ALLAN EDGARD SILVA FREITAS

SIMULAÇÃO DE SISTEMAS DISTRIBUÍDOS HÍBRIDOS E DINÂMICOS

Tese apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Estadual de Feira de Santana e Universidade Salvador, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Dr. Raimundo José de Araújo Macêdo

Salvador
2013

Sistema de Bibliotecas da UFBA

Freitas, Allan Edgard Silva
Simulação de Sistemas Distribuídos Híbridos e Dinâmicos / Allan Edgard Silva Freitas. – 2013.
136p. : il.

Inclui apêndices.
Orientador: Prof. Dr. Raimundo José de Araújo Macêdo.
Tese (doutorado) – Universidade Federal da Bahia, Instituto de Matemática. Universidade Estadual de Feira de Santana, Universidade Salvador, 2013.

1. Métodos de Simulação. 2. Simulação (Computadores). 3. Sistemas operacionais distribuídos (Computadores). 4. Tolerância a falhas (Computação). 5. Rede de computador - Protocolos - Avaliação. I. Macêdo, Raimundo José de Araújo. II. Universidade Federal da Bahia. Instituto de Matemática. III. Universidade Estadual de Feira de Santana. IV. Universidade Salvador. V. Título

CDD - 519.2
CDU - 519.119

TERMO DE APROVAÇÃO

ALLAN EDGARD SILVA FREITAS

SIMULAÇÃO DE SISTEMAS DISTRIBUÍDOS HÍBRIDOS E DINÂMICOS

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da UFBA-UEFS-UNIFACS.

Salvador, 17 de Maio de 2013

Professor e orientador Raimundo José de Araújo Macêdo, Ph.D.
Universidade Federal da Bahia

Professor Elias Procópio Duarte Júnior, Ph.D.
Universidade Federal do Paraná

Professor Luciano Paschoal Gaspary, Dr.
Universidade Federal do Rio Grande do Sul

Professor George Marconi de Araújo Lima, Ph.D.
Universidade Federal da Bahia

Professor Sérgio Gorender, Dr.
Universidade Federal da Bahia

Ao meu melhor trabalho desenvolvido durante esta Tese, meu filho Luan, e a Luana, minha esposa, pela co-autoria.

AGRADECIMENTOS

- Aos meus amigos pela compreensão por todos os momentos que não cultivamos juntos enquanto este trabalho era realizado;
- Aos professores do IFBA, em especial a Romildo, Flavia, Manoel, Sandro, Pablo e a todos do curso de ADS;
- Aos técnicos do IFBA, em especial a Lazaro, Edna, Marcio, Thiago e a todos da DGTI;
- Aos meus alunos do IFBA;
- Aos professores, técnicos, em especial aos do Departamento de Ciencia da Computação e do Centro de Processamento de Dados;
- Aos colegas do doutorado da UFBA, em especial o Alírio;
- À UFBA por ter me proporcionado toda minha formação universitária desde a graduação;
- Aos revisores do SBRC, DAIS, LADC, WoSiDA, SBESC e JPDC pelo trabalho voluntário que contribuiu na melhoria desta tese;
- Aos professor Macêdo pela excelente orientação e por compartilhar sempre um pouco da sua grande experiência de pesquisa;
- Aos membros da banca por contribuírem na análise deste trabalho;
- Ao meu filho Luan e à minha esposa Luana, por todo o apoio e por a cada dia me fazerem uma pessoa melhor;
- Aos meus pais, Joselito e Vera, e aos meus irmãos, Charles e Fernanda, por tudo que significam para mim;
- Ao ser inteligente que torna o milagre da vida possível: Deus.

Salvador, 17 de Maio de 2013
Allan Edgard Silva Freitas

As a profession, we seem to specialise in re-inventing the wheel, and in inventing jargon that, by accident or design, obscures the fact of re-invention.

—BRIAN RANDELL AND JOHN E. DOBSON (1986)

RESUMO

Sistemas distribuídos são usualmente definidos como um conjunto de processos residentes em sítios variados de uma rede de computadores e que se comunicam através de canais de comunicação. Processos e canais são caracterizados por comportamentos temporais síncronos ou assíncronos a depender dos recursos subjacentes (sistemas operacionais e subsistema de comunicação). Diferentemente dos sistemas convencionais, as características temporais dos sistemas híbridos e dinâmicos variam com o tempo, de acordo com a disponibilidade de recursos e ocorrência de falhas. Tais sistemas estão se tornando cada vez mais comuns nos dias de hoje devido à crescente diversidade, heterogeneidade e onipresença das redes e dispositivos computacionais. Devido à sua grande complexidade, tais sistemas são difíceis de serem testados ou verificados. Nesta tese, introduzimos um novo ambiente de simulação para tais ambientes, onde diversos modelos de falhas e comportamentos temporais podem ser associados dinamicamente a processos e canais de comunicação. Tal ambiente foi utilizado no desenvolvimento e na avaliação de desempenho de diveros protocolos distribuídos, como, por exemplo, de um protocolo de comunicação em grupo adequado aos ambientes híbridos e dinâmicos, onde é possível adaptar o comportamento dos algoritmos conforme o estado percebido do sistema (*self-aware*), de um protocolo de comunicação em grupo auto-configurável e de uma versão adaptativa do clássico protocolo PBFT, utilizado para replicação ativa de estado em ambientes sujeitos a falhas bizantinas.

Palavras-chave: métodos de simulação, simulação (computadores), sistemas operacionais distribuídos (computadores), tolerância a falhas (computação), rede de computador - protocolos - avaliação.

ABSTRACT

Distributed systems are defined as a set of processes that communicate with each other by message passing, through communication channels, and may be located at several computers spread over a communication network. These processes and communication channels are usually characterized by synchronous or asynchronous timeliness behavior, according to the characteristics of underlying system (operating system and communication sub-system). Unlike conventional systems, the timeliness characteristics of dynamic and hybrid distributed systems may vary over time, according to the availability of resources and occurrence of failures. Such systems are becoming common today because of the increasing diversity and heterogeneity of computer networks and associated devices. Due their high complexity, these systems are difficult to test or verify. In this paper, we introduce a simulation tool for such environments, where distinct fault models and timeliness properties can be dynamically assigned to processes and communication channels. Such a tool was used in development and performance evaluation of distributed protocols, as, for instance, a group communication protocol for a self-aware hybrid and dynamic distributed system, a self-manageable group communication protocol and an adaptive version of the well-known Pratical Byzantine Fault-Tolerance, a classical protocol for achieve state-machine replication under byzantine faults.

Keywords: simulation techniques, simulation (computers), distributed operating systems (computers), fault-tolerance (computing), computer network - protocol - evaluation.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE CÓDIGOS-FONTE

Capítulo 1

INTRODUÇÃO

Sistemas distribuídos consistem de processos que executam em uma infraestrutura computacional e comunicam-se por meio de troca de mensagens, através de canais de comunicação fim-a-fim, os quais abstraem toda a infraestrutura de uma rede de computadores. De modo a permitir o melhor estudo destes sistemas, e o desenho das soluções computacionais associadas, caracterizamos modelos de sistemas distribuídos, os quais definem, por exemplo, hipóteses sobre falhas dos componentes (processos e canais) e sobre as latências de processamento e comunicação fim-a-fim. Quanto às hipóteses associadas a tais latências de processamento e comunicação fim-a-fim (comportamento temporal), destacam-se na literatura, os modelos clássicos, assíncrono e síncrono.

O modelo síncrono nos permite utilizar o determinismo da existência de limites máximos associados às latências de processamento e de comunicação fim-a-fim, de modo a facilitar o desenho de algoritmos distribuídos, como o consenso distribuído, os quais podem, por exemplo, detectar de forma correta a falha de um processo por *crash*. Por outro lado, a incerteza inerente ao modelo assíncrono, no qual não são conhecidos limites máximos associados às tais latências, como, por exemplo, na Internet, torna impossível a resolução, nas mesmas hipóteses de falhas, de dados problemas básicos da computação distribuída [1].

Alguns modelos intermediários têm sido propostos, representando situações em que, por exemplo, embora o sistema não seja síncrono, parte dos componentes apresentam comportamento síncrono, ou mesmo onde em um dado momento há limites temporais associados a todo o sistema. Mesmo nestas condições, a existência de tais limites pode otimizar o desenho de algoritmos distribuídos. Tais modelos representam ambientes com características híbridas ou dinâmicas.

Em cenários atuais, como, por exemplo, os de computação em nuvem, em que recursos de diferentes sítios possam ser combinados, como em uma nuvem federada [2]. Ou seja, uma nuvem composta por um conjunto de *clusters*, cada qual possivelmente localizado em sítios distintos conectados pela Internet. Em cada *cluster* da nuvem, se pode assumir um comportamento síncrono, mas a nuvem federada não é um sistema síncrono. Pode ainda ser possível

renegociar a qualidade de serviço (*QoS*) para a comunicação entre nuvens, estabelecendo canais de comunicação através da Internet com um dado conjunto de garantias, inclusive temporais, e tais canais podem ser renegociados de acordo com a existência dinâmica de recursos na infra-estrutura subjacente.

Outro cenário atual, que reflete, por exemplo, grades de energia, ambientes industriais e *utilities*, é o de *Cyber-Physical Systems*[3]. Estes sistemas são compostos por computadores, e componentes computacionais, sensores e atuadores que monitoram e controlam processos físicos, por meio, por exemplo de laços de controle. Tais sistemas podem ser implantados inclusive em larga escala, com a planta e computador de controle distribuídos em ambientes físicos distintos. Diferentes partes de um *Cyber-Physical System* possuem diferentes características. Por exemplo, uma parte dos componentes pode estar integrada entre si por meio de uma rede de tempo real, mas parte dos componentes se comunica por meio da Internet.

Para permitir construir soluções computacionais adequadas a cenários como os dos exemplos, em que componentes podem possuir comportamentos distintos que podem ainda variar ao longo do tempo, diferentes propostas de modelos foram introduzidas em [4–8].

Em [4, 6], um modelo de sistema distribuído híbrido e dinâmico é apresentado em conjunto com um algoritmo de consenso uniforme adaptado a este ambiente. Em [5, 8], o modelo foi generalizado para que processos e canais de comunicação pudessem variar entre síncrono e assíncrono. Além disto, foi apresentado um algoritmo de comunicação em grupo capaz de lidar com esses ambientes híbridos e dinâmicos, e finalmente, em [7, 9] foi introduzido o modelo híbrido *partitioned synchronous* que requer um número menor de garantias temporais do que o modelo síncrono e onde foi provado ser possível a implementação de detectores perfeitos.

O número máximo de componentes que podem falhar em um sistema distribuído sem comprometer o progresso da computação depende intrinsecamente da qualidade temporal dos componentes (processos e canais) deste sistema. Ambientes representados por modelos temporais híbridos e dinâmicos possuem qualidade de serviço temporal dinâmica de seus componentes, de acordo com o número de componentes síncronos [6].

No desenho de algoritmos que possam atuar nestes modelos, o projetista do sistema deve exercitar os extremos: (i) propor algoritmos que mantenham suas propriedades válidas nos casos síncrono e assíncrono e (ii) que gradualmente incrementem seu desempenho de acordo com o nível de sincronia existente no ambiente. Uma etapa importante para o sucesso do desenho destes algoritmos é a avaliação de desempenho, considerando as hipóteses do modelo de sistema que caracteriza o ambiente de execução dos mesmos.

A avaliação de desempenho de sistemas distribuídos pode ser efetuada de três principais formas [10]: medição, modelo analítico e simulação. O uso de medição consiste em mensurar as variáveis de interesse em uma implementação real do sistema avaliado. O modelo analítico descreve, por meio de um modelo matemático, o comportamento de interesse do sistema avaliado; diferentes ferramentas matemáticas podem ser utilizadas para compor este modelo matemático, como, por exemplo, utilizar a teoria das filas para representar o tempo de atendimento de um

serviço, ou, redes de Petri para representar a transição de estados nos diferentes componentes de um sistema.

Ambientes homogêneos, como síncrono ou assíncrono, são compostos por componentes com mesmas características, o que facilita a implementação de ambientes reais de testes necessários para a avaliação por medição. Da mesma forma, a composição de um modelo analítico em que os componentes possuam as mesmas características, facilita a análise matemática da combinação destes componentes, como, por exemplo, na análise do tempo de serviço devido a combinação de uma sequencia de filas de serviço.

Contudo, compor um ambiente com componentes com características distintas entre si, híbrido, em ambiente de laboratório para efetuar medição pode ser custoso, envolvendo, por exemplo, a integração e configuração de um grande número de diferentes equipamentos de infraestrutura.

Mesmo no caso de modelagem analítica, em que avaliamos um modelo matemático que abstrai detalhes do sistema em avaliação, uma avaliação de um ambiente híbrido em geral poderá assumir a análise de configurações de pior e de melhor caso, e será complexa na avaliação de cenários em que esta configuração varia de forma dinâmica ao longo do tempo.

O uso de simulação permite, de forma similar a modelagem analítica, avaliar um modelo que apresenta uma abstração do sistema a ser avaliado, representando as características de maior interesse deste. Porém, esta avaliação não requer uma análise matemática detalhada de todo o conjunto, em face que os componentes do sistema podem ser expressos por modelos mais simples e a simulação representa a interação entre estes. A simulação assim é baseada na composição do sistema por meio destes componentes e na execução deste em diferentes cenários de interesse, de modo similar à medição.

A combinação de ambas estas características da simulação nos permite uma flexibilidade de uso adequada para a avaliação de sistemas distribuídos híbridos e dinâmicos, apresentando menor custo de implementação que a medição, e requerendo menor complexidade de análise que o modelo analítico.

Simuladores genéricos de eventos discretos, como o SMPL [11], ou mesmo simuladores de redes de computadores, como NS-2 [12], NS-3 [13] e Omnet++ [14], não apresentam o nível de abstração adequado para a simulação de sistemas distribuídos.

Mesmo ambientes adequados a simulação de ambientes distribuídos, como Neko [15] e Simmcast [16], não possuem todas as carecterísticas desejáveis para a simulação de sistemas distribuídos em cenários híbridos e dinâmicos. Esta simulação deve permitir a avaliação de algoritmos distribuídos por meio da execução dos processos com acesso a relógio físico local a este, e a comunicação fim-a-fim entre os processos, por meio de canais de comunicação.

Dentre as características, destacamos a composição de cenários heterogêneos (híbridos) e dinâmicos, o comportamento de tempo real, com agendamento de tarefas e sua execução sob uma janela temporal restrita, e a associação dos distintos modelos de falhas aos componentes do sistema distribuído, processos e canais de comunicação.

1.1 OBJETIVOS, CONTRIBUIÇÕES E ESTRUTURA DA TESE

Esta motivação define o objetivo da tese: a avaliação de desempenho de sistemas distribuídos híbridos e dinâmicos, por meio da proposta de um *framework* de simulação adequado a estes ambientes. Tal proposta permite avaliar os diferentes modelos de sistemas distribuídos, desde os casos clássicos síncrono e assíncrono, até os sistemas parcialmente síncronos e sistemas ditos híbridos ou dinâmicos, em que as características dos componentes (processos e canais de comunicação) podem variar no espaço ou no tempo.

Esta tese apresenta como contribuição principal o *framework* de simulação **HDDSS** (*Hybrid and Dynamic Distributed System Simulator*). O **HDDSS** é capaz de simular sistemas distribuídos híbridos e dinâmicos, bem como sistemas distribuídos convencionais, como síncronos, assíncronos e parcialmente síncronos, e caracterizar o comportamento destes componentes e diferentes hipóteses para modelos de falhas associados a estes por meio de funções determinísticas ou probabilísticas.

Este *framework* é baseado em um modelo de simulação com um nível de abstração adequado para a avaliação de desempenho de algoritmos em sistemas distribuídos, caracterizando o comportamento fim-a-fim, a evolução dos atributos ao longo do tempo e a probabilidade de ocorrência de eventos relacionados ao modelo de falhas.

Como contribuições adicionais, destaca-se ainda a avaliação e o desenvolvimento de protocolos para cenários de comunicação em grupo e replicação bizantina.

Os algoritmos de comunicação em grupo foram desenvolvidos a partir do mecanismo *timed causal blocks* [5]. No primeiro cenário, o protocolo proposto se adapta a um sistema distribuído com diferentes níveis de qualidade de serviço de seus componentes, e que provê aos processos do algoritmo distribuído informação sobre esta qualidade de serviço, caracterizando um sistema *self-aware*: o algoritmo utiliza-se da informação disponível no ambiente e adapta-se de forma dinâmica a sua execução, apresentando execução otimizada de acordo com o nível de sincronia existente em seus componentes.

Em um segundo cenário, consideramos o caso de construção de um protocolo de comunicação em grupo auto-gerenciável: a partir dos mecanismos de comunicação em grupo utilizados anteriormente, propomos o ajuste de parâmetro de configuração do protocolo sensível a carga de trabalho do ambiente, de modo a se adequar ao nível de consumo de recursos desejado por uma aplicação. A abordagem de construção de algoritmos auto-gerenciáveis é contribuição de outro trabalho de tese apresentado em [17], o qual modelou os mecanismos de auto-ajuste deste protocolo. Esta tese contribuiu no desenvolvimento do mecanismo básico do protocolo, os pontos de observação e atuação deste pelo mecanismo auto-gerenciável e, em especial, na sua avaliação de desempenho.

Por fim, o protocolo clássico de replicação tolerante a falhas bizantinas PBFT apresentado em [18] é estudado e a proposta adaptativa de [17] aplicada, de modo que a definição de um conjunto de parâmetros operacionais do protocolo se ajusta automaticamente de acordo com o

monitoramento do ambiente ao longo da execução. A contribuição específica desta tese a este trabalho conjunto foi na avaliação de desempenho do protocolo adaptativo em comparativo ao clássico PBFT.

A estrutura deste documento está organizado da seguinte forma: o Capítulo 2 apresenta modelos de sistemas distribuídos; o Capítulo 3 discute a avaliação por simulação de algoritmos que executem em sistemas distribuídos e a proposta de modelo de simulação; o Capítulo 4 apresenta o *framework* de simulação proposto; o Capítulo 5 apresenta o desenvolvimento e avaliação realizados na construção de protocolos desenvolvidos para os problemas de comunicação em grupo e de replicação bizantina; e, por fim, o Capítulo 6 tece considerações finais.

Capítulo 2

MODELOS DE SISTEMAS DISTRIBUÍDOS

Um sistema distribuído é um conjunto de processos, os quais podem estar localizados em diferentes nós processadores geograficamente distribuídos, que cooperam entre si para prover um serviço computacional distribuído [19]. Para obter progresso na computação de tal serviço, cada processo de um sistema distribuído executa um algoritmo local que, por meio de mecanismos de interação com os demais processos, formam a computação distribuída.

Dada a localização em diferentes nós processadores, sistemas distribuídos não dispõem de compartilhamento de memória, logo o mecanismo de interação utilizado para comunicação entre processos é a troca de mensagens através da infraestrutura subjacente de rede de comunicação. Em face das particularidades desta infraestrutura, a interação entre os processos está sujeita a latências, falhas de comunicação e incertezas.

Os cenários de interesse desta tese são os de sistemas distribuídos, ou seja, aqueles em que a comunicação entre os processos ocorre por meio de uma infraestrutura subjacente de comunicação, como, por exemplo, uma rede de computadores, e nos quais os processos não possuem acesso a um relógio único e global. Desta forma, esta tese utiliza o termo sistema distribuído na sua forma estrita, se referenciando aos cenários de sistemas em que os processos são fracamente acoplados – diferentes daqueles encontrados em ambientes centralizados ou muitprocessados.

Mesmo com esta restrição ao cenário de interesse, diferentes atributos do ambiente de execução definem um conjunto de condições que os algoritmos distribuídos devem observar para o sucesso de sua execução. Por exemplo, quais os tipos de falhas a que estão sujeitos os componentes do sistema? Os algoritmos podem utilizar os relógios das máquinas locais como informação temporal? Qual a precisão entre estes relógios? Há limites temporais para o recebimento de uma resposta a uma requisição remota?

As respostas a estas perguntas variam de acordo com o ambiente de execução. Mesmo para um algoritmo com um código pequeno e simples, o fato de partes deste algoritmo serem

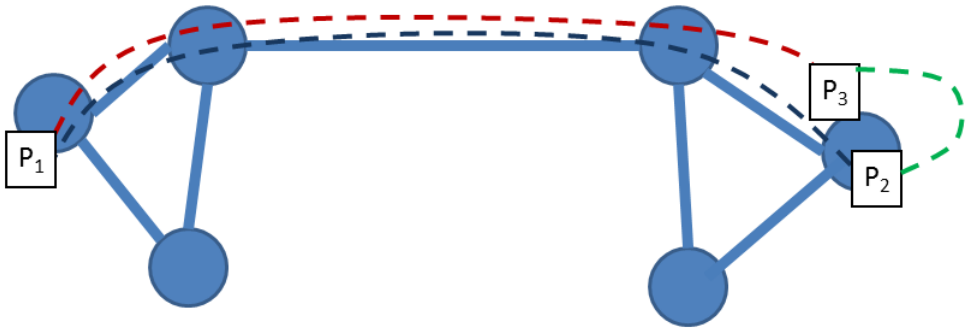


Figura 2.1. Um exemplo de sistema distribuído composto por 3 processos.

executadas em paralelo em diferentes processos em nós processadores distintos, com passos intercalados entre si de uma forma não determinística[1], implica em diferentes traços de execução, mesmo para um mesmo conjunto de entradas[2] [19].

De modo a melhorar a compreensão destes ambientes distribuídos e possibilitar o projeto correto de algoritmos distribuídos, as condições que definem os atributos destes ambientes de execução podem ser representadas por meio de modelos. Um modelo abstrai detalhes da infraestrutura de execução e comunicação subjacente ao sistema distribuído. Modelos devem ser genéricos o suficiente para representar um conjunto de ambientes de execução em que possam ser aplicados. Modelos também podem representar de forma adequada condições relevantes ao desenho do algoritmo distribuído.

2.1 DOS MODELOS DE SISTEMAS DISTRIBUÍDOS

Um sistema distribuído pode ser modelado por um grafo $DS(\Pi, \chi)$. O grafo DS é composto por um conjunto finito Π de $n > 1$ processos, denominado, $\Pi = \{p_1, p_2,...,p_n\}$, e um conjunto finito de canais $\chi = \{(p_i, p_j)|p_i, p_j in \Pi\}$. Vértices representam processos e arestas representam canais de comunicação. Cada canal de comunicação é uma abstração de toda a infraestrutura de comunicação subjacente entre dois processos e modela a interação entre os processos associados por meio de troca de mensagens. O grafo DS é completo, ou seja, há um canal bidirecional (p_i, p_j) conectando cada par de processos de Π. A figura 2.1 apresenta um exemplo de sistema distribuído com 3 processos, onde as linhas tracejadas são os canais de comunicação estabelecidos entre fim-a-fim entre os processos sob a infraestrutura subjacente de processamento e comunicação.

Cada processo p_i executa um algoritmo local A_i, parte do algoritmo distribuído A associado ao sistema distribuído DS. A execução de A_i define a execução local de uma sequência de passos. Esta sequência de passos considera a entrada de dados local e a interação com os

[1] Por exemplo, o atraso da rede de computadores no exato momento do envio de uma requisição pode determinar de forma variável o tempo de ativação de uma parte do algoritmo executada de forma remota.

[2] Ao longo desta tese, discutiremos que tais condições que alteram o traço de execução, como o atraso da rede, também podem ser considerados parte da entrada, em uma definição ampla.

demais processos de Π (troca de mensagens). Cada passo representa uma mudança de estado local.

Assumimos a existência de um tempo global, discreto, o qual não é conhecido dos processos. Cada processo executa em um nó computacional e pode ter acesso a um relógio físico local, se existir. Cada processo p_i mantém seu estado local $x_i(t)$ no instante de tempo global t. Este estado local evolui de acordo com a ocorrência de um histórico de eventos H_i percebidos pelo processo p_i, como um envio ou recebimento de uma mensagem m ($send_i(m)$ ou $receive_i(m)$), uma computação local ou uma entrada de dados local.

O modelo apresentado até então é genérico o suficiente para representar a ampla gama de sistemas distribuídos de interesse desta tese. Assumimos que o número n de processos é previamente conhecido, de modo que este modelo não representa sistemas com alto índice de *churn*[3], em que o número total de processos não é previamente conhecido, salvo se este número fosse grande o suficiente para representar o total de processos do sistema ao longo da execução.

Os modelos de sistemas distribuídos conhecidos na literatura diferem-se, dentre outros aspectos, no comportamento temporal de seus componentes, o que define o modelo temporal, e, na forma da ocorrência de falhas, ou seja, se e como os componentes desviam ou não do comportamento especificado, dito modelo de falhas.

2.1.1 Do modelo de falhas

Sistemas computacionais são compostos por componentes, os quais podem desviar do comportamento especificado. Se um componente do sistema desvia do comportamento especificado, têm-se uma falha do sistema computacional. A falha pode implicar em alteração do estado do sistema, ocasionando o erro. O erro pode ocasionar um defeito, ou seja o serviço provido pelo sistema computacional apresenta resultado diferente do comportamento especificado. O defeito de um sistema computacional pode ser visto como uma falha de um outro sistema que o utilize como componente. Os termos falha, erro e defeito, ora descritos são a tradução da nomenclatura respectiva *fault*, *error* e *failure*, apresentada em [21]. Por exemplo, suponha na execução de um programa em um sistema computacional, a ocorrência de uma falha em uma placa de memória pode implicar no erro de leitura do valor de uma dada variável no programa, e este erro no cálculo de um resultado incorreto da computação (defeito).

As hipóteses de falhas de componentes, processos e canais, são importantes para caracterizar a execução do sistema distribuído e desenvolver mecanismos de tolerância a falhas. Tais hipóteses são determinadas por meio do modelo de falhas. Ao prover serviços computacionais por meio de um sistema distribuído, tais serviços podem ser desenhados de modo a prevenir ou tolerar falhas – utilizando, por exemplo, replicação de processos–, de modo a assegurar a execução dos serviços mesmo na presença de até um dado subconjunto de componentes falhos

[3] O termo *churn* descreve o fenômeno de rotatividade de processos no sistema, i.e. chegada e saída de processos do conjunto de processos do sistema. Em condições de *churn*, um algoritmo distribuído deve ser projetado de forma distinta de cenários com um conjunto fixo de processos [20].

