplicada ao sistema ter sido a mesma em todos os experimentos comparados na Tabela 3.3. Para reforçar este argumento, apresenta-se os

resultados de um outro experimento, com $n = 7$ processos, $\Delta = 1400\mu s$ e, para se obter mesma carga de 400 Mb/s aplicada à rede, $S = 10$ KB. A eficiência obtida foi de 94.36%, os TTRs limitados por Δ foram 94.33% e as mensagens atrasadas foram 2.24%. Porém, a taxa de mensagens perdidas foi de 3.40%, isto é, foi ainda superior à taxa de 2.95% de perdas observada para $n = 5$, $\Delta = 1000\mu s$ e $S = 10$ KB (conforme a Tabela 3.3).

3.5 Discussão dos Resultados

A série de experimentos reportados neste capítulo confirma a hipótese inicial de que os aglomerados de alto desempenho, dado que seja possível controlar a carga aplicada ao sistema, apresentam um comportamento predominantemente síncrono. O controle de carga implementado pelo mecanismo de sincronização mostrou-se eficiente, e foi suficiente para permitir que a execução do sistema fosse organizada em rodadas, com uma semântica síncrona. Em adição, com a escolha apropriada de suas durações de referência, em função do número de processos e do tamanho das mensagens difundidas, as rodadas mostram-se eficientes como rodadas de comunicação. Por outro lado, a verificação de comportamentos temporais discrepantes em todos os experimentos reafirma a assincronia do sistema e confirma que ele foi corretamente modelado como assíncrono temporizado.

Capítulo 4

Protocolo de Difusão Síncrona Totalmente Ordenada

Neste capítulo apresenta-se um novo protocolo para o problema de Difusão Totalmente Ordenada, chamado de Difusão Síncrona Totalmente Ordenada (DSTO). O protocolo é destinado a sistemas em que não ocorrem falhas de processos ou aos períodos da execução de um sistema em que ainda não ocorreram falhas de processos. O protocolo também foi projetado tendo em vista sistemas que tenham um comportamento predominantemente síncrono, de modo que seja possível organizar as computações em rodadas.

O protocolo de DSTO é descrito por partes, distribuídas em três seções deste capítulo. Na Seção 4.2 apresenta-se a intuição e uma visão geral do funcionamento do protocolo. Na Seção 4.3 descreve-se uma primeira versão do protocolo, destinada a um modelo de computação restritivo, que na prática só pode ser implementado em sistemas síncronos. Na Seção 4.4 as restrições de sincronia são relaxadas, com a especificação de uma versão do protocolo que tolera falhas de desempenho. Como resultado, obtém-se um protocolo com semântica síncrona, pois ele só obtém progresso quando o sistema se comporta de forma síncrona, mas cuja corretude não está associada à sincronia efetivamente apresentada pelo sistema, pois sua segurança é construída a partir de predicados assíncronos.

Antes da descrição do protocolo de DSTO, a Seção 4.1 apresenta uma visão geral do comportamento das soluções de DTO existentes, com destaque aos mecanismos que elas empregam para obter a ordenação total das mensagens. E, finalmente, após a descrição do protocolo, a Seção 4.5 apresenta os principais trabalhos a ele relacionados.

4.1 Resolução de Difusão Totalmente Ordenada

São três as funcionalidades que, de modo geral, todo protocolo de Difusão Totalmente Ordenada deve implementar. A primeira funcionalidade é fazer com que as mensagens

difundidas sejam recebidas por todos os processos do sistema, de modo a permitir que a todos processos corretos seja finalmente entregue um mesmo conjunto de mensagens — como exige a propriedade de Acordo Uniforme. A segunda funcionalidade é a construção da sequência de mensagens a ser entregue por cada instância de DTO, de modo a se garantir que a entrega das mensagens se dê em todos os processos segundo uma mesma ordem total — o que é a premissa fundamental de DTO, definida pela propriedade de Ordem Total Uniforme. A terceira funcionalidade, que impacta diretamente na implementação das duas anteriores, é a tolerância a falhas de processos ou a processos irresponsivos.

A implementação da primeira funcionalidade citada equivale, por si só, à resolução de um outro problema de comunicação em grupo, conhecido como Difusão Confiável Uniforme [36]. A especificação deste problema é dada exatamente pelas três primeiras propriedades de DTO: Validade, Acordo Uniforme e Integridade Uniforme, descritas na Seção 2.4. Como a sua especificação não determina em que ordem as mensagens devem ser entregues pelos processos, a Difusão Confiável Uniforme é um problema mais simples que DTO. A sua solução consiste, essencialmente, em adiar a entrega de uma mensagem recebida até que se tenham garantias de que ao menos um processo correto — ou uma maioria — também a tenha recebido. Com esta estratégia, aliada ao uso de mecanismos de confirmação e retransmissão de mensagens, faz-se com que todos os processos corretos finalmente entreguem o mesmo conjunto de mensagens por eles recebidas. Em adição, faz-se com que processos falhos entreguem uma mensagem se, e somente se, algum processo correto também a recebeu — o que, portanto, garante que o Acordo seja Uniforme. Assim, por conta da sua semântica e de suas múltiplas implementações, várias das soluções de DTO descritas empregam soluções de Difusão Confiável como caixas pretas — ou seja, blocos constituintes, implementados à parte — para difundir suas mensagens [27].

A segunda funcionalidade, essencial para uma solução de DTO, é um algoritmo ou uma estratégia que determine a ordem com que as mensagens recebidas devem ser entregues pelos processos. Em levantamento realizado por Defago et al [27] com dezenas de protocolos de DTO desenvolvidos entre a década de 1970 e 2004, foram identificadas três classes de soluções para DTO, que se diferenciam pelo mecanismo básico de ordenação por elas empregado na ausência de falhas. Dentro destas três classes, foram caracterizadas cinco subclasses que condensam as estratégias que podem ser empregadas, individualmente ou em conjunto, para gerar a ordenação de mensagens [27]:

- Sequenciador fixo
- Sequenciador rotativo
- Privilégio no envio
- Histórico de comunicação

- Acordo na recepção

A seguir apresentam-se, de forma resumida, os pontos principais que caracterizam as três grandes classes, isto é, os três mecanismos básicos para a ordenação das mensagens.

Sequenciadores Uma das estratégias empregadas para se resolver DTO consiste em delegar a um processo distinto a tarefa de determinar, de forma centralizada, a ordem na qual as mensagens devem ser entregues por todos os processos. Ao processo que assume esta função dá-se o nome de *sequenciador* e o procedimento para a difusão de mensagens, tendo o sequenciador como intermediário, é o seguinte. Para difundir suas mensagens, os processos emissores as enviam—diretamente, através de canais ponto-a-ponto, ou a todos os processos, via difusão — para o sequenciador. O papel do sequenciador é associar a cada mensagem recebida um número de sequência e difundi-las — mensagens e números de sequência — aos demais processos. Os destinatários das mensagens, então, entregam as mensagens recebidas segundo a ordem estabelecida pelos seus números de sequência.

A classe de soluções baseadas em sequenciadores pode ainda ser dividida em duas subclasses [27]. Na subclasse *sequenciador fixo* um processo é eleito no início de cada execução para atuar como sequenciador e este processo assim permanece durante toda a execução ou até que ele falhe. Exemplos de protocolos desta subclasse são o Isis [13] e o Amoeba [37], que fazem parte de soluções mais complexas de comunicação em grupo. Em contrapartida, na subclasse *sequenciador rotativo* a função de associar números de sequência às mensagens é assumida por vários processos, mesmo que nenhum deles falhe durante a execução. As soluções desta subclasse — que são, de modo geral, otimizações da solução proposta por Chang e Maxemchuck [18] — organizam os processos em um anel lógico, pelo qual circula a função de sequenciador, sintetizada pelo próximo número de sequência que deve ser empregado. A transferência da função de sequenciador, nestas soluções, também constitui uma confirmação do recebimento das mensagens previamente difundidas, que são entregues somente quando são confirmadas por todos os sequenciadores. Com isto, a custo de uma maior complexidade e maiores latências para a entrega de mensagens, consegue-se que a tarefa — e, portanto, a carga — de ordenar, (re)difundir e confirmar o recebimento das mensagens seja distribuída por vários processos [27].

É característico das soluções desta classe o esforço de se resolver, de forma eficiente, o problema de se difundir as mensagens de forma confiável aos destinatários, com o uso de mecanismos de confirmação e topologias especiais para a propagação das mensagens. Por outro lado, as soluções não suportam nativamente a ocorrência de falhas de processos, cujo tratamento é delegado a soluções de gestão de grupos [12], responsáveis pela detecção de falhas e a remoção — muitas vezes forçada — de processos suspeitos. Em adição, para suportar a falha do processo sequenciador e garantir uma transição segura para uma nova configuração de execução, as soluções empregam ou implementam complexos

procedimentos de recuperação, a fim de definir o conjunto de mensagens já recebidas e entregues pelos processos e o próximo número de sequência a ser empregado. Como resultado, as soluções desta classe — em particular as da subclasse sequenciador fixo — normalmente não garantem a versão uniforme das propriedades DTO, uma vez que o comportamento de processos falhos não é controlado pelo protocolo [27, 35]. Além disto, apesar destas soluções serem assíncronas [27], os mecanismos de gerência de grupos que elas empregam para tolerar falhas não podem ser implementados neste modelo [16].

Ordenação no Envio Em contraste com as soluções que empregam sequenciadores, nas soluções pertencentes a esta classe, a ordenação das mensagens é realizada de forma totalmente distribuída, sem intermediários entre os emissores e os destinatários. A ordem com que as mensagens devem ser entregues é estabelecida pelos emissores, que, ao difundir cada mensagem, determinam quais outras mensagens devem precedê-la na ordenação total. Cabe aos destinatários, por sua vez, decidir se uma mensagem recebida pode ser entregue de forma segura ou se sua entrega deve ser adiada, por depender do recebimento de outras mensagens. A forma como as informações de precedência são sintetizadas e a estratégia empregada para transformar estas relações de ordem parcial em uma ordem total definem duas subclasses para as soluções que realizam a ordenação no envio.

O princípio da primeira subclasse, a de *privilégio no envio*, é fazer com que apenas um processo difunda suas mensagens a cada instante da execução e que este privilégio de difundir mensagens circule dentre os emissores. Ao receber este privilégio, cada emissor difunde um conjunto de mensagens suas — ordenadas entre si, por exemplo, por números de sequência — e então sinaliza que este privilégio foi passado para outro emissor. Todos os processos sabem, então, que o conjunto de mensagens por ele difundidas — que pode, inclusive, ser vazio — deve ser entregue após as mensagens difundidas pelo emissor que o precede. Soluções desta subclasse normalmente organizam os processos em um anel lógico, por onde as mensagens difundidas e o privilégio de envio circulam. Como exemplo tem-se o Train [22], em que os processos recebem as mensagens previamente difundidas, as confirmam e adicionam suas próprias mensagens, sempre que o "trem passa" por eles. Outro exemplo importante é o Totem [6], que dá suporte a uma solução mais complexa de comunicação em grupo e emprega a topologia de anel para o envio e a difusão (com diversas variações nas propriedades que são atendidas) de mensagens.

A segunda subclasse de soluções em que a ordenação das mensagens se dá no envio é a de *histórico de comunicação*. Ao contrário da primeira subclasse, nesta estratégia não se restringe quando os processos podem difundir suas mensagens — o que constitui a principal limitação das soluções de privilégio de envio. Por outro lado, esta solução exige que os destinatários mantenham informação global do sistema, para que eles possam decidir quando uma mensagem pode ser entregue de forma segura. Esta informação, em

soluções assíncronas, é condensada pelos chamados relógios lógicos [41], que sintetizam dados de precedência entre mensagens enviadas em um sistema e permitem a construção —com a adição de critérios para "desempate" de mensagens sem precedências entre si— de uma ordem total. Já em soluções síncronas, em que se considera a existência de relógios sincronizados, esta informação é sintetizada pelo registro do instante do tempo "físico" em que ocorreu a difusão de cada mensagem. Assim, são estes "carimbos de tempo" (físicos ou lógicos) que fornecem a informação necessária para que os processos possam retardar a entrega de uma mensagem até que ela se torne estável [27] — isto é, até que se tenham garantias que nenhuma outra mensagem a pode preceder na ordenação total.

Finalmente, nota-se que em todas as soluções desta classe, cada processo precisa conhecer e se comunicar, direta ou indiretamente, com todos os processos. Portanto, na ocorrência de falhas de processos estas soluções deixam de progredir, visto que os processos se tornam incapazes de construir e manter visões globais do sistema. Assim, para voltarem a ter progresso, as soluções precisam reformar o grupo de processos participantes, removendo aqueles que estejam inativos, e decidir quais das mensagens difundidas pelos processos removidos ainda devem ser entregues e quais devem ser descartadas. Estas duas tarefas de recompor o grupo de processos [12] e de construir uma nova visão global consistente [11] são complexas [16] e, por conta disto, normalmente implementadas separadamente. Por outro lado, uma vantagem das soluções desta classe — por exemplo, em relação às que empregam sequenciadores — é que elas, por serem simétricas e trabalharem com o conceito de estabilidade de mensagens, são, em sua maior parte, uniformes [7, 27].

Ordenação por Acordo Nesta última classe de soluções para DTO, a ordem na qual as mensagens difundidas são entregues é definida por comum acordo entre os processos destinatários. Assim, os emissores difundem suas mensagens, que são recebidas pelos processos, normalmente em diferentes ordens, e cuja ordem de entrega é definida por um procedimento auxiliar. Neste procedimento os vários processos entram em acordo quanto a um ou alguns dos seguintes pontos: os números de sequência a serem associados a certas mensagens, o próximo conjunto de mensagens que deve ser entregue ou a aprovação ou não de uma pré-ordenação que foi proposta por algum dos processos.

De modo geral, para obter um acordo em relação a qualquer um destes pontos, os processos executam uma instância de *consenso*. O Consenso é um problema fundamental de computação distribuída, que é definido em termos de valores propostos pelos processos e valores decididos (ou escolhidos) pelos processos. Uma solução de consenso deve cumprir as seguintes três propriedades, apresentadas em suas versões uniformes [17, 32]:

i. Se um processo decide um valor, então este valor foi proposto por algum processo;

ii. Dois processos não decidem valores diferentes;

iii. Se (ao menos) um processo correto propõe um valor, então finalmente todos os processos corretos decidem um valor.

Nota-se que estas propriedades tornam-se análogas às propriedades de DTO quando substitui-se proposição e decisão de valores pela difusão e entrega de mensagens. De fato, os dois são problemas relacionados e a resolução de DTO pode ser reduzida à execução de instâncias de consenso, com conjuntos de mensagens como valores de entrada [17].

A principal razão para que a redução de DTO para consenso não ser empregada em grande parte das soluções desenvolvidas é o alto custo para a resolução do consenso [27]. Por outro lado, as soluções desta classe, como Paxos [43], Fast Paxos [44] e Chandra e Toueg [17], herdam todas as propriedades dos algoritmos de consenso uniforme por elas empregados. Como consequência, o progresso destas soluções é garantido mesmo em condições minimais de sincronia [15] e elas são ótimas em termos de número de falhas toleradas [39, 45]. Além disto, estas soluções de consenso não têm sua segurança violada mesmo em situações de total assincronia, o que faz com que delas sejam obtidas as soluções mais robustas para DTO [53]. Finalmente, é importante notar que os procedimentos para a tolerância a falhas, baseados em detecção de falhas e/ou em gerência de grupos, das classes apresentadas anteriormente também dependem da resolução de instâncias de consenso. Assim, enquanto na classe de ordenação por acordo emprega-se consenso durante toda a execução, nas demais soluções o consenso é empregado apenas em situações (espera-se que) excepcionais, em que há falhas e/ou instabilidade do sistema.

Apresentados os principais aspectos na resolução do problema de Difusão Totalmente Ordenada, com destaque para os principais mecanismos empregados para a ordenação das mensagens, a partir da próxima seção descreve-se o protocolo de DTO proposto neste trabalho, denominado protocolo de Difusão Síncrona Totalmente Ordenada.

4.2 Visão Geral do Protocolo

Os princípios gerais e a intuição que guiaram o desenvolvimento do protocolo proposto no decorrer deste capítulo são descritos nesta seção. O protocolo de DSTO considera que o sistema é composto por um conjunto fixo de processos, com cardinalidade conhecida, que estão completamente conectados entre si por canais de comunicação. Apesar de não ser um requisito do protocolo, ele foi desenvolvido para ambientes de rede com suporte nativo à difusão de mensagens. Para garantir a segurança do protocolo é necessário que uma maioria dos processos seja correta, mas para obter progresso — isto é, para que se difunda e entregue novas mensagens — o protocolo requer que todos os processos estejam

ativos e participem da execução, difundindo novas mensagens no início de cada rodada. Em adição, considera-se que, caso os processos falhem, eles falham por parada.

O protocolo progride em rodadas e o seu funcionamento é o seguinte. No início de cada rodada, cada processo envia para todos os processos do sistema uma nova mensagem para ser totalmente ordenada. Mesmo se o processo não tiver mensagens para ordenar, ele também participa da rodada, ao enviar uma mensagem nula — que é uma mensagem especial, sem conteúdo da aplicação. Supondo que na rodada não ocorram falhas, todos os processos, ao final da rodada, recebem todas as mensagens nela difundidas por cada um dos processos, inclusive a própria mensagem. Na fase de processamento desta mesma rodada, após verificar que todas as mensagens esperadas foram recebidas, cada processo aplica às mensagens não nulas recebidas uma função determinista de ordenação — por exemplo, que as ordene pelo identificador das mensagens ou de seus emissores. Como resultado, ao final da rodada cada processo obtém uma *mesma* subsequência de mensagens difundidas: a permutação ordenada de todas as mensagens que poderiam ser difundidas naquela rodada — visto que cada processo difunde uma mensagem por rodada e que toda mensagem recebida por um processo em uma rodada foi difundida naquela mesma rodada.

Em uma descrição mais detalhada, seja r a rodada acima descrita, na qual o processo recebeu todas as mensagens esperadas e, a partir da ordenação delas, construiu a r-ésima subsequência de mensagens. Diz-se também, neste caso, que o processo teve *sucesso* na rodada r, pois ele recebeu todas as mensagens esperadas e não detectou a ocorrência de falhas. Dando prosseguimento ao protocolo, no início da rodada seguinte, a rodada $r + 1$, o processo irá enviar uma nova mensagem (com conteúdo da aplicação ou nula) para ser totalmente ordenada. Com o envio desta mensagem, o processo, em particular, também confirma para os demais processos que ele obteve sucesso na rodada r. Assim, se ao final da rodada $r + 1$ o processo também receber todas as mensagens esperadas, ele tem a confirmação de que todos os processos tiveram sucesso na rodada r e, portanto, que eles já construíram a (mesma) r-ésima subsequência de mensagens. De forma que, ao obter sucesso na rodada $r + 1$, o processo detecta que a r-ésima subsequência de mensagens se tornou estável e que ela pode, portanto, ser entregue à aplicação de forma segura.

Como todos os processos executam a mesma sequência de rodadas, a concatenação das subsequências de mensagens construídas ao final de cada rodada por cada processo gera uma sequência totalmente ordenada de mensagens. Em adição, com o adiamento da entrega das subsequências construídas a cada rodada até que se tenham garantias de sua estabilidade, garante-se a uniformidade da solução — visto que qualquer processo entrega uma subsequência de mensagens se, e somente se, todos os processos também já a tiverem construído. De modo que, enquanto não houver falhas de processos ou de desempenho, garante-se o progresso da solução de DTO, com a entrega de uma mensagem difundida por cada processo a cada rodada. E as mensagens difundidas por um processo são entregues

por todos os processos dentro de duas rodadas, o que representa o custo mínimo, em termos de comunicação para a resolução de DTO na ausência de falhas — por redução à resolução do Consenso Uniforme, que requer ao menos dois passos de comunicação [39].

Assim, apenas com a organização da computação em rodadas, enquanto não houver falhas de processos ou de desempenho, é possível resolver DTO através deste protocolo trivial. O que se mostra na próxima seção é que com a ocorrência de falhas o protocolo — que é destinado para execuções livres de falhas — não violará as propriedades de segurança de DTO. Mostra-se também como mensagens pendentes, que poderiam já ter sido entregues por algum processo antes da ocorrência de falhas, serão finalmente entregues a todos os processos corretos — com o que se garante o Acordo Uniforme de DTO. Nota-se, porém, que em caso de falhas de processos, para que a solução de DTO volte a progredir, faz-se necessário *reconfigurar* o sistema, a fim de remover os processos falhos ou irresponsivos. O problema de reconfiguração é complexo, principalmente na presença de falhas e de assincronia, e a sua resolução é obtida basicamente a partir de duas abordagens. A primeira abordagem [11, 12], que é uma parte constituinte dos serviços de gerência de grupos (brevemente discutidos na seção anterior), consiste na transição entre *visões* do conjunto de processos participantes (e corretos). A segunda abordagem [47] consiste na definição, via consenso, de uma nova configuração e na construção de uma barreira que, quando alcançada pelos processos, faz com que eles abandonem a configuração vigente e ingressem na nova configuração definida (caso eles façam parte dela). A descrição detalhada deste procedimento de recuperação, que permitiria ao protocolo proposto voltar a operar da forma normal está fora do contexto deste trabalho.

Em seguida, a Seção 4.4 mostra como este protocolo pode ser adaptado para tolerar a ocorrência de falhas de desempenho. O que significa que o protocolo passa a tolerar a perda de mensagens pelos canais de comunicação, atrasos discrepantes na realização de etapas de processamento e na entrega de mensagens aos seus destinatários, além de falhas no mecanismo de sincronização que é responsável por gerar as rodadas.

4.3 Protocolo para o Modelo de Rodadas Síncronas

Nesta seção apresenta-se uma primeira versão do protocolo de DSTO, que é destinada ao modelo de rodadas síncronas, descrito na Seção 2.3. Computações neste modelo são organizadas em rodadas e apresentam a propriedade de que todas as mensagens enviadas por um processo correto no início de uma rodada são recebidas por todos os processos corretos no decorrer da mesma rodada. O que, em particular, faz com que a detecção de falhas neste modelo seja trivial, uma vez que a ocorrência de falhas leva ao não recebimento de alguma das mensagens esperadas em uma rodada. O que se descreve nesta seção é como o protocolo de DSTO — cujo funcionamento básico, na ausência de falhas, foi descrito

na seção anterior — procede quando uma falha é detectada por um processo ou quando, pela assincronia que passe a ser apresentada pelo sistema, a propriedade fundamental das rodadas síncronas deixe de ser válida. Em ambos os casos, o protocolo deixará de progredir, mas irá assegurar o cumprimento das propriedades de segurança de DTO.

Dada a propriedade fundamental das rodadas síncronas, o resultado imediato da ocorrência de falhas em uma rodada r, é que um processo correto p não irá receber parte das mensagens referentes a esta rodada por ele esperadas. Assim, como p não obtém sucesso na rodada r, ele não irá entregar à aplicação a subsequência $r-1$, construída na rodada precedente, e não irá difundir uma nova mensagem na rodada $r+1$ seguinte. Como consequência, mesmo que um processo q tenha obtido sucesso na rodada r[1], ele necessariamente não terá sucesso na rodada $r+1$ e também irá deixar de difundir e de entregar novas mensagens. Em outras palavras, o que esta primeira versão do protocolo de DSTO determina a um processo que detecte a ocorrência de uma falha, é que ele se retire da execução e não envie, nem entregue mais mensagens. Com isto, o processo que detectou a falha impede que os demais processos obtenham sucesso nas rodadas subsequentes, e acaba por forçá-los a finalmente também abandonar a execução do protocolo.

Já a decisão de que mensagens serão entregues nas rodadas que ficaram pendentes — no exemplo acima empregado, das rodadas r e $r-1$, no caso do processo p, e das rodadas r e $r+1$, no caso do processo q — é obtida através da execução de instâncias de consenso. Nota-se que os processos podem abandonar a execução do protocolo de DSTO em rodadas distintas e que alguns deles podem ter entregado subsequências de mensagens que não foram entregues pelos demais. Assim, nas instâncias de consenso que seguem a detecção de uma falha, deve-se garantir que as mensagens já entregues por alguns processos sejam necessariamente entregues aos demais. Portanto, no exemplo empregado, como o processo q já tinha entregado a subsequência referente à rodada $r-1$, é necessário forçar que o valor decidido nesta instância de consenso — que será o conjunto de mensagens entregue pelo processo p e pelos demais processos corretos — seja esta mesma subsequência.

Para descrever como isto pode ser garantido é necessário detalhar o funcionamento da maioria das soluções de Consenso Uniforme. Os protocolos de consenso — o problema foi formalmente definido no final da Seção 4.1 — normalmente executam em rodadas. Devido à assincronia do sistema ou à ocorrência de falhas, diferentes processos podem participar de diferentes rodadas simultaneamente. Em cada uma das rodadas, os processos buscam um acordo em qual valor será decidido naquela rodada e, caso eles não consigam chegar a um acordo, uma nova rodada é iniciada. A questão, porém, é que é possível que em uma mesma rodada alguns processos tenham informação suficiente para obter o valor consensual, enquanto outros processos declarem a rodada falha e iniciem uma nova. O que

[1]É possível que um processo falhe durante a difusão de sua mensagem de certa rodada, de forma que uma parte dos processos irá recebê-la até o final da rodada, enquando os demais não a receberão.

os primeiros — que decidiram um valor — devem garantir, é que os demais, nas próximas rodadas, não possam decidir um valor diferente daquele por eles decidido. Da mesma forma, ao iniciar uma nova rodada e buscar consenso em um novo valor, os processos devem se assegurar que nas rodadas anteriores nenhum valor poderia ter sido decidido; e, se algum valor puder ter sido decidido, é neste valor que se deve buscar consenso. De forma que as soluções de consenso que operam em rodadas — que têm Paxos [43] e Chandra e Toueg [17] como as mais conhecidas — têm um funcionamento que, de modo geral, pode ser resumido da seguinte forma. Um valor só pode ser decidido por qualquer processo se ao menos um dos processos corretos — isto é, ao menos $f+1$ processos, onde f o número máximo de falhas — tiver *fixado* este mesmo valor. Assim, em cada rodada que tem início, averígua-se se algum valor foi fixado por algum dos processos que dela participam. Em caso afirmativo, como este valor pode já ter sido decidido, só ele pode ser proposto para decisão nesta rodada; caso contrário, qualquer valor proposto por um processo pode ser escolhido. Esta averiguação, para ser correta, deve levar em consideração ao menos um processo correto. Portanto, somente em rodadas das quais participam uma maioria dos processos — a rigor, de que participam $f+1$ processos — pode-se decidir valores.

Retornando ao exemplo acima empregado, relativo ao protocolo de DSTO, tem-se o processo p que não obteve sucesso na rodada r. O processo p, por construção do algoritmo, entregou a subsequência $r-2$ na rodada precedente à detecção da falha, na qual ele também construiu a subsequência $r-1$, que ainda não foi por ele entregue. Assim, ao abortar a execução do protocolo de DSTO, o processo p marcará a instância $r-2$ de consenso como decidida, tendo como valor de decisão a última subsequência que ele entregou à aplicação; e também fixará a subsequência $r-1$ (a última por ele construída) como valor para tal instância. Por sua vez, o processo q, que abortou a execução do protocolo de DSTO na rodada seguinte — a rodada $r+1$ — marcará como decidida a instância $r-1$, tendo como valor a última subsequência por ele entregue, e fixará a última subsequência construída como valor para a instância r. De modo que, apesar do processo p não ter entregado a subsequência $r-1$, o valor fixado por ele à instância de consenso que decidirá o conjunto de mensagens a ser entregue como esta subsequência é a mesma subsequência já entregue pelo processo q. Portanto, mesmo que o processo q abandone a execução logo após entregar a subsequência $r-1$, o processo p e os demais processos corretos irão finalmente entregar esta mesma subsequência entregue por q.

Em resumo, dado que o protocolo proposto adia a entrega de subsequências até o momento em que elas se tornam estáveis, tem-se que um processo qualquer entrega uma subsequência t se, e somente se, todos os processos já tiverem construído a mesma subsequência t. Na ocorrência de falhas, então, os processos abandonam o protocolo e, com isto, forçam os demais processos a fazerem o mesmo e a não entregar, nem difundir novas mensagens. Em seguida, os processos iniciam instâncias de consenso relativas às roda-

das pendentes, a fim de decidir quais subsequências de mensagens devem ser entregues. Nestas instâncias, os processos fixam, como únicos valores que podem ser decididos, as subsequências de mensagens por eles já construídas. De modo que se garante que, se uma subsequência foi entregue por algum processo durante a execução normal do protocolo, todos os processos corretos irão finalmente entregá-la, respeitando sua ordenação original, pois somente este valor poderá ser decidido nas instâncias de consenso iniciadas.

```
1  Variaveis:
2      sequencia[][]          /* Sequencia de mensagens a serem entregues */
3
4  Inicio da rodada r=1:
5      sequencia[0] := ∅
6      difunde(1, carregaMensagem())
7
8  Fim de rodada r com mensagens M_r:
9      if (|M_r| = n) then    /* Rodada com sucesso */
10         sequencia[r] = ordena(M_r)
11         if (r > 1) entrega(sequencia[r-1])
12         difunde(r+1, carregaMensagem())
13     else                   /* Falha ou assincronia detectada */
14         if (r > 1) decide(r-2, sequencia[r-2])
15         fixa(r-1, sequencia[r-1])
16         consenso(r-1)
17         consenso(r)
18     endif
```

Algoritmo 4.1: Primeira versão do protocolo de Difusão Síncrona Totalmente Ordenada.