(processos ou canais de comunicação).

Um processo correto p_i – ou seja, que não falha ao longo da execução – executa um algoritmo pré-determinado em função da ocorrência do histórico de eventos H_i. Um canal bidirecional (p_i, p_j) correto, também dito confiável, permite que o evento $send_i(m) \in H_i$ de envio de uma mensagem m no tempo global t_1 em p_i, se reflita no evento $receive_j(m) \in H_j$ de recebimento de uma mensagem m no tempo global $t_2 > t_1$ em p_j. O modelo de falhas associado a um componente do sistema distribuído *DS* define as hipóteses em que e como o componente pode desviar da especificação correta ao longo da execução.

Em sistemas reais, processos podem falhar de diferentes modos. A forma como é equipada a infraestrutura subjacente para execução dos processos, como, por exemplo, nós processadores, sistema operacional, conjunto de APIs etc., e a infraestrutura subjacente de comunicação, como, por exemplo, rede de computadores, define quão fortes são as hipóteses sobre a robustez do sistema à ocorrência de falhas em processos e canais de comunicação.

Dentre as hipóteses de falhas de processos da literatura destacamos as falhas arbitrárias, relacionadas a execução deliberada fora da especificação do algoritmo. Por exemplo, um processo comprometido, ou executando sobre um sistema operacional comprometido por um atacante malicioso, pode prejudicar a execução do algoritmo – como, por exemplo, ao parar sua execução ou alterar mensagens enviadas ou recebidas. A leitura incorreta de uma variável de memória também pode ocasionar resultados inesperados ao longo da execução do algoritmo. Falhas arbitrárias são também conhecidas como falhas bizantinas em referência ao problema apresentado em [22].

Podemos equipar a infraestrutura subjacente, protegendo o ambiente de comprometimento de segurança e de falhas de hardware, como por exemplo, utilizando paridade no subsistema de memória. Desta forma, pode-se reduzir substancialmente a vulnerabilidade do sistema a um conjunto de falhas, tornando mais fortes as hipóteses do modelo. Ainda assim, este ambiente pode estar sujeito à parada de execução, devido, por exemplo, a queda de energia elétrica ou ao defeito do sistema operacional em face a uma falha não recuperável. A não execução do próximo passo computacional é dita omissão e quando o sistema falha por omissão sem se recuperar, temos a chamada falha por colapso (*crash*). Se for possível o reinício do componente defeituoso e que o processo continue sua execução após o reinício em um ponto de execução anterior ao do colapso, o modelo de falhas é dito *crash-recovery*.

Os algoritmos apresentados nesta tese poderão fazer referência às hipóteses de falhas de processos associadas aos modelos de falhas por *crash*, omissão, *crash-recovery* e bizantino. Em geral, os algoritmos assumem condição de execução tolerando um dado número k de processos falhos de acordo com o modelo.

Canais de comunicação podem também falhar de forma arbitrária ou bizantina: mensagens podem ser perdidas ao longo da rede ou a infraestrutura subjacente de comunicação pode ser comprometida. Por exemplo, se um nó de roteamento intermediário for comprometido, mensagens podem ser alteradas ao serem roteadas, ou mesmo descartadas deliberadamente, ou ainda

criadas mensagens em nome de um processo pertencente à computação distribuída. Se utilizarmos mecanismos de segurança como assinatura digital e estabelecimento de criptografia fim-a-fim na comunicação entre processos, ainda assim, é possível que mensagens sejam descartadas, em face, por exemplo, do congestionamento dos nós roteadores intermediários. Assim, canais de comunicação podem falhar por omissão.

Mecanismos de verificação fim-a-fim com retransmissão podem ser empregados, e os canais podem ser quasi-confiáveis [23]: se os processos do canal mantém-se corretos, mesmo em face à omissão, a mensagem será retransmitida pelo remetente o quanto for necessário e será recebida a termo pelo destinatário. Podemos ainda assumir que se os canais forem quasi-confiáveis, eles simulam o comportamento de canais confiáveis, onde a eventual falha do canal é reproduzida como falha do processo remetente ou destinatário imediatamente antes do envio ou da recepção da mensagem. Canais quasi-confiáveis são implementáveis sobre canais *fair-lossy* por meio de retransmissão de mensagens [24]. Um canal de comunicação *fair-lossy* não cria ou duplica mensagens e assume que se um processo p enviar um infinito número de mensagens ao processo q, o mesmo receberá um infinito número de mensagens de p.

Os algoritmos apresentados na tese podem fazer referência às hipóteses de falhas de canais associadas aos modelos bizantino, por omissão, e (quasi-)confiáveis.

2.1.2 Do modelo temporal

No projeto de algoritmos distribuídos, o modelo temporal apresenta quais hipóteses são asseguradas pelo ambiente distribuído, quanto a limites temporais para a execução de passos computacionais por cada processo e para a troca de mensagens. Tais propriedades determinam a forma de interação entre processos, permitindo, por exemplo, pontos de sincronização entre os processos, que auxiliam na determinação do traço global da execução do algoritmo distribuído.

O desenho de algoritmos distribuídos pode utilizar as hipóteses temporais asseguradas pelo ambiente distribuído na construção de mecanismos de tolerância a falhas. De fato, a habilidade de resolver certos problemas de tolerância a falhas nos sistemas distribuídos está fortemente relacionada às hipóteses garantidas pelo modelo de sistema escolhido. No curso da pesquisa em sistemas distribuídos, diferentes modelos temporais foram propostos, destacando-se o síncrono e o assíncrono, extremos do comportamento temporal.

No modelo assíncrono, de hipóteses mais fracas, assumimos que os componentes do sistema distribuído, processos e canais de comunicação, executam passos de forma arbitrária, sem limites temporais conhecidos. A velocidade relativa da execução[4] entre dois processos não é conhecida. Ou seja, não é possível que os algoritmos assumam quaisquer hipotéses baseadas no tempo de execução dos passos computacionais ou do tempo gasto para troca de mensagens.

Desde que o modelo assíncrono apresente as hipóteses mais fracas, algoritmos desenhados para tais sistemas são mais genéricos e portáveis, podendo ser utilizados em quaisquer am-

[4]Razão entre o processo mais rápido e o mais lento do sistema distribuído.

bientes distribuídos. Contudo, estes algoritmos podem não apresentar bom desempenho em comparação a algoritmos que se utilizem de garantias temporais de ambientes distribuídos com hipóteses mais fortes.

Por outro lado, o modelo síncrono é o de hipóteses mais fortes. Assumimos a definição de modelo síncrono de [19]: os componentes do sistema realizam passos de forma sincronizada, isto é, a execução ocorre em rodadas síncronas. Esta descrição de modelo embora não seja representativa, permite o desenho de algoritmos distribuídos que utilizem da sincronia entre os processos para obter progresso na computação distribuída. A velocidade relativa de execução entre dois processos, dita ϕ, é igual a um. Ou seja, processos executam à mesma velocidade, avançando a cada rodada de forma síncrona.

O comportamento temporal dos processos de acordo com o modelo síncrono apresentado por [19] é de difícil implementação na sua forma estrita em um ambiente real. Se processos executam de forma distribuída em diferentes nós processadores, cada nó possui acesso a um relógio físico local. Estes relógios são compostos por cristais de quartzo que, face às imperfeições do processo de construção, possuem frequencias de oscilação distintas, computando um mesmo período temporal de forma distinta entre si. Desta forma, o mesmo passo computacional executado por processos distintos é executado em janelas temporais distintas. Ou seja, temos a velocidade relativa entre processos $\phi \neq 1$. Processos dispostos em nós processadores distintos avançam de forma distinta ao longo do tempo.

Considerando a imprecisão do processo de fabricação dos relógios de hardware reais, o desvio de cada relógio físico em relação ao tempo real é limitado, e assim a velocidade relativa ϕ é limitada. Desta forma, dada a taxa de desvio máxima ρ de um relógio correto em relação ao tempo real, e a existência de um relógio físico H_i correto associado ao nó de execução do processo p_i, o intervalo de tempo real $T - T'$ é mensurado pelo relógio físico H_i pelo intervalo temporal $H_i(T) - H_i(T')$, onde $(1-\rho)(T-T') \leq H_i(T) - H_i(T') \leq (1+\rho)(T-T')$.

Assumindo que o ambiente de execução de p_i possui hipóteses temporais fortes, como, por exemplo, sistema operacional de tempo real, cada passo computacional invocado pelo processo p_i no tempo $H_i(T)$ mensurado por seu relógio local será executado até o limite temporal $H_i(T) + \varepsilon$, onde ε é um valor positivo conhecido.

Ainda, a comunicação inter-processos ocorre por meio de troca de mensagens. Considere que o ambiente provê garantias temporais na comunicação, como, por exemplo, em redes *Token Ring* [25]–, de modo que há um limite temporal Δ conhecido para o envio/recepção de uma mensagem de um processo a outro. Em geral, tais limites de troca de mensagens são significativamente superiores à janela de execução de um passo computacional. Na ausência de falhas, o evento $send_i(m) \in H_i$ de envio de uma mensagem m no tempo global t_1 em p_i, se reflete no evento $receive_i(m) \in H_j$ de recebimento de uma mensagem m no tempo global $t_2 > t_1$ em p_j, de modo que o limite temporal do recebimento da mensagem é $t_2 \leq t_1 + \Delta$.

Estas premissas definem uma configuração síncrona [26]: canais de comunicação possuem limites temporais conhecidos e definidos e processos, velocidade relativa ϕ de execução entre

processos limitada.

Tal modelo com velocidade relativa entre processos limitada e a existência dos limites temporais na comunicação, permite a ressincronização periódica dos relógios associados aos nós de processamento de cada processo – mantendo-se precisão relativa entre tais relógios–, e possibilita simular[5] o modelo de sistema síncrono apresentado em [19], executando passos computacionais por processo associado a rodadas síncronas.

Ao longo desta tese, assumiremos que sistemas síncronos são os que atendem a esta configuração. Esta abordagem é coerente com a literatura que apresenta modelos com mesmas hipóteses para caracterizarem ambientes síncronos reais, como em [28].

Os modelos extremos apresentados, assíncrono e síncrono, implementam de forma distinta a tolerância a falhas para problemas de computação distribuída. Por exemplo, a difusão confiável – na presença de canais confiáveis e de processos que podem falhar por colapso–, pode ser resolvida em ambos modelos [19]. Contudo, o consenso distribuído [19], para as mesmas hipóteses do modelo de falhas, pode ser resolvido no modelo síncrono, mas não no modelo assíncrono [1].

Na ausência de memória compartilhada, e com isso, na impossibilidade dos processos compartilharem o estado global atual do sistema [29], o consenso distribuído permite aos processos concordarem no valor de uma computação. Desta forma, o consenso distribuído torna-se um dos principais blocos algorítmicos utilizados no desenho de algoritmos distribuídos. E desta forma, é um balizador de interesse relevante na avaliação de quão fortes são as hipóteses do modelo, temporal e de falhas, do sistema distribuído.

Considerando o problema do consenso distribuído, dada a hipótese que processos falhem por colapso e canais são confiáveis, detectar os processos falhos pode ser parte importante da solução. Em [30], apresenta-se o uso de componentes denominados detectores de defeitos. Um detector de defeito é um componente que monitora e suspeita de processos falhos. A precisão e completude dos detectores de defeitos (categorizados em diferentes classes) em face das hipóteses temporais existentes no modelo do sistema são apresentadas em [30]. Por exemplo, dadas as hipóteses fortes do modelo síncrono, o consenso síncrono pode utilizar detectores de defeitos perfeitos (classe $\mathcal{P}$) e obter fortes completude e precisão na detecção de processos falhos por colapso, possibilitando terminar de forma confiável a computação.

2.1.2.1 Dos modelos parcialmente síncronos

No desenho de algoritmos distribuídos, as hipóteses fracas do modelo assíncrono oferecem poucas garantias para o projetista face à presença de falhas. Contudo, obter as garantias necessárias para as fortes hipóteses do modelo síncrono pode não ser possível. Em face disto, diversos ambientes ditos parcialmente síncronos foram exercitados, em que propriedades mais

[5] O termo simular neste contexto difere de seu significado utilizado ao longo desta tese. O contexto se refere a ação de simulação de modelos: um modelo de sistema simula o comportamento de outro, ao apresentar as características e garantias deste [27].

fortes que as existentes em ambientes assíncronos permitem um comportamento síncrono adequado ao longo do tempo em todo o sistema ou em parte deste de modo que a execução de algoritmos como o consenso distribuído seja possível [26].

Discutimos inicialmente configurações com processos síncronos ($\phi \neq 1$, porém limitado), em que, mesmo que a comunicação ocorra de forma síncrona, os processos consultam relógios que se desviam ao longo do tempo e periodicamente podem ser resincronizados. Este modelo representa comportamento observado em cenários reais, mas com garantias estritas para o sistema de comunicação (síncrono).

Manter tais garantias temporais estritas em toda infraestrutura de comunicação, ou seja, canais de comunicação síncronos, ao longo de todo o tempo de execução de comunicação pode ser uma premissa muito forte em um conjunto significativo de cenários. Uma hipótese menos forte, apresentada em [26], é a da existência de um limite temporal Δ para a troca de mensagens por meio de canais de comunicação apenas a partir de um instante de tempo em que o sistema de comunicação torna-se estável, dito *Global Estabilization Time – GST*. Tal estabilidade deve se manter ao longo da execução da computação distribuída.

Desta forma, assumimos que os canais de comunicação apresentam comportamento síncrono, com limite temporal Δ conhecido, ao longo de um período de estabilidade do sistema iniciado no instante GST até ao menos $GST + L$, onde L é o tempo necessário para a conclusão da computação distribuída.

A existência de um instante a partir do qual limites temporais são conhecidos permite construir algoritmos que tentem inferir o início do período de estabilidade e utilizar as garantias temporais existentes na convergência da computação distribuída. Por exemplo, o modelo parcialmente síncrono ora apresentado, se equipado com detectores de defeitos da classe $\Diamond\mathcal{S}$ permite a terminação do algoritmo de consenso distribuído na presença de canais confiáveis e de processos que podem falhar por colapso [30].

Um outro exemplo é o do modelo temporizado assíncrono[31]: neste modelo, o ambiente distribuído, mesmo sujeito a períodos de instabilidade, possui longos períodos de estabilidade no comportamento temporal, o que, de forma similar à hipótese GST, garante que serviços providos pelo ambiente progridam na computação distribuída.

Ainda que não seja possível garantir as hipóteses fortes de um ambiente síncrono, um sistema pode ser composto de um plano de controle síncrono associado ao plano de aplicação, que utiliza as garantias existentes no ambiente distribuído. Esta é a proposta da Base Computacional Temporizada [32] (*Timely Computing Base – TCB*): um sistema não síncrono é equipado com componentes síncronas, que formam uma *spanning tree* de comunicação síncrona, ou *wormholes*, tais componentes formam um plano de controle que provê serviços aos processos distribuídos. O uso de infraestrutura de execução com suporte a tempo real, como, por exemplo, sistema operacional de tempo real, em cada nó processador, em conjunto com a possibilidade de priorização de trafego e reserva parcial de recursos ao longo da rede é uma forma possível de implantação deste plano de controle.

Embora o modelo *TCB* assuma que há componentes com diferentes qualidades de serviço (componentes síncronas que compõem o plano de controle e componentes assíncronas que compõem o plano de aplicação), o modelo é estático, uma vez que não há adaptação a possibilidade de mudança da qualidade de serviço das componentes do sistema.

De forma similar, em [33] um plano de controle que possibilita a execução do consenso distribuído é obtido por meio de uma *spanning tree* síncrona. Este trabalho evolui em [4, 6] com o modelo *HA*, possibilitando a mudança da qualidade de serviço dos componentes do sistema ao longo da execução: canais de comunicação variam o comportamento entre assíncrono e síncrono.

Ao invés de se basear em um *wormhole* ou em uma *spanning tree* síncrona, o modelo *HA* provê uma semântica segura para as aplicações sobre o estado dos processos e da comunicação entre estes. O modelo assume que o ambiente distribuído permite monitorar a qualidade de serviço negociada para os canais de comunicação, por meio, por exemplo, de mecanismos de *QoS* como *DiffServ* [34], e a definição de garantias temporais de execução de processos, por meio, por exemplo, do suporte de sistemas operacionais de tempo real.

O modelo *HA* assume as hipóteses de falhas de processos por *crash* e de canais confiáveis, e a partir da observação acerca da modificação da qualidade de serviço dos canais de comunicação, bem como da detecção de defeitos, permite a cada processo classificar os demais processos do sistema nos sub-conjuntos $live$, $down$ e $uncertain$ – os quais expressam uma semântica sobre o estado de operação dos processos.

Desta forma, um conjunto de regras mapeia a construção local dos sub-conjuntos $live_i$, $down_i$ e $uncertain_i$ para cada processo p_i, conforme a observação do ambiente e a percepção anterior destes sub-conjuntos. Dentre as regras, destaca-se a hipótese que apenas degradação e não melhorias na qualidade de serviço de canais de comunicação são percebidas pelos processos durante a execução do algoritmo distribuído. Ou seja, o conjunto $live_i$ mantém-se ou é monotonicamente decrescente em tempo de execução (processos anteriormente em $live_i$ podem ser percebidos em $uncertain_i$ por p_i se a qualidade de serviço decai). Este conjunto de regras de transição de estados fornece uma semântica conservadora, mas segura, a qual permite a execução do consenso distribuído, adaptando-se ao longo da execução o quórum, de modo que este contemple todos processos em $live$ e uma maioria de processos em $uncertain$.

Em [7, 9] foi introduzido o modelo híbrido *partitioned synchronous*. Neste modelo define-se partições síncronas: dois ou mais processos síncronos conectados entre si por meio de canais síncronos. A existência de partições síncronas mesmo que conexas por meio de canais de comunicação assíncronos, requer um número menor de garantias temporais do que o modelo síncrono, e permite a execução de serviços temporizados em cada partição, provendo informações de interesse para o sistema como um todo. Por exemplo, se todos processos pertencem a partições síncronas, e assumindo que ao menos um processo seja correto em cada partição, é possível a implantação de detectores de defeitos perfeitos que permitem a execução do consenso distribuído. Mesmo que processos não estejam em qualquer das componentes síncronas,

pode-se tirar proveito das partições síncronas existentes para melhorar a robustez das aplicações de tolerância a falhas.

Vale salientar que a implementação de detectores perfeitos no modelo *partitioned synchronous* não requer a existência de um *wormhole* síncrono [32] ou *spanning tree* síncrona [33], onde seria possível implementar ações síncronas globais em todos os processos, como sincronização interna de relógios.

2.1.3 Um modelo genérico de comportamento temporal híbrido e dinâmico

A partir dos avanços da literatura na proposta e construção de modelos de sistema distribuídos parcialmente síncronos, verifica-se que os diferentes modelos apresentam diferentes hipóteses para as condições de sincronia dos componentes do sistema: variando ao longo do tempo (dinâmicos, como, por exemplo, a hipótese *GST*) ou ao longo do espaço (híbridos, como por exemplo, partições síncronas do modelo *partitioned synchronous*).

As possibilidades de (re)configuração de recursos na infraestrutura de comunicação, como, por exemplo, *DiffServ* [34], de alocação de garantias temporais na execução, como, por exemplo, por meio de sistemas operacionais de tempo real, e mesmo da existência destes recursos ao longo de todo o tempo de execução e em toda a infraestrutura subjacente de execução e de comunicação definem diferentes possibilidades de ambientes distribuídos híbridos e dinâmicos.

Sistemas distribuídos que expressem comportamento temporal de forma híbrida e dinâmica podem ser caracterizados a partir do modelo apresentado em [5], definindo um modelo genérico. Esta tese apresenta este modelo genérico de sistema distribuído e aborda a avaliação de desempenho de algoritmos que executam sob tais condições. Tal modelo generaliza a percepção da qualidade de serviço temporal expressa no modelo HA [6] e no modelo *partitioned synchronous* [7]. Esta generalização permite expressar tanto o modelo síncrono e o modelo assíncrono quanto os diferentes modelos parcialmente síncronos como instâncias deste modelo genérico em que um conjunto de hipóteses adicionais é mantido.

Um sistema distribuído híbrido e dinâmico, caracteriza-se pelo sistema distribuído composto por um grafo $DS(\Pi, \chi)$, com o algoritmo A_i e o estado x_i locais a cada processo p_i – de acordo com o apresentado no início da seção 2.1.

De modo a representar as diferentes configurações possíveis para o comportamento temporal, assumimos que cada componenente de DS (processo $p_i \in \Pi$ ou cada canal $(p_i, p_j) \in \chi$) possui um atributo de comportamento temporal, que define a capacidade de sincronia deste componente.

De forma similar a [6], pode-se definir este atributo como a qualidade de serviço (QoS) do comportamento temporal do componente distribuído. Podemos representar esta qualidade de serviço pela função QoS, definida da forma:

$$QoS : \Pi \cup \chi \times t \rightarrow \{timely, untimely\} \tag{2.1}$$

Esta função mapeia o comportamento temporal de cada componente de *DS* para cada instante de tempo global t.

Um processo é dito *untimely* se não fornece quaisquer garantias temporais na execução de passos, ou seja, sua velocidade relativa aos demais processos não é conhecida e limitada. Assumimos que processos síncronos são *timely*: processos *timely* guardam entre si uma velocidade relativa ϕ limitada, possuem acesso a relógios de hardware local que se desviam no máximo de ρ.

A execução de cada passo computacional invocado pelo processo p_i no tempo t mensurado por seu relógio local ocorre em até o limite temporal $t + \varepsilon$, com ε limitado. No caso ideal, quando os processos são estritamente síncronos (conforme [19]), $\phi = 1$ e $\rho = 0$.

Da mesma forma, um canal de comunicação é dito *untimely* se não se conhecem garantias temporais no tempo de envio de mensagens por este canal. Assumimos que canais de comunicação são *timely* se o limite temporal do recebimento da mensagem enviada no tempo global t_1 é $t_2 \leq t_1 + \Delta$, com Δ limitado e conhecido.

O sistema distribuído *DS* é dito híbrido e dinâmico se seus componentes podem ser, de modo distinto, *timely* ou *untimely*, e se esse comportamento variar ao longo do tempo, refletindo, por exemplo, a renegociação da *QoS* na infraestrutura subjacente de rede ou no escalonamento de tarefas em um sistema operacional.

Podemos caracterizar propriedades do sistema que definem como os componentes de *DS* mantém o seu comportamento temporal entre *timely* ou *untimely* ao longo da execução. Um modelo de sistema distribuído mais específico, que representa *DS* pode ser definido a partir do conjunto de propriedades que se mantém em *DS* na execução. Desta forma, propriedades definem os modelos de sistema distribuído específicos ora apresentados.

Na construção de um sistema distribuído em que os algoritmos alterem sua execução em face dos componentes com características *timely* e *untimely* do sistema, é necessário perceber a qualidade de serviço destes componentes. Desta forma, supondo que é possível construir mecanismos de monitoramento da qualidade de serviço da infraestrutura subjacente aos componentes do sistema distribuído, podemos obter um estimador $\widehat{QoS}$ da função *QoS*. Se houver mudança da qualidade de serviço em tempo de execução, $\widehat{QoS}$ pode fornecer a dinâmica desta variação, permitindo ao ambiente tornar-se cônscio de seu estado (self-awareness) [35].

Podemos definir a hipótese de observação de *QoS* como uma propriedade que caracteriza a existência de um óraculo $\widehat{QoS}$ que associa a qualidade de serviço percebida *timely* ou *untimely* a cada componente do sistema. Este óraculo permite a construção de modelos de sistema distribuídos, como o modelo HA [4, 6], em que mesmo em face das características híbridas e dinâmicas do ambiente, é possível o desenho de algoritmos, como o consenso distribuído, que se adaptam de acordo com a percepção do ambiente.

2.1.4 Dos sistemas *self-aware*

Diferentes cenários permitem a alocação de recursos e definição de níveis de serviços de forma apenas parcial, como, por exemplo, no cenário de nuvens federadas [2], em que a possibilidade de alocação de recursos interdomínios pode não ser possível, ou em sistemas computacionais úbiquos e pervarsivos, como os *Cyber-Physical Systems* [3], em que dispositivos distribuídos podem, por exemplo, utilizar laço de controle computacional para a auto-regulação do sistema, tendo que observar, total ou parcialmente, requisitos temporais de tempo real, os quais podem não ser assegurados em todo o ambiente se o mesmo estiver distribuído ao longo de redes como, por exemplo, a Internet.

Algoritmos adaptados a estes cenários podem se basear no modelo genérico que expressa o comportamento de sistemas distribuídos híbridos e dinâmicos, bem como adaptar-se a detecção de propriedades que indiquem hipóteses mais fortes de sincronia ao longo do tempo de execução, como as dos modelos parcialmente síncronos apresentados. Isto é particularmente possível se os sistemas forem equipados com um estimador $\widehat{QoS}$, o que os torna sistemas *self-aware* [35, 36]. Um sistema distribuído *self-aware* é cônscio da *QoS* oferecida pelos seus componentes, podendo prover informação parcial desta *QoS* para os processos que o compõem. Esta possibilidade de detecção dinâmica da *QoS* é uma hipótese adicional ao modelo, que se mantém de acordo com a existência de uma infraestrutura subjacente de monitoramento.

Esta percepção *self-aware* é utilizada no desenho de alguns dos algoritmos distribuídos avaliados nesta tese e utilizados em cenários que permitam o auto-ajuste face a tais facilidades de percepção da configuração do ambiente. Uma breve discussão sobre sistemas distribuídos *self-aware* é apresentada em [36].

2.2 CONCLUSÕES

Este capítulo apresentou de forma breve modelos para sistemas distribuídos, caracterizando as hipóteses de falhas e as de comportamento temporal. Modelos para o comportamento temporal são apresentados a partir dos comportamentos extremos, síncrono e assíncrono. Então, apresentamos variações entre estes comportamentos, demostrando diferentes modelos da literatura em que o comportamento temporal varia ao longo do tempo ou do espaço. Sistemas distribuídos sujeitos a tais variações caracterizam-se como híbridos e dinâmicos, e representam os cenários de interesse desta tese.

A partir das propostas existentes na literatura, generalizamos um modelo de sistema, o qual, instanciado com conjuntos adicionais de hipóteses, caracteriza desde os extremos, síncrono e assíncrono, a modelos parcialmente síncronos – estáticos ou dinâmicos (variação ao longo do tempo) e homogêneos ou híbridos (componentes distribuídos com diferentes comportamentos temporais).

Esta tese estuda a avaliação de algoritmos distribuídos em ambientes que podem apresentar comportamento híbrido e dinâmico. Desta forma, apresentamos um modelo genérico que

nos permite instanciar modelos especializados a cada ambiente e prover os algoritmos da semântica adequada para que estes adotem estratégias apropriadas de acordo com o conjunto de garantias mantido pelo sistema. Estendemos este modelo ainda, equipando-o quando possível com um oráculo de *QoS* de modo a permitir que os algoritmos distribuídos possam adaptar o comportamento à percepção de *QoS* existente no ambiente.

Capítulo 3

SIMULAÇÃO DE SISTEMAS DISTRIBUÍDOS

No projeto e implementação de algoritmos em sistemas computacionais, a avaliação de desempenho é uma etapa importante. Para efetuar esta avaliação de desempenho são em geral utilizadas três técnicas [10]: a abordagem analítica, em que a análise matemática do modelo permite inferir os resultados esperados; a simulação, em que os algoritmos operam em um ambiente computacional que simula o ambiente de execução real; e a medição, no qual os algoritmos são testados em ambientes reais.

O objeto de interesse desta tese é a avaliação de algoritmos distribuídos que executem em sistemas distribuídos híbridos e dinâmicos, desta forma apresentaremos uma breve discussão sobre a avaliação de desempenho neste contexto.

Sendo o ambiente representado por um modelo de sistema distribuído híbrido e dinâmico, o uso de medição impõe um alto custo, uma vez que pode não ser viável compor um cenário de medição com números grandes de processos e canais de comunicação, cujas configurações variam com o espaço e o tempo (híbridos e dinâmicos).

O uso de abordagem analítica é adequado para estimar o comportamento assintótico dos algoritmos, sendo assim bastante útil para estimar o comportamento de pior-caso. Contudo, esta abordagem pode não ser adequada ao caso médio, devido à complexidade em representar os sistemas distribuídos em geral, e os sistemas híbridos e dinâmicos em particular (e os algoritmos relacionados), por meio de modelos como os baseados em teorias das filas [10] ou redes de Petri. Modelos baseados em teoria das filas podem ser demasiadamente complexos para análise, se, por exemplo, parâmetros como o tempo de serviço variam de forma arbitrária ao longo do tempo.

Por fim, devemos observar a possibilidade do uso de simulação. Por meio de simulação pode-se compor diferentes cenários, que permitem exercitar de forma controlada a execução de algoritmos distribuídos, de acordo com o modelo de sistema em estudo. Cenários híbridos e dinâmicos, podem ser construídos e, conforme as facilidades do ambiente de simulação, eventos podem ser agendados de forma controlada, de forma determinística ou probabilística, como a

degradação da *QoS* de um canal de comunicação, ou a ocorrência de falhas de um ou mais componentes do sistema.

Este capítulo discutirá inicialmente aspectos gerais de simulação de sistemas computacionais. Estes aspectos são um recorte do que é apresentado em maior detalhes em [10]. A seguir, introduzimos aspectos específicos da simulação e modelos de simulação de sistemas distribuídos e dos seus simuladores existentes na literatura, bem como os desafios apresentados para a simulação dos sistemas distribuídos híbridos e dinâmicos.

3.1 SIMULAÇÃO DE SISTEMAS

A avaliação de desempenho de um sistema computacional por meio da simulação se baseia na execução de um modelo de sistema. Este modelo de sistema simulado abstrai o sistema real: os detalhes mais relevantes para a avaliação do sistema devem ser representados. Em geral, o nível de detalhes considerado em um modelo de sistema simulado é maior que o utilizado no estudo de um modelo analítico. Quanto maior o nível de detalhes do modelo de sistema simulado, maior o tempo de desenvolvimento do modelo, contudo mais complexo é o sistema simulado e maior a probabilidade de *bugs*.

O nível de detalhes simulado deve observar o grau de conhecimento existente sobre o sistema. Ou seja, deve-se utilizar uma representação mais generalista do comportamento, abstraindo detalhes, quando estes não forem suficientemente conhecidos ou relevantes para o estudo em questão. Por exemplo, na representação do serviço de leitura/escrita em disco, se o padrão do movimento de um disco (rotação do disco e movimentação do cabeçote) não for suficientemente conhecido, é preferível utilizar uma função de probabilidade que apresente um comportamento esperado para o tempo de acesso ao disco, do que simular de forma detalhada todo o sistema [10].

A simulação é a execução de um modelo do sistema, não do sistema em si. O modelo do sistema deve ser validado e verificado: deve-se observar se o modelo guarda a semelhança desejada com o sistema real (validação), ou seja, se baseia nas hipóteses adequadas sobre este; e se a implementação deste modelo de simulação foi executada sem erros de programação (verificação). A validação em geral é realizada por meio de análise dos resultados obtidos na simulação em comparação ao uso de medição ou de modelagem analítica. Por sua vez, a verificação pode ser realizada por meio de técnicas comuns da engenharia de software para verificação e eliminação de erros de programação ao longo do desenvolvimento do sistema.

A simulação deve ser realizada de modo que os resultados obtidos na execução do sistema representem a execução típica (em regime), descartando o efeito temporário (transiente) do início da execução, os quais podem ser fortemente dependentes das condições iniciais do sistema. O problema de determinar uma janela da execução que represente o sistema avaliado em regime de modo a obter uma avaliação de desempenho adequado é abordado na seção 3.1.4.

A simulação de um modelo de sistema utiliza em geral distribuições de probabilidade. Uma

distribuição de probabilidade provê valores aleatórios que representam parte do comportamento do sistema, como, por exemplo, a carga de trabalho do sistema. A escolha adequada de uma distribuição probabilística permite abstrair os detalhes da implementação de parte do sistema a ser simulado. Considere o exemplo de uma aplicação que, por meio de ativações, invoca um protocolo distribuído: se, por medição de um sistema real, obtivemos a média, o desvio-padrão e o histograma do tempo entre as ativações, e verificamos a semelhança deste histograma com uma distribuição de probabilidade com os mesmos parâmetros, podemos em simulação utilizar esta função para representar as ativações que caracterizam a carga do sistema, sem implementar a aplicação.

3.1.1 Simulação de eventos discretos

Diferentes técnicas de simulação podem ser aplicadas para sistemas computacionais, como, por exemplo, o método de Monte Carlo, a simulação de eventos discretos e a emulação. Monte Carlo não considera aspectos temporais, ou seja, é apropriado para caracterizar fenômenos cujas características não se alteram ao longo do tempo. Emulação permite reproduzir o comportamento de um sistema computacional por meio de outro sistema computacional, como, por exemplo, a emulação de uma arquitetura pela execução do conjunto de instruções desta sobre outra arquitetura computacional.

Um modelo de eventos discretos permite simular a execução de um sistema computacional, em que eventos ocorrem ao longo do tempo entre os componentes que compõem o sistema. Desta forma, este modelo é adequado para a simulação de sistemas distribuídos.

Um simulador de eventos discretos define um relógio de simulação, o qual determina a evolução do tempo ao longo da simulação. Um escalonador de eventos permite agendar no tempo a ocorrência de eventos a serem processados pelo simulador. O estado do sistema evolui de acordo com o processamento dos eventos. Este processamento é realizado por procedimentos computacionais associados a ocorrência dos eventos.

Novos eventos podem ser criados de acordo com gatilhos associados aos dados de entrada da simulação, como, por exemplo, o de chegada de uma nova tarefa na simulação de um protocolo de escalonamento de tarefas em CPU. A caracterização do comportamento da simulação (gatilhos de eventos e dos procedimentos de processamento destes) normalmente é realizada por meio de distribuições de probabilidade. Contudo, esta caracterização também pode utilizar traços (medições) obtidas previamente em ambientes reais.

3.1.2 Métricas

Em uma avaliação de desempenho, em geral, as hipóteses em análise são validadas por meio da observação dos valores obtidos nas métricas de interesse. Por exemplo, na avaliação de protocolos em execução em um sistema distribuído, diferentes métricas podem ser utilizadas para a avaliação de desempenho, de acordo com o algoritmo distribuído avaliado. Em protocolos de

comunicação em grupo, o tempo médio de atraso de entrega de mensagens da aplicação é uma métrica de interesse. Da mesma forma, o *throughput* de um serviço replicado é outra métrica possível.

Um simulador de eventos discretos deve permitir definir quais métricas serão utilizadas, computando-as no processamento dos eventos, de modo a obter dados agregados sobre os mesmos na execução de uma simulação. Resultados intermediários ao longo da simulação (traços) devem ser possíveis de ser obtidos, bem como um relatório de saída final apresentando os valores obtidos para as métricas utilizadas na avaliação de desempenho.

3.1.3 Análise de resultados

Definidas as métricas de interesse, é desejável que o relatório de saída apresente um sumário dos valores observados em simulação. A forma mais simples de sumário dos valores observados em uma métrica é a **média amostral**. Consideremos uma amostra $\{x_1, x_2, ..., x_n\}$ de n valores observados, a média amostral x_{mean} é dada por:

$$x_{mean} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{3.1}$$

A média amostral por si não expressa de forma suficiente um sumário adequado do conjunto de dados da amostra. Para isto, é necessário exprimir o quanto os dados do conjunto variam a partir da média (variabilidade).

Ao observar um conjunto de dados com alta variabilidade, vê-se que os valores podem se distanciar mais e mais frequentemente da média amostral que um conjunto com baixa variabilidade. Diferentes graus de variabilidade para uma mesma métrica em sistemas distintos indicam comportamentos distintos, embasando a comparação entre os sistemas, i.e. um sistema A com desempenho médio levemente inferior a um sistema B pode ser preferido pelo fato de apresentar baixa variabilidade em torno desta média, enquanto o sistema B com alta variabilidade alterna momentos de baixo desempenho com os de alto desempenho. Uma medida de variabilidade é a **variância amostral**. Consideremos a amostra $\{x_1, x_2, ..., x_n\}$ de n valores observados, a variância amostral x_{var} é dada por:

$$x_{var} = \frac{1}{n-1} \sum_{i=1}^{n} (x_{mean} - x_i)^2 \tag{3.2}$$

A raiz quadrada desta medida, o **desvio-padrão** $x_\sigma = \sqrt{x_{var}}$, é em geral uma medida mais utilizada para expressar a variabilidade.

Em alguns casos, é importante constar no sumário da métrica os valores máximos e mínimos do conjunto de dados observados ao longo da simulação.

Se reproduzirmos diferentes vezes um mesmo experimento, obtemos **réplicas** do experimento. Cada réplica deve utilizar sequencias distintas correspondentes às distribuições de probabilidade usadas em simulação (ver sementes de simulação – Seção 3.1.6). Cada réplica *i*

possui um conjunto distinto R_i de dados observados para cada métrica de interesse. Se, em cada réplica do experimento, mensurarmos um valor para uma métrica ou para o sumário da métrica, como, por exemplo, a média amostral, esta medida é uma estimativa do valor real. A partir de um conjunto de estimativas, pode-se obter um intervalo de confiança (c_1, c_2) que expresse com um grau de probabilidade $1-\alpha$, dito nível de confiança, a probabilidade do valor v estar no intervalo:

$$P(c_1 < v < c_2) = p = 1-\alpha \tag{3.3}$$

Onde α é dito nível de significância – o nível de confiança e o de significância são normalmente expressos em percentagem.

A partir do teorema do limite central, o intervalo de confiança associado a uma medida, dado a média x_{mean} e o desvio-padrão x_σ de $n > 30$ valores observados nas réplicas do experimento é dado por:

$$(x_{mean} - z_{[1-\alpha/2]}.s/\sqrt{n}, x_{mean} + z_{[1-\alpha/2]}.s/\sqrt{n}) \tag{3.4}$$

Onde $z_{1-\alpha/2}$ é o quantil de ordem $1-\alpha/2$ da distribuição normal padronizada z (cuja média é igual a 0 e desvio-padrão, igual a 1). O nível de confiança é geralmente um valor próximo a 100%, sendo frequente o uso de nível de confiança de 95%. Para 95% de intervalo de confiança, utilizamos $z_{0,975} = 1,958$.

Para um conjunto pequeno de dados, $n \leq 30$, assumindo uma distribuição de probabilidade normal na população de dados observada, utilizamos o quartil de ordem $1-\alpha/2$ da distribuição t de Student com $n-1$ graus de liberdade.:

$$(x_{mean} - t_{[1-\alpha/2,n-1]}.s/\sqrt{n}, x_{mean} + t_{[1-\alpha/2,n-1]}.s/\sqrt{n}) \tag{3.5}$$

Pode-se utilizar o intervalo de confiança de uma métrica para comparar dois sistemas distintos. Se for possível a observação em pares, deve-se observar a média entre a diferença entre as medições de ambos os sistemas. Se o intervalo de confiança desta média incluir o zero, não há diferenças significativas entre os sistemas. Caso contrário, os sistemas são diferentes e o sinal da média indica qual sistema é melhor.

Mesmo em caso de observações não pareadas, pode-se verificar se há sobreposição dos intervalos de confiança medidos em cada sistema. Se houver, não há diferença significativa entre estes.

3.1.4 Remoção de transientes

Uma análise de desempenho justa deve considerar o desempenho do sistema em regime, ou seja, após as perturbações inerentes da operação inicial do sistema. O estado inicial de instabilidade do sistema é denominado período transiente, o qual pode ser excluído do cálculo das

métricas analisadas. Uma técnica comum em simulações é a de **longo período de execução**: o período de execução da simulação deve ser longo suficiente de modo que o efeito das condições iniciais sobre os resultados sejam minimizados.

Contudo, se o consumo de recursos for excessivo, ou não for possível determinar quão longo deve ser o período de execução, pode-se remover a parte inicial dos dados coletados em tempo de simulação. Essa remoção pode ser de uma parte de tamanho pré-determinado (**truncamento**), ou determinada por meio de análise estatística sobre o conjunto de dados coletados em tempo de simulação. Por exemplo, dentre tais técnicas de remoção de transientes, tem-se o método de remoção de dados iniciais – o mesmo tenta identificar um ponto de inflexão, antes do qual há o período transiente e a partir do qual a operação em regime.

Suponha uma amostra de n valores observados. Este método consiste em detectar o l-ésimo valor da amostra a partir do qual há mudança significativa na média da amostra observada. Ou seja, mensuramos a média x_{mean} de todos os valores da amostra, e a média x^l_{mean} dos valores da amostra após o corte dos primeiros l valores iniciais. Computa-se a seguir a mudança relativa rd_l entre estas médias.

$$rd_l = \frac{x_{mean} - x^l_{mean}}{x_{mean}} \tag{3.6}$$

Executa-se um traço com as mudanças relativas rd_l, variando-se l de 1 a $n-1$ até identificar o ponto da trajetória de rd_l a partir do qual a curva suaviza-se. Este ponto é o ponto de inflexão até o qual os l valores iniciais são removidos do conjunto de dados.

3.1.5 Término da simulação

De forma similar à discussão sobre a remoção do período transiente da simulação, o término desta deve ser calculado de forma cuidadosa. A determinação do tempo de simulação pode ser *ad hoc* observando as características da evolução do sistema ao longo do tempo, por exemplo, em casos de sistemas que se mantém sempre em estado transiente. O período da simulação não deve ser curto o suficiente para que haja alta variabilidade nos dados, nem longo desnecessariamente. Podemos tentar ajustar o período de simulação de modo a obter um conjunto suficientemente grande de n amostras que mantenha um intervalo de confiança reduzido. Na coleta das amostras devem-se observar as características das métricas de interesse de modo a não mensurar, por exemplo, o efeito de computações iniciadas imediatamente antes do término da simulação, mas não completadas até o fim desta.

3.1.6 Geração de números aleatórios e distribuições de probabilidade

Um dos módulos principais de um simulador é o de geração de valores aleatórios, de modo que possam ser compostas distribuições de probabilidade, como, por exemplo, lognormal, exponencial e Weibull. Em geral, o módulo gerador de números aleatórios permite gerar uma

distribuição uniforme de números entre 0 e 1. A partir desta, outras distribuições de probabilidade são geradas.

Uma sequencia de valores (pseudo-)aleatórios é obtido por meio de uma função que a partir de um valor inicial x_0 denominado semente, gera uma sequencia de números x_i, em que x_{i+1} é computado a partir de x_i. A semente x_0 determina toda a cadeia. A maioria das linguagens de programação atuais possui geradores de números aleatórios adequados para o uso em simulação. Devem ser observados contudo alguns cuidados, dentre os quais:

- a cadeia gerada deve ser suficientemente longa para evitar repetições em uma mesma simulação;
- os valores devem ser independentes entre si, com baixa correlação entre os números da mesma cadeia;
- deve-se utilizar uma cadeia para cada variável independente; ao se utilizar a mesma cadeia para duas variáveis independentes estas terão forte correlação, prejudicando o resultado da simulação.

Uma discussão extensa sobre a gerador de números pode ser encontrada em [10]. Diferentes distribuições de probabilidade podem ser compostas e utilizadas a partir de um gerador de números aleatórios. A seguir descreve-se brevemente algumas distribuições de probabilidade utilizadas em simulações ao longo deste trabalho de tese.

Uma distribuição discreta bastante utilizada é a distribuição de **Bernoulli**. Esta distribuição assume os valores 0 ou 1, associados, respectivamente, à falha ou ao sucesso da ocorrência de um evento. Se p é a probabilidade de sucesso, têm-se $P(0) = 1 - p$ e $P(1) = p$ e $P(0 < x < 1) = 0$. As demais características da distribuição de **Bernoulli** são: a média é dada por p e a variância é dada por $p(1-p)$.

A distribuição normal ou gaussiana é bastante utilizada para caracterizar variáveis aleatórias em que a aleatoriedade é causada por diferentes fontes independentes em conjunto, como, por exemplo, ao modelar um comportamento em que diversos fatores não são incluídos no modelo. Ainda, a distribuição normal aproxima o cálculo de outras distribuições para um número grande de observações. A distribuição normal é caracterizada por uma função densidade de probabilidade $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$, $-\infty < x < +\infty$, onde μ é a média e σ, a variância.

A distribuição **exponencial** é utilizada em geral para caracterizar modelos de fila, como, por exemplo, o tempo entre chegadas de eventos independentes a uma fila. Na distribuição **exponencial**, uma variável aleatória x assume valores iguais ou maiores que zero, caracterizados por uma função densidade de probabilidade $f(x) = \frac{1}{\lambda} e^{-x/\lambda}$, $0 < x < +\infty$, onde λ é a média e λ^2, a variância.

Uma variação da distribuição de probabilidade **normal** é a distribuição **lognormal**. Esta função é caracterizada por uma função de densidade de probabilidade $f(x)$, tal que $ln(x)$ segue

uma distribuição normal. Sendo a função de distribuição de probabilidade $f(x) = \frac{1}{\sigma x\sqrt{2\pi}} e^{-(lnx-\mu)^2/2\sigma^2}$, $0 \leq x < +\infty$, onde μ é a média e σ, a variância de $ln(x)$.

A função de densidade de probabilidade de uma distribuição **lognormal** possui "cauda longa", podendo caracterizar, por exemplo, o tempo de transmissão de mensagens, que varia em torno de um tempo típico, contudo em alguns momentos, pode se apresentar muito maior que a média.

A distribuição **uniforme** define a probabilidade de uma variável aleatória x assumir um valor no intervalo $a \leq x \leq b$. A função de distribuição de probabilidade é $f(x) = \frac{1}{b-a}$, a qual possui como características: média no ponto médio do intervalo, dado por $\frac{a+b}{2}$ e variância, dada por $(b-a)^2/12$. Esta distribuição permite, por exemplo, caracterizar a distância entre nós em uma rede sem fio.

Uma versão discreta da distribuição **uniforme** define a probabilidade de assumir um valor inteiro entre $\{a, a+1, a+2, ..., b\}$, com a e b inteiros, neste caso a função massa de probabilidade é $f(x) = \frac{1}{b-a+1}$, a qual possui como características: média no ponto médio do intervalo, dado por $\frac{a+b}{2}$ e variância, dada por $\frac{(a-b+1)^2-1}{12}$.

Outras distribuições, como binomial, Poisson, Gamma, t de Student, Weibull, etc., e uma apresentação mais exaustiva podem ser obtidas em [10].

3.2 SIMULADORES DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS

A área de simulação tem produzido ao longo dos anos um grande número de ferramentas de simulação, que permitem modelar sistemas distribuídos e de rede. Tais ferramentas suportam a criação de ambientes controlados para avaliação de desempenho e desenvolvimento de protocolos sob os ambientes simulados [37].

A seguir, de forma breve, discutiremos possibilidades existentes de simulação apresentadas na literatura. Uma discussão extensa sobre simuladores e ferramentas de simulação existentes há cerca de 10 anos é apresentada em [37]. Deve-se observar que cada ferramenta de simulação tem um objetivo que motivou a sua construção. Por exemplo, há ambientes de propósito geral, que permitem representar sistemas computacionais por grafo de transição de eventos e o serviço provido por canais e processos por meio de filas, como o SMPL [11], isto requer que uma grande quantidade de código tenha que ser construída para simular protocolos de computação distribuída.

Uma ampla gama de ambientes de simulação são ditos simuladores de redes de computadores. Estes simuladores simulam os elementos básicos de infra-estrutura de uma rede de computadores, como nós, roteadores, comutadores. Tais ambientes permitem simular o comportamento de diferentes tecnologias de redes. Por meio desta infra-estrutura simulada, pode-se avaliar, por exemplo, protocolos de roteamento.

Um exemplo clássico deste tipo de simulador é o NS-2 [12], simulador de eventos discretos

baseado em C++ e *Object Tcl*. O NS-2 provê simulação de protocolos, os quais acessam topologias pre-definidas a partir de *scripts oTcl*. Permite o estudo de políticas de filas, controle de congestionamento, roteamento, desempenho de protocolos, dentre outros. Um desenvolvedor de algoritmos de sistemas distribuídos deve compor os modelos de comportamento temporal e de falhas a partir desta infra-estrutura subjacente. Por exemplo, ao simular perda em enlaces, o NS-2 permite simular o modelo de falhas de omissão em canais de comunicação. Em [38] é apresentada uma proposta de extensão do NS-2 para simular falhas de *crash* de processos e sugeridas abordagens não implementadas para simulação de outros modelos de falhas (fail-stop, temporização, falhas de valor e falhas bizantinas).

Um aspecto que deve ser observado é que o nível de abstração para simulação de sistemas distribuídos neste ambiente não é adequado, e embora NS-2 possua uma grande quantidade de código de protocolos de rede já desenvolvido, o tempo para aprendizado do ambiente é longo e o código complexo o suficiente de modo a ser propenso a *bugs*, em especial gerando efeitos inesperados na simulação [39]. Mesmo o NS-3 [13], refatoramento do NS-2 em Java, e o OMNET++ [14], outro ambiente de simulação baseado em eventos discretos, que combina C++ e uma linguagem de descrição em alto nível, denominada NED. Ambos ambientes possuem a mesma filosofia de simulação de redes de computadores, a qual não nos permite uma simulação de sistemas distribuídos baseada no comportamento fim-a-fim dos seus componentes.

Ambientes de simulação para aplicações específicas sobre redes de computadores também são encontradas na literatura. Um exemplo é o do GloMoSim [40], focado em redes MANET, a partir do ambiente de simulação Parsec [41]. O próprio ambiente de simulação Parsec utiliza uma linguagem de simulação própria, distinta de linguagens de uso geral como C++ e Java, o que dificulta a possibilidade de reuso de código usado na simulação em uma aplicação real, uma característica desejável para desenvolvedores que utilizem ambientes de simulação.

Outra aplicação específica é a avaliação de algoritmos P2P em redes de *overlay*. Simuladores como PeerSim [42] e PlanetSim [43] representam a rede de *overlay* na forma de um grafo, com comportamento caracterizado, respectivamente, ou através de traços coletados em ambientes reais ou mantida como uma camada intermediária, sobre a camada de rede física de acordo com diferentes topologias possíveis.

Estes ambientes específicos têm forte ênfase nos detalhes das camadas relacionadas às aplicações de interesse e não a representação dos modelos que caracterizem o sistema distribuído, em que o foco é na comunicação fim-a-fim [44].

Na literatura destacam-se dois ambientes com grau de abstração adequado à simulação de sistemas distribuídos, o Neko [15] e o Simmcast [16]. Estes ambientes são utilizados pela comunidade de sistemas distribuídos e focam na representação de comportamento fim-a-fim da comunicação (canais) e dos processos. Ambos permitem o reuso de código para uso em protótipo em uma rede de computadores real e são baseados em *frameworks* de simulação de eventos discretos em Java: SimJava [45] e JavaSim [46].

Originalmente, apenas o modelo de falha por *crash* é implementado no Neko. Em [47], o

framework é estendido com a implementação de diversos modelos de falhas. Como um *framework* de simulação, o mesmo pode ser estendido para implementação de outros modelos de falhas. Um ponto importante é que o ambiente não permite a composição de cenários mais complexos, híbridos (i.e. com diferentes *QoS* nos canais de comunicação simulados), ou de alterações na dinâmica dos canais na simulação. O agendamento de tarefas é adequado a ambientes assíncronos, pois suas primitivas de agendamento de tarefas não garantem a execução em uma janela de tempo estrita. Essas restrições inviabilizaram a utilização do Neko em cenários síncronos de tempo real e na avaliação de aplicações distribuídas em ambientes híbridos e dinâmicos.