Com a descrição do procedimento de tolerância a falhas de processos, a primeira versão do protocolo de DSTO, destinada ao modelo síncrono de computação, tem o seu pseudocódigo apresentado no Algoritmo 4.1. Ele emprega algumas funções genéricas, destacadas em negrito, que realizam as operações já descritas no decorrer das últimas seções. A função `carregaMensagem()` obtém uma nova mensagem da aplicação para ser difundida pelo protocolo e, caso a aplicação não tenha mensagens a difundir, gera uma mensagem nula. Esta função corresponde à entrada do algoritmo de DTO. A função `entrega()`, por sua vez, é a saída do protocolo e oferece à aplicação uma nova subsequência de mensagens, cuja ordenação foi requisitada e que já foram ordenadas. Já as funções `decide(instância, valor)` e `fixa(instância, valor)` são relativas à solução auxiliar de consenso que é empregada pelo protocolo e tiveram sua semântica já descrita. Finalmente, as duas chamadas de `consenso(instância)` referem-se ao início de duas instâncias de consenso a fim de se decidir, com a detecção de uma falha, as subsequências a serem entregues relativas às duas rodadas pendentes para o processo.

Nota-se que as instâncias de consenso são executadas à parte, em paralelo à execução do protocolo de DSTO, e que todos os processos delas participam, reportando os valores que nela já foram decididos ou participando na escolha de um valor de consenso.

4.4 Protocolo Tolerante a Falhas de Desempenho

Nas seções anteriores definiu-se um novo protocolo para DTO, destinado a execuções sem falhas de processos, chamado de Difusão Síncrona Totalmente Ordenada. O protocolo progride em rodadas e a sua corretude, na ausência de falhas de processos, se embasa em duas hipóteses. A primeira hipótese é que todos os processos executam uma mesma sequência de rodadas e que cada processo difunde, a cada rodada, uma nova mensagem para ser totalmente ordenada. A segunda hipótese é que, ao final de cada rodada, todos os processos recebem o mesmo conjunto de mensagens, que contém todas as mensagens que poderiam ter sido difundidas naquela rodada, uma de cada processo. Para que estas duas hipóteses —em particular, a segunda— sejam válidas, é necessário que o sistema apresente, durante toda execução, um comportamento estritamente síncrono. O que significa que, tanto os processos e os canais de comunicação, como o mecanismo que gera as rodadas não podem cometer falhas de desempenho, sob pena de que a execução normal do protocolo seja abortada, por suspeita de que possam ter ocorrido falhas de processos.

Este não é o comportamento desejado para o protocolo de DSTO, visto que da forma como ele foi apresentado no início deste capítulo, ele foi projetado tendo em vista sistemas com comportamento predominantemente síncrono, e não estritamente síncrono. Nesta seção, então, desenvolve-se uma nova versão do protocolo de DSTO, que agrega à primeira a tolerância a falhas de desempenho. Da mesma forma que a versão original, a nova versão do protocolo também considera que a execução é organizada em rodadas e o progresso da solução também está associado a rodadas em que todos os processos tem sucesso em receber as mensagens nelas difundidas. Porém, ao contrário da primeira solução, as rodadas não precisam ter a semântica de rodadas síncronas — descritas na Seção 2.3 — e podem ser implementadas em sistemas como menos restrições temporais —, como é o caso das rodadas obtidas pelo mecanismo de sincronização, que foi descrito no Capítulo 3. Para construir a nova versão para o protocolo de DSTO, então, serão relaxadas as duas hipóteses acima citadas, que fundamentam a construção da versão original do protocolo.

A primeira hipótese — de que todos os processos enviam mensagens em todas as rodadas — não pode ser cumprida quando um processo comete uma falha de desempenho ou quando o mecanismo de sincronização falha em fazê-lo iniciar uma nova rodada. Assim, nesta nova versão do protocolo, a ordem com que as mensagens difundidas são entregues não será mais definida pelas rodadas nas quais elas foram enviadas, mas por números de sequência a elas associados pelos seus emissores. De modo que, ao enviar uma nova men-

sagem (com conteúdo da aplicação ou nula) para ser totalmente ordenada, um processo anexa a ela o próximo número de sequência disponível. Esta associação é definitiva: sempre que esta mensagem for enviada, a ela estará anexado o mesmo número de sequência a ela assinalado por seu emissor em seu primeiro envio. Nota-se que esta substituição não alteraria o funcionamento do protocolo original: dado que se defina um número de sequência inicial i_0 comum, na rodada r todos os processos enviariam mensagens com números de sequência $r + i_0$ e se construiria a mesma subsequência de mensagens.

Com esta primeira alteração no protocolo de DSTO, na fase de processamento de cada rodada os processos empregam a seguinte regra de transição. Se o processo obteve sucesso na rodada e recebeu mensagens com números de sequência i, ele constroi a i-ésima subsequência de mensagens. Da mesma forma que no protocolo original, a construção da subsequência i confirma a estabilidade da subsequência anterior e, portanto, o processo entrega a subsequência $i - 1$ à aplicação. Na rodada seguinte, o processo gera uma nova mensagem, associa a ela o número de sequência $i + 1$, e a envia para todos os processos. Porém, se o processo não obteve sucesso na rodada, ele não pode gerar a subsequência i esperada, ele não entrega nenhuma mensagem e, na rodada seguinte, ele *reenvia* a sua i-ésima mensagem — isto é, a mensagem a qual ele já associou o número de sequência i. Com isto, o protocolo tolera a situação — possível com a remoção da primeira hipótese de sincronia citada — de um ou mais processos não participarem de uma rodada: ao invés de se abortar a execução, os processos reenviam a última mensagem por eles difundida, o que possibilita aos processos que perderam a rodada finalmente recebê-las.

A segunda hipótese de sincronia do protocolo original de DSTO, já com o relaxamento da primeira hipótese, pode ser reescrita da seguinte forma: ao final de cada rodada, todo processo que dela participou recebe um mesmo conjunto de mensagens, enviadas naquela mesma rodada por todos os processos que dela participaram. Com esta hipótese ainda válida, o progresso da solução de DTO só ocorre nas rodadas em que todos os processos participam e (re)enviam suas i-ésimas mensagens, a partir das quais se constroi a i-ésima subsequência de mensagens. Enquanto isto não ocorre, os processos que participam das rodadas apenas (re)enviam suas i-ésimas mensagens, mas não obtém progresso.

Quando esta segunda hipótese de sincronia é removida, tem-se uma nova situação que pode ocorrer durante a execução do protocolo. Torna-se possível que em uma rodada, da qual todos os processos participam, uma parte dos processos tenha sucesso, enquanto os demais não o tenham, por não terem recebido todas as mensagens para ela esperadas. Como consequência, uma parte dos processos reenviará na rodada seguinte suas i-ésimas mensagens, enquanto outros processos irão gerar e enviar suas mensagens $i + 1$ para que elas sejam ordenadas. Mas como esta situação levaria o protocolo a estados inconsistentes, faz-se necessário redefinir o conceito de rodadas com sucesso. Assim, nesta nova versão do protocolo, um processo tem sucesso em uma rodada se, e somente se, ele recebe uma

mensagem proveniente de cada processo participante e se todas as mensagens recebidas na rodada possuírem o mesmo número de sequência. Seja i este número de sequência, comum a todas as mensagens recebidas pelo processo durante a rodada, diz-se que o processo teve sucesso em receber as mensagens i naquela rodada. De modo que a condição de progresso do protocolo passa a ser a seguinte: um processo envia sua mensagem $i+1$ se, e somente se, ele teve sucesso em receber as mensagens i em alguma rodada precedente.

Esta modificação na regra que define quando uma rodada tem sucesso, porém, não resolve por inteiro a situação acima descrita. Além de evitar que os processos misturem mensagens com diferentes números de sequência, é necessário fazer com que os processos que não tiveram sucesso em rodadas que os demais tiveram sucesso, consigam finalmente construir as subsequências de mensagens que para eles ainda estão pendentes. A forma como se recupera estes processos que ficaram "para trás" é fazer com que os demais processos *retrocedam* na ordenação das mensagens e reenviem as suas mensagens com o número de sequência anterior ao corrente. Assim, se um processo que envia em uma rodada sua i-ésima mensagem, recebe nesta mesma rodada uma mensagem com número de sequência $i-1$, ele deve retroceder um passo e reenviar sua mensagem $i-1$ na rodada seguinte. Nota-se que este processo já teve sucesso em receber estas mensagens e já construiu a subsequência $i-1$, mas ele deve reenviar esta mensagem por quantas rodadas forem necessárias, até que em alguma rodada ele obtenha sucesso em receber (novamente) as mensagens $i-1$. Quanto isto finalmente ocorrer, o processo não precisa gerar a subsequência $i-1$ e ele não deve entregar mensagens à aplicação: ele apenas volta à situação anterior e reenvia, na rodada seguinte, a sua i-ésima mensagem.

Este procedimento de retrocesso é necessário para ressincronizar os processos que se atrasam — ou, informalmente, que "ficam para trás" — na ordenação das mensagens. Quando ele é finalizado com sucesso, todos os processos voltam a enviar mensagens com um mesmo número de sequência e buscam construir a mesma subsequência de mensagens a ser entregue à aplicação. É importante destacar que os processos que retrocedem uma vez, não retrocedem uma segunda vez antes de ter sucesso no primeiro retrocesso e voltar ao número de sequência original. Esta propriedade advém da construção do protocolo e pode ser informalmente justificada da seguinte forma. Seja um processo p que em uma rodada r teve sucesso em receber as mensagens i e, portanto, progrediu para o número de sequência $i+1$. Como p teve sucesso, todos os processos participaram da rodada r e todos eles tinham i como subsequência para ser aprovada naquela rodada. Assim, em qualquer rodada $r' > r$, um processo só poderia enviar sua mensagem i, caso ele não tenha tido sucesso na rodada r, ou sua mensagem $i+1$, caso ele tenha tido sucesso na rodada r. Como p alcançou o número de sequência $i+1$, então necessariamente houve uma rodada como r, de modo que nenhum processo poderia enviar, após a rodada r, mensagens com números de sequência menores que i. Portanto, se um processo p alcança o número de

sequência $i + 1$, ele nunca irá retroceder a números de sequência inferiores a i.

4.4.1 Versão Final do Protocolo

Com a descrição de como se dá a recuperação de falhas de desempenho, através do reenvio de mensagens já difundidas e do procedimento de retrocesso, conclui-se a descrição desta nova versão do protocolo de DSTO. O pseudocódigo para esta nova versão — a versão final do protocolo — é apresentado no Algoritmo 4.2 e descreve-se a seguir as modificações realizadas em relação ao protocolo original, apresentado no Algoritmo 4.1.

Inicialmente, os identificadores de rodadas não são mais empregados para referenciar mensagens, que estão associadas a números de sequência. O número de sequência da próxima subsequência de mensagens que o processo deve construir é armazenado pela variável `base`, que é inicializada com i_0 — que é o primeiro número sequência que pode ser empregado. Já a variável `atual`, que também inicializada com i_0, armazena sempre o número de sequência da mensagem que foi enviada na rodada em curso. Enquanto o processo não retrocede na solução a fim de recuperar outros processos, o valor de `atual` será igual ao valor de `base`, pois o processo envia a cada rodada a sua mensagem para a próxima subsequência deve ser construída. Quando o processo retrocede, o que ocorre quando ele recebe uma mensagem com número de sequência inferior ao da mensagem que ele enviou, `atual` passa ser uma unidade menor que `base`. Como consequência, quando o processo retrocede, mesmo que ele tenha sucesso em uma rodada, ele não gera uma nova subsequência de mensagens e nem entrega a última subsequência gerada à aplicação: ele apenas incrementa o valor de `atual` e, portanto, desfaz o retrocesso.

As circunstâncias em que o processo progride na solução de DTO por obter sucesso nela correspondem à condicional (o `if`) principal do Algoritmo 4.2: se `n` mensagens são recebidas na rodada e se todas elas possuem o mesmo número de sequência, que deve ser igual a `atual`. Neste caso, o processo incrementa a variável `atual` e, portanto, progride para o próximo número de sequência. Se, em adição, o processo não estava em procedimento de recuperação (`base` é igual a `atual`), ele: (i) ordena as mensagens recebidas a fim de gerar a próxima subsequência de mensagens a ser entregue à aplicação; (ii) obtém uma nova mensagem da aplicação, que terá como número de sequência o valor incrementado de `atual`; e (iii) entrega à aplicação a última subsequência por ele gerada.

Quando o processo não tem sucesso na rodada e também não é obrigado a retroceder ao número de sequência anterior, o estado de suas variáveis não se altera, de modo que ele reenvia, na rodada seguinte, a mesma mensagem por ele enviada na rodada finalizada. Assim, ao contrário do Algoritmo 4.1, a não obtenção de sucesso por um processo em uma rodada não é considerada como sendo decorrente de falhas de processos, mas sim de falhas (transientes) de desempenho. Considera-se que as falhas de processo sejam detec-

tadas por um mecanismo auxiliar, externo ao protocolo de DSTO, que apenas sinaliza ao protocolo a ocorrência (ou a suspeita) de uma falha. O recebimento deste sinal faz com que o processo abandone a rotina principal e execute a rotina de recuperação do protocolo, apresentada no Algoritmo 4.2. Supõe-se também que um processo que receba mensagens relativas às instâncias de consenso empregadas para recuperar o protocolo abandone a rotina principal e inicie a execução desta rotina de recuperação. Porém, enquanto isto não ocorre, o protocolo de DSTO, dada a ausência dos processos que já abandonaram sua execução, não pode mais obter progresso, de modo que o estado dos processos — em particular, a variável `base` — não é alterado após a detecção da primeira falha.

```
1  Variaveis:
2      sequencia[][]          /* Sequencia de mensagens a serem entregues */
3      mensagens[]            /* Mensagens enviadas pelo processo */
4      base := i_0            /* Subsequencia de mensagens esperada */
5      atual := i_0           /* Subsequencia de mensagens corrente */
6
7  Inicio da primeira rodada r = r_0:
8      mensagens[i_0] := carregaMensagem()
9      difunde(i_0, mensagens[i_0])
10
11 Fim de rodada r com mensagens M_r:
12     seq := min_i{i = seq(m) : m ∈ M_r}
13     max := max_i{i = seq(m) : m ∈ M_r}
14     if (|M_r| = n and atual = max = seq) then /* Rodada com sucesso */
15         if (base = atual) then              /* Nova subsequencia */
16             base := base + 1
17             sequencia[atual] := ordena(M_r)
18             mensagens[atual + 1] := carregaMensagem()
19             if (atual > i_0) entrega(sequencia[atual - 1])
20         endif
21         atual := atual + 1
22     else if (seq < atual) then             /* Retrocesso para base - 1 */
23         atual := seq
24     endif
25     difunde(atual, mensagens[atual])
26
27 Falha foi detectada:
28     if (base > i_0 + 1) decide(base - 2, sequencia[base - 2])
29     if (base > i_0) fixa(base - 1, sequencia[base - 1])
30     consenso(base - 1)
31     consenso(base)
```

Algoritmo 4.2: Protocolo de Difusão Síncrona Totalmente Ordenada.