Já o Simmcast possui uma arquitetura focada em um conjunto de *building blocks*, como, por exemplo, **Nodo**, ***Thread***, **Caminho**, **Mensagem**, **Grupo** e **Rede**. Ao criar uma simulação, o desenvolvedor pode estender tais componentes e implementar a funcionalidade a ser avaliada. Na simulação de algoritmos distribuídos, o Simmcast permite simular falhas de forma limitada: caminhos podem descartar mensagens de forma probabilística, ou Nodos podem falhar por contenção de recursos através de filas de tamanho finito – é possivel simular diferentes falhas no *framework*.

O Simmcast se baseia em um conceito de relógio global de eventos, dito ***TimeWheel***, permitindo agendar eventos temporizados para cada **Thread** simulada de acordo com a linha de tempo global. O modelo arquitetural não prevê a simulação de relógio físico local a cada processo, o que permitiria representar dessincronia entre processos ao longo do tempo. Este requisito também é importante, pois a noção temporal em um ambiente distribuído é um conhecimento local, dada a impossibilidade de conhecimento do estado global do sistema pelos processos.

3.3 DESAFIOS PARA SIMULAÇÃO DE SISTEMAS DISTRIBUÍDOS HÍBRIDOS E DINÂMICOS

A avaliação de desempenho em sistemas distribuídos por meio de simuladores deve caracterizar a execução dos algoritmos sob um sistema que expresse o modelo de sistema distribuído desejado. Isto é, o simulador deve permitir implementar as hipóteses temporais, de falhas e o comportamento fim-a-fim do modelo. Esta avaliação pode ser realizada com alto grau de abstração dos detalhes da infra-estrutura subjacente de comunicação (tecnologias de redes de computadores e seus protocolos) e processamento (escalonamento de tarefas etc.).

Um simulador de sistemas distribuídos deve, então, simular desde os modelos homogêneos clássicos, síncrono e assíncrono, os modelos parcialmente síncronos, até os modelos que expressem sistemas distribuídos cujas características dos seus componentes sejam distintas (híbridos) ou o comportamento destes varie ao longo do tempo (dinâmicos).

A avaliação deve representar o comportamento de execução de processos, os quais possuem acesso apenas ao relógio físico local a este associado, e a comunicação fim-a-fim, por meio de canais de comunicação. Deve-se permitir compor diferentes níveis de qualidade de serviço

para os canais de comunicação, representando, por exemplo, canais de comunicação síncronos e assíncronos, e para a execução local a cada processo, como, por exemplo, o comportamento de tempo real, como agendamento de tarefas e sua execução sob uma janela temporal restrita. O desenvolvedor deve poder associar os distintos modelos de falhas aos componentes do sistema distribuído, processos e canais de comunicação.

3.4 CONCLUSÕES

Este capítulo apresentou de forma breve aspectos de simulação de sistemas de eventos discretos. Discutimos de forma breve as etapas necessárias para a avaliação de sistemas por meio de simulação, apresentadas de forma extensa em [10].

Apresentamos também uma breve análise comparativa de trabalhos correlatos em simulação, desde simuladores de eventos discretos genéricos à simuladores de redes de computadores e de sistemas distribuídos, identificando as fragilidades existentes e os desafios para a simulação de sistemas distribuídos híbridos e dinâmicos.

Capítulo 4

HDDSS: SIMULADOR DE SISTEMAS DISTRIBUÍDOS HÍBRIDOS E DINÂMICOS

A partir do **Modelo Genérico de Comportamento Temporal Híbrido e Dinâmico**, apresentado no capítulo 2, propomos um modelo de simulação de sistema distribuídos que caracteriza o comportamento dos componentes do sistema e seus atributos para a execução do ambiente de simulação.

Este modelo de simulação é utilizado para o desenvolvimento de um ambiente de simulação denominado *Hybrid and Dynamic Distributed System Simulator* (**HDDSS**). Este ambiente permite não somente a simulação de sistemas distribuídos híbridos e dinâmicos, mas a simulação de quaisquer sistemas distribuídos convencionais (e.g., síncrono, assíncrono e variações de sistemas parcialmente síncronos).

O **HDDSS** permite a especificação do comportamento dos componentes básicos de um sistema distribuído: processos e os canais. Este comportamento expressa a comunicação fim-a-fim entre os processos, as características temporais dos componentes e as hipóteses de falhas associadas a estes.

Definimos eventos relevantes na execução de uma aplicação distribuída, por meio de geradores de eventos, em que podemos determinar, por exemplo, a carga de trabalho do sistema, e como o comportamento dos componentes varia ao longo da execução.

Este capítulo descreve o modelo de simulação e o **HDDSS**, apresentando sua estrutura e exemplos de uso do ambiente.

4.1 UM MODELO DE SIMULAÇÃO DE SISTEMAS DISTRIBUÍDOS HÍBRIDOS E DINÂMICOS

O objetivo em propor um modelo de simulação para sistemas distribuídos híbridos e dinâmicos é caracterizar um modelo com um nível de abstração adequado para a avaliação de

desempenho de algoritmos em sistemas distribuídos, caracterizando o comportamento fim-a-fim, a evolução dos atributos ao longo do tempo e a probabilidade de ocorrência de eventos relacionados ao modelo de falhas.

A partir do modelo genérico de sistema distribuído, assumimos que todo sistema distribuído $DS(\Pi, \chi)$, constitui-se de um conjunto Π de n componentes processos, dado por:

$$\Pi = \{p_1, p_2, ..., p_n\} \tag{4.1}$$

e por um conjunto χ de canais de comunicação que caracterizam a comunicação fim-a-fim entre tais processos:

$$\chi = \{(p_i, p_j) | p_i, p_j \in \Pi\}. \tag{4.2}$$

Cada processo p_i executa uma sequencia de passos. Cada passo determina uma mudança no seu estado local. O ritmo desta execução é determinado pelo avanço do relógio local ao nó em que o processo executa. Este relógio local se desvia (avança ou atrasa) em relação ao relógio real com uma taxa de desvio (*drift rate*) máxima de ρ.

O comportamento temporal de processos e canais pode ser caracterizado por funções associadas a cada processo p_i e a cada canal de comunicação (p_i, p_j), respectivamente, denominadas f^i e $f^{i,j}$, determinando, de forma respectiva, o tempo de processamento de um passo computacional e a latência de envio de mensagens. Se esta função associada a um componente do sistema tiver valor limitado e conhecido ao longo da execução, o componente é $timely$, caso contrário, $untimely$.

Cada componente do sistema, processo p_i e cada canal de comunicação (p_i, p_j) executa um autômato que caracteriza seu comportamento. Este comportamento pode se alterar ao longo da execução por meio do modelo de falhas associado, respectivamente, $FaultModel^i$ e $FaultModel^{i,j}$. O modelo de falhas associado a um componente é caracterizado por uma função de ativação, respectivamente, g^i e $g^{i,j}$ para $FaultModel^i$ e $FaultModel^{i,j}$. Esta função de ativação determina, ao longo da execução, a ocorrência de falhas caracterizadas no modelo, como, por exemplo, a ocorrência de falhas por *crash* em um processo ou a omissão de entrega de mensagens por um canal de comunicação.

Por fim, podemos definir um conjunto de eventos run_i associado a cada processo p_i que caracteriza as ativações associadas ao protocolo executado pelo processo, ou seja, invocações que geram troca de mensagens de aplicação. Considere o exemplo de uma camada de aplicação de banco de dados que utiliza um algoritmo de comunicação em grupo para replicação ativa [48]: sob diferentes conjuntos *run*, o algoritmo distribuído apresentará execuções distintas, face ser um conjunto distinto de ativações. Este conjunto *run* define uma carga de trabalho de comunicação implementada no sistema.

Desta forma, caracterizamos a execução em um sistema distribuído pelo comportamento fim-a-fim de seus componentes: comportamento temporal (caracterizado por f^i e $f^{i,j}$, respec-

tivamente, para cada processo p_i e cada canal de comunicação (p_i, p_j)); hipóteses de falhas (caracterizado pelo modelo de falhas $FaultModel^i$ e $FaultModel^{i,j}$ associado, respectivamente, a cada processo p_i e a cada canal de comunicação (p_i, p_j) e pela forma como tais falhas ocorrem ao longo da execução – dado, respectivamente, pelas funções de ativação g^i e $g^{i,j}$); e, pela ativação do protocolo distribuído (caracterizado pelo conjunto de eventos run_i associado a cada processo p_i), determinando uma carga de trabalho de comunicação implementada no sistema.

Tais comportamentos são determinados pela configuração do ambiente, pela execução de camadas de software de aplicação e pela infra-estrutura subjacente do sistema. Em um ambiente de simulação, representar tal complexidade do ambiente real não é desejável, uma vez que representaria um maior esforço de detalhamento desta infraestrutura e complexidade na execução da simulação.

Como proposta de abstração adequada para a simulação, tanto o comportamento temporal de canais e processos, quanto a ocorrência de falhas (modelo de falhas) e a carga de trabalho, caracterizada pelos eventos de ativação da computação distribuída, podem ser expressos por funções probabilísticas ou determinísticas. Estas funções refletem a probabilidade conhecida de ocorrência dos comportamentos possíveis para cada componente da computação distribuída.

Diferentes trabalhos da literatura relacionam, tanto a carga de trabalho, quanto o comportamento temporal (atraso de comunicação fim-a-fim) dos canais de comunicação (i.e. em redes IP), com funções de distribuição de probabilidade bem conhecidas. Em [49] distribuições de Pareto, lognormal e normal são analisadas de forma comparativa a traços de uma rede WAN, por outro lado, em [50] traços de aplicações distribuídas que utilizam o protocolo HTTP apresentam comportamento caracterizado por distribuição lognormal.

A partir da caracterização destas funções de ativação que determinam carga de trabalho, comportamento temporal e de falhas, associamos comportamentos probabilísticos ou determinísticos que permitam inferir o comportamento fim-a-fim da computação distribuída, de modo a abstrair a camada subjacente de infraestrutura.

4.2 VISÃO GERAL DO HDDSS

O modelo de simulação proposto no capítulo 3 é a base da proposta do *framework* **HDDSS**. Processos e canais de comunicação utilizam funções que caracterizam o seu comportamento temporal. A carga de trabalho devido à ativação de chamadas de um processo, como, por exemplo, na invocação do protocolo distribuído, também pode ser caracterizada por meio de uma função, bem como demais atributos relacionados à execução do processo. Cada processo possui acesso a um relógio local, o qual pode se desviar do tempo global de uma taxa de desvio máxima de ρ. Podemos associar diferentes modelos de falhas aos processos e canais de comunicação, sem alterar o modo original de implementação, a ativação de falhas relacionadas ao modelo associado pode ser caracterizada por funções.

As funções associadas às latências de processamento e comunicação, à ativação de falhas

e à carga de trabalho, podem ser determinísticas, ou representar distribuições de probabilidade características, ou traços coletados de aplicações reais. A forma como caracterizamos os processos e canais por meio destas funções pode ser estática, ou ainda ser dinamicamente alterada em tempo de simulação. Por exemplo, podemos reproduzir a hipótese GST de estabilidade dos limites temporais a partir de um dado momento da execução.

Por este meio, podemos caracterizar desde os cenários extremos clássicos, síncrono e assíncrono, como prover configurações que implicam em cenários híbridos (diferentes funções associadas a diferentes componentes), caso do TCB ou do Spa, ou cenários dinâmicos, em que a qualidade de serviço de componentes pode, por exemplo, degradar ao longo da execução, caso do HA (ver Capítulo 2).

Por meio desta proposta, o desenvolvedor define um modelo de simulação que caracteriza o modelo de sistema do cenário de interesse, e implementa o protocolo, caracterizando o algoritmo associado aos processos.

Além desta simulação em alto nível, adicionalmente, o *framework* possibilita definir em detalhes uma infraestrutura de comunicação, ou seja, rede de comunicação, e de processamento subjacente, ou seja, nós processadores. Esta possibilidade permite, se necessário, implementar cenários de simulação com um maior grau de detalhes do ambiente de avaliação.

4.3 ESTRUTURA DO HDDSS

O ambiente de simulação **HDDSS** é um *framework* de simulação em Java, ou seja, provê um conjunto básico de componentes para o desenvolvimento genérico de simulações de aplicações distribuídas, ao mesmo tempo que permite construir novos componentes a partir deste conjunto básico, determinando novos comportamentos, de forma a criar novos cenários específicos para a simulação das aplicações. O fluxo de execução da simulação é determinada pelo núcleo de execução do **HDDSS**, que executa o cenário configurado no ambiente de simulação, criando as instâncias correspondentes dos componentes da aplicação distribuída.

De acordo com a taxonomia para ambientes de simulação apresentada em [37], podemos classificar o **HDDSS** como um ambiente de simulação baseado em um motor de execução serial de eventos discretos, dito núcleo de execução, orientado por eventos e pelo tempo (tarefas podem ser agendadas na ocorrência de eventos ou pelo tempo). O framework simula um sistema distribuído baseado em troca de mensagens. Este ambiente é composto por um modo simulação e um modo protótipo, o qual permite reproduzir um ambiente de testes (prototipação) em que a troca de mensagens ocorre por meio de sockets UDP em uma rede de computadores real.

A figura 4.1 ilustra uma visão simplificada dos componentes de uma simulação. O componente principal do núcleo de execução é a classe **Simulator**, a qual executa uma instância de simulação de acordo com um arquivo de configuração. O arquivo de configuração define a configuração inicial de processos e canais de comunicação e a dinâmica do sistema.

Cada processo é uma instância em execução do programa definido por uma classe derivada

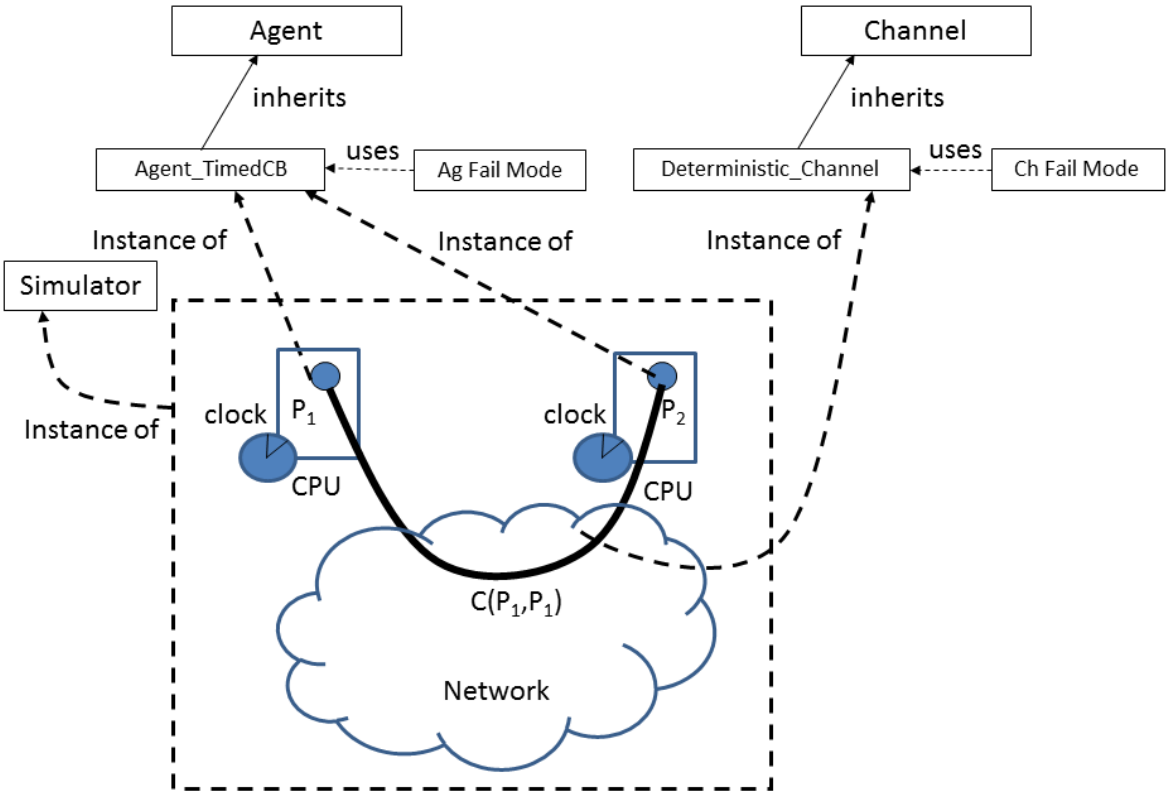


Figura 4.1. Visão geral da estrutura do simulador.

de **Agent**. A troca de mensagens em modo simulação ocorre por meio de canais de comunicação definidos entre processos, cada canal de comunicação é uma instância de uma classe derivada de **Channel**. Uma configuração inicial de processos e canais de comunicação, incluindo as hipóteses de falhas associadas, é definida no início da simulação e pode ser alterada ao longo desta, de acordo com um cenário pré-definido que caracteriza a dinâmica do sistema, este cenário é uma instância de uma classe derivada de **Scenario** – por meio de **Scenario**, pode-se redefinir em tempo de simulação o comportamento associado a processos e aos canais de comunicação que os interconectam.

Para controlar uma simulação, pode-se definir gatilhos de eventos por meio de uma instância da classe **EventGenerator**. Gatilhos definem ações a serem executadas, as quais podem ser ativadas por meio de funções determinísticas ou probabilísticas ou por meio de funções *time-triggered*, ou seja, ativadas a partir de um dado instante t do tempo.

De forma a implementar o modelo genérico de sistema distribuído proposto, caracterizamos atributos de cada componente do sistema (processo ou canal) por meio de funções que expressam comportamento predeterminado ou aleatório; a classe **Randomize** é utilizada para definir funções de distribuição (probabilísticas ou determinísticas) que caracterizam os atributos dos componentes.

Diferentes funções ou parâmetros podem caracterizar o comportamento dos diferentes componentes do sistema. Deve-se ressaltar que em tempo de simulação, pode-se substituir de forma dinâmica uma função por outra se alterando as características de quaisquer atributos dos componentes do sistema. Esta biblioteca de funções inclui as disponíveis no pacote de simulação funções SMPL [11]. Ainda, deve-se notar que um mecanismo de integração com a linguagem R [51], por meio da classe **IntegrationR**, permite o uso de todo o portifólio de funções estatísticas e todo poder computacional deste ambiente estatístico. Por fim, podemos utilizar como funções predefinidas dados de traços coletados de medições em ambiente real.

Todo processo possui associado um relógio físico local que determina o passo de sua execução. Cada relógio físico local em modo simulação é uma instância da classe **ClockVirtual**, simulando o comportamento do relógio de modo que cada relógio físico local diverge do tempo real com (possivelmente distintas) taxas de desvio limitadas por ρ. Tarefas de tempo real podem ser gatilhadas (de acordo com **EventGenerator**) por um processo de acordo com seu relógio físico local.

De forma a simular a troca de mensagens, pode-se definir o comportamento de canais de comunicação, configurados a partir de funções probabilísticas (instância de **ChannelProbabilistic**), como, por exemplo, as funções de Erlang e Lognormal, ou por meio de comportamento determinístico ou de traços medidos de ambiente real.

Como citado, diferentes hipóteses de falhas (definidas por modelos de falhas) podem estar associadas a cada componente (processo ou canal de comunicação), por meio de instâncias da classe **FaultModelAgent** ou da classe **FaultModelChannel**, respectivamente.

Por exemplo, pode-se simular mudanças de valores na troca de mensagens entre processos (*Bizantine value fault model*), processos podem arbitrariamente falhar por *crash* (*Crash fault model*), processos podem falhar por *crash* e recuperar (*Crash-recovery fault model*) ou tão somente podem omitir a resposta de uma computação (*Omission fault model*), ou ainda processos podem apresentar-se corretos.

Quanto a canais de comunicação, estes podem perder mensagens (*Omission fault model*), arbitrariamente corromper mensagens (*Byzantine fault model*), ou ainda apresentar comportamento confiável.

Na criação do ambiente, as instâncias de cada componente são definidas por meio do padrão de projeto *Factory*, utilizando reflexão. Os atributos definidos no arquivo de configuração são inicializados no *Factory* de cada componente respectivo, por meio da invocação, por reflexão, do método associado. A comunicação entre os processos ocorre por meio de troca de mensagens através da invocação de *createMessage(< parameters of a message >)*, gerando uma instância de **Message**. Cada processo utiliza *buffers* de comunicação (de envio e de recepção) e um *buffer* de aplicação (de entrega). O método *execute()* define os passos da computação do algoritmo executado por cada processo. Os métodos *receive(Message m)* e *deliver(Message m)* recebem mensagens e entregam estas ao processamento da aplicação, respectivamente, de acordo com os respectivos *buffers*.

Em modo protótipo, o núcleo da execução é definido pela instância de **TestBed** que permite a execução em um ambiente real: processos podem ser hospedados em máquinas distintas; nós mapeamos o relógio físico local real da máquina hospedeira para o uso pelo processo; e associamos *endpoints* UDP a cada identificador de processo de modo a permitir a troca de mensagens. O *framework* pode ser estendido para permitir, por exemplo, *endpoints* TCP e JAVA RMI.

Para permitir a depuração, há um modo de execução *debug* que permite obter traços detalhados da execução e a observação de eventos de interesse de uma dada avaliação de desempenho.

Diferentes métricas podem ser utilizadas na avaliação de desempenho, de acordo com os objetivos do algoritmo distribuído avaliado. Por exemplo, ao avaliar protocolos de comunicação em grupo, a latência média de entrega é uma métrica de interesse. A vazão é normalmente uma métrica comum de avaliação de diversos serviços distribuídos. A classe **Statistics** permite gerar métricas associadas aos eventos observados (e.g. contagem de eventos, médias, desvio padrão, máximos, mínimos). Por meio do pacote de classes **report**, um relatório estatístico destas métricas pode ser obtido em formato *CSV* ou *human-readable*.

A estrutura básica do simulador ora apresentada permite a simulação de uma computação distribuída e o comportamento fim-a-fim da comunicação inter-processos por meio de canais de comunicação, de modo a realizar a avaliação de desempenho de acordo com o modelo de sistema distribuído desejado. Contudo, como um *framework*, adicionamos facilidades de simulação que permitem reproduzir o comportamento das camadas de infraestrutura subjacente.

Este comportamento permite, por exemplo, representar a interferência no comportamento do sistema, devido à atividade do próprio protocolo avaliado. Por exemplo, considere que o incremento no número de mensagens transmitidas pode afetar as próprias taxas de transmissão nos *switches* e roteadores da rede de computadores. No ambiente de simulação, o efeito subjacente dos nós hospedeiros de cada processo é simulado por meio de instâncias de **RunTimeContainer** e o efeito subjacente da rede de computadores é simulado por meio de uma instância de uma classe derivada de **Network**.

Cada processo executa em um nó hospedeiro, o qual possui sua infraestrutura subjacente caracterizada por uma instância da classe **RunTimeContainer**. Tal instância possui um processador que executa o processo, o relógio físico local associado ao processo e um contexto local, este instância da classe **RunTimeSupport**. Cada processador é instância de uma classe derivada de **CPU** – a escolha apropriada do modelo de processador pode influir na forma como são computados custos de processamento, por exemplo, de tarefas criptográficas. Um modelo simples que abstrai custos de processamento no cômputo da simulação é o uso de **CPUZeroDelay**. O contexto local de um processo mantém, por exemplo, informações percebidas pelo nó hospedeiro.

De forma similar, podemos caracterizar os efeitos do enfileiramento no roteamento na infraestrutura subjacente de comunicação por meio de uma instância de uma classe derivada de **Network**. Por exemplo, um efeito de enfileiramento em um nó roteador central com uma la-

Nome da Classe	Descrição
Agent	Representa um programa em execução (Processo)
Channel	Modela o comportamento fim-a-fim de um canal de comunicação
FaultModelAgent	Caracteriza o modelo de falhas aplicado a um processo
FaultModelChannel	Caracteriza o modelo de falhas aplicado a um canal de comunicação
Scenario	Representa a dinâmica do sistema, definindo como processos e canais de comunicação podem mudar seus atributos ao longo da simulação
RunTimeContainer	Representa um nó hospedeiro: infraestrutura de execução (processador, relógio físico local, contexto local) de um processo
ClockVirtual	Representa um relógio físico local (incluindo *clock drifting*) de um nó hospedeiro
RunTimeSupport	Mantém o contexto de execução local de um nó hospedeiro
CPU	Represente o processador de um nó hospedeiro, o qual executa o processo
Network	Representa a infraestrutura subjacente da rede de computadores utilizada na comunicação interprocessos

Tabela 4.1. Resumo das classes associadas aos principais blocos básicos.

Nome da Classe	Descrição
Randomize	Provê funções probabilísticas ou determinísticas para caracterizar componentes
EventGenerator	Define gatilhos para ações em um processo baseado em parâmetros determinísticos ou probabilísticos
IntegrationR	Provê uma interface para invocar funções da linguagem R
Statistics	Agrega e sumariza eventos para criar métricas de saída
Report (package)	Apresenta os resultados da simulação em formato *CSV* ou *human-readable*

Tabela 4.2. Resumo das classes principais de suporte.

tência determinística de processamento adicionada ao atraso de entrega de cada mensagem é obtido por meio de **NetworkDeterministic**.

Por meio de **Network**, podemos modelar a infraestrutura de rede de modo a computar seu efeito na entrega de mensagens. É possível, por exemplo, modelar o efeito de uma camada de enlace baseada em Ethernet (CSMA-CD), ou de nós roteadores intermediários. Como um *framework*, é possível estender tal classe e obter o comportamento do enlace por meio, por exemplo, da interação obtida em um ambiente de simulação de redes de computadores, como o NS-2 [12]. Tais extensões não foram objeto de implementação.

A Tabela 4.1 apresenta, de forma resumida, as classes associadas aos principais blocos básicos, os quais são instanciados diretamente ou por meio de classes derivadas de forma a compor os componentes do sistema distribuído e da infraestrutura subjacente. A Tabela 4.2 resume as classes de suporte, utilizadas para caracterizar o comportamento probabilístico ou determinístico, para ativar ações baseadas nestes comportamentos, e para computar e apresentar os resultados da simulação.

O **HDDSS** na versão desta tese (revisão 102) possui 58 classes concretas e 15 classes abstratas e interfaces, contabilizando 4.335 linhas de código. O código-fonte foi registrado no Núcleo de Inovação Tecnológica da UFBA e é acessível via Google Code (projeto **HDDSS**). O Apêndice A apresenta uma visão geral do diagrama das principais classes do **HDDSS**.

4.3.1 Uso do ambiente de simulação

O ambiente de simulação proposto permite avaliar protocolos sob diferentes modelos de computação distribuída. Por exemplo, considere o cenário de uma nuvem federada [2], em que

clusters de nós são conectados entre si por meio da Internet. Cada *cluster* da nuvem representa um grupo de computadores interconectados em uma rede local, onde, pode-se assumir um comportamento síncrono, contudo a nuvem como um todo não é um sistema síncrono. Este ambiente pode ser representado por meio do modelo de sistema distribuído híbrido denominado síncrono particionado [7], no qual cada *cluster* local é um sub-sistema síncrono, interconectado por canais de comunicação assíncronos aos clusters remotos, como se apresenta na Figura 4.2.

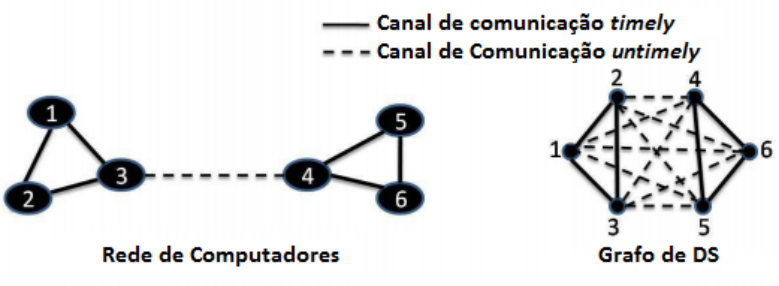


Figura 4.2. Exemplo de cenário híbrido – modelo síncrono particionado.

Um trecho de um arquivo de configuração deste cenário é apresentado na Figura 4.3. A seguir, descrevemos as variáveis desta configuração: $FinalTime$ representa o tempo total de simulação; $NumberofAgents$ representa o número total de processos simulados; $MaximumDeviation$ representa a taxa de desvio máxima dos relógios (em 10^3); $ClockVirtual$ representa o uso do relógio simulado em todos processos; *channel* define os tipos de canais de comunicação a serem instanciados: $channel[0]$ define um canal com latência determinística no qual há limites temporais mínimos e máximos para a transmissão de mensagens, e $channel[1]$ define um canal probabilístico em que a latência é determinada por uma distribuição NORMAL, função da linguagem R; e *scenario_ps* é o nome do cenário o qual será simulado, o qual utiliza as definições de canais apresentadas para configurar um cenário de dois clusters síncronos, com canais de comunicação assíncronos entre os processos de clusters distintos, na forma do modelo síncrono particionado.

```
FinalTime = 100000
NumberOfAgents = 6
MaximumDeviation = 2
clock = Clock_virtual

scenario = Scenario_PS
scenario.NumberOfAgentsPerPartition = 3 3
scenario.SynchronousChannelType = 0
scenario.AsynchronousChannelType = 1

channel[0] = ChannelDeterministicInterval
channel[0].DeltaMin = 5
channel[0].DeltaMax = 10

channel[1] = ChannelProbabilistic
channel[1].Distribution = R("rnorm(1000000,mean=20,sd=10)")
```

Figura 4.3. Exemplo de arquivo de configuração que caracteriza cenário híbrido.

4.4 ESTUDO COMPARATIVO: PROTOCOLO DE DIFUSÃO CONFIÁVEL *SENDER-INITIATED*

Para avaliar a validade do uso do simulador, foram realizados diferentes experimentos *ad hoc* com simulação de cenários, como o de sincronização de relógios em ambientes síncronos e difusão de mensagens em ambientes assíncronos. Para apresentar a efetividade do uso do simulador, apresentamos um comparativo dos resultados obtidos pelo ambiente de simulação com os de um modelo analítico para a avaliação de desempenho de um protocolo de difusão confiável dito *sender-initiated*.

Um estudo clássico de escalabilidade de protocolos de difusão confiável é apresentado em [52]. Este compara resultados analíticos de vazão de protocolos de difusão confiável: de um protocolo dito *sender-initiated* que utiliza ACKs, e de duas variações de protocolos *receiver-initiated* que utilizam NACKs.

Utilizaremos os resultados do protocolo *sender-initiated*. Este protocolo, denominado protocolo A, se baseia na difusão pelo emissor de uma sequência de pacotes numerados: um pacote de número i é transmitido; de modo a verificar o recebimento do pacote, todos os receptores devem confirmar por meio de ACK; caso a confirmação não ocorra em uma janela temporal pré-estabelecida (*time-out*), o pacote é retransmitido, até que todos os receptores confirmem por ACK o mesmo; somente após esta confirmação é transmitido o pacote $i+1$ da mesma forma e assim por diante.

O modelo analítico proposto em [52] assume que os canais estão sujeitos a perda de pacotes (falhas por omissão). E estas falhas são mutualmente independentes entre si com probabilidade p da perda de mensagens no canal. Ainda, assume-se a hipótese que os pacotes de ACK do receptor para o emissor não são nunca perdidos, em face serem pequenos.

A vazão Λ^A de envio com sucesso do protocolo A é dada em função do tempo médio $E\left[X^A\right]$ de difusão com sucesso de um pacote:

$$E\left[X^A\right] = E[M]\,E[X_p] + (E[M]-1)E[X_t] + R.E[M]\,(1-p)E[X_a] + E\left[X_f\right] \tag{4.3}$$

$$\Lambda^A = \frac{1}{E[X^A]} \tag{4.4}$$

onde $E[M]$ é o número médio de retransmissões por pacote, $E[X_p]$ é o tempo médio de processamento da transmissão de um pacote, $E[X_t]$ é o tempo médio de processamento da interrupção originada pelos *time-outs*, R é o número de receptores, $E[X_a]$ é o tempo médio de processamento do pacote e $E\left[X_f\right]$ é o tempo médio para o recebimento de dados da camada superior.

Este modelo analítico de [52] permite inferir o valor de $E[M]$ a partir do número de receptores R. O modelo apresenta diferentes resultados numéricos de vazão de acordo com a probabilidade de perda p. Implementamos o mesmo modelo analítico por meio da linguagem R

[51] e reproduzimos exatamente os resultados apresentados no artigo, validando nossa implementação desenvolvida na linguagem R.

Implementamos com o **HDDSS** o protocolo A, por meio de cenários caracterizados por distribuição uniforme (média de 10 unidades de tempo e desvio padrão de 5 unidades de tempo), e *time-out* de 15 unidades de tempo. Estes valores foram escolhidos de forma *ad-hoc* em face de não haver referência aos valores utilizados em [52]. Cada experimento simulou um período de 10000 unidades de tempo, sendo replicado 3 vezes, em que o emissor continuamente envia uma sequencia de mensagens, conforme o protocolo descrito.

O cenário implementado utilizou a mesma hipótese de configuração híbrida dos canais apresentada no artigo: os canais do emissor para os receptores aplicam o modelo de falhas por omissão, com a probabilidade de perda de pacotes de $p = 0.05$, os canais dos receptores para o emissor são confiáveis.

Em unidades de tempo, definimos os parâmetros do modelo analítico: $E\left[X_f\right] = 5$, $E\left[X_a\right] = 5$, $E\left[X_t\right] = 1$ e $E\left[X_p\right] = 1$. Os eventos correspondentes na simulação, quando existiam, foram ajustados da mesma forma.

Os resultados da simulação são apresentados de forma comparativa com o resultado numérico do modelo analítico na figura 4.4. Os resultados foram normalizados para uma escala de 0 a 1, onde 1 representa a vazão máxima obtida.

Como pode-se observar, os resultados da simulação obtidos foram similares aos resultados numéricos do modelo analítico, mesmo não sendo possível determinar de forma exata todos os parâmetros – exceto que a curva da simulação apresenta queda um pouco mais rápida que a curva do modelo analítico com uma discrepância máxima entre os valores de ambos os traços de 0.043 na escala de 0 a 1. Verificamos que a simulação valida a tendência esperada pelo modelo analítico.

4.5 CONCLUSÕES

Sistemas distribuídos híbridos e dinâmicos têm componentes cujos comportamentos podem variar ao longo do tempo, de acordo, por exemplo, com a disponibilidade de recursos ou a ocorrência de falhas. A avaliação de tais sistemas não é uma tarefa trivial e a simulação é uma ferramenta poderosa nestes cenários, especialmente quando há um número considerável de componentes do sistema. Este capítulo apresentou um *framework* de simulação baseado em um modelo genérico de sistema distribuído, capaz de simular sistemas distribuídos híbridos e dinâmicos, bem como sistemas distribuídos convencionais, como síncronos, assíncronos e parcialmente síncronos.

O simulador permite definir diferentes comportamentos temporais e modelos de falhas e associar estes dinamicamente a canais de comunicação e processos. O ambiente de simulação apresenta um alto grau de abstração adequado à avaliação da computação distribuída ao mesmo tempo que permite representar detalhes da infraestrutura subjacente, se for necessário. Obtemos

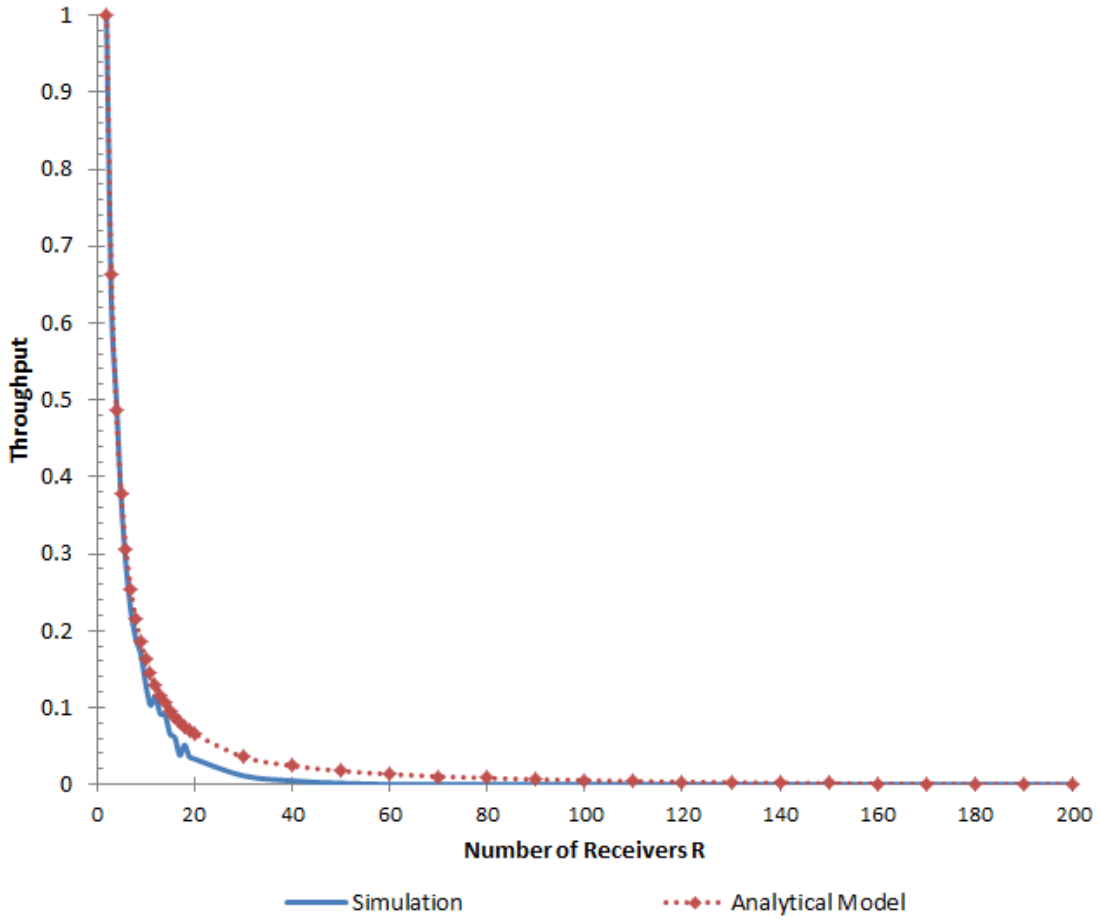


Figura 4.4. Gráfico dos resultados obtidos na avaliação por simulação e por modelagem analítica do protocolo de difusão *sender-initiated*.

resultados adequados na comparação com um modelo analítico para um problema clássico, de modo que o *framework* de fato apresentou-se adequado para simular o mesmo comportamento de um modelo analítico conhecido.

Capítulo 5

AVALIAÇÃO DE PROTOCOLOS TOLERANTES A FALHAS POR MEIO DO HDDSS

Diferentes mecanismos podem ser utilizados como *building blocks* para a construção de aplicações distribuídas tolerantes a falhas. Dentre tais mecanismos, são considerados neste capítulo o uso de protocolos de computação em grupo em ambientes sujeitos a falhas por *crash* de processos, o de protocolos de replicação tolerantes a falhas bizantinas e o de consenso em ambientes híbridos e dinâmicos.

Mecanismos de comunicação em grupo em ambientes sujeitos a falhas por *crash* podem ser utilizados, por exemplo, para aumentar a resiliência de aplicações distribuídas por meio da replicação ativa de servidores. No desenho de protocolos para este fim, mecanismos mais básicos como difusão confiável, consenso distribuído e detectores de defeito devem ser combinados entre si na montagem da solução. Em ambientes mais hostis, por exemplo, em que a infraestrutura computacional pode estar sujeita a intrusão, o provimento de um serviço replicado deve ser tolerante a falhas bizantinas.

Estes *building blocks* avaliados podem ser utilizados em um largo espectro de cenários para prover soluções úteis ao desenvolvimento de aplicações tolerante a falhas. Neste capítulo, é feita, de forma comparativa, uma avaliação do desempenho de novas propostas para tais *building blocks* em contextos de aplicação nos quais os mecanismos convencionais já desenvolvidos não obtêm um desempenho adequado.

5.1 COMUNICAÇÃO EM GRUPO

Desde a década de 80, que protocolos de comunicação em grupo se apresentam como uma poderosa abstração para o desenho de aplicações distribuídas tolerantes a falhas em uma variedade de cenários que expressam diferentes modelos de sistemas distribuídos, de sistemas síncronos a sistemas assíncronos. Embora similar em princípios, diferentes especificações e implementações foram propostas para os diferentes modelos de sistema [53–57]. De forma resumida, um protocolo de comunicação em grupo provê o serviço de entrega do mesmo conjunto

de mensagens a um mesmo grupo de processos que cooperam em uma computação distribuída. Desta forma, um protocolo de comunicação em grupo abstrai, para camadas de aplicação superiores, a complexidade e incerteza do sistema de comunicação subjacente, como falhas ou entrega de mensagens fora de ordem, permitindo, por exemplo, o provimento de replicação ativa de servidores.

O protocolo de comunicação em grupo, por sua vez, utiliza um conjunto de algoritmos básicos, como difusão confiável e consenso. Por exemplo, o problema de determinar a visão deste grupo, ou seja, o conjunto de processos que o compõe, conhecido como *membership*, pode utilizar como componente da solução o consenso distribuído, ou seja, os processos concordam em um mesmo e único conjunto de processos que formam a visão atual deste grupo. Ainda, o problema da entrega ao grupo em uma mesma ordem de um mesmo conjunto de mensagens, dito difusão atômica confiável, é equivalente ao consenso distribuído, uma vez que os processos envolvidos executam uma forma de acordo [58].

De acordo com o modelo de sistema que caracteriza o cenário de atuação do protocolo de comunicação em grupo, pode-se determinar a qualidade de serviço que pode ser obtida, em termos de garantias na entrega das mensagens e na visão dos membros do grupo.

Em sistemas síncronos, há limites temporais conhecidos para a transmissão de mensagens e processamento. As hipóteses deste modelo simplificam o tratamento de falhas, devido à certeza que se um processo não envia uma mensagem (ou não realiza seu processamento) dentro de um limite temporal conhecido, pode-se determinar que este processo falhou por *crash*. Esta certeza permite que algoritmos como consenso e difusão atômica confiável tenham solução possível, mesmo em face da ocorrência de falhas por *crash*, em sistemas distribuídos síncronos [59–61].

Contudo, em sistemas puramente assíncronos, não há tais limites temporais para transmissão de mensagens e processamento. Esta ausência de requisitos temporais torna o sistema menos sensível a variações nas condições operacionais, como, por exemplo, períodos indefinitamente longos para transmissão de mensagens. Ainda, isto torna impossível a solução de algoritmos como o consenso, e consequentemente, o *membership*, em face da possibilidade de ao menos um processo falhar, sem que hipóteses adicionais de garantias sejam consideradas no modelo do sistema [1].

Hipóteses adicionais podem ser providas, por exemplo, por sistemas parcialmente síncronos, como a hipótese GST, a qual admite a existência de um instante do tempo, a partir do qual há limites temporais que se mantém, criando condições necessárias para a resolução de problemas fundamentais de tolerância a falhas, como consenso e difusão atômica confiável [26, 58]. Baseado nestas premissas, foi possível a implementação de serviços de comunicação em grupo em sistemas parcialmente síncronos [56, 57].

O provimento dos serviços de comunicação em grupo implica no monitoramento de mudanças no *membership* do grupo, devido a falhas, entradas ou saídas espontâneas de processos, e, por isso, na troca de mensagens de controle relacionadas a este monitoramento, as quais implicam em *overhead* adicional, que altera as condições de operação do protocolo. A escolha

adequada, de acordo com a carga do ambiente e dos parâmetros relacionados à operação do protocolo, determina a sua velocidade, ou seja, o atraso na entrega de mensagens, e custo, ou seja, o *overhead*, imposto pelas mensagens de controle – este problema de configuração adequada dos parâmetros, em especial em face de mudanças da carga do ambiente ao longo do tempo, não é tratado de forma adequada na maioria das implementações.

Um outro problema não completamente tratado é o de desenvolvimento de protocolos de comunicação em grupo em ambientes com componentes ora síncronos ora assíncronos (sistemas distribuídos híbridos e dinâmicos) [54]. Enquanto uma solução segura seria considerar o pior caso (sistema totalmente assíncrono, ou ao menos, com hipótese GST), a existência de limites temporais pode ser utilizada na convergência do protocolo, de modo a obter uma solução genérica, aplicável no caso assíncrono, e ótima no caso síncrono e adaptável conforme o nível de sincronia do sistema.

Aqui, apresenta-se de forma breve o problema de comunicação em grupo, caracterizando as propriedades de segurança e terminação e métricas a serem consideradas na avaliação de desempenho destes protocolos. Em seguida, a construção de um protocolo de comunicação em grupo que considera a informação temporal existente baseada em um modelo de sistema distribuído híbrido e dinâmico. Por fim, é apresentada a avaliação de desempenho de dois protocolos de comunicação em grupo, especialmente projetados para suprir as deficiências de soluções existentes nos cenários de interesse apresentados.

Um outro esforço de desenvolvimento, apresentado na tese de [17], é o de construção de um protocolo de comunicação em grupo em que o ajuste dos parâmetros de operação considera um nível de serviço desejado (auto-gerenciável). Este protocolo utilizou a base de desenvolvimento anterior e a sua avaliação de desempenho foi uma das contribuições desta tese.

Exceto quando mencionado, quanto ao estudo de protocolos de comunicação em grupo, esta seção assume que processos possam falhar por *crash*, e que canais de comunicação são confiáveis, e possuem a propriedade de FIFO, mecanismos que podem ser implementados, por exemplo, por meio de uso de ACKs, retransmissão e controle de fluxo fim-a-fim.

5.1.1 Mecanismos básicos dos protocolos de comunicação em grupo

As considerações sobre os mecanismos de comunicação em grupo ora discutidas assumem a existência de um único grupo, estas considerações podem ser facilmente estendidas para o suporte a múltiplos grupos. Ainda, em ambientes como os de redes WAN ou de redes sem fios, a rede pode temporariamente estar sujeita a partições: alguns nós podem ficar isolados dos demais, e quando isto ocorre um grupo pode ser dividido em diferentes subgrupos (ou partições). Protocolos de comunicação em grupo abordam este problema de duas formas distintas: partição primária e particionamento [57]. Na abordagem de partição primária, apenas uma das partições mantém o grupo e os processos isolados desta precisam ser adicionados novamente ao grupo. A abordagem de partição primária é a utilizada aqui no desenvolvimento dos protocolos

de comunicação em grupo ora descritos.

Processos formam um único grupo g, o qual possui como configuração inicial Π, ou seja, $g = \{p_1, p_2,...,p_n\}$. Neste grupo g, podemos definir quatro operações básicas: $join(g)$ – requisição de um processo que deseja entrar no grupo g; $leave(g)$ – requisição de um processo que deseja sair do grupo g; $send(g,m)$ – envio de uma mensagem m por um processo ao grupo g; e $deliver(g,m)$ – entrega a camada de aplicação de uma mensagem previamente recebida pelo processo que foi enviada por um membro do grupo g.

Cada processo p_i do grupo g instala visões na forma $v_i(g) \subseteq \Pi$. Uma visão indica o conjunto de membros do grupo que se reconhecem mutuamente como ativos e pertencentes ao grupo em um dado instante da existência do mesmo. A visão $v_i(g)$ pode se alterar ao longo do tempo em função de (suspeitas de) falhas de processos ou quando processos deliberadamente saem ou entram no grupo. Processos podem falhar apenas por *crash*, podendo haver ou não a hipótese de recuperação (*crash-recovery*). Em face de mudanças na visão do grupo, uma nova visão é instalada pelo sub-protocolo de membership. Cada visão instalada por um processo é associada a um número de versão (visão) que incrementa monotonicamente de acordo com as instalações das visões de grupo: $v_i^k(g)$ denomina a visão de número de versão k instalada por p_i. Quando conveniente, e sem que haja ambiguidade, o identificador do processo que mantém a visão pode ser omitido (e.g., $v^k(g)$), ou o identificador do grupo (e.g., v_i^k).

Quando um grupo g é criado, cada membro do grupo p_i instala a mesma visão inicial $v_i^1 = \Pi$. Quaisquer modificações subsequentes na configuração do grupo resultará na instalação de novas visões, formando a sequência $v_i^1, v_i^2, ..., v_i^k$, onde k é o número da versão atual da visão de acordo com o histórico de visões. Um processo p_i difunde mensagens somente aos processos pertencentes a visão atual. A difusão de mensagens ocorre por meio do sub-protocolo de difusão confiável com ordenação total, o qual é responsável, de forma sucinta, pela difusão e entrega do mesmo conjunto de mensagens, preservando a mesma ordem de entrega, aos processos pertencentes à visão atual.

Os protocolos de comunicação em grupo podem utilizar diferentes mecanismos para a difusão atômica confiável, mantendo a ordenação total. Em [57], são apresentadas algumas propostas de mecanismos básicos para a difusão: (i) **sequenciador**, um processo líder, o qual se mantém ao longo da computação ou é alterado periodicamente, é responsável por realizar a difusão atômica confiável e indicar a sequencia que compõe o fluxo de mensagens enviado ao grupo; (ii) **baseado em privilégio**, um processo envia ao grupo apenas quando possui permissão, a qual é repassada a cada membro do grupo por meio de um *token*; (iii) **histórico de comunicação**, em que informações, como, por exemplo, o relógio físico de envio de cada mensagem (em sistemas síncronos), ou o relógio lógico de envio de cada mensagem [62] (em sistemas não síncronos), permite inferir ordem, mesmo que parcial na sequencia de eventos locais a cada processo, e são utilizadas para determinar a sequencia ordenada de mensagens no grupo, e (iv) **acordo no destino**, um subprotocolo de acordo é executado entre os nós pertencentes ao grupo para determinar o conjunto de mensagens e a sua ordem de entrega – esta abordagem, apresentada em

[58], reduz a difusão atômica confiável com ordenação total a uma sequencia de problemas de consenso. No escopo desta tese, os protocolos apresentados são baseados em sequenciadores fixos e no histórico de comunicação.

De modo a definir a especificação adequada de funcionamento de protocolos de comunicação em grupo, são definidas propriedades de segurança e de terminação para o protocolo de comunicação em grupo. Tais propriedades de segurança e terminação estão relacionadas à instalação de visões (*membership*) e a entrega do conjunto de mensagens (difusão atômica confiável com ordenação total). De acordo com as diferentes implementações de protocolos de comunicação em grupo, há diferentes conjuntos possíveis de propriedades que caracterizam o nível de serviço oferecido [54, 56, 57].

Como um cenário motivador para o uso de protocolos de comunicação em grupo, destaca-se o da replicação ativa de servidores. Para provermos as propriedades necessárias a este cenário, destacamos as propriedades associadas à entrega de mensagens pela difusão atômica confiável baseada em [57] adaptadas à existência de visões de grupo: a propriedade de **validade**, que garante a terminação, e as propriedades de **acordo uniforme**, **integridade uniforme** e **ordem causal e total uniforme** que garantem a segurança.

A **validade** implica que uma mensagem enviada em uma dada visão será, possivelmente, entregue em uma visão futura a qual reflete, possivelmente, uma nova configuração de grupo devido a ocorrência de crashes, joins ou leaves. Há um limite temporal Δ_1 para esta entrega conhecido *a priori* quando todos processos da visão $v_i^s(g)$ são síncronos e conectados entre si por meio de canais de comunicação síncronos, ou seja, um sistema distribuído síncrono, e pode ser determinado de acordo com o modelo do sistema. De outra forma, o sistema não é síncrono e Δ_1 é um valor positivo arbitrário, mas que pode não ser determinado *a priori*.