Nota-se que a rotina para o tratamento de falhas é idêntica a que foi apresentada no Algoritmo 4.1, com a diferença de que se substituiu o identificador da rodada em que a falha foi detectada pela variável `base`. A razão para esta associação é que a variável `base` no Algoritmo 4.2 tem uma semântica idêntica a do identificador das rodadas no Algoritmo 4.1: ela representa a última mensagem que foi difundida pelo processo e a subsequência pendente para o processo, que é a primeira ainda não construída por ele. Em adição, o valor de `base` só é incrementado nas rodadas que o processo obtém progresso, que são rodadas em que o sistema se comporta de forma síncrona e os processos estão sincronizados — que é o que se supõe que ocorra no Algoritmo 4.1 em todas as rodadas antes da primeira falha ser detectada. Finalmente, se na primeira versão do protocolo dois processos detectam a ocorrência de falhas em no máximo duas rodadas consecutivas, nesta nova versão, os valores da variável `base` em dois processos quaisquer, por construção do protocolo, a qualquer instante da execução, diferem no máximo em uma unidade.

4.4.2 Discussão do Protocolo

Segundo a categorização para as soluções de DTO apresentada na Seção 4.1, o protocolo de DSTO se enquadra na classe de ordenação do envio, pois quando se difunde uma mensagem com um número de sequência i determina-se a priori (já no seu envio) que ela pertencerá à i-ésima subsequência de mensagens que será entregue por todos os processos. Das soluções da subclasse de privilégio de envio, o protocolo herda a característica de que os processos só são autorizados a difundir novas mensagens em certas circunstâncias: no início de cada rodada, com a condição de que a última mensagem difundida deve já ter sido aprovada pelo processo. Porém, ao contrário das soluções desta subclasse, todos os processos difundem suas mensagens em conjunto e a ordem com que elas serão entregues é determinada por informações anexadas à mensagem, o que é característico de soluções da subclasse histórico de comunicação. Em adição, como é comum a todas as soluções da classe de ordenação no envio, para que o protocolo possa construir uma ordenação total para as mensagens é necessário que o processo mantenha uma visão global do sistema. De modo que os processos precisam receber periodicamente mensagens dos demais processos, nas quais eles informam seus estados, condensados pelos números de sequência das subsequências que eles buscam construir em cada rodada.

A construção de uma visão global do sistema ao final de uma rodada é um requisito necessário para que o protocolo obtenha progresso naquela rodada. Porém, não é um requisito suficiente, visto que só se tem progresso se todos os processos estiverem com seus estados sincronizados, isto é, se todos os processos (re)enviarem, no início da rodada em questão, mensagens com um mesmo número de sequência. Estes dois requisitos, de se obter uma visão completa e uma visão sincronizada em uma rodada, são outra forma

de definir o conceito, apresentado na Seção 4.2 e estendido na Seção 4.4, de rodada com sucesso para um processo. Nota-se, entretanto, que a obtenção de uma rodada com sucesso ainda não é um requisito suficiente para que o processo — e para que a solução de DTO como um todo — obtenha progresso naquela rodada[2]. A razão para tal é que o sucesso de um processo na rodada pode resultar, no pior caso, apenas nele concluir um procedimento de retrocesso, e que este procedimento pode ainda ter sido mal sucedido, pois algum dos processos que deveriam ser "recuperados" pode não ter tido sucesso na mesma rodada. Assim, faz-se necessário definir um predicado de progresso completo para o protocolo, que determine condições suficientes e necessárias para que mensagens sejam entregues.

Define-se o predicado de progresso para o protocolo de DSTO em termos de rodadas com *sucesso global*. Uma rodada r tem sucesso global quando todos os processos obtém sucesso nela, ou seja, quando todos os processos recebem, até o final da rodada r, todas as mensagens com um número de sequência i, uma proveniente de cada processo, que poderiam ter sido difundidas. O que se mostra a seguir é que em toda rodada com sucesso global o protocolo obtém progresso, isto é, que ao menos um processo entrega uma nova subsequência de mensagens. O caso mais simples — e, espera-se, o mais frequente nas execuções — é uma rodada r em que todos os processos enviam suas i-ésimas mensagens pela primeira vez e na qual se tem sucesso global. Neste caso, todos os processos entregam a subsequência $i-1$ e enviam, na rodada seguinte, suas mensagens $i+1$. Caso isto não ocorra, tem-se a possibilidade de que nenhum processo tenha sucesso na rodada r ou de que apenas uma parte dos processos tenha sucesso na rodada r. Neste último caso, diz-se que a rodada r teve sucesso parcial e, na rodada seguinte, parte dos processos enviarão mensagens i, enquanto outros enviarão mensagens $i+1$, e não se obtém progresso. Esta situação permanece até que em uma rodada $r' > r$ todos os processos que tiveram sucesso na rodada r estejam em procedimento de retrocesso, reenviando suas mensagens i. Caso a rodada r' tenha sucesso global, os processos que não tiveram sucesso na rodada r irão, então, entregar a subsequência $i-1$, enquanto os demais processos irão apenas concluir o procedimento de retrocesso. Assim, demonstra-se de modo informal que em toda rodada com sucesso global, algum processo entrega uma subsequência de mensagens — isto é, que o sucesso global é *suficiente* para o progresso. Além disto, para que todos os processos finalmente construam sua i-ésima sequência de mensagens e, portanto, entreguem a subsequência $i-1$ anterior é *necessário* que em alguma rodada, em que todos os processos enviam suas i-ésimas mensagens, o protocolo obtenha sucesso global.

Este predicado de que o progresso é determinado pela ocorrência de rodadas com sucesso global faz com que os processos que executam o protocolo de DSTO sempre progridam em conjunto na solução do problema e entreguem as mensagens difundidas, em geral, quase que simultaneamente. Com isto, é necessário que de fato os processos

[2]Entende-se por progresso da solução de DTO a entrega de novas mensagens para a aplicação.

progridam em uma mesma velocidade para que o protocolo tenha progresso, uma vez que um único processo dessincronizado dos demais retém o progresso do sistema como um todo. Por outro lado, por parte da aplicação distribuída que usa o protocolo de DSTO para difundir suas mensagens, o avanço síncrono, em bloco, dos processos é interessante, pois propicia que todas as instâncias (ou réplicas) da aplicação estejam, na maior parte do tempo, em estágios idênticos da computação distribuída por elas realizada.

Finalmente, a tolerância a falhas de processos ou à assincronia que pode acometer o sistema em certos períodos da execução é realizada através da redução de DTO a instâncias de consenso. Deixa-se à implementação de consenso empregada a tarefa de, ou reconfigurar o sistema — conforme foi brevemente descrito no final da Seção 4.2 — e reiniciar a execução do protocolo de DSTO em uma nova configuração, livre de processos irresponsivos ou falhos, ou prosseguir a solução de DTO através de um outro protocolo qualquer de Difusão Totalmente Ordenada que tolere falhas de processos.

4.5 Trabalhos Relacionados

A maior parte das soluções de DTO desenvolvidas não se destina ao modelo síncrono de computação: como parâmetro, das mais de 60 soluções levantadas em [27], apenas 10 são síncronas. A grande maioria destas soluções (9 das 10) se enquadra na classe ordenação no envio, o que provavelmente decorre da possibilidade de se construir e de se manter visões globais do sistema neste modelo — o que inclui a detecção perfeita de falhas.

Um primeiro grupo de protocolos síncronos da classe de ordenação no envio são os que empregam relógios sincronizados, essencialmente para duas funções. A primeira função, empregada nas soluções da subclasse de histórico de comunicação, é a de "carimbar" as mensagens com o seu instante de envio e empregar este instante, tanto como chave para ordenação das mensagens, como para detectar sua estabilidade. Dentre estes protocolos, destaca-se o HAS [24] e suas variantes, destinadas a diferentes modelos de falhas. A segunda função para a qual relógios sincronizados são empregados é particionar a execução em períodos contíguos de tempo, nos quais um único processo é autorizado a difundir suas próprias mensagens — o que caracteriza a subclasse de privilégio no envio. Esta estratégia, proveniente de mecanismos para o controle de acesso à rede e conhecida como TDMA (*Time Division Multiple Access*), é indicada para sistemas especializados, com garantias de tempo real, como é caso das soluções desenvolvidas para o MARS [40].

O segundo grupo de protocolos síncronos desta classe emprega as noções referentes à passagem do tempo e a existência de prazos para a realização de operações com intuito de otimizar o funcionamento de estratégias, que, a priori, são de semântica assíncrona. Este é o caso de algumas soluções da subclasse de privilégio no envio, que empregam temporizadores e predicados temporais para garantir uma melhor eficiência na rotação

deste privilégio. Como exemplo, tem-se o as soluções desenvolvidas por Cristian et al [26], tanto para o modelo síncrono de computação, como para o assíncrono temporizado, que garantem propriedades de tempo real na passagem do privilégio de difundir mensagens. Também pertencem a este grupo algumas soluções que empregam a estratégia de *merge* determinístico [3], categorizada na subclasse de histórico de comunicação em [27]. Estes protocolos não restringem quando os processos podem difundir suas mensagens, mas fazem com que a entrega se dê em *round-robin*, isto é, que os destinatários entreguem uma mensagem de cada emissor de cada vez. Também neste caso, as restrições temporais impostas pelo modelo síncrono permitem que se otimize o tempo de retenção de uma mensagem até que ela se torne estável e possa ser entregue à aplicação [3].

De todos os protocolos de DTO que foram encontrados na literatura, o único que emprega rodadas com semântica similar às empregadas pelo DSTO é o protocolo proposto por Gopal e Toueg [34]. Este protocolo organiza a execução em rodadas síncronas e determina que em cada rodada um único processo, chamado de transmissor, difunda suas mensagens. Os demais processos aguardam a mensagem difundida pelo transmissor da rodada e, em paralelo, difundem suas confirmações para as mensagens difundidas na rodada precedente. Quando um número suficiente de confirmações é recebido para as mensagens difundidas em uma rodada, elas se tornam estáveis, e serão entregues no final da rodada seguinte. Assim, na rodada r todos os processos confirmam as mensagens difundidas pelo transmissor da rodada $r-1$ e, tendo sucesso neste procedimento, entregam as mensagens difundidas pelo emissor da rodada $r-2$. O resultado é um protocolo da subclasse de privilégio no envio, que implementa a versão uniforme de DTO e que, na ausência de falhas, entrega mensagens 3 rodadas após a sua difusão [34].

O protocolo tolera falhas por parada e por omissão, que devem ser limitadas a uma minoria dos processos [34]. Quando a falha de um transmissor é detectada em uma rodada r, os processos iniciam instâncias de consenso para decidir as mensagens a serem entregues relativas às rodadas $r-1$ e $r-2$. O protocolo de consenso empregado nas etapas de recuperação também é descrito no trabalho, executa em rodadas síncronas e tolera falhas por omissão. Estas instâncias são resolvidas em $f+1$ rodadas — onde f é o número máximo de falhas — e depois os processos retornam ao fluxo normal de execução. Na realidade, o próprio protocolo de DTO proposto pelos autores tem a semântica de um protocolo de consenso síncrono, no qual um único processo — o transmissor — tem exclusividade de propor novos valores a cada rodada. Os demais processos, por sua vez, quando recebem o valor proposto por este processo distinto o aceitam — o que, na ausência de falhas, pode ser otimizado de modo a requerer apenas duas rodadas [51].

Porém, a solução de DTO que mais se aproxima do protocolo proposto neste trabalho é a abstração *Synchronized Phase Systems* (SPS) ou Sistemas de Fases Sincronizadas [28]. A abstração de fase definida neste trabalho compartilha muitas características com o

conceito de visão empregado por sistemas de gerência de grupos [12]. Os processos em uma mesma fase compartilham uma mesma visão do conjunto de processos corretos e as fases são identificadas por valores únicos e totalmente ordenados, anexados a todas as mensagens. Dentro de cada fase a execução é organizada em rodadas que, porém, têm uma semântica assíncrona: uma rodada termina quando o processo recebe uma mensagem de cada processo pertencente à sua fase, ou então, quando se tem uma suspeita de falha. Neste último caso o processo inicia uma nova fase, o que também ocorre quando ele recebe uma mensagem com identificador de fase superior ao que é empregado por ele. Para implementar estas funcionalidades, o SPS supõe que os canais de comunicação sejam, apesar de assíncronos, confiáveis e emprega detectores não confiáveis de falhas [17].

O protocolo de DTO construído sobre o SPS, chamado de ATR [28], opera da seguinte forma. Os processos podem difundir mensagens a qualquer momento, ao enviá-las para todos os processos de sua fase. Em cada rodada da fase, os processos difundem uns para os outros os conjuntos de mensagens recebidas no decorrer da roda precedente e busca-se acordo numa subsequência comum de mensagens a serem entregues. Na ausência de suspeitas de falhas, o ATR necessita ao menos duas rodadas para entregar um conjunto de mensagens — além das difusões iniciais das mensagens, realizadas pelos seus emissores. A exigência de que de cada fase participe pelo menos uma maioria dos processos e a ordenação total das fases embasa a corretude da solução, que também implementa a versão Uniforme de DTO. Por outro lado, o ATR apresenta uma alta complexidade, tanto em termos de sua complexa implementação, como em termos do número de difusões para entrega de um conjunto de mensagens e da quantidade de informação que cada mensagem precisa carregar. Além disto, seus requisitos de comunicação e de sincronia — isto é, a classe de detectores de falhas empregada — são mais exigentes que os de outras soluções da classe de acordo na recepção, como Paxos [43] ou Chandra e Toueg [17].

Capítulo 5

Avaliação do Protocolo de Difusão Síncrona Totalmente Ordenada

Com a especificação, a implementação e a análise de um mecanismo de sincronização, responsável por organizar as computações em rodadas, descritas no Capítulo 3, tem-se o ambiente necessário para a implementação do protocolo de Difusão Síncrona Totalmente Ordenada que foi descrito no Capítulo 4. Este capítulo apresenta a implementação do protocolo de DSTO e os experimentos realizados a fim de se avaliar o seu desempenho. Os resultados obtidos são analisados e o desempenho do protocolo de DSTO é comparado com o desempenho de outros protocolos de DTO encontrados na literatura.

5.1 Métricas de Avaliação de Desempenho

Uma medida interessante na avaliação do desempenho de protocolos de DTO é a sua *latência* para a entrega das mensagens difundidas. A latência para a entrega de uma mensagem é definida como o tempo decorrido entre o instante em que a requisição para que ela seja difundida é recebida pelo protocolo e o instante em que o protocolo entrega a mensagem à aplicação. Em termos analíticos, as latências são mensuradas pelo número de *passos de comunicação* necessários para que uma mensagem difundida seja entregue à aplicação. Nesta modelagem, um passo de comunicação equivale ao atraso máximo para que uma mensagem seja entregue aos seus destinatários por canal de comunicação, seja ele ponto-a-ponto ou de difusão [38]. Quando se trata do modelo síncrono, este atraso é conhecido, mas em modelos parcialmente síncronos, este atraso é o valor esperado para o atraso máximo de comunicação em "boas" condições de execução do sistema.