Em grupos estáticos, ou seja, onde não há mudanças na visão do grupo, uma primitiva de difusão confiável [61] satisfaz todas propriedades de segurança: quaisquer dos processos do grupo, mesmo sujeito a falhas, entrega no máximo uma única vez uma mensagem (**integridade uniforme**), e esta entrega implica na entrega da mesma mensagem a todos processos corretos (**acordo uniforme**) e na mesma ordem (**ordem total uniforme**), exceto a **ordem causal**. Contudo, **ordem causal** é necessária para replicação ativa de servidores [48, 62].

Ainda, em face da ocorrência de *crashes*, *joins* ou *leaves*, a execução de *membership* dinâmico, isto é, no qual há a evolução da visão do grupo ao longo da execução, é necessária para, por exemplo, manter um número especificado de réplicas ativas. Em algumas aplicações, como, por exemplo, no balanceamento de carga entre processos distribuídos, o conhecimento prévio desta visão atual do grupo é um fator importante para determinar quando ocorre a entrega de mensagens. A instalação de visões de grupo, *membership*, pode ainda observar outros aspectos, como, por exemplo, em sistemas síncronos ainda a existência de limites temporais.

Desta forma, propriedades de terminação devem assegurar que qualquer falha por *crash* será detectada a termo [1] e que uma nova visão será instalada sem os processos falhos, pro-

[1] Deve-se observar no caso de sistemas assíncronos, uma suspeita de falha pode também resultar na instalação

vendo completude forte ao mecanismo de detecção de defeitos [30], e ainda, que requisições espontâneas de *join* e *leave* também serão processadas a termo, gerando novas visões de grupo correspondentes.

Os limites temporais relacionados a instalação de visões de grupo na ocorrência de *crashes*, *joins* ou *leaves* podem somente ser calculados quando o sistema é síncrono – em caso de sistema assíncrono, tais limites são valores positivos arbitrários, mas que podem não ser determinados *a priori*.

Quanto à segurança, a visão de grupo deve evoluir de forma única, todos os processos devem instalar a mesma sequência de visões na mesma ordem crescente (**Sequência Única de Visões** e **Monotonicidade Local**). Um processo só deve participar de uma visão ao qual pertence (**auto-inclusão**) e o ingresso ou saída de um processo de uma visão para uma visão subsequente deve ser justificado (**inclusão** e **exclusão justificadas**) somente por requisições espontâneas de *join* ou de *leave*, ou ainda por suspeita de falha por *crash*.

As propriedades de **Monotonicidade Local** e **Sequência Única de Visões** combinadas provêm a propriedade de **Acordo nos Sucessores** [56]. E esta última em conjunto com o **Acordo Uniforme na Entrega de Mensagens** apresentada provêm a conhecida propriedade de **Sincronia de Visões** [56].

5.1.2 Desenvolvimento de protocolo de comunicação em grupo *self-aware*

Em sistemas distribuídos híbridos e dinâmicos, os componentes do sistema (processos e canais de comunicação) podem alternar ao longo da execução de forma não necessariamente homogênea em distintos níveis de qualidade de serviço (*timely* e *untimely*). Tais sistemas podem se aplicar a cenários cuja a alocação de recursos para definição de níveis de serviços possa ser realizada de forma apenas parcial. Estes cenários englobam desde nuvens federadas [2], em que não é possível a alocação de recursos interdomínio, aos *Cyber-Physical Systems* [3], em que dispositivos distribuídos se auto-regulam de acordo com um laço de controle computacional, e requisitos temporais, como os de tempo real no escalonamento e na comunicação entre os componentes, podem ser necessários em parte do sistema, mas podem não se manter no seu todo, se, por exemplo, o mesmo estiver distribuído ao longo de redes como a Internet.

A possibilidade de percepção dos requisitos temporais existentes torna o sistema distribuído auto-cônscio (*self-aware*) e permite prover os algoritmos de computação distribuída com subsídios de modo a auto-adaptarem-se, alterando a execução em face dos componentes com características $timely$ e $untimely$ do sistema. Considerando as hipóteses temporais processos síncronos e canais de comunicação $timely$ e $untimely$, baseado no modelo HA [4, 6] pode-se negociar níveis de serviço fim-a-fim entre processos e perceber a qualidade de serviço existente. Isto pode ser obtido por meio de um mecanismo subjacente de provimento e monitoramento da qualidade de serviço, que assegure, por exemplo, limites temporais em canais de comunicação.

---

de uma nova visão que exclui o processo suspeito.

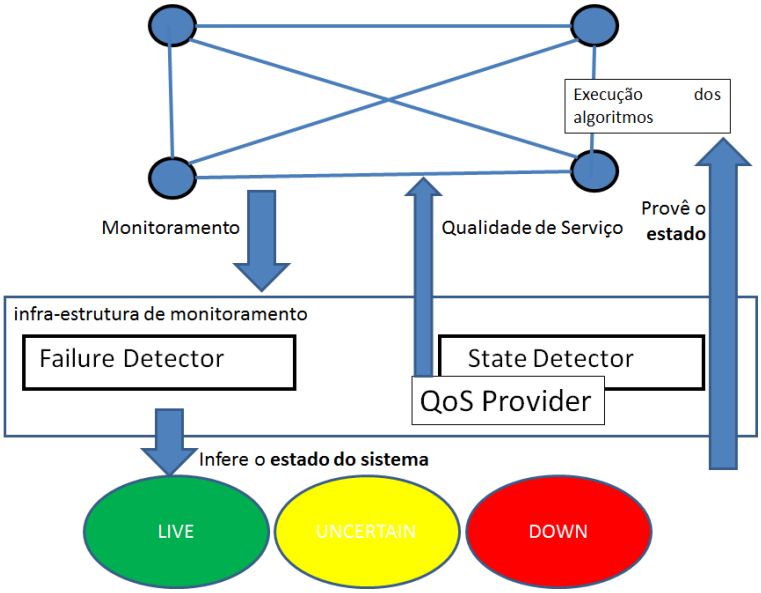


Figura 5.1. Exemplo de sistema distribuído *self-aware*, com estrutura de monitoramento e provimento de qualidade de serviço.

Esta informação é utilizada para classificar os processos em conjuntos (*live, uncertain e down*). Este modelo de sistema permite o desenho de algoritmos, como o consenso distribuído, que se adaptam de acordo com a percepção do ambiente.

A estrutura de monitoramento associa a qualidade de serviço percebida *timely* ou *untimely* a informação de conjuntos que caracteriza os processos do sistema. A *QoS* representada por estes conjuntos é um corte consistente percebido por cada processo do estado global do sistema, dada a impossibilidade de compor um instantâneo real do estado global de todo o sistema [29].

A Figura 5.1 apresenta um exemplo de sistema distribuído *self-aware*, com uma estrutura de provisionamento de recursos para obter qualidade de serviço (*QoS provider*) e um monitor da *QoS* obtida do sistema (*State Detector*) – estes mecanismos são apresentados em [6], e em conjunto com um mecanismo de monitoramento de processos ativos baseado em *heartbeat*, detector de defeitos (*failure detector*), provê a informação dos conjuntos *live, uncertain e down* utilizados para o ajuste do comportamento do algoritmo distribuído.

Aqui, estes conceitos apresentados são estendidos de modo a construir um protocolo de comunicação em grupo em que cada processo utiliza da informação do estado do sistema obtido pelo monitoramento, na forma dos mesmos conjuntos (*live, uncertain e down*) e da percepção de *timely* e *untimely* de cada canal de comunicação, de modo a adaptar o comportamento e otimizar a execução em face da existência de componentes síncronos.

Para o desenvolvimento desta abordagem, um mecanismo pré-existente de comunicação em grupo, baseado em histórico de comunicação, o mecanismo de blocos causais, é utilizado, o qual provê um protocolo de comunicação em grupo para sistemas assíncronos, garantindo as propriedades adequadas à execução destes protocolos [63, 64]. Tal mecanismo de blocos causais utiliza matrizes baseadas na informação dos relógios lógicos de envio das mensagens

– utilizamos a informação dos relógios físicos, estendendo o mecanismo em blocos causais temporais (timed causal blocks), que permite utilizar informação temporal, se existente, obtendo execução otimizada em sistemas síncronos.

A seguir, o mecanismo de blocos causais é descrito brevemente, e apresentados o protocolo genérico desenvolvido baseado na extensão *timed causal blocks* e uma avaliação de desempenho comparativa com protocolos clássicos de comunicação em grupo em ambientes assíncronos e síncronos.

5.1.2.1 Mecanismo de blocos causais

O mecanismo de blocos causais permite implementar a difusão atômica confiável com ordenação total em um protocolo de comunicação em grupo. Cada processo p_i mantém um relógio lógico, denominado *block counter* de p_i, ou BC_i. BC_i é uma variável inteira cujo valor incrementa monotonicamente. Quando um grupo g é criado, o respectivo BC_i de cada p_i é iniciado com um inteiro não-negativo – sem perda de generalidade, todos podem ser iniciados com o valor zero. Mensagens transmitidas são rotuladas com o *block counter* do emissor, e, de forma similar ao relógio lógico de Lamport [62], este rótulo obedece a causalidade potencial – ou seja, a relação *happen-before* indicada por $\rightarrow$. Ou seja, se $m \rightarrow m'$, então o rótulo temporal associado com *m'* será maior que o associado com *m*. Contudo, o inverso nem sempre se mantém: se duas mensagens possuem diferentes rótulos temporais, estas podem ser concorrentes. De forma similar à evolução do relógio lógico de Lamport em face a eventos de envio/recepção, o *block counter* avança em face de eventos de envio/entrega, uma vez que uma mensagem transmitida pode ter dependência causal apenas de mensagens que foram entregues previamente.

Imediatamente antes de um processo p_i enviar uma mensagem m ao grupo g, este incrementa em um o valor de BC_i. Este valor incrementado torna-se o rótulo temporal de m, como número do bloco no campo $m.b$. p_i pode atualizar o valor de BC_i ainda se houver a entrega de uma mensagem m (denoted $delivery_i(m)$) cujo valor $m.b$ associado é maior que o valor atual de BC_i. Desta forma, estas duas regras definem o incremento de BC_i:

- *CA1* (Incremento do *block counter* em face de $send_i(m)$): Antes de p_i difundir m, este incrementa em um o valor de BC_i, e associa este valor atualizado ao rótulo temporal $m.b$; e,
- *CA2* (Incremento do *block counter* em face de $delivery_i(m)$): Antes de p_i entregar m, este atualiza o valor de BC_i para $max\{BC_i, m.b\}$.

A partir das regras *CA1* e *CA2*, podemos definir as seguintes propriedades que associam precedência de eventos, onde $send_i$ indica um evento de *send* executado pelo processo p_i:

- *pr1*: $send_i(m) \rightarrow send_i(m') \Rightarrow m.b < m'.b$
- *pr2*: $\forall m, p_j : deliver_j(m) \rightarrow send_j(m') \Rightarrow m.b < m'.b$; e,

- *pr3*: $\forall$ *m'*, *m"* : *m'.b* = *m".b* $\Rightarrow$ *m'* and *m"* are concurrent.

As propriedades *pr1* e *pr2* seguem diretamente de *CA1* e *CA2*, respectivamente. Em conjunto, implicam que para quaisquer duas mensagens *m*, *m'*: *send(m)* $\rightarrow$ *send(m')* $\Rightarrow$ *m.b* < *m'.b*. A propriedade *pr3* define que duas mensagens difundidas com mesmo número de bloco são necessariamente concorrentes e foram envadas por processos distintos, uma vez que *CA1* veda o envio de dois eventos de *send* executado pelo mesmo processo com o mesmo valor de *BC*.

Matriz de Blocos Causais

Cada processo p_i representa mensagens concorrentes (propriedade *pr3*) enviadas/recebidas com mesmo número de bloco por meio de um bloco causal. A construção destes blocos causais resulta na noção de uma matriz de blocos que é uma forma de representação matricial das mensagens enviadas/recebidas com diferentes números de bloco. Um bloco causal é um vetor de tamanho igual ao número *n* de processos de Π. Sempre que uma mensagem com novo número de bloco, dito *B*, difundida por um processo p_i é enviada ou recebida por um processo, este cria um novo vetor de tamanho *n*; para qualquer mensagem difundida (enviada/recebida) com o número de bloco *B*, define-se a *i-ésima* entrada do vetor em '+'; e, para quaisquer novas mensagens difundidas com mesmo número de bloco *B* enviadas/recebidas de outro processo $p_j, j \neq i$, define-se a *j-ésima* entrada do vetor em '+'. Desta forma, os blocos causais atendem a seguinte propriedade:

- *PR1* : em um mesmo bloco causal, apenas mensagens concorrrentes são representadas.

Em uma aplicação de comunicação em grupo, se a difusão de mensagens pelos processos ocorre de forma contínua, o número de blocos causais construídos incrementará ao longo do tempo. Cada processo ordena seus blocos causais de forma crescente do número de bloco que representam, formando uma matriz de blocos denominada *BM*. De forma que *BM[B]* representa o bloco causal de número de bloco *B*. Para o *BM* associado a quaisquer dos processos, podemos definir a seguinte propriedade:

- *PR2*: considere as mensagens *m* e *m'*, representados em *BM[B]* e *BM[B'*], respectivamente, se há precedência causal entre elas de forma que $m \rightarrow m'$, então $B < B'$.

Por exemplo, a Figura 5.2 representa a matriz de blocos de um processo dos 6 que compõem um dado grupo. Esta representa todas mensagens enviadas ou recebidas por este dado processo que possui a matriz. Por exemplo, a matriz da Figura 5.2 indica que o número de blocos das últimas mensagens recebidas dos processos p_1 e p_2 são, respectivamente, 4 e 5.

Completude de Blocos

De acordo com a propriedade *PR1*, todas mensagens que pertencem a um dado bloco causal, por exemplo, *BM[B]*, são concorrentes. E pela propriedade *PR2*, pode-se implementar um mecanismo de entrega de mensagens que mantenha a ordem causal baseado nos números de

	p_1	p_2	p_3	p_4	p_5	p_6
1	+			+		
2		+			+	+
3	+		+	+		
4	+				+	
5		+				

Figura 5.2. Exemplo de matriz de blocos para um grupo de 6 processos.

bloco. Assim, se uma mensagem recebida representada em *BM[B]*, $B > 1$, é entregue somente depois que todas mensagens difundidas que possam ser representadas em *BM[B']*, para todo $B' < B$, forem previamente entregues.

Se assumimos a hipótese de canais de comunicação par-a-par FIFO, a qual é facilmente implementável por meio de mecanismos de controle de conexão fim-a-fim, como o uso do protocolo TCP, pode-se determinar de forma fácil se um dado bloco causal é completo, ou seja, se o mesmo indica a recepção/envio de todas as mensagens com o mesmo número de bloco associado. Por exemplo, considere a Figura 5.2, bloco 2 está completo em face dos processos p_2, p_5, e p_6 haverem enviado mensagens com número de bloco 2, e os processos p_1, p_3, e p_4, enviado mensagens com número de bloco 3. As condições de completude de um bloco são definidas de forma precisa por meio do Lema 5.1.1 [63, 64]:

Lema 5.1.1 *Um bloco causal BM[B], $B \geq 1$, mantido por um processo p_i, é dito completo, se e somente se $\forall j$, $1 \leq j \leq n$, a seguinte condição se mantém:*

- *LC1: a j-ésima entrada de BM[B] ou (i) está marcada com '+' or (ii) está vazia e existe B', $B' > B$, tal qual j-ésima entrada de BM[B'] está marcada com '+'.*

Prova do Lema 5.1.1 Considere que *LC1* é satisfeito pelo bloco causal *BM[B]*, $\forall j$, $1 \leq j \leq n$. De acordo com (i), p_j contribui enviando previamente uma mensagem indicada em *BM[B]*. De acordo com as regras *CA1* e *CA2*, um processo sempre difunde mensagens com números de bloco monotonicamente crescentes, e, considerando a hipótese de canais de comunicação FIFO, mensagens transmitidas são recebidas na ordem sequencial de seus números de blocos, logo, de acordo com (ii), se p_j difundiu mensagem com número de bloco B' (registrado em *BM[B']*, $B' > B$), é garantido que p_j não difundido previamente (e não difundirá) quaisquer mensagens para *BM[B]*. Desta forma, o bloco causal BM[B] está completo. □

Obviamente, dado *PR2*, a existência de *BM[B]*, $B > 1$, implica na existência de um bloco causal imediatamente anterior *BM[B']*, e que $B' < B$ (e $B' = B - 1$ considerando que o contador de blocos é incrementado pela unidade). Desde que a hipótese de canais de comunicação FIFO se mantém, blocos causais de *BM* mantidos por um processo se completarão na ordem sequencial de seus números de bloco.

É possível utilizar a informação de completude dos blocos para a entrega de mensagens, atendendo a ordenação total: na ordem do número de bloco, mensagens de blocos completos podem ser entregues; em cada bloco completo as mensagens são entregues sempre em uma ordem pré-definida, como, por exemplo, em ordem crescente ao identificador do processo remetente. hipóteses temporais, o que não permite determinar, por exemplo

O mecanismo de blocos causais ora apresentado não apresenta considerações quanto a hipóteses temporais, o que não permite determinar, por exemplo, limites temporais de terminação para a completude de blocos, e, por consequente, para a entrega de mensagens.

Por exemplo, se apenas um processo permanece inativo, sem enviar mensagens, este não contribui com a completude dos blocos causais que se formam, e consequentemente as mensagens associadas não serão entregues. De modo a garantir terminação na completude de blocos causais, utilizamos um simples mecanismo, denominado *time-silence*: este mecanismo garante que todo processo, mesmo em face de longos períodos de inatividade, deve contribuir para a completude de blocos causais existentes. Este mecanismo é descrito a seguir.

Mecanismo de *Time-silence*

Cada processo p_i mantém variáveis inteiras $SENT_i$ e $RECD_i$, indicando o maior número de bloco de mensagens que p_i enviou ou recebeu, respectivamente. Ao receber uma mensagem com um novo número de bloco B, p_i verifica se $SENT_i \geq B$. Se $SENT_i < B$, um *timeout* local é configurado, denominado *time-silence ts(B)*, com um período de tempo pré-determinado *ts*. Este *timeout* indica a janela temporal na qual p_i deve enviar uma mensagem com número de bloco igual ou maior a B. Em face a expiração deste *timeout*, então p_i difunde uma mensagem de conteúdo nulo com número de bloco $m.b. = RECD_i$. Esta ação contribui com a completude de todos os blocos criados $BM[B'], B' \leq RECD_i$. E define uma nova regra CA3 de incremento de BC_i:

- CA3 (Avanço do contador devido ao $timesilence_i$): Antes de p_i difundir uma menssagem de conteúdo nulo, este define $m.b = RECD_i$ e $BC_i = m.b$.

Observe que ao longo da execução se $RECD_i$ torna-se muito maior que $SENT_i$, devido a inatividade do processo, o mecanismo de *time-silence* torna, a termo, $SENT_i = RECD_i$.

Pode-se construir protocolos de comunicação em grupo baseado no mecanismo de blocos causais, os quais possuem garantias de terminação na completude dos blocos em face do uso do *time-silence*.

Contudo, não determinamos limites temporais para esta completude. Se assumimos hipóteses de sincronia no sistema distribuído – taxa de desvio máxima entre os relógios, presença de limites temporais na execução e latência de canais de comunicação, o mecanismo de blocos causais pode utilizar tais hipóteses para determinar estes limites temporais de completude, o que é obtido na extensão apresentada em [5]: blocos causais temporizados, ou *timed causal blocks* (*TimedCB*). Os *timed causal blocks* permitem inferir limites temporais associados à completude de blocos, dado que há limites temporais conhecidos associados ao sistema distribuído.

Em síntese, na presença de um ambiente (processos e canais) síncrono, temos limites temporais conhecidos para a completude dos blocos, denominados *timed causal blocks*, medidos pelo relógio local de p_i, a partir do instante t_i de criação do bloco, são determinados pelo Lema 5.1.2:

Lema 5.1.2 *O limite temporal para um bloco BM[m.b], criado no tempo local t_i, se tornar completo em p_i, considerando canais de comunicação confiáveis, mensurado pelo seu relógio local é:*

- *TC1: $(t_i + ts(m.b) + 2\Delta_{max})(1+\rho)$, se a mensagem m que criou o bloco foi enviada por p_i*
- *TC2: $(t_i + ts(m.b) + 2\Delta_{max} - \Delta_{min})(1+\rho)$, se a mensagem m que criou o bloco foi recebida por p_i*

onde Δ_{max} e Δ_{min} são, respectivamente, os limites temporais máximo e mínimo da transmissão da mensagem através do canal de comunicação entre p_i e p_j e a taxa máxima de desvio entre relógios é dada por ρ.

Prova do Lema 5.1.2 Considere que um bloco causal BM[m.b] é criado por um processo p_i quando este envia ou quando este recebe uma mensagem m e o bloco *BM[m.b]* não existia previamente. Primeiro, se *BM[m.b]* é criado por p_i no instante de tempo local t_i quando p_i envia m. Neste caso, p_i deve aguardar até $t_i + ts(m.b) + 2\Delta_{max}$ para receber uma mensagem (nula ou não) de cada processo p_j com número de bloco $\geq m.b$, $i \neq j$. O primeiro Δ_{max} é o limite temporal máximo para m alcançar p_j, e *ts(m.b)* é o limite temporal máximo para p_j enviar uma mensagem *m'* a *BM[m.b]*, em que $m'.b \geq m.b$[2]. Por fim, o último Δ_{max} é o limite temporal para p_i receber *m'*.

Agora considere o segundo caso, *BM[m.b]* é criado por p_i no tempo local t_i quando p_i recebe m de um processo p_j. Imediatamente após *BM[m.b]* ser criado em p_i, *ts(m.b)* é definido para expitar em $ti + ts(m.b)$ – como um limite temporal para p_i enviar uma mensagem a *BM[m.b]*. Como todos os demais $p_j, i \neq j$, também recebem m (assumindo que os canais de comunicação são confiáveis) e definem *ts(m.b)*, p_i deve aguardar até que $t_i + (\Delta_{max} - \Delta_{min}) + ts(m.b) + \Delta_{max}$. A diferença $(\Delta_{max} - \Delta_{min})$ deve-se ao caso onde p_j pode receber m em Δ_{max} enquanto p_i recebe m em Δ_{min}. Por fim, *ts(m.b)* é o limite temporal máximo para p_j enviar uma mensagem *m'* a *BM[m.b]* e Δ_{max} é o limite temporal para p_i receber *m'*. □

5.1.2.2 O protocolo proposto

A partir do mecanismo de *timed causal blocks* [5], uma avaliação preliminar deste mecanismo é apresentada em [65], um protocolo genérico de comunicação em grupo é proposto em

[2] No pior caso *m'* será gerada pelo mecanismo de *time-silence* se p_j ficar inativo sem enviar nenhuma mensagem de aplicação até expirar o *timeout* dado por *ts(m.b)*.

[8], e, por fim, este é estendido em [35] – contribuições parciais desta tese. Esta abordagem genérica considera um ambiente híbrido (canais de comunicação podem ser *timely* ou *untimely*), onde o algoritmo proposto percebe o contexto do ambiente e adapta-se a este contexto.

Conforme descrito em seções anteriores, é possivel prover um mecanismo de difusão atômica de mensagens que garanta ordem total a partir da informação de completude dos blocos causais. Considere um cenário sem falhas: uma mensagem *m* enviada ao grupo alcança todos os processos destinos, em face de todos estes e do processo emissor não falharem por crash ao longo da execução. Contudo, em caso de falhas, alguns processos podem não receber a mensagem *m*. De modo a garantir as propriedades necessárias à difusão atômica confiável, não podemos assim descartar uma mensagem imediamente após a sua recepção e entrega à camada de aplicação: pode ser necessária sua retransmissão aos processos que não a tenham recebido, de modo a garantir que todos processos do grupo a receberam. Esta garantia implica no **acordo na entrega de mensagens**, o que pode ser obtido por meio da informação de completude dos blocos de acordo com as seguintes condições:

- *safe1*: uma mensagem *m* pode ser entregue se *BM[m.b]* é um bloco completo;
- *safe2*: as mensagens são entregues em ordem crescente de seus números de bloco; uma ordem pré-determinada e fixa de entrega se aplica as mensagens de mesmo número de bloco.

A entrega baseada na completude dos blocos, utilizada na versão do protocolo proposta em [8], não atende aos requisitos de diversas aplicações, uma vez que o **acordo na entrega de mensagens** apenas se aplica a processos corretos. Considere, por exemplo, o cenário de replicação ativa de servidores [48]: se apenas uma réplica recebe um comando e o executa imediatamente antes de falhar, esta réplica evolui a um estado distinto das demais réplicas, e na sua recuperação a sincronização de estados é mais custosa em face da réplica defeituosa ter uma trajetória distinta das demais.

A solução para isto é o **acordo uniforme na entrega de mensagens**, ou seja, a entrega mesmo em processos que falham por *crash* ao longo da execução, só deve ocorrer se também for entregue em processos corretos. No exemplo da replicação ativa de servidores, a réplica mesmo imediatamente antes de falhar só executaria ações também entregues a processos corretos, mantendo-se a mesma trajetória na evolução do estado.

Pode-se obter este comportamento de acordo uniforme por meio da estabilidade das mensagens. Uma mensagem é estável se for recebida por todos os membros do grupo (e instável caso contrário). A entrega de mensagens somente após as mesmas tornarem-se estáveis atende ao **acordo uniforme na entrega de mensagens**, e as mesmas só devem ser descartadas se forem conhecidas estáveis por todos os processos (super-estáveis).

Acordo Uniforme na Entrega de Mensagens. Para determinar a estabilidade das mensagens associadas a um bloco causal, membros do grupo informam em cada mensagem transmitida o último bloco completo (*last complete block* – LCB). Por meio da coleta do LCB informado por

todos processos, um processo p_i pode calcular o último bloco estável (*last stable block* – LSB), como $LSB = min\{LCB_j, \forall p_j \in g\}$. Ou seja, todos os blocos de p_i de número menor ou igual a *LSB* são estáveis em p_i.

As seguintes condições de entrega garantem a entrega do mesmo conjunto de mensagens na mesma ordem de acordo com o **acordo uniforme na entrega de mensagens**:

- *stable1*: uma mensagem *m* pode ser entregue se *BM[m.b]* é um bloco estável;
- *stable2*: as mensagens são entregues em ordem crescente de seus números de bloco; uma ordem pré-determinada e fixa de entrega se aplica as mensagens de mesmo número de bloco.

A terminação na entrega, garantida pelo mecanismo de *time-silence* deve atuar de modo que todos blocos completos se tornem a termo estáveis. Ou seja, após a completude de um bloco, define-se um *timeout* com janela temporal ts, e na sua expiração uma mensagem de conteúdo nulo informa o novo valor de *LCB* a todos processos. Neste caso, a mensagem nula não possui número de bloco e não cria novos blocos.

O Lema 5.1.3 define os limites temporais para a estabilidade de blocos na presença de um ambiente (processos e canais) síncrono:

Lema 5.1.3 *O limite temporal para um bloco BM[m.b], criado no tempo local t_i, se tornar estável em p_i, mensurado pelo seu relógio local é:*

- *ST1:* $(t_i + 2ts(m.b) + 3\Delta_{max})(1+\rho)$, *se m foi enviada por* p_i
- *ST2:* $(t_i + 2ts(m.b) + 3\Delta_{max} - \Delta_{min})(1+\rho)$, *se m foi recebida por* p_i

Prova do Lema 5.1.3 Considere o Lema 5.1.2, uma vez que o bloco $B[m.b]$ é criado no instante de tempo local t_i no processo p_i, o qual enviou m, a sua completude ocorre em $TC1$, e em $TC2$ em um processo p_j que recebeu m. Desconsiderando o efeito do *clock drift* (taxa de desvio), de fato, $TC1$ e $TC2$ ambos referem-se ao mesmo instante de tempo global t (mas não conhecido localmente), porque no limite p_j receberá m em Δ_{max}, logo $t_j = t_i + \Delta_{min}$, medido pelo relógio local de p_i, mas para outro processo p_k que recebeu o bloco em $t_i + \Delta_{max}$, e contribui para a sua completude em p_j somente após o *time-silence* $t.s.(m.b)$ e o atraso máximo de transmissão Δ_{max}, logo medindo pelo relógio local de p_i: $t_j + ts(m.b) + 2\Delta_{max} - \Delta_{min}) = (t_i + \Delta_{min}) + ts(m.b) + 2\Delta_{max} - \Delta_{min}) = (t_i + ts(m.b) + 2\Delta_{max}) = TC1$. Logo $TC2$ medido pelo relógio local de p_j é $TC1$ medido pelo relógio local de p_i.

Assim, após os limites temporais para completude de blocos, uma mensagem para notificar tal completude será enviada no máximo em $ts(m.b)$ (*time-silence*) e recebida após, no máximo, Δ_{max}, de modo que os limites para estabilidade serão obtidos após $ts(m.b) + \Delta_{max}$ dos limites para completude. □

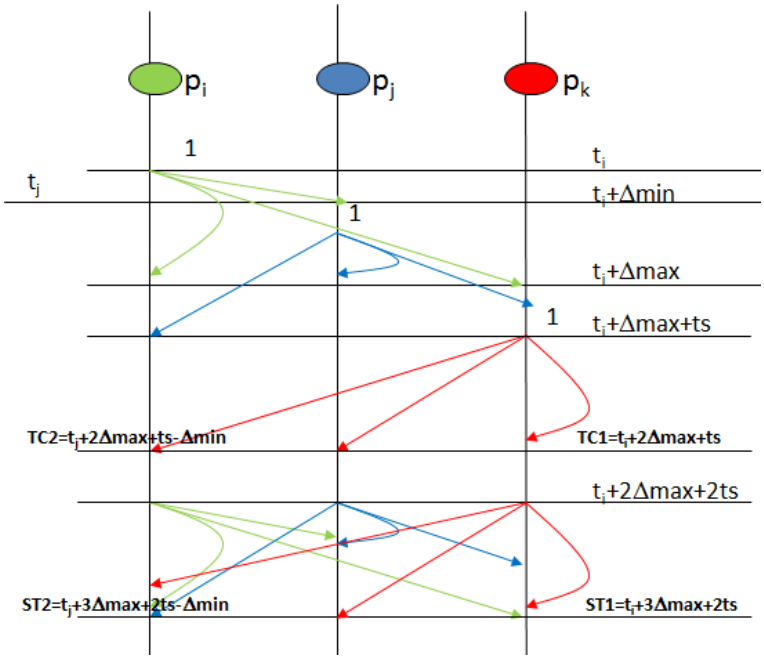


Figura 5.3. Limites temporais para completude e estabilidade do mecanismo *timed causal blocks*.

A Figura 5.3 ilustra de forma gráfica os limites temporais para completude ($TC1$ e $TC2$) e estabilidade ($ST1$ e $ST2$) calculados pelo instante de tempo local a cada processo em que o bloco é criado, considerando o caso de p_i que cria o bloco ao enviar m e de p_j que cria o bloco ao receber m.

A visão do grupo evolui em face a falhas de processos por *crash*, ou de requisições espontâneas de processos de entrada (*join*) ou saída (*leave*) do grupo. Consideremos o caso de falhas. Se assumimos que processos e canais de comunicação são síncronos, a completude e estabilidade de blocos deve ocorrer de acordo com os limites temporais apresentados nos Lemas 5.1.2 e 5.1.3, caso não ocorra temos uma indicação de falha do processo que não contribuiu para a completude ou estabilidade dos blocos.

Suponha que canais de comunicação sejam assíncronos, podemos arbitrar um valor razoável para Δ_{max}, de modo que a medição da latência dos canais de comunicação tenha alta probabilidade de atender a tal limite temporal. Neste caso assíncrono, a violação dos limites temporais de completude e estabilidade de blocos indica tão somente uma suspeita da falha do processo. Em face à indicação ou mesma a suspeita da falha de um processo do grupo, é necessária a instalação de uma nova visão, com a entrega prévia das mensagens instáveis ainda não entregues ao grupo.

No caso de requisições de *join* e *leave*, podemos utilizar o conceito de estabilidade de blocos. Ou seja, o processo que quer ingressar ou sair do grupo difunde a requisição ao grupo (no caso de *join* o envio é feito por meio de um processo que já é membro do grupo). Uma vez que a requisição (*join* ou *leave*) é entregue estável uma nova visão é instalada a partir do número do bloco associado à requisição.

Esta abordagem obedece a sequência única de visões, uma vez que toda requisição de *join*

ou de *leave* deve ocorrer na mesma visão $v^r(g)$ para quaisquer dois processos p_j e p_k pertencentes ao grupo. Associando a sequência de visões à sequência de blocos estáveis, uma vez que a estabilidade garante que a informação é conhecida por todos processos e os blocos tornam-se estáveis na mesma sequência em todos processos do grupo.

Algoritmos Os algoritmos que detalham o funcionamento do procoloco são descritos a seguir. O Algoritmo 5.1 é ativado por um evento de envio/recepção de uma mensagem *m*, definindo *timeouts* de completude e estabilidade na criação de blocos, armazenando a mensagem *m* em um *buffer* local e ativando a tarefa de entrega (Algoritmo 5.2). Esta tarefa entregará mensagens estáveis de acordo com as condições *stable1* e *stable2*. Suponha que um processo p_k falha por *crash* e, como consequência, expira o *timeout* em p_i associado a completude de *BM[m.b]*. De modo a prover a entrega de mensagens, uma nova visão do grupo *g* deve ser instalada excluindo p_k (e quaisquer outros processos falhos).

Algoritmo 5.1: Protocolo proposto para comunicação em grupo self-aware: executado por um processo p_i em face de um evento de *send/receive* de uma mensagem *m*.

```
1:  if BM[m.b] does not exist then
2:      create BM[m.b]
3:      if p_i = m.sender then
4:          set completion timeout TC1 for BM[m.b]
5:          set stability timeout ST1 for BM[m.b]
6:      else
7:          set completion timeout TC2 for BM[m.b]
8:          set stability timeout ST2 for BM[m.b]
9:      end if
10: end if
11: store m at a local buffer and signal delivery task (Algorithm 5.2)
```

Algoritmo 5.2: Protocolo proposto para comunicação em grupo self-aware: tarefa de entrega de mensagens.

```
1:  updates LCB and LSB according to the current view and cancels related timeouts for complete or stable
    causal blocks
2:  while exists (stable and not delivered causal blocks) do
3:      deliver stable messages with the lowest block number according to stable-safe1 and stable-safe2
4:      if exists ( join(g) or leave(g) requests delivered in current view v_i^k ) then
5:          apply the requests and compound a new view v_i^{k+1}
6:          if p_i ∉ v_i^{k+1} then
7:              terminate p_i (* p_i requested previously to leave *)
8:          else if v_i^k ≠ v_i^{k+1} then
9:              install the new view v_i^{k+1} at p_i
10:             updates LCB and LSB according to the current view
11:         end if
12:     end if
13: end while
```

Logo, de modo a garantir a instalação desta nova visão, executamos uma difusão confiável da mudança de visão associada ao número de bloco expirado, *rmcast(ChangeViewRequest,B)*,

onde $B = m.b$, de acordo com o Algoritmo 5.3. Todos os processos remanescentes devem receber tal difusão e executar a tarefa de mudança de visão (Algoritmo 5.4).

Enfim, a tarefa de mudança de visão utiliza de difusão confiável para disseminar entre todos os processos do grupo o valor do *LSB* e o conjunto de mensagens instáveis conhecidas por cada processo. Desta forma, todas as mensagens instáveis não ainda entregues serão conhecidas e a partir do cálculo do máximo valor do *LSB* coletado (LSB_{max}), pode-se inferir quais blocos são estáveis em um processo, mas não conhecidos como estáveis em outro.

Uma condição de guarda define um quórum adequado para o conjunto de processos em funcionamento: este quórum considera a qualidade de serviço de processos e canais de comunicação baseado nas informações atualizadas sobre os conjuntos *live*, *down* e *uncertain*. Um algoritmo de consenso adaptativo apresentado em [6] é utilizado para obter o acordo no conjunto de processos ativos, no valor de LSB_{max} e no conjunto de mensagens instáveis a ser entregue, de modo a determinar visões idênticas a ser instalados em todos os membros do grupo (propriedade **acordo uniforme** do consenso). O consenso adaptativo progride de acordo com a qualidade de serviço existente. Ao fim do consenso, o conjunto de mensagens instáveis é armazenado no buffer local, o valor de *LSB* é atualizado e as mensagens são entregues de acordo com as condições de entrega *stable1* e *stable2*.

Se um processo p_i não foi incluído na nova visão (em face de não ter enviado seu conjunto de mensagens instável), o mesmo é terminado (linha 12 do Algoritmo 5.4). A nova visão é instalada se é diferente da visão atual (linhas 13-14 do Algoritmo 5.4), caso contrário, somente as mensagens instáveis são entregues sem a necessidade de nova visão.

Algoritmo 5.3: Protocolo proposto para comunicação em grupo self-aware: executado por um processo p_i em face da expiração de um timeout relacionado a um bloco B.

```
1: rmcast(ChangeViewRequest,B)
```

Algoritmo 5.4: Protocolo proposto para comunicação em grupo self-aware: executado por um processo p_i em face da recepção de uma requisição de mudança de visão *(ChangeViewRequest,B)*.

1: **if** *(unstable,B,LSB, leaving$_i$) was already been sent by p_i* **then**
2: *exit*
3: **end if**
4: *block ordinary delivery at delivery task*
5: **if** *p_i requested a leave with no join request after that* **then**
6: $leaving_i = true$
7: **else**
8: $leaving_i = false$
9: **end if**
10: *rmcast(unstable, B, LSB, leaving$_i$)*
11: *wait until ($\forall p_j \in \Pi$: received (unstable,B,LSB) from p_j or $p_j \in down_i$ or $FD_i(p_j) = true$) and for majority of uncertain: received (unstable,B,LSB,leaving$_j$) from p_j*
12: *let allunstable$_i$ be the union of the unstable sets received from all p_j and LSB_{max} the maximum value of LSB collected.*
13: *let V_i be set of all p_j from which (unstable,B, LSB, false) was received.*
14: *consensus(B,(V,allunstable$_i$,LSB_{max}))*
15: *store messages from allunstable not yet received by p_i, sets $LSB = LSB_{max}$ and invokes Delivery Task.*
16: $v_i^{r+1} = V \cup v_i^r - v_i^k$ (* v_i^r *is the current view after Delivery Task* *)
17: **if** $p_i \notin v_i^{r+1}$ **then**
18: *terminate p_i (* p_i was removed due to a false suspicion from a $p_j, i \neq j$ *)*
19: **else if** $v_i^{r+1} \neq v_i^r$ **then**
20: *install the decided view V at p_i as v_i^s where $s = r+1$*
21: **end if**
22: *signal delivery task (Algorithm 5.2) for resuming ordinary message delivery*

Devemos observar que em um cenário síncrono, a expiração de timeout associado à completude de um bloco corresponde a um detector perfeito de defeitos. Caso os canais de comunicação sejam assíncronos, temos tão somente uma suspeita da falha do processo. Em ambientes cuja qualidade de serviço varia ao longo do tempo, o nível de serviço provido pela detecção de defeitos varia, conforme o subconjunto de processos pertencentes aos conjuntos $live$, $down$ e $uncertain$.

Em configuração híbrida, não necessariamente dinâmica, a aplicação também terá a execução otimizada. Considere o cenário de nuvem federada [2], ou seja, *clusters* de nós conectados entre si por meio da Internet. Cada *cluster* da nuvem representa um grupo de computadores interconectado em uma rede local, onde pode-se assumir um comportamento síncrono, contudo a nuvem como um todo não é um sistema síncrono. Esta configuração é representada pelo modelo síncrono particionado [7], e podemos utilizar o algoritmo associado ao detector de defeitos perfeito neste ambiente para a determinação do conjunto *down*: a expiração de bloco B em p_i devido a não contribuição de um processo p_j localizado na mesma partição síncrona de p_i indica a certeza da falha de p_j. Neste cenário, verificada a cobertura do detector perfeito por todos os processos, a expiração de bloco envolvendo os processos p_i e p_j em diferentes parti-

ções síncronas (e portanto com comunicação assíncrona entre si), ou seja, associada a suspeita de falha, não torna necessária a execução da mudança de visão.

5.1.2.3 Avaliação de desempenho

O *framework* **HDDSS** proposto nesta tese é adequado para esta e as demais avaliações de desempenho ora apresentadas pela facilidade de composição de cenários com componentes (processos e canais de comunicação) com diferentes níveis de qualidade de serviço, bem como uso de funções de probabilidade adequadas que caracterizem, dentre outros, a carga do sistema e hipóteses de falhas associadas aos componentes. Ainda, podemos caracterizar um conjunto de variáveis locais a cada processo que define informação sensoriada e que os valores variam de acordo com a dinâmica do sistema. Esta funcionalidade nos permite melhor representar o comportamento de sistemas distribuídos self-aware que caracterizamos neste capítulo.

A avaliação de desempenho compara o nosso protocolo com abordagens clássicas nos casos síncrono e assíncrono. No caso síncrono, comparamos com o *Periodic Group Creator* [28, 61] e no caso assíncrono com o protocolo de comunicação em grupo do sistema operacional distribuído *Amoeba* [66].

Os protocolos foram escolhidos por sua simplicidade. O *Periodic Group Creator* utiliza a informação dos relógios físicos locais para a entrega de mensagens, baseado na existência de sincronização periódica. O *Amoeba* utiliza um sequenciador fixo que determina a ordem da entrega de mensagens.

O *Periodic Group Creator* utiliza de relógios físicos para prover um protocolo de comunicação em grupo baseado no histórico de comunicação, em comparativo a nossa abordagem que utiliza *timed causal blocks*, combinando a informação de relógios físicos (dado pelos timeouts associados aos blocos) e de relógios lógicos (números de blocos associados as mensagens). Quanto ao *Amoeba*, o mesmo implementa um paradigma de sequenciador fixo, em que um processo líder define a sequencia das mensagens enviadas ao grupo.

Para simplicidade, denominamos os protocolos acima tão somente de *PGC* e *Amoeba*, respectivamente, e nosso protocolo de *TimedCB*.

Caso Síncrono

Modelo de Simulação

De modo a caracterizar um ambiente síncrono, configuramos um cenário no qual há uma latência máxima pré-determinada para a comunicação fim-a-fim, o qual pode ser estabelecido por meio de um comutador de rede Ethernet com níveis de qualidade de serviço configurados fim-a-fim, de modo que obtemos uma latência máxima $\Delta_{max} = 15ms$ – assumimos uma latência mínima $\Delta_{min} = 1ms$ e que o atraso dos canais de comunicação é muito maior que a latência de processamento. Cada processo tem acesso a um relógio físico local, e estes mantêm uma taxa de desvio máxima limitada por ρ_{max} da ordem de 10^{-6}. Para operação do protocolo PGC há a execução subjacente de um algoritmo de sincronização de relógios [67], cuja execução

periódica limita a distorção ε entre os relógios dos processos.

Métricas de Desempenho

Para avaliar o desempenho de um protocolo de comunicação em grupo, utilizamos duas métricas: o *overhead* do protocolo, dado pela carga adicional imposta a rede, ou seja, a quantidade de mensagens de controle em comparativo com o total de mensagens transmitida; e a *latência na entrega de mensagens*, este medido da recepção da mensagem pelo processo à entrega correspondente pela camada de aplicação.

O *overhead* do protocolo está associado a necessidade de mensagens de controle para, por exemplo, manter o *membership* na presença de falhas, ou instalar uma nova visão. Já o atraso na entrega de mensagens pode ser devido a necessidade de se observar condições de guarda que garantam, por exemplo, a estabilidade da mensagem, assegurando o **acordo uniforme na entrega de mensagens**.

Descrição dos Experimentos

No protocolo *PGC*, cada processo periodicamente envia mensagens para verificação do *membership* com período π. O algoritmo de difusão atômica confiável se basea em *flooding* e a entrega de mensagens utiliza a informação dos relógios físicos, considerando-se a latência máxima da rede, o número máximo de retransmissões k e a distorção máxima ε entre os relógios sincronizados – esta baseada na configuração do protocolo de sincronização de relógios físicos utilizado.

Devemos observar que, na ausência de mensagens de aplicação, o *TimedCB* utiliza o mecanismo de *time-silence* para garantir a completude de blocos a cada período ts. Este mecanismo nos permite monitorar o *membership* de forma similar ao *PGC*, e, desta forma, pode-se comparar os protocolos em cenários que $ts = \pi$. Consideramos que o mecanismo de *flooding* do protocolo *PGC* tolera tão somente a falha de até 2 processos no caminho ($k = 2$, devido a retransmissões) e que faz uso do mecanismo de sincronização de relógios. No *TimedCB* não é necessária a sincronização de relógios, uma vez que os limites temporais são todos estimados a partir do relógio físico local a cada processo.

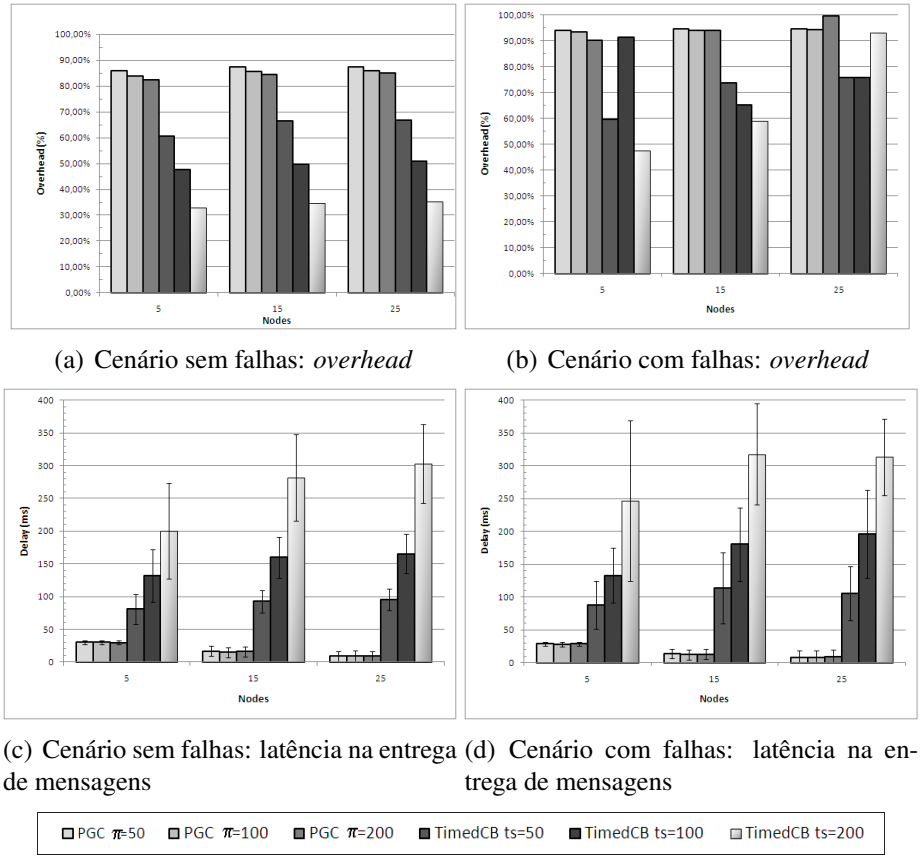
Os fatores de simulação considerados são o tamanho do grupo (5, 15 e 25) e o período ts e π (50, 100 e 200ms). De forma a simular a geração de mensagens de aplicação, caracterizamos a carga de trabalho em cada processo por meio de uma função de distribuição de probabilidade de Bernoulli, de modo que simulamos uma taxa média de 10 mensagens enviadas por cada processo ao grupo a cada segundo. As simulações são realizadas em cenários com e sem falhas.

Resultados e Discussão

Analisando os resultados do *overhead* – Figura 5.4.a e 5.4.b, verifica-se que o protocolo TimedCB apresenta, na maioria dos casos, *overhead* menor ou igual ao PGC. Conforme esperado, quanto maior o período (ts do *TimedCB* e π do *PGC*), e consequentemente menor a frequência de envio de mensagens de controle, menor será o *overhead*. Deve-se observar que na presença de falhas, quando é executado o consenso, o protocolo *TimedCB* aumenta seu *overhead*, ainda que mantenha-se menor que o *PGC*.

Uma vez que o *TimedCB* utiliza *piggy-backing* para portar informações de controle do protocolo nas mensagens de aplicação, o mesmo apresenta menor *overhead*. Mesmo em face de falhas quando o *TimedCB* executa o consenso para mudança de visão, verifica-se que este é mais eficiente que o *PGC*, que mantém o *overhead* constante, determinado pelo período de monitoramento, pela retransmissão do mecanismo de difusão atômica confiável e pelo mecanismo de sincronização de relógios. O *TimedCB* por sua vez somente gera mensagens de protocolo na ausência de mensagens de aplicação (mecanismo de *time-silence*) e na ocorrência de falhas (consenso).

Em análise dos resultados para a latência na entrega de mensagens – Figura 5.4.c e 5.4.d (média com o respectivo desvio padrão indicado em linha de intervalo), o *PGC* com a resiliência escolhida ($k = 2$) se apresenta mais eficiente que o *TimedCB*, com comportamento similar nos cenários com e sem falhas. Pode-se observar que o *TimedCB* apresenta um leve aumento nas latências na presença de falhas, em função do custo de execução do consenso para a entrega das mensagens não estáveis.



(a) Cenário sem falhas: *overhead*

(b) Cenário com falhas: *overhead*

(c) Cenário sem falhas: latência na entrega de mensagens

(d) Cenário com falhas: latência na entrega de mensagens

Figura 5.4. Gráficos dos resultados de simulação de *TimedCB* e *PGC* em cenário síncrono.

Contudo, o atraso de entrega do TimedCB pode ser otimizado para menores períodos de monitoramento (parâmetro ts do mecanismo de *time-silence*) ou em presença de maior carga de trabalho da aplicação. Ainda, deve-se notar que esta incrementa no PGC com o aumento da resiliência da difusão atômica confiável ($k > 2$). Para o *TimedCB*, o custo sempre é o mesmo

(o da execução do consenso), não importa o número de falhas toleradas (de 1 a $n-1$). Desta forma, uma maior resiliência (tolerância de maior quantidade de processos sujeitos a falhas) pode ser obtida de forma mais otimizada pelo protocolo *TimedCB*. Na ausência de falhas, o protocolo *TimedCB* se adapta e nenhum preço adicional é pago.

Caso Assíncrono

Modelo de Simulação

Um ambiente totalmente assíncrono em que processos imersos na Internet se comunicam entre si por meio de canais de comunicação com latência determinada por meio de uma função de distribuição de probabilidade exponencial ($mean = 20ms$).

Métricas de Desempenho

As métricas de desempenho são as mesmas utilizadas no caso síncrono: o *overhead* do protocolo, dado pela carga adicional imposta a rede; e a *latência na entrega de mensagens*, este medido da recepção da mensagem pelo processo à entrega correspondente pela camada de aplicação.

Descrição dos Experimentos

O protocolo proposto é avaliado em comparação a uma versão do *Amoeba* que provê acordo uniforme na entrega de mensagens. Neste protocolo, uma mensagem ao grupo é enviada inicialmente ao sequenciador, o qual a difunde ao grupo; os processos do grupo precisam enviar o ACK da mensagem ao sequenciador o qual espera por uma maioria (quórum) antes de enviar uma mensagem de controle permitindo a entrega. Considerando a natureza assíncrona do cenário, e dado o valor escolhido para a latência média dos canais de comunicação ($20ms$) com comportamento exponencial, estimamos um valor de $\Delta_{max} = 100ms$, a ser utilizado pelo *TimedCB* para suspeita de falha na expiração de um bloco causal.