Em particular, foi provado que algoritmos para a versão Uniforme de DTO têm latência mínima de dois passos de comunicação, que só pode ser obtida na ausência de falhas (ou da suspeita de falhas) de processos, quando se considera o modelo síncrono de computação,

onde a detecção de falhas é perfeita [17]. Este resultado advém do limite inferior para a resolução do Consenso Uniforme, descrito no final da Seção 4.1, visto que DTO se reduz ao consenso na presença de falhas [15] e que ambos os problemas compartilham a propriedade de Acordo Uniforme[1]. Por outro lado, quando se considera a versão não Uniforme da propriedade de Acordo, torna-se possível resolver Consenso e DTO (não Uniforme), em apenas um passo de comunicação, no melhor caso [20, 38].

Na prática, porém, a latência para a entrega de mensagens é afetada por diversos fatores, que incluem atrasos de processamento e atrasos de enfileiramento. Este último atraso se deve aos mecanismos de controle de concorrência (ou mecanismos de contenção) que são empregados pelos algoritmos, que podem fazer com que uma mensagem cuja difusão é requisitada seja retida em uma fila de espera antes de ser enviada. Em particular, quando a taxa de requisições recebidas da aplicação se aproxima da taxa de difusões simultâneas que os mecanismos de paralelização — controle de concorrência e filas — do algoritmo permitem realizar, o atraso de enfileiramento passa a ser relevante e crescente, quando não dominante, nas latências aferidas. O mesmo ocorre, de certa forma, com relação aos processos e os canais de comunicação: quando a carga que eles devem atender se aproxima da sua capacidade máxima de processamento e transmissão, os atrasos para a realização das tarefas necessárias à difusão tornam-se imprevisíveis e discrepantes em relação ao seu comportamento habitual, o que afeta diretamente as latências aferidas.

Uma outra métrica importante na avaliação de protocolos de DTO é a sua *vazão*, que é definida como a quantidade máxima de mensagens que o protocolo é capaz de entregar à aplicação por unidade de tempo. Esta definição, porém, não leva em consideração o tamanho das mensagens, que é um fator que influencia a vazão, uma vez que o seu principal limitante, na prática, é a capacidade de transmissão de dados da rede. Assim, na avaliação do desempenho de protocolos de DTO é comum empregar o termo vazão para representar a quantidade de bytes que podem ser entregues à aplicação por unidade de tempo. Com uma definição ou com outra, a vazão de um protocolo representa a carga por ele suportada, em termos do número de requisições de difusão que ele é capaz de atender por unidade de tempo, sem que as mensagens precisem ser retidas — e, portanto, tenham suas latências afetadas, como foi acima descrito.

5.2 Desempenho Experimental do Protocolo

A fim de se avaliar o desempenho apresentado pelo protocolo de DTO proposto no Capítulo 4, ele foi implementado sobre — isto é, como a aplicação cliente — o mecanismo de sincronização descrito no Capítulo 3. Assim, é o mecanismo de sincronização que for-

[1] O Acordo Uniforme determina que uma mensagem entregue (ou um valor decidido, no caso do consenso) por um processo qualquer deva ser entregue (ou decidido) por todos os processos corretos.

nece ao protocolo de DSTO a abstração de rodadas necessária para a sua implementação e que gerencia toda a comunicação por ele realizada. Como foi descrito na Seção 3.1, a interação entre o mecanismo de sincronização e a aplicação que é a sua cliente — que é, neste caso, o protocolo de Difusão Síncrona Totalmente Ordenada — ocorre ao final de cada rodada. Com o término da rodada r, por conta do início de uma rodada $r' > r$, a aplicação é invocada para processar o conjunto M_r de mensagens que foram recebidas na rodada r e, como resultado deste processamento, a aplicação retorna ao mecanismo de sincronização a próxima mensagem que deve ser enviada, referente à rodada r'.

As mensagens difundidas pelo protocolo de DSTO — que correspondem às cargas úteis das mensagens difundidas pelo mecanismo de sincronização, conforme descrito na Seção 3.2 — são sempre compostas por quatro campos. O primeiro campo é um inteiro e representa o número de sequência da mensagem m da aplicação carregada pela mensagem do protocolo — que é denotado por $seq(m)$ no Algoritmo 4.2 da Seção 4.4, que descreve a versão do protocolo de DSTO tolerante a falhas de desempenho. O segundo campo, que é também um inteiro, armazena o identificador do processo que difundiu a mensagem e é empregado pela função `ordena()` do Algoritmo 4.2, como chave de ordenação para gerar as subsequências de mensagens a serem entregues à aplicação. O terceiro campo representa o instante — aferido segundo o relógio do processo, em nanossegundos — em que a mensagem m foi enviada pela primeira vez, e é empregado para computar a latência para a entrega à aplicação de cada mensagem difundida. O quarto campo, por sua vez, armazena a mensagem m da aplicação em si, representada nos experimentos por uma sequência com S bytes aleatórios, onde S é um dos parâmetros dos experimentos.

Os experimentos realizados para avaliar o desempenho do protocolo de DSTO são idênticos aos realizados para avaliar o mecanismo de sincronização, descritos na Seção 3.2. Consideram-se execuções livres de falhas de processos, de modo que o procedimento para a tolerância falhas do protocolo de DSTO, descrito na Seção 4.3 não é empregado. Um processo pré-selecionado atua como sincronizador durante toda a execução e ele não é suspeito pelos demais processos, de modo que toda execução ocorre dentro de uma mesma fase. Ao sincronizador é fornecido o período Δ a ser empregado na geração e difusão de tiques, que é dado em microssegundos ($1\mu s = 10^{-6}s$) e corresponde à duração esperada para as rodadas. Como determina o Algoritmo 4.2, em toda rodada de que participa, o processo (re)envia uma mensagem, com tamanho fixo S. A este tamanho deve-se adicionar os dois cabeçalhos — do mecanismo de sincronização e do protocolo — que, em conjunto, agregam um excedente de 260 bytes à mensagem. De modo que dada a duração Δ esperada para as rodadas e o número n de processos participantes, a carga teoricamente imposta à rede pelo protocolo de DSTO é de $8n(S + 260)/\Delta$ Mb/s.

Na execução do protocolo de DSTO, se um processo obtém sucesso em uma rodada e gera uma nova subsequência i com as mensagens recebidas, ele entrega a subsequência

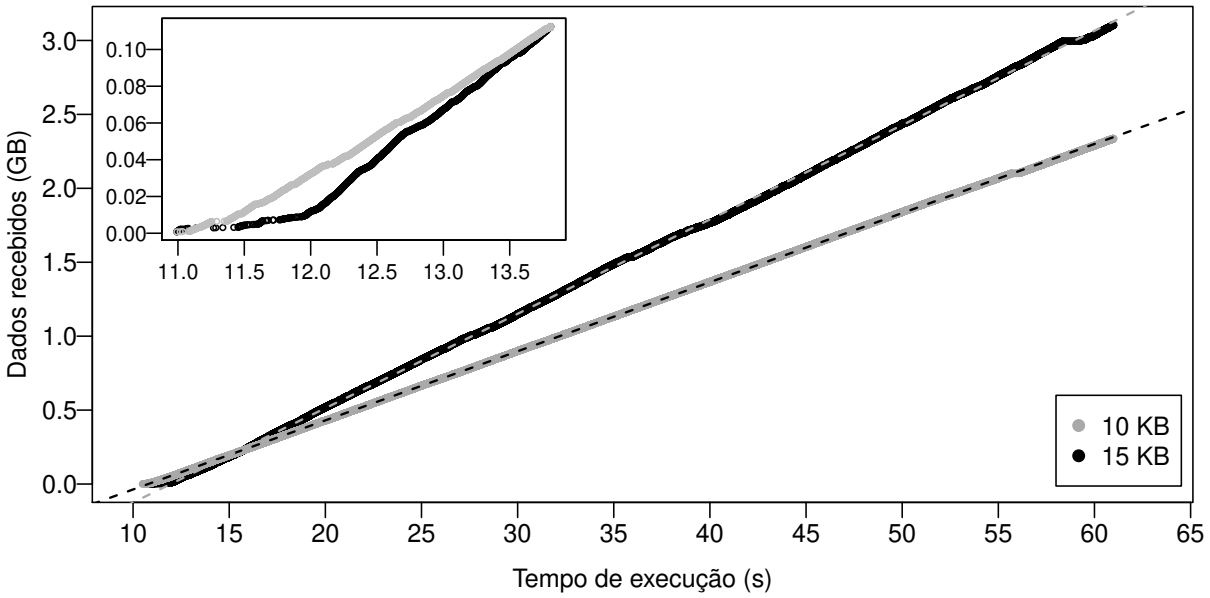


Figura 5.1: Quantidade de Gigabytes recebidos a cada instante da execução, com $n = 5$, $\Delta = 1000\mu s$ e mensagens de 10 e 15 KB. As retas tracejadas são as com melhor ajuste aos pontos representados e suas inclinações são as vazões efetivas das duas execuções.

$i - 1$, a última gerada, composta por n mensagens de S bytes cada. O instante, segundo o relógio do processo, em que cada subsequência foi entregue é registrado, o que permite que se compute a *latência* para a entrega do bloco de mensagens — pela diferença com o instante de envio da mensagem desta subsequência que foi difundida pelo processo em questão. O registro dos instantes em que cada subsequência foi entregue por um processo também permite computar a vazão por ele obtida na execução do experimento, conforme ilustrado na Figura 5.1. Nela estão representados dois conjuntos de pontos, relativos a duas execuções independentes do protocolo. Cada ponto representa o instante em que o processo entregou uma nova subsequência de mensagens e o total de bytes recebidos pela aplicação até aquele instante, após a entrega da nova subsequência. As duas retas traçadas na Figura 5.1 são as retas que apresentam o melhor ajuste em relação aos dois conjuntos de pontos representados, e os seus coeficientes angulares constituem o que se define como a *vazão efetiva* obtida pelo processo em cada uma das execuções.

Algumas observações são necessárias em relação ao método empregado para se obter os indicadores de desempenho experimental do protocolo de DSTO. Em relação à vazão obtida na execuções, nota-se, a partir da análise da Figura 5.1, que em alguns períodos a entrega de novas subsequências se dá de forma contínua, quase que linear no tempo, enquanto em outros períodos as entregas estão mais espaçadas no tempo, o que se reflete em descontinuidades no gráfico. Ao se computar uma reta de melhor ajuste, denota-se a

taxa instantânea média de entrega das mensagens, tal como é observada pela aplicação, já com um reajuste estatístico relativo aos períodos em que as entregas são mais espaçadas. Em relação às latências, é importante destacar que elas correspondem ao tempo que o protocolo de DSTO leva para entregar uma mensagem e não considera o tempo em que as mensagens ficam retidas antes de poderem ser difundidas. Assim, para estimar a latência real de uma mensagem é necessário levar em conta que, além da latência aferida, ela pode ficar retida até a duração Δ de uma rodada do protocolo.

5.3 Vazão Efetiva do Protocolo

A vazão obtida pelo protocolo de DSTO, dadas as circunstâncias em que são realizados os experimentos, ou seja, dado que cada processo tem uma nova mensagem para difundir sempre que for possível difundir uma nova mensagem, é determinada por dois fatores. O primeiro fator é o número máximo de mensagens ou a quantidade máxima de bytes que os processos — que todos os processos, em conjunto — podem difundir por unidade de tempo. Esta taxa máxima de envio de mensagens pelos processos corresponde à taxa máxima de entrega de mensagens pelos processos, uma vez que uma nova mensagem é difundida por um processo sempre que ele entrega uma nova subsequência de mensagens. A construção do protocolo determina que uma subsequência de mensagens, contendo uma mensagem de cada um dos n processos, seja entregue, no melhor caso de execução, a cada rodada. Assim, dado o tamanho S das mensagens difundidas e a duração Δ de referência para as rodadas executadas pelos processos, a vazão máxima teórica que pode ser obtida, também chamada de *vazão ótima* do protocolo de DSTO é dada por (nS/Δ) MB/s.

A vazão ótima do protocolo, porém, só é obtida em situações ideais de execução, em que as rodadas induzidas pelo mecanismo de sincronização têm, na média, a duração Δ esperada e em que todos os processos têm sucesso em todas as rodadas executadas. Isto é, a vazão ótima do protocolo é a vazão obtida nos períodos nos quais o sistema se comporta de forma síncrona e não ocorrem falhas de desempenho. A frequência com que estes períodos de sincronia ocorrem — isto é, com que o sistema é capaz de cumprir os prazos estabelecidos, condensados pela duração Δ de referência para as rodadas — define o segundo fator que determina a vazão obtida pelo protocolo de DSTO. Ou, em uma definição mais precisa, o segundo fator determinante para a vazão obtida é proporção de rodadas em que são úteis ao protocolo, isto é, nas quais se obtém *sucesso global*[2].

As rodadas nas quais se obtém sucesso global são necessariamente rodadas em que não ocorrem falhas de desempenho, das quais participam todos os processos e nas quais todas as mensagens difundidas são recebidas a tempo por todos os processos. A frequência com

[2]Conforme descrito na Seção 4.4.2, uma rodada tem sucesso global quando todos os processos têm sucesso nela, de forma que ao menos um processo entrega uma nova subsequência de mensagens à aplicação.

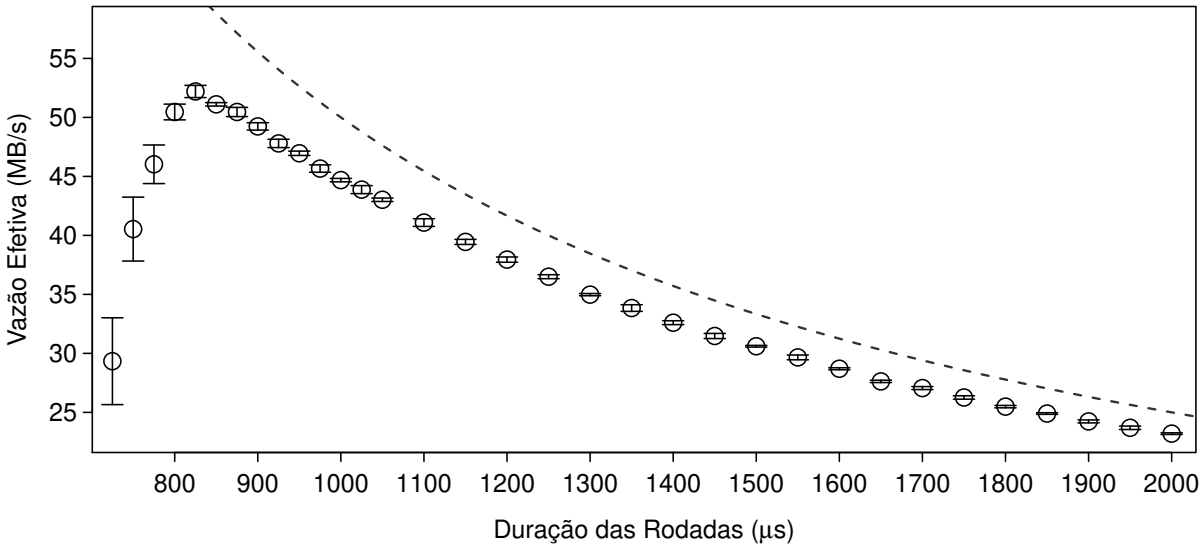


Figura 5.2: Vazão efetiva obtida nos experimentos com o protocolo de DSTO, com $n = 5$ processos e mensagens de $S = 10$ KB, em função da duração Δ de referência para as rodadas. A curva tracejada denota a vazão ótima para o experimentos, em função de Δ.

que rodadas com estas características ocorrem durante a execução, conforme foi avaliado no Capítulo 3, depende essencialmente da duração Δ de referência para as rodadas: quanto maior for a duração Δ das rodadas, maior a probabilidade que o sistema cumpra os prazos estabelecidos e, portanto, maior a proporção de rodadas com sucesso. Por outro lado, quanto menor for Δ, maior a probabilidade de falhas de desempenho e menor será a proporção de rodadas úteis para o protocolo, em termos de obtenção de progresso. A questão, porém, é que a duração das rodadas também determina quantas mensagens podem ser difundidas pelos processos por unidade de tempo, e, portanto limita a vazão ótima do protocolo, que é o primeiro fator determinante da vazão. Assim, se por um lado rodadas mais longas aumentam a probabilidade dos processos obterem sucesso nelas, por outro elas reduzem a vazão máxima que pode ser obtida pelo protocolo.