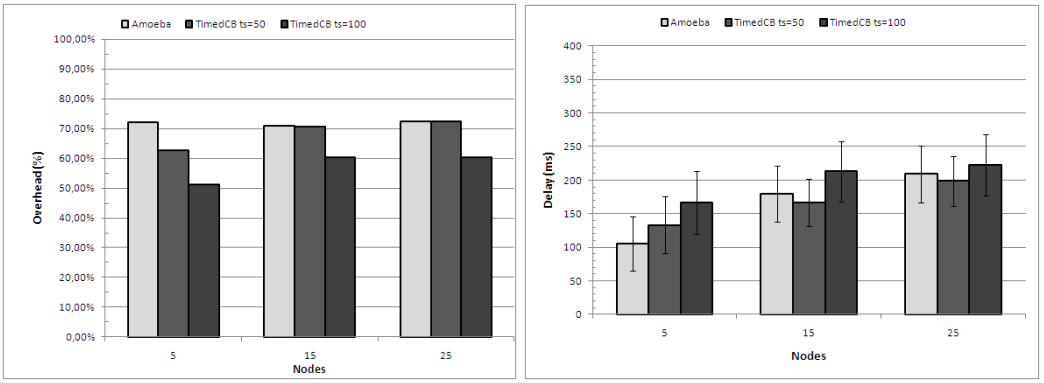
Os fatores de simulação considerados são o tamanho do grupo (5, 15 e 25) e o período ts (50 e $200ms$). De forma a simular a geração de mensagens de aplicação, caracterizamos a carga de trabalho em cada processo por meio de uma função de distribuição de probabilidade de Bernoulli, de modo que simulamos uma taxa média de 10 mensagens enviadas por cada processo ao grupo a cada segundo. As simulações são realizadas no cenários sem falhas.

Resultados e Discussão

Analisando os resultados do *overhead* e da latência na entrega de mensagens – Figura 5.5, verifica-se que o Amoeba produz *overhead* constante, enquanto o *overhead* do *TimedCB* varia conforme a escolha do parâmetro ts e a carga de trabalho imposta. Quanto a latência na entrega de mensagens, os valores de ambos protocolos são similares.

5.1.3 Desenvolvimento de um protocolo de comunicação em grupo auto-configurável

Em cenários bastante dinâmicos de computação elástica, como, por exemplo, os de computação em nuvens [68], usuários são tarifados de acordo com a quantidade de recursos computacionais que as aplicações consomem, como, por exemplo, rede de computadores, processa-



(a) Cenário sem falhas: *overhead*

(b) Cenário sem falhas: latência na entrega de mensagens

Figura 5.5. Gráficos dos resultados de simulação de *TimedCB* e *Amoeba* em cenário assíncrono.

mento, armazenamento. Neste contexto, mecanismos subjacentes (como protocolos de comunicação em grupo), devem estar alinhados a estes requisitos e auto-ajustarem o uso de recursos de acordo com mudanças na carga de trabalho da nuvem ou nos requisitos da aplicação da nuvem. Este ajuste deve ser realizado por meio da configuração dinâmica dos parâmetros de operação, que em ambientes estáticos é feita em tempo de desenho de acordo com o ponto de operação estimado.

Desta forma, há a motivação do desenvolvimento de protocolos distribuídos que possam auto-gerenciar sua configuração, se ajustando às condições operacionais do ambiente ao longo da execução e aos requisitos de operação definidos pela aplicação usuária. Por exemplo, suponha um ambiente de computação em nuvem onde um módulo de gerenciamento de níveis de serviço denominado *SLA Manager* define o nível de serviço negociado para a aplicação distribuída que executa na nuvem. Como uma ilustração, considere o esboço de um gerenciador de recursos da nuvem, *Resource Cloud Manager* (esboço no Algoritmo 5.5), o qual de acordo com o SLA atual, requisita mudanças nos pontos de operação (*set-point*) para o consumo de recursos dos protocolos subjacentes, como, por exemplo, de um protocolo de comunicação em grupo auto-gerenciável, o qual é utilizado em todas as réplicas em execução. Uma mudança neste *set-point* pode implicar no aumento de consumo de recursos, provendo entrega de mensagens mais rápida, ou, ao contrário, reduzir o consumo de recursos para as aplicações associadas, implicando em maior retardo na entrega das mensagens.

Algoritmo 5.5: Esboço de um pseudo-algoritmo para prover adaptação a níveis de serviço de um gerenciador de recursos da nuvem.

```
1 let cm be a instance of ResourceCloudManager;
2 let rm be a instance of ReplicationManager;
3 on event receive of ⟨SLARenegotiation⟩ request by cm do
4     forall the ⟨rm ∈ CloudApplication⟩ do
5         trigger ⟨GroupProtocol, ChangeSetPoint⟩;
6         do other stuff...;
```

O protocolo proposto é uma variação do protocolo base apresentado na Seção 5.1.2.2 construído sob o mecanismo de *timed causal blocks*. Este protocolo base é provido com um mecanismo que o torna auto-gerenciável, percebendo o nível de serviço requerido para o usuário, em termos de consumo de recursos, e ajustando os seus parâmetros operacionais.

A seguir descrevemos, de forma breve, o protocolo base proposto, a abordagem auto-gerenciável e uma avaliação de desempenho comparativa entre o protocolo base e a abordagem auto-gerenciável.

5.1.3.1 O protocolo base

Assumimos um modelo de sistema parcialmente síncrono, com hipótese *GST*. Ou seja, a termo o sistema torna-se estável, sendo mantidos limites temporais que permitem a terminação da computação distribuída. Sob estas premissas de modelo, construímos um protocolo de comunicação em grupo básico, variação de [35],

O protocolo base é uma variação do protocolo da Seção 5.1.2.2 adaptado a este modelo de sistema parcialmente síncrono. Esta variação assume que toda a mudança de visão requer uma maioria dos processos ativos, em face do sistema ser parcialmente síncrono com hipótese *GST* e não haver um mecanismo subjacente que permita inferir qualidade de serviço dos processos do grupo. Os limites temporais de completude e estabilidade de blocos indicam mera suspeita de falha, a partir da hipótese que Δ_{max} indica com alta probabilidade um valor razoável para a latência máxima a ser observada nos canais de comunicação.

Desta forma, mantemos o Algoritmo 5.6 que no envio/recepção de uma mensagem *m* define *timeouts* de completude e estabilidade, armazena a mensagem *m* e ativa a tarefa de entrega (Algoritmo 5.7). Este entrega mensagens estáveis de acordo com as condições *stable1* e *stable2*. Em caso de suspeita devido a expiração de blocos o Algoritmo 5.8 ativa a mudança de visão.

O Algoritmo 5.9 de mudança de visão assume um ambiente assíncrono, requerendo desta forma um quórum de maioria de processos corretos na visão atual, e executa o protocolo de consenso baseado em um detector de defeitos $\diamond\mathcal{S}$ apresentado em [58] – o qual garante a terminação do consenso sob a hipótese *GST*. Este consenso assume como quórum uma maioria de processos ativos. De modo a obter o acordo em uma visão idêntica a ser instalada em todos os membros do grupo. A nova visão é instalada em todos os membros do grupo, exceto em um processo suspeito de forma errônea que for excluído desta nova visão, neste caso o mesmo é terminado.

5.1.3.2 O protocolo proposto

A proposta auto-gerenciável estende o protocolo de comunicação em grupo base apresentado, de modo a tratar da escolha apropriada do parâmetro de configuração do protocolo (no caso, o *time-silence*). Escolher o valor apropriado do *time-silence* implica em uma relação de compromisso: a) períodos longos para *time-silence* diminuem o *overhead* do protocolo e

Algoritmo 5.6: Protocolo base proposto para comunicação em grupo: executado por um processo p_i em face de um evento de *send/receive* de uma mensagem m;

```
1:  if BM[m.b] does not exist then
2:      create BM[m.b]
3:      if p_i = m.sender then
4:          set completion timeout TC1 for BM[m.b]
5:          set stability timeout ST1 for BM[m.b]
6:      else
7:          set completion timeout TC2 for BM[m.b]
8:          set stability timeout ST2 for BM[m.b]
9:      end if
10: end if
11: store m at a local buffer and signal delivery task (Algorithm 5.2)
```

Algoritmo 5.7: Protocolo base proposto para comunicação em grupo: tarefa de entrega de mensagens.

```
1:  updates LCB and LSB according to the current view and cancel related timeouts for complete or stable
    causal blocks
2:  while exists a causal block stable and not delivered do
3:      deliver stable messages with the lowest block number according to stable-safe1 and stable-safe2
4:      if exists requests join(g) or leave(g) delivered in current view v_i^k then
5:          apply the requests and compound a new view v_i^{k+1}
6:          if p_i ∉ v_i^{k+1} then
7:              terminate p_i (* p_i requested previously to leave *)
8:          else if v_i^k ≠ v_i^{k+1} then
9:              install the new view v_i^{k+1} at p_i
10:             updates LCB and LSB according to the current view
11:         end if
12:     end if
13: end while
```

o consumo de recursos, contudo, podem determinar um tempo de bloqueio maior na entrega de mensagens, pois caso um processo não envie mensagem de aplicação para completude dos blocos, este só contribui para a completude após o período dado pelo *time-silence*; b) no outro lado, períodos curtos de *time-silence* podem reduzir o tempo de bloqueio, porém ao custo do aumento do *overhead* e do consumo de recursos, o que pode ser problemático quando o ambiente estiver sujeito a alta carga de trabalho, implicando em aumento do atraso fim-a-fim e tempo de bloqueio na entrega de mensagens.

A abordagem, contribuição da tese [17], propõe o uso de teoria de controle realimentado [69] para o ajuste de parâmetros de protocolos de computação distribuída, de acordo com um dado nível de serviço. A ideia básica do controle realimentado é que um controlador utilize de sensores que observam as saídas atuais de um objeto controlado (ou planta), e executa um algoritmo de controle (ou lei de controle) para definir novos valores de entrada de modo que as saídas sigam o valor de referência (*set-point*) para o comportamento desejado [70].

Neste caso, a planta controlada é o protocolo de comunicação baseado no mecanismo de *timed causal blocks*. O **sensor** coleta e pré-processa dados sobre o ambiente computacional e desempenho do protocolo de comunicação em grupo, como, por exemplo, latência fim-a-fim dos

Algoritmo 5.8: Protocolo base proposto para comunicação em grupo: executado por um processo p_i em face da expiração de um timeout relacionado a um bloco B.

```
1: rmcast(ChangeViewRequest,B)
```

Algoritmo 5.9: Protocolo base proposto para comunicação em grupo: executado por um processo p_i em face da requisição de mudança de visão *(ChangeViewRequest,B)*.

```
1: if (unstable,B,LSB) was already been sent by p_i then
2:     exit
3: end if
4: block ordinary delivery at delivery task
5: rmcast(unstable, B,LSB)
6: wait until (∀p_j ∈ v_i^k: received (unstable,B,LSB) from the majority of processes in view v_i^k) and (∀p_j ∈ v_i^k:
   either received (unstable,B,LSB) from p_j or FD(p_j) = true)
7: let allunstable_i be the union of the unstable sets received from all p_j and LSB_max the maximum value of LSB
   collected.
8: let v_i^{k+1} be set of all p_j from which (unstable,B) was received.
9: consensus(B,(v_i^{k+1}, allunstable_i, LSB_max))
10: store messages from allunstable not yet received by p_i, sets LSB = LSB_max and apply stable-safe1 and
    stable-safe2 to blocks that get stable.
11: if p_i ∉ v_i^{k+1} then
12:     terminate p_i (* p_i was removed due to a false suspicion from a p_j, i ≠ j *)
13: else if v_i^k ≠ v_i^{k+1} then
14:     install the decided view v_i^{k+1} at p_i
15: end if
16: signal delivery task (Algorithm 5.2) for resuming ordinary message delivery
```

canais de comunicação, *overhead* e carga de trabalho. O **controlador** utiliza desta informação para ajustar de forma dinâmica o *time-silence* de acordo com o valor de referência. Este valor de referência é definido em termos do nível de consumo de recursos contratado pela aplicação. A Figura 5.6 apresenta uma visão simplificada do laço de controle.

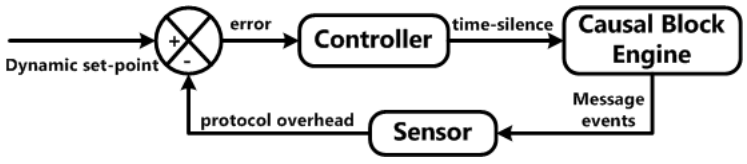


Figura 5.6. Laço de controle para auto-configuração do protocolo de comunicação em grupo proposto.

Deve-se observar que não é possível determinar de forma exata a quantidade de recursos disponíveis no ambiente. Desta forma, o consumo de recursos é estimado por meio de uma metáfora que associa as latências do ambiente aos recursos disponíveis em um dado momento: a maior disponibilidade de recursos (como largura de banda, memória, processamento etc.) produziria menores latências de comunicação e processamento, e vice-versa.

Uma vez que o ambiente é assíncrono, não conhecemos previamente as latências máxima e mínima do sistema, Δ_{max} e Δ_{min}, e devemos, por meio de observação do sensor, obter as estimativas δ_{max} e δ_{min}.

A partir destes valores observados, e de uma estimativa da latência de comunicação atual do sistema, construímos a metáfora do consumo de recursos do sistema, utilizando uma relação linear entre a variação da latência observada, dita *jitter* e a quantidade de recursos consumidos *rc* do ambiente, representada na Figura 5.7: definimos o $rc_{max} = 1$ e o $rc_{min} = 0$, se *rc* é alto, proxímo de 1, supomos que, como o consumo é alto, há poucos recursos de comunicação e processamento disponíveis, com latências próximas do máximo observado; já se *rc* é baixo, próximo de 0, há pouco consumo, e assim maior disponibilidade de recursos, e as latências observadas tornam-se baixas.

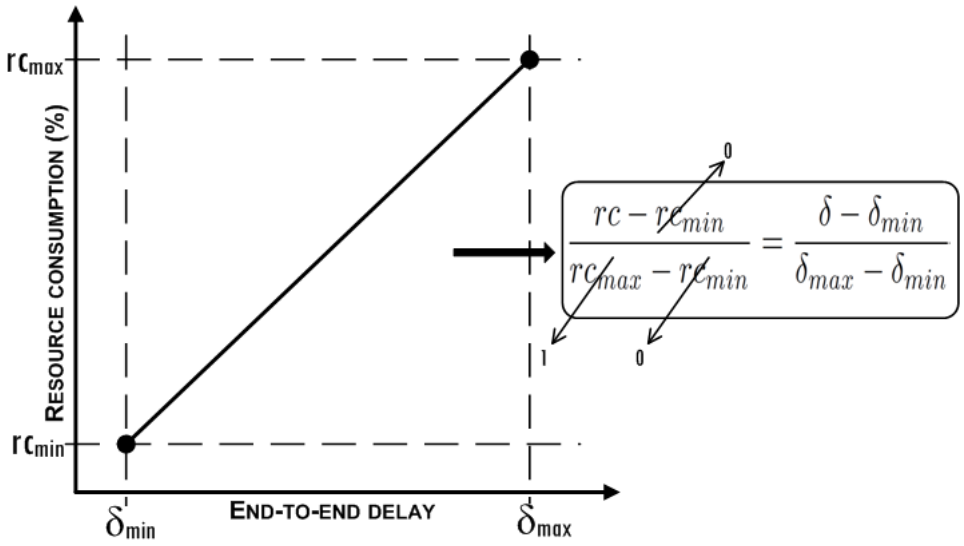


Figura 5.7. Gráfico da relação proposta entre as latências de comunicação fim-a-fim do sistema e o consumo de recursos.

É importante ressaltar que a relação entre consumo de recursos e latência possivelmente não apresenta linearidade. O escopo deste trabalho não inclui a identificação do sistema de modo a estimar a relação mais adequada, de modo que experimentamos a relação linear para avaliar a eficiência do algoritmo proposto a partir desta premissa.

Deve-se observar que o valor de referência depende da carga de trabalho do sistema e do tamanho do grupo (número de nós membros). Uma vez que estas condições podem variar em tempo de execução, o valor de referência pode ser calculado de forma dinâmica de acordo com o consumo de recursos desejados, a partir de: a) requisitos do usuário em termos de consumo de recursos desejado; b) consumo atual de recursos do ambiente; e c) número de membros ativos do grupo.

Assumimos que o *overhead* do protocolo influi no consumo de recursos, de forma proporcional: quanto maior o *overhead* do protocolo ovh_P, maior o número de mensagens enviadas pelo protocolo, e assim maior o percentual de recursos consumidos. A partir dos valores máximo e mínimo observados para o *overhead* (ovh_{max} e ovh_{min}), determinamos esta relação de acordo com a Figura 5.8. Devemos observar que quando todos processos continuamente enviam mensagens ao grupo, não há atuação do mecanismo de *time-silence* e nenhuma mensagem de controle

($ovh_{min} = 0$); por outro lado, quando apenas um membro do grupo está ativo, todos os demais $n-1$ membros contribuem sempre para a completude por meio do *time-silence* ($ovh_{max} = \frac{n-1}{n}$).

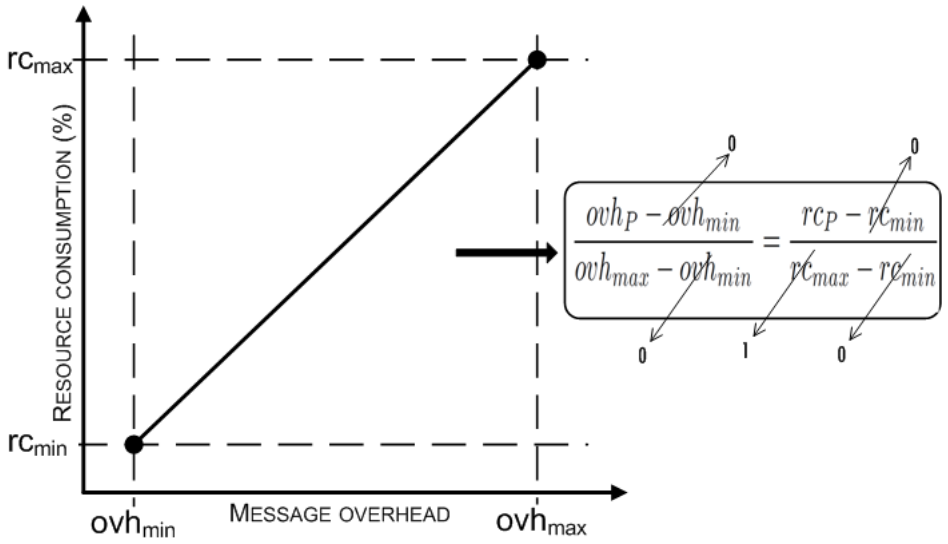


Figura 5.8. Gráfico da relação entre o *overhead* do protocolo de comunicação em grupo proposto e o consumo de recurso.

Deve-se notar que quanto maior o período ts, menor a frequência de atuação do *time-silence* e menor o *overhead* devido às mensagens de controle do protocolo. Assumindo linearidade nesta relação e a relação linear entre *overhead* e consumo de recursos, podemos associar a um dado consumo de recursos desejado (*set-point* definido pelo usuário), um valor de *overhead* e, por meio de uma lei de controle, o ajuste necessário no período ts para o *time-silence*.

Apresentamos a seguir, detalhes da implementação dos componentes sensor e controlador do protocolo auto-gerenciável – estes componentes e a proposta de construção de algoritmos auto-gerenciáveis são contribuição da tese apresentada em [17]. Esta tese contribui na caracterização do protocolo básico de comunicação em grupo, identificação de métricas e dos pontos de sensoreamento e de atuação específicos para o protocolo auto-gerenciável, sua implementação e sua avaliação – os resultados foram apresentados inicialmente em [71] e, posteriormente, de forma estendida em [72].

Componente Sensor

O componente sensor é caracterizado por dois métodos: sensoreamento (Algoritmo 5.10) e transdução (Algoritmo 5.11), os quais são executados, respectivamente, na recepção e na entrega de mensagens.

Na recepção, o sensor contabiliza o número de mensagens recebidas (*nrecv*), o número de mensagens nulas (*nts*) (Algoritmo 5.10, linhas 1–4), e estima a latência da viagem (RTT – *round-trip-time* (Algoritmo 5.10, linha 5).

Na entrega, o sensor converte as métricas mensuradas (transdução), estimando: i) o valor atual e máximo do *overhead* do protocolo (*ovh* e ovh_{max}), linhas 1–2 do Algoritmo 5.11; ii) os valores máximo, mínimo e a média das latências de comunicação fim-a-fim (δ_{max}, δ_{min} e δ_{mean}),

Algoritmo 5.10: Protocolo auto-configurável proposto para comunicação em grupo: mecanismo sensor – método Sensor.sensing() – executado por um processo p_i na recepção de uma mensagem m.

1: $nrecv_i \leftarrow nrecv_i + 1$
2: ***if*** *m is a null message* ***then***
3: $\quad nts_i \leftarrow nts_i + 1$
4: ***end if***
5: *estimate the rtt delay*

linhas 3–15 do Algoritmo 5.11 – deve-se observar que os valores estimados utilizam fator de esquecimento (linhas 7 e 14–15) de modo a obter amostras mais significativas que representem limites temporais adaptativos para os blocos causais; iii) o consumo de recursos atual (*rc*), linha 16 do Algoritmo 5.11; e iv) carga de trabalho, representada por meio do intervalo entre chegadas (*TBA – time between arrivals*), linha 17 do Algoritmo 5.11 – medida pelo intervalo em que mensagens de aplicação provenientes de processos remotos são recebidas por p_i. Por fim, considerando o máximo intervalo entre chegadas observado, o sensor estima um limite máximo para o *time-silence* (ts_{max}), linha 19 do Algoritmo 5.11.

Algoritmo 5.11: Protocolo auto-configurável proposto para comunicação em grupo: mecanismo sensor – método Sensor.transducing() – executado por um processo p_i na entrega de uma mensagem m.

1: *estimate the actual overhead (*ovh_i*) by* $ovh_i \leftarrow nts_i/nrecv_i$
2: *estimate the maximum overhead by* $ovh_{max} = (n-1)/n$
3: *estimate the communication delay (*δ*) by* $\delta \leftarrow rtt/2$
4: ***if*** δ_{mean} *is unset* ***then***
5: $\quad$ *do* $\delta_{mean} \leftarrow \delta$, $\delta_{max} \leftarrow \delta$ *and* $\delta_{min} \leftarrow \delta$
6: ***end if***
7: *compute* $\delta_{mean} \leftarrow \alpha * \delta_{mean} + (1-\alpha) * \delta$
8: ***if*** $\delta_{max} < \delta$ ***then***
9: $\quad$ *compute* $\delta_{max} \leftarrow (1+\beta) * \delta$
10: ***end if***
11: ***if*** $\delta_{min} > \delta$ ***then***
12: $\quad$ *compute* $\delta_{min} = \delta$
13: ***end if***
14: *compute* $\delta_{min} \leftarrow \phi * \delta_{min} + (1-\phi) * \delta$
15: *compute* $\delta_{max} \leftarrow \phi * \delta_{max} + (1-\phi) * \delta$
16: *compute* $rc \leftarrow (\delta_{mean} - \delta_{min})/(\delta_{max} - \delta_{min})$
17: *store the inter-arrival time interval of the messages from the sender process to* p_i *doing* $TBA[m.sender] \leftarrow clock_i - A[m.sender]$
18: *update the vector of arrival time* $A[m.sender] \leftarrow clock_i$;
19: *estimates* $ts_{max} \leftarrow (1+\beta) * max(TBA)$

Componente Controlador

O controlador (Algoritmo 5.12) mapeia o percentual de recursos disponíveis a serem alocados (diferença entre a quantidade de recursos desejada *rcd* e a quantidade de recursos consumidos *rc*) em um *set-point* dinâmico do *overhead* do protocolo. A escolha da quantidade de recursos desejada é uma decisão da aplicação usuária, de acordo com o cenário de execução do protocolo. Uma lei de controle proporcional define ts em função da diferença entre o *overhead*

atual e o desejado (ovh_{ref}): se o valor de *overhead* atual (*ovh*) é menor que o desejado (ovh_{ref}), então reduz-se o valor de *ts*, consumindo mais recursos da rede e diminuindo o tempo de bloqueio; caso contrário, se o valor atual de *overhead* é maior que o desejado, então o valor de *ts* é incrementado, diminuindo as mensagens de controle e o consumo de recursos da rede e aumentando o tempo de bloqueio.

Algoritmo 5.12: Protocolo auto-configurável proposto para comunicação em grupo: mecanismo sensor – método Controller.controlling() – executado por um processo p_i na entrega de uma mensagem *m*.

```
1:  read the user-defined resource consumption requirement rcd
2:  compute the set-point (ovh_ref) by ovh_ref ← ovh_max * (rcd − rc)
3:  error ← ovh_ref − ovh
4:  ts_var ← (−1/ts_max) * ovh_max * error
5:  action ← K_p * ts_var * h
6:  ts ← ts + action
7:  if ts < 0 then
8:      ts ← 0
9:  end if
10: if ts > ts_max then
11:     ts ← ts_max
12: end if
```

Em cada evento de entrega de mensagens, o controlador atua, lendo os requisitos do usuário para o consumo de recursos (*rcd*). A partir deste, o controlador combina *rcd* com o consumo de recursos estimado (*rc*) e o *overhead* máximo estimado (ovh_{max}), definindo o valor de referência para o *overhead* (ovh_{ref}), linhas 1–2 do Algoritmo 5.12. Então, o controlador computa a diferença (*error*) entre o overhead estimado (*ovh*) e o desejado (ovh_{ref}), linha 3 do Algoritmo 5.12. O ajuste ts_{var} ao valor de *ts* é computado a partir desta diferença, linha 4 do Algoritmo 5.12 – este cálculo considera a relação linear entre *overhead* e *time-silence*: se *time-silence* é mínimo (zero) então o *overhead* é máximo (ovh_{max}); caso o *time-silence