Esta relação contraditória entre os dois fatores determinantes para a vazão obtida pelo protocolo de DSTO pode ser observada na Figura 5.2, que apresenta a vazão efetiva aferida para o protocolo em uma série de experimentos, nos quais se variou a duração Δ de referência das rodadas. As execuções contaram com $n = 5$ processos, que difundiram mensagens com $S = 10$ KB nas 300 mil rodadas iniciadas pelo sincronizador. Cada experimento foi executado cinco vezes e os pontos e as barras da Figura 5.2 denotam, respectivamente, as médias e os desvios padrão das vazões efetivas neles aferidas. Já a linha tracejada denota a vazão ótima do protocolo, em função da duração esperada para as rodadas, que representa o limite superior para a vazão que pode ser obtida pelo protocolo.

Vazão / Δ	$2000\mu s$	$1800\mu s$	$1600\mu s$	$1400\mu s$	$1200\mu s$	$1000\mu s$
Valor Ótimo	25.00	27.78	31.25	35.71	41.67	50.00
Valor Médio	23.19	25.49	28.70	32.60	37.95	44.68
Desvio Padrão	0.069	0.102	0.084	0.164	0.222	0.144
Eficiência	92.79%	91.75%	91.83%	91.28%	91.08%	89.37%

Vazão / Δ	$950\mu s$	$900\mu s$	$850\mu s$	$825\mu s$	$800\mu s$	$750\mu s$
Valor Ótimo	52.63	55.56	58.82	60.61	62.50	66.67
Valor Médio	46.96	49.23	51.11	52.20	50.46	40.53
Desvio Padrão	0.184	0.312	0.145	0.517	0.667	2.710
Eficiência	89.22%	88.63%	86.89%	86.13%	80.75%	60.80%

Tabela 5.1: Valores ótimos, médios e seus desvios padrão aferidos em MB/s para a vazão do protocolo de DSTO, em função da duração de referência das rodadas, em experimentos com $n = 5$ e $S = 10$ KB. As eficiências são as razões entre os valores médios e ótimos.

Nota-se que, como previsto, com o aumento da duração Δ de referência para as rodadas, as vazões tendem a se aproximar das vazões ótimas, o que significa que a proporção de rodadas executadas em que o protocolo tem sucesso global torna-se maior.

Assim, para rodadas com $\Delta = 2000\mu s$ a vazão ótima computada foi de 25 MB/s e a vazão efetiva aferida para o protocolo foi em média de 23.19 MB/s, o que significa que em cerca de 92.8% das rodadas o processo entregou mensagens para aplicação, ou que a *eficiência* do protocolo foi de 92.8%. Ao se reduzir paulatinamente a duração das rodadas, até que se atinja $\Delta = 1000\mu s$, a vazão ótima cresce continuamente até 50 MB/s, de modo que mesmo com a contínua queda da eficiência do protocolo, até atingir cerca de 89.4%, obtém-se vazões efetivas crescentes, que alcançam uma média de 44.68 MB/s, como pode ser notado nos resultados relacionados na parte superior da Tabela 5.1. Ao se reduzir ainda mais Δ, este padrão se mantém — a redução da eficiência do protocolo, contrabalanceado pelo aumento da vazão ótima — e nota-se um aumento da vazão efetiva obtida até a duração limite $\Delta = 825\mu s$, conforme apresentado na porção inferior da Tabela 5.1. Nesta duração limite, o protocolo de DSTO obtém sua maior vazão efetiva para esta configuração com $n = 5$ processos e mensagens com $S = 10$ KB, que é dada por 52.20 ± 0.52 MB/s, o que corresponde a uma eficiência de cerca de 86.1%.

Com durações inferiores a $825\mu s$ para as rodadas, porém, nota-se uma grande queda na eficiência do protocolo, que se deve basicamente aos prazos impostos ao sistema para que as rodadas sejam completadas se tornarem muito estreitos, reduzidos, de modo que

seu cumprimento por processos e canais torna-se cada vez menos provável. Diz-se neste caso que o sistema está *saturado* ou que se chegou ao limite do sistema para aquela configuração, visto que não se pode mais melhorar o desempenho do protocolo de DSTO com o uso de durações mais agressivas — isto é, mais curtas — para as rodadas.

5.4 Latência para a Entrega de Mensagens

O segundo aspecto analisado na execução experimental do protocolo de DSTO é o tempo que decorre entre a difusão de uma mensagem e a sua entrega à aplicação, isto é, a latência do protocolo. Conforme apresentado na Seção 4.2, a latência para que as mensagens sejam entregues pelo protocolo em condições ideais de execução é de duas rodadas. Assim, a *latência ótima* do protocolo de DSTO é dada por 2Δ, onde Δ é a duração de referência para as rodadas — e, portanto, independe do número de processos ou do tamanho máximo das mensagens difundidas por cada processo, como ocorre com a vazão ótima. A latência ótima só é obtida, dada a construção do protocolo, se na rodada em que uma mensagem foi difundida — isto é, enviada pela primeira vez pelo processo — o protocolo obtém sucesso global e, na rodada seguinte, pelo menos este processo que difundiu a mensagem obtenha sucesso. No caso geral, a latência de entrega de uma mensagem i será dada pelo número de rodadas transcorridas entre a sua difusão e a ocorrência de duas rodadas com sucesso global, nas quais as mensagens i e $i + 1$ sejam recebidas por todos os processos.

Assim, mais do que a vazão — que é um valor médio — a latência para a entrega de mensagens é determinada pelo comportamento instantâneo do protocolo, isto é, de seu comportamento nas rodadas que seguem a difusão de uma mensagem. De forma que, se períodos relativamente curtos de assincronia — que geram descontinuidades na distribuição de entregas, conforme ilustrado na Figura 5.1 — não têm um impacto tão grande na vazão efetiva computada, eles são suficientes para gerar latências muito superiores ao valor ótimo e claramente discrepantes em relação aos valores aferidos para as latências de entrega da maior parte das mensagens. Como consequência, enquanto a vazão efetiva é mais influenciada pela frequência de períodos de assincronia na execução, nas latências médias o maior impacto é a duração dos períodos de assincronia.

Este comportamento é ilustrado na Figura 5.3, que apresenta valores médios para as latências aferidas em experimentos do protocolo em que se varia a duração Δ de referência das rodadas. Os resultados são obtidos em experimentos com $n = 5$ processos, com mensagens difundidas de $S = 15$ KB, em cinco execuções independentes, com 300 mil rodadas iniciadas em cada uma delas. Os círculos pretos representam os valores médios para as latências de todas mensagens difundidas, dados em milissegundos, e as barras que os cruzam são os desvio padrão aferidos, desenhados apenas quando eles superam $0.1ms$. Nota-se que, como esperado, as latências médias convergem para quase a latência ótima

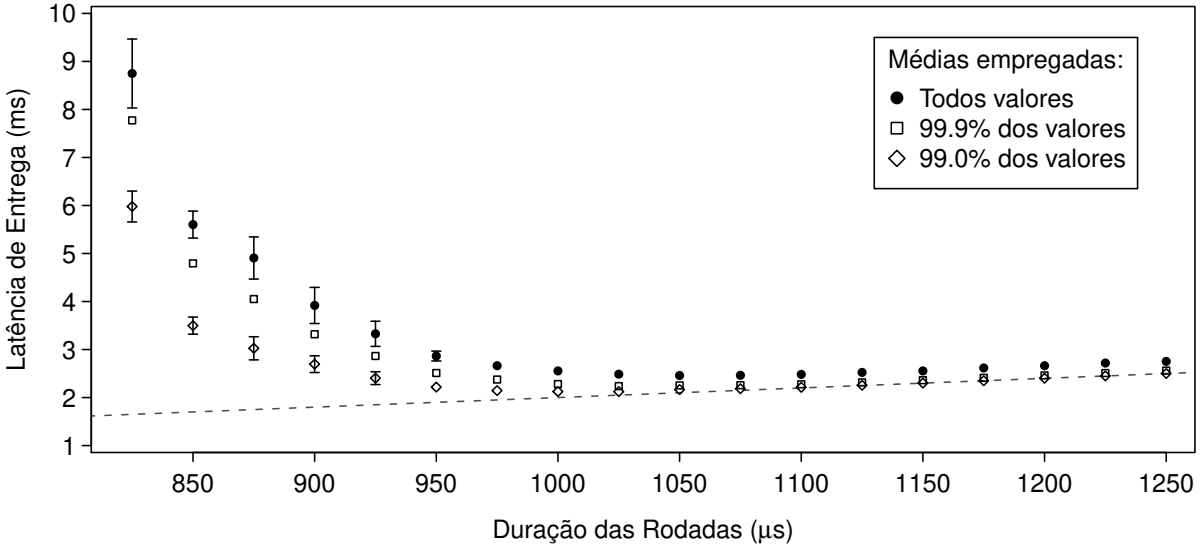


Figura 5.3: Latências médias (de todos valores e dos valores inferiores aos percentis 0.99 e 0.999) para a entrega das mensagens difundidas em experimentos com o protocolo de DSTO, com $n = 5$ e $S = 15$ KB, em função da duração Δ de referência para as rodadas.

— representada pela linha tracejada — com o aumento da duração das rodadas, mas não atingem o valor ótimo em nenhum dos experimentos. Como referência, com $\Delta = 1250\mu s$, a latência ótima é $2.5ms$ e a latência aferida nos experimentos foi de 2.753 ± 0.014 ms. Porém, quando se descarta, ao aferir-se a média, os valores de latência que ultrapassam o 0.999-percentil da distribuição de latências, o valor obtido é 2.5614 ± 0.0006 ms. E, com o descarte de mais pontos, de todos que ultrapassam o 0.99-percentil, o valor médio das latências passa a ser 2.5002 ± 0.0003 ms — que é praticamente a latência ótima para o experimento. São estas médias, computadas a partir do descarte de até 0.1% e 1% das maiores latências mensuradas — as que ultrapassam o 0.999-percentil e o 0.99-percentil — que são representadas, respectivamente, pelos quadrados e losangos da Figura 5.3.

O que é possível notar com a análise da Figura 5.3 é, primeiramente, que como ocorre com as vazões efetivas, há um valor ótimo para as latências, que ocorre com $\Delta = 1050\mu s$. Para duração de rodadas superiores, apesar das latências se tornarem mais próximas à latência ótima, por conta desta latência ótima crescer linearmente com o tamanho das rodadas, as latências médias computadas são maiores. Por outro lado, com rodadas mais curtas, a probabilidade de se obter sucesso nas rodadas torna-se menor, os períodos de assincronia tornam-se mais longos e, com isto, as latências médias também aumentam, assim como a sua variabilidade. Isto é, como resultado dos prazos tornarem-se mais restritivos, o comportamento do protocolo de DSTO torna-se mais instável, o que faz com que as latências de entrega das mensagens aumentem. Além deste comportamento,

nota-se também que, apesar das latências médias não atingirem valores ótimos, a grande maioria das mensagens difundidas são entregues com latências muito próximas à ótima. Assim, na configuração com melhor latência obtida (dentre os experimentos apresentados na Figura 5.3), com $\Delta = 1050\mu s$, a latência média foi de 2.46 ± 0.03 ms; porém, mais de 99.9% das mensagens difundidas tiveram latências de, em média, 2.25 ± 0.01 ms; e mais de 99% das mensagens foram entregues com latência média de 2.17 ± 0.01 ms — valor que é muito próximo à latência ótima para esta configuração, que é de 2.1 ms.

5.5 Discussão do Desempenho do Protocolo

Os resultados ilustrados pela Figura 5.2 e pela Figura 5.3 representam o comportamento do protocolo de DSTO que foi observado em todos experimentos realizados. Assim, para diferentes configurações de número de processos — foram verificados $n = 3, 5, 7$ — e de tamanhos das mensagens difundidas —foram verificados de 5 KB até 25 KB— a vazão efetiva e a latência média do protocolo formam uma curva, cujo valor ótimo para a duração de referência das rodadas a divide em duas regiões. Na região com valores maiores de Δ, os resultados demonstram-se mais comportados, com menor variância e uma suave e contínua queda na vazão efetiva, acompanhada por um aumento linear da latência média. Na outra região, nota-se um comportamento mais instável do sistema, com maior variância nos resultados obtidos em diferentes execuções do protocolo e uma queda brusca da vazão efetiva, acompanhada por um aumento considerável das latências médias.

Estes resultados demonstram que para diferentes configurações é possível obter uma *duração ótima* para as rodadas, com a qual se atinge o *ponto de saturação* da rede, que se reflete na obtenção do maior desempenho do protocolo de DSTO naquela configuração. Por maior desempenho aqui se entende, tanto a maior vazão efetiva computada, como a menor latência média para a entrega das mensagens. O fato destas duas métricas de desempenho atingirem seus valores ótimos para uma mesma duração de referência das rodadas é uma característica interessante do protocolo de DSTO, que não é comum em protocolos de DTO, que costumam, de certa forma, sacrificar a obtenção de baixas latências para obter altas vazões, ou controlar a vazão para obter menores latências.

A Figura 5.4 apresenta dados do comportamento ótimo obtido em três configurações, todas com $n = 5$ processos e nas quais se varia o tamanho S das mensagens difundidas, que possuem 10, 15 e 17.5 KB. Os dados referentes ao desempenho obtido na duração ótima das rodadas nestas configurações são, por sua vez, apresentados na Tabela 5.2. Nota-se que com o aumento do tamanho das mensagens, a duração ótima para as rodadas também cresce, o que se deve prioritariamente a maior latência para o envio destas mensagens, conforme discutido na Seção 3.3.3. Em adição, as latências médias são maiores, visto que as latências ótimas crescem, e também se observa uma maior vazão efetiva aferida nos

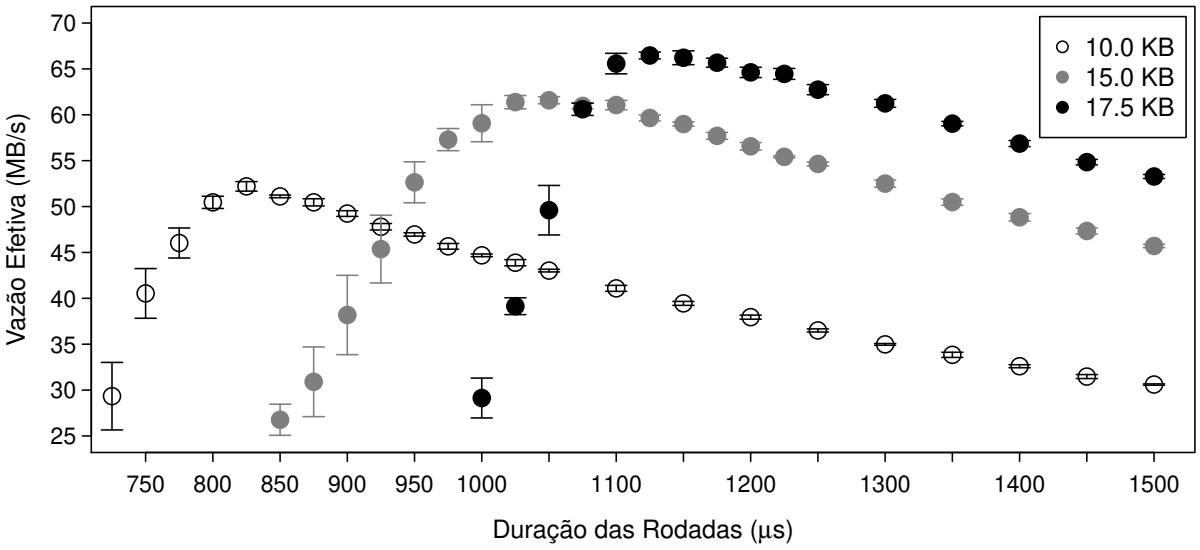


Figura 5.4: Vazão efetiva obtida nos experimentos com o protocolo de DSTO, com $n = 5$ e mensagens de 10, 15 e 17.5 KB, em função da duração Δ de referência para as rodadas.

experimentos, o que é um reflexo do aumento da vazão ótima do protocolo.

Com mensagens da aplicação maiores que 17500 bytes, porém, não se obtém melhores vazões efetivas; pelo contrário, o comportamento do protocolo torna-se mais instável e tem-se uma considerável redução na eficiência das rodadas, o que provavelmente se deve à alta carga "instantânea" que passa a ser aplicada à rede no início de cada rodada, com o envio (quase que) simultâneo de mensagens maiores. Esta maior instabilidade da rede também é observada quando se aumenta o número de processos que participam do protocolo. Em particular, os resultados tornam-se menos interessantes — em termos da vazão ótima obtida pelo protocolo — quando mais que sete processos participam dos experimentos. O impacto do aumento do número de processos que difundem mensagens regularmente no sistema e do tamanho das mensagens por eles difundidas foi discutido na Seção 3.4, onde se concluiu que estes dois fatores reduzem a eficiência obtida nas rodadas de comunicação — de modo que a queda da eficiência do protocolo de DSTO nestas circunstâncias já era, de certa forma, esperada. Além disto, é importante destacar que as soluções de DTO que têm seu desempenho comparado com o obtido pelo protocolo de DSTO (esta comparação é realizada na Seção 5.7) também obtém seu desempenho ótimo em experimentos com a mesma escala empregada nos experimentos descritos neste capítulo — normalmente com $n = 5$ processos. E com o aumento do número de processos participantes, estas soluções de DTO também sofrem com perda de desempenho, que se reflete na redução da vazão obtida e/ou no aumento das latências médias aferidas.

Configurações	10 KB, $\Delta = 825\mu s$	15 KB, $\Delta = 1050\mu s$	17.5 KB, $\Delta = 1125\mu s$
Vazão Efetiva	50.20 ± 0.52 MB/s	61.58 ± 0.38 MB/s	66.46 ± 0.37 MB/s
Latência Média	1.942 ± 0.036 ms	2.459 ± 0.032 ms	2.659 ± 0.020 ms
Latência (99.9%)	1.734 ± 0.007 ms	2.253 ± 0.014 ms	2.442 ± 0.008 ms
Latência (99.0%)	1.676 ± 0.005 ms	2.167 ± 0.011 ms	2.330 ± 0.010 ms

Tabela 5.2: Desempenho experimental ótimo do protocolo de DSTO com $n = 5$ processos.

5.6 Protocolos para DTO de Alto Desempenho

As dezenas de protocolos elaborados para Difusão Totalmente Ordenada implementam várias estratégias para a solução de DTO, sendo que várias delas são otimizadas para certos ambientes computacionais ou circunstâncias de execução [27]. Porém, na maior parte deles, o foco da solução é a obtenção de boas latências e vazões teóricas e apenas uma minoria deles apresentam resultados obtidos em experimentos ou em execuções reais. Em relação ao ambiente computacional dos aglomerados de alto desempenho, para o qual o protocolo de DSTO foi projetado, dados sobre o desempenho experimental de soluções de DTO são ainda mais escassos, mesmo sendo este o ambiente computacional empregado na quase totalidade dos centros de processamento de dados modernos [1].

Na prática, aplicações que necessitam uma semântica de comunicação com ordenação total, acabam por empregar pacotes completos[3] e muito mais complexos que dão suporte à comunicação e à gerência de grupos de processos. Estes pacotes disponibilizam uma série de funcionalidades de comunicação em grupo, com diferentes semânticas e garantias para a troca de mensagens, que normalmente incluem a Difusão Totalmente Ordenada. Dentre os pacotes livres de comunicação em grupo, que também fornecem a funcionalidade de DTO, destacam-se os dois que são mais empregados: o JGroups[4] e o Spread[5].

O JGroups [10] implementa DTO através da estratégia de sequenciador fixo, que, como foi discutido na Seção 4.1, tem como vantagens sua simplicidade e bom desempenho, mas que concentra a tarefa e a informação necessária à ordenação das mensagens em um único processo, chamado de sequenciador. Como, além disto, com intuito de melhorar seu desempenho, o JGroups otimiza o procedimento de confirmação das mensagens, a solução de Difusão Totalmente Ordenada por ele implementada não é Uniforme [7, 27, 35], o que limita sua empregabilidade em soluções com requisitos de alta consistência.

Já o Spread [4] implementa DTO através de uma variante do Totem [6], que é um

[3]Do inglês *toolkits*.
[4]Disponível em `http://www.jgroups.org`
[5]Disponível em `http://www.spread.org`

protocolo da subclasse de privilégio no envio, já citado na Seção 4.1. Assim como a maior parte das soluções desta classe, o Spread organiza os processos em um anel lógico, pelo qual circula o privilégio — isto é, a exclusividade — de difundir mensagens. Por este mesmo anel os processos confirmam o recebimento das mensagens previamente difundidas, de modo que, com o retorno do privilégio a um processo, ele tem a confirmação de que todos os processos receberam suas mensagens, que podem então ser entregues à aplicação de forma segura. De modo que, ao contrário do JGroups e a custo de uma maior latência na entrega das mensagens, o Spread implementa a versão Uniforme de DTO [27, 35].

Em relação à tolerância a falhas, o JGroups e o Spread implementam seus próprios mecanismos de gerência de grupo, responsáveis por remover os processos (suspeitos de serem) falhos e por adicionar novos processos que requisitam a entrada no grupo — o que, no caso do Spread, requer a reconstrução do anel. Apesar destas duas soluções não se focarem especificamente na obtenção de Difusão Totalmente Ordenada de alto desempenho (visto que esta é apenas uma das funcionalidades por elas disponibilizadas), o JGroups e o Spread são projetos estáveis, desenvolvidos e atualizados há vários anos. Além disto, eles são altamente configuráveis e podem ser otimizados para operar em vários ambientes computacionais, inclusive em aglomerados de alto desempenho. De modo que o desempenho obtido por estas duas soluções na implementação desta primitiva é um importante parâmetro de comparação para novos protocolos de DTO desenvolvidos.

Afora as soluções que dão suporte à computação distribuída baseada em grupos de processos e que implementam primitivas de comunicação com a semântica de ordenação total de mensagens, descreve-se a seguir dois protocolos recentes que têm como principal objetivo a obtenção de alto desempenho em DTO: o LCR [35] e o Ring-Paxos [49].

O LCR foi proposto por Guerraoui et al [35] e é um protocolo de DTO Uniforme que visa, especificamente, a obtenção da maior vazão possível no ambiente dos aglomerados de alto desempenho. Nele, os processos também são organizados em um anel lógico — como ocorre no Spread e no Ring-Paxos — e estabelecem conexões TCP com seus sucessores e seus antecessores no anel. As mensagens difundidas pelos processos circulam pelo anel em um sentido e no sentido contrário circulam suas confirmações. Uma mensagem só é entregue quando se torna estável — isto é, quando seu recebimento é confirmado por todos os processos — e a sua ordenação é realizada através de relógios lógicos [41] anexados às mensagens, de modo que ele se enquadra na subclasse de histórico de comunicação, descrita na Seção 4.1. O LCR desenvolve um mecanismo próprio para o controle de fluxo, o que, junto com uma série de otimizações no protocolo TCP, permite a obtenção do maior fluxo de dados possível através de conexões ponto-a-ponto. Como resultado, a vazão experimental por ele obtida — após um período inicial que os autores chamam de "aquecimento", necessário para que o TCP atinja sua taxa máxima de transmissão — chega a 95% da capacidade nominal da rede e permanece quase constante com o aumento

de processos no anel [35]. Por outro lado, até pela construção e o objetivo do protocolo, a latência na entrega de mensagens cresce linearmente com o tamanho do anel [49].

A principal ressalva, porém, em relação ao funcionamento do LCR é o seu mecanismo de tolerância a falhas. Assim como Spread, o LCR emprega serviços de um serviço de gestão de grupos e descreve um algoritmo para a recuperação do sistema e a reconstrução do anel em caso de falhas. Porém, a detecção de falhas per si é relegada também ao protocolo TCP: um processo considera seu sucessor ou antecessor falho quando a conexão TCP com algum deles é perdida [35]. Os autores argumentam que este mecanismo seria equivalente a ter-se um detector perfeito de falhas [17], apesar de ser demonstrado que detectores desta classe só podem ser implementados em sistemas síncronos [15].

O Ring-Paxos, proposto por Marandi et al [49], é também um protocolo de DTO Uniforme, que implementa uma variante de Paxos [43] e, portanto, é um algoritmo da classe ordenação por acordo. O Ring-Paxos também arranja os processos em um anel lógico, formado por $f+1$ processos, onde $f < n/2$ é número máximo de falhas de processos toleradas. O processo responsável por este anel — equivalente ao coordenador de Paxos — é o único que pode difundir mensagens e age como uma espécie de sequenciador único do sistema. Porém, os números sequência propostos por este processo às mensagens devem ser aceitos e o recebimento das mensagens deve ser confirmado por todos os processos do anel, antes que tais mensagens possam ser entregues. Os processos monitoram o responsável pelo anel através de detectores não confiáveis de falhas [15] e, quando há suspeitas em relação a ele, o protocolo passa a se comportar como Paxos. Com a diferença de que quando um novo processo é eleito coordenador, ele escolhe f processos para compor um novo anel sob sua responsabilidade, antes de reiniciar a difusão de novas mensagens.

O Ring-Paxos é resultado da observação de que quando vários processos difundem mensagens simultaneamente nos aglomerados de alto desempenho, em que os processos são interconectados por um comutador Ethernet de alta velocidade, as taxas de perdas de mensagens pela rede tornam-se altas[6]. Esta seria, segundo os autores, uma das razões para que Paxos e outras soluções em que vários processos difundem mensagens de modo simultâneo não obterem bons desempenhos em DSTO neste ambiente computacional [49]. O Ring-Paxos, então, para contornar este problema, reserva a um único processo — que é o responsável pelo anel — o papel de difundir mensagens, enquanto os demais enviam somente confirmações, que são mensagens pequenas e apenas circulam pelo anel.

[6]O Ring-Paxos, assim como o protocolo de DSTO, emprega IP-*multicast* na difusão de mensagens. Esta observação sobre o comportamento da rede foi confirmada neste trabalho e é discutida na Seção 3.4.

5.7 Estudo Comparativo de Protocolos de DTO

O trabalho que apresenta o Ring Paxos [49] realiza um estudo comparativo do desempenho experimental de cinco protocolos de DTO Uniforme: o LCR [35], o Spread [5], o próprio Ring Paxos e duas implementações de Paxos [43], uma desenvolvida pelos mesmos autores do Ring Paxos e outra descrita em [5]. O ambiente experimental empregado para a comparação destes protocolos é muito similar ao empregado para avaliar o desempenho do DSTO, o que permite a comparação direta dos resultados. Assim, os dados de desempenho que são apresentados em relação ao LCR, ao Ring Paxos e ao Spread — do qual não se conseguiu dados quanto à sua latência para a entrega de mensagens — provém do trabalho acima citado [49]. São apresentados também dados de desempenho de duas implementações de Paxos. A primeira, que será chamada de Paxos4SB, é a com melhor desempenho dentre as descritas em [5], segundo os experimentos apresentados no próprio trabalho, também realizados em um ambiente experimental idêntico ao nosso. A segunda implementação é o Treplica [55], que é de autoria de nosso grupo de pesquisa.

Estes cinco protocolos são comparados ao protocolo de DSTO na Tabela 5.3, que apresenta dados relativos a execuções livres de falhas com $n = 5$ processos[7]. Na coluna relativa à vazão obtida pelos protocolos, os percentuais são relativos à vazão nominal da rede, que em todos os experimentos foi de 1 Gb/s. Assim, como o LCR atinge 95% da vazão nominal da rede, sua vazão é de 950 Mb/s, que equivale a 118.75 MB/s e é a maior vazão dentre as soluções avaliadas. Por outro lado, ele também é o que apresenta a maior latência para a entrega das mensagens, dada a necessidade delas e de suas confirmações circularem por todo anel. O mesmo valeria para o Spread que, apesar de não se ter dados precisos quanto à sua latência, emprega um procedimento similar, além de pertencer à subclasse de soluções de privilégio no envio, que tem como característica normalmente apresentar altas latências na entrega de mensagens, conforme descrito na Seção 4.1.

As demais colunas da tabela 5.3 apresentam algumas características das soluções de DTO comparadas. Exceto o LCR, todas as soluções supõem canais de comunicação não confiáveis e empregam o protocolo UDP para o transporte de mensagens — o que pode ser justificado pela possibilidade de se empregar a primitiva de difusão IP-*multicast*. Como o LCR, além de empregar TCP, também supõe detecção perfeita de falhas, ele é o único que foi classificado como um protocolo que faz suposições fortes de sincronia. O Spread, apesar de ser classificado um protocolo com suposições fracas de sincronia, é o único, além do LCR, que emprega serviços de gestão de grupo [11, 12] para tolerar a ocorrência de falhas. Assim, considera-se que os requisitos de sincronia que ele requer para seu correto funcionamento são mais fortes que os dos demais, que empregam consenso [16].

[7]Apesar de ter sido descrito na Seção 5.6, o JGroups não foi incluído neste estudo comparativo por não resolver a versão Uniforme do problema de Difusão Totalmente Ordenada.

Protocolo	*Vazão*	*Latência*	*Sincronia*	*Canais*	*Topologia*	*Falhas*
LCR [35]	95%	4.6ms	Forte	TCP	Anel	Grupo
Ring-Paxos [49]	90%	4.2ms	Fraca	UDP	Roda	Consenso
DSTO (17.5 KB)	53%	2.7ms	Fraca	UDP	Estrela	Consenso
DSTO (15.0 KB)	49%	2.5ms	Fraca	UDP	Estrela	Consenso
DSTO (10.0 KB)	41%	1.9ms	Fraca	UDP	Estrela	Consenso
Spread [4]	18%	—	Fraca	UDP	Roda	Grupo
Treplica [56]	10%	4.3ms	Fraca	UDP	Estrela	Consenso
Paxos4SB [5]	6.5%	4.2ms	Fraca	UDP	Estrela	Consenso

Tabela 5.3: Tabela comparativa de soluções uniformes para DTO, com $n = 5$ processos.

A coluna referente à topologia empregada pelos protocolos refere-se à organização lógica dos processos, visto que em todos experimentos cujos resultados são apresentados, a topologia da rede é em estrela, cujo nó central é um comutador Ethernet que interconecta todos processos e que implementa difusão em nível físico. O que se denomina topologia *roda* consiste em um anel formado por $n - 1$ nós e um nó central conectado a todos eles. Este nó central, no caso do Spread é o processo que tem privilégio de difundir suas mensagens a cada instante da execução, e no Ring-Paxos é o processo responsável pelo anel, que também é o único que pode difundir mensagens. Assim, é interessante notar que as três principais soluções que foram levantadas e que são destinadas à obtenção de alto desempenho em DTO — LCR, Ring-Paxos e Spread — organizam os processos em anel. A razão principal desta escolha é evitar a *colisão* de mensagens no nó central e nos canais que saem do comutador de rede, ou ao menos minimizá-las, com o envio de mensagens pequenas, que servem prioritariamente como confirmação, através do anel.

Em relação aos resultados de desempenho, notam-se três "categorias" de protocolos de DTO que podem ser comparados ao DSTO. Na primeira delas tem-se Ring-Paxos e LCR, que apresentam altas vazões, da ordem da capacidade da rede, através de uma boa configuração de temporizadores e uma boa organização do fluxo de mensagens pelo anel, de modo a explorar esta topologia da melhor forma. Nota-se, porém que apesar do LCR ter uma vazão superior, o Ring-Paxos é implementado sobre um modelo computacional muito mais genérico, que é minimal para a resolução de DTO [15], e que, em adição, considera canais de comunicação não confiáveis. Além disto, a latência na entrega de mensagens do Ring-Paxos cresce pouco com o aumento do número de processos, enquanto no LCR ela cresce linearmente com o tamanho do anel [49]. Na segunda categoria estão soluções completas e mais complexas de comunicação em grupo, como o Spread e o JGroups — dada a análise do desempenho do Spread, JGroups e LCR feita em [35]. Estas soluções, apesar de toda sua otimização para os aglomerados e a Ethernet, apresentam um

desempenho inferior ao do protocolo de DSTO. Na última categoria têm-se as soluções baseadas em consenso, que especificamente implementam Paxos [43]. As suas limitações em termos de desempenho são conhecidas e, em particular, são citadas no restante dos trabalhos [4, 35, 49, 27]. Já suas principais qualidades, que são a capacidade de obter progresso em condições minimais em termos de falhas de processos e de sincronia, são, em particular, empregadas pelo protocolo de DSTO para tolerar falhas.

Os dados de desempenho apresentados na Tabela 5.3 para o protocolo de DSTO são os resultados ótimos obtidos em experimentos com três configurações, em que se empregou três tamanhos para as mensagens difundidas: 10 KB, 15 KB e 17.5 KB — resultados que foram apresentados com mais detalhes na Tabela 5.2. Nota-se que há um compromisso entre latência e vazão efetiva obtidas em cada configuração, mas que mesmo a latência mais alta aferida, de 2.7 ms, é consideravelmente inferior — e, portanto, melhor — que as apresentadas pelos demais protocolos, que são superiores a 4 ms. De modo que o protocolo de DSTO apresenta, não só uma latência teórica ótima de dois passos de comunicação, mas também uma latência prática, que inclui o custo de reenvio várias mensagens, bastante reduzida. Em relação à vazão apresentada, como já foi discutido, o protocolo situa-se numa faixa intermediária de desempenho, com vazões efetivas entre 40% e cerca de 65% da capacidade nominal da rede, a depender do tamanho das mensagens difundidas.

Um último aspecto — que, apesar de ser de difícil medição — é destacado refere-se à complexidade na implementação e verificação dos protocolos avaliados. O protocolo de DSTO é um protocolo extremamente simples, que se vale da prerrogativa de construção de visões globais do sistema para construir a sequência de mensagens a ser entregue a cada processo. O protocolo considera a organização das computações em rodadas, o que é implementado, também de forma simples, pelo mecanismo de sincronização descrito no Capítulo 3. A implementação de Paxos ou de suas variantes, como o Ring-Paxos, é uma tarefa complexa [5, 55], assim como a verificação de sua corretude. O mesmo vale para soluções mais completas de computação em grupo, como é o caso do Totem e do JGroups. Já o LCR, apesar de ter uma implementação razoavelmente simples, considera uma série de propriedades de sincronia, cujo cumprimento em ambientes reais de computação é questionável [17]. De forma que, apesar do desempenho do protocolo de DSTO ser inferior aos do LCR e do Ring-Paxos, o protocolo proposto apresenta um interessante compromisso entre simplicidade, requisitos de sincronia e desempenho experimental.

Capítulo 6

Conclusões

Este trabalho apresentou uma solução para Difusão Totalmente Ordenada, destinada a um dos ambientes computacionais mais empregados atualmente para processamento intensivo de dados e para a implementação de aplicações com requisitos de alta disponibilidade: os aglomerados de alto desempenho. A solução de DTO proposta explora, essencialmente, três características deste ambiente computacional: o comportamento predominantemente síncrono por ele apresentado, dado que se controle a carga que lhe é aplicada; a existência de uma primitiva eficiente de comunicação por difusão, implementada em nível de rede; e a confiabilidade do *hardware* nele empregado, que sustenta a hipótese de que a ocorrência de falhas de processos, principalmente em aglomerados pequenos, é um evento raro. Estas características dão sustentação a um modelo computacional, que é forte o bastante para a resolução de Difusão Totalmente Ordenada de forma simples e eficiente.

As contribuições desta dissertação situam-se, tanto no âmbito teórico, de projeto de algoritmos distribuídos para Difusão Totalmente Ordenada, como no âmbito prático, de implementação de soluções de comunicação em grupo com alto desempenho. A principal contribuição teórica é o protocolo de Difusão Síncrona Totalmente Ordenada, destinado ao modelo assíncrono temporizado de computação, ao qual, até onde se averiguou, apenas dois protocolos foram destinados, ambos descritos no mesmo trabalho de Cristian et al [26]. Apesar de o protocolo necessitar de informação global para obter progresso — como ocorre com várias das soluções de DTO descritas na literatura [27] — e, portanto, não tolerar nativamente a ocorrência de falhas de processos, o tratamento de falhas é realizado via a execução de instâncias de consenso[1], ao contrário da maior parte das soluções com este requisito, que normalmente empregam complexos serviços de gestão de grupos [27].

Em relação às contribuições práticas, destaca-se o mecanismo de sincronização, que se mostrou eficiente no controle da carga aplicada ao sistema e na organização de suas

[1]Existem diversas implementações estáveis e eficientes para consenso uniforme disponíveis, também destinadas ao ambiente dos aglomerados de alto desempenho, como as que são descritas em [5, 56].

computações em rodadas de comunicação, com uma semântica síncrona. Em adição, os resultados dos experimentos descritos neste mesmo capítulo indicam que o ambiente dos aglomerados de alto desempenho, dado que se empregue um mecanismo simples — como o descrito — para controle de carga, apresenta um comportamento que reflete as hipóteses consideradas pelo modelo assíncrono temporizado de computação distribuída [24]. Além do mecanismo de sincronização, a principal contribuição prática é a implementação do protocolo de DTO proposto. O protocolo apresentou um desempenho experimental, em termos de vazão, comparável ao de outras soluções de DTO mais complexas [49] ou com requisitos mais restritivos de sincronia [35], e em termos de latência, resultados superiores aos obtidos por elas. Reitera-se que a solução desenvolvida é simples de ser implementada e verificada, e que não foram feitas otimizações nos protocolos de comunicação empregados — por exemplo, os *buffers* utilizados são do tamanho padrão do sistema operacional. De modo que acreditamos que o desempenho do protocolo pode ser um pouco superior, com a escolha apropriada de certos parâmetros — o que se torna um problema de otimização.

Os trabalhos futuros podem ser classificados em duas vertentes. A primeira consiste em desenvolver estratégias para tornar a solução de DTO proposta mais escalável, sem com isto modificar a sua semântica de funcionamento ou reduzir o seu desempenho. A limitação que impede que esta versão do protocolo seja executada por um conjunto maior de processos é a perda de eficiência da primitiva de difusão de mensagens oferecida pela Ethernet quando muitos processos difundem mensagens (quase que) simultaneamente — limitação que já tinha sido observada em [49]. A segunda vertente consiste em desenvolver e implementar a integração do protocolo de DTO proposto neste trabalho, com outras soluções de DTO de semântica assíncrona e que tolerem nativamente falhas de processos. Esta integração permitiria que a solução de DTO continuasse a obter progresso com a ocorrência de falhas ou em períodos de comportamento assíncrono do sistema, mesmo que com um desempenho provavelmente inferior ao obtido por esta solução "síncrona".

Referências Bibliográficas

[1] Dennis Abts and Bob Felderman. A guided tour of data-center networking. *Commun. ACM*, 55(6):44–51, June 2012.

[2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distrib. Comput.*, 13(2):99–125, 2000.

[3] Marcos Kawazoe Aguilera and Robert E. Strom. Efficient atomic broadcast using deterministic merge. *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*, D(91128):209–218, 2000.

[4] Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. The spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Johns Hopkins University, 2004.

[5] Yair Amir and Jonathan Kirsch. Paxos for system builders. In *LADIS '08: Proceedings of Large-Scale Distributed Systems and Middleware*, New York, September 2008.

[6] Yair Amir, Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal, and Paul Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, November 1995.

[7] Emmanuelle Anceaume. A comparison of fault-tolerant atomic broadcast protocols. In *Proceedings of the Fourth Workshop on Future Trends of Distributed Computing Systems*, pages 166–172, Lisbon, Portugal, sep 1993. Institute of Electrical & Electronics Engineers (IEEE).

[8] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, jan-mar 2004.

[9] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985.

[10] Bela Ban. Design and implementation of a reliable group communication toolkit for java. Technical report, Cornell University, 1998.

[11] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[12] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[13] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions Computer Systems*, 9(3):272–314, August 1991.

[14] Daniel Cason and Luiz E. Buzato. Difusão Síncrona Totalmente Ordenada de Mensagens em Sistemas Assíncronos Temporizados. Technical Report IC-11-25, Institute of Computing, University of Campinas, December 2011.

[15] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[16] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 322–330, New York, NY, USA, 1996. ACM.

[17] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[18] Jo-Mei Chang and Nicholas F. Maxemchuk. Reliable broadcast protocols. *ACM Transactional on Computer Systems (TOCS)*, 2(3):251–273, August 1984.

[19] Bernadette Charron-Bost, Rachid Guerraoui, and André Schiper. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *Proceedings International Conference on Dependable Systems and Networks, 2000 (DSN 2000)*, page 523, Los Alamitos, CA, USA, 2000. IEEE.

[20] Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus. *Journal of Algorithms*, 51(1):15–37, 2004.

[21] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22:49–71, 2009. 10.1007/s00446-009-0084-6.

[22] Flaviu Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, 1991.

[23] Flaviu Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4:175–187, 1991.

[24] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: from simple message diffusion to byzantine agreement. In *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years', Twenty-Fifth International Symposium on*, page 431, jun 1995.

[25] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10:642–657, 1999.

[26] Flaviu Cristian, Shivakant Mishra, and Guillermo Alvarez. High-performance asynchronous atomic broadcast. *Distributed Systems Engineering*, 4(2):109, 1997.

[27] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[28] Carole Delporte-Gallet and Hugues Fauconnier. Real-time fault-tolerant atomic broadcast. In *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*, pages 48–55, 1999.

[29] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[30] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[31] Christof Fetzer and Flaviu Cristian. Fail-awareness in timed asynchronous systems. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 314–321, New York, NY, USA, 1996. ACM.

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

[33] Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, PODC '98, pages 143–152, New York, NY, USA, 1998. ACM.

[34] Ajei Gopal and Sam Toueg. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In Jean-Claude Bermond and Michel Raynal, editors, *Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 110–123. Springer Berlin / Heidelberg, 1989. 10.1007/3-540-51687-5_36.

[35] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28(2):5:1–5:32, July 2010.

[36] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Dep. of Computer Science, Cornell University, may 1994.

[37] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the Amoeba group communication system. *2012 IEEE 32nd International Conference on Distributed Computing Systems*, 0:436, 1996.

[38] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *SIGACT News*, 32(2):45–63, 2001.

[39] Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47 – 52, 2003.

[40] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: the Mars approach. *Micro, IEEE*, 9(1):25 –40, feb 1989.

[41] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[42] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984.

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

[44] Leslie Lamport. Fast Paxos. *Distrib. Comput.*, 19(2):79–103, October 2006.

[45] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, June 2006.

[46] Leslie Lamport and Nancy Lynch. Handbook of theoretical computer science (vol. b). chapter Distributed computing: models and methods, pages 1157–1199. MIT Press, Cambridge, MA, USA, 1990.

[47] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41:63–73, March 2010.

[48] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[49] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *DSN 2010: 40th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 527–536, Chicago, USA, June 2010.

[50] David L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[51] Michel Raynal. Consensus in synchronous systems: a concise guided tour. Technical report, Institut de Reserche en Systèmes Aléatoires, Université de Rennes 1, July 2002.

[52] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[53] Péter Urbán, Xavier Défago, and André Schiper. Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be? In *SRDS '01: Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 190–193, New Orleans, LA, USA, October 2001. IEEE Computer Society.

[54] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. Practical TDMA for datacenter ethernet. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

[55] Gustavo Maciel Dias Vieira and Luiz Eduardo Buzato. Treplica: Ubiquitous replication. In *SBRC '08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*, Rio de Janeiro, Brasil, May 2008.

[56] Gustavo Maciel Dias Vieira and Luiz Eduardo Buzato. The performance of Paxos and Fast Paxos. In *SBRC '09: Proc. of the 27th Brazilian Symposium on Computer Networks and Distributed Systems*, pages 291–304, Recife, Brasil, May 2009.

[57] Uwe Wilhelm and André Schiper. A hierarchy of totally ordered multicasts. In *Reliable Distributed Systems, 1995. Proceedings., 14th Symposium on*, pages 106 – 115, sep 1995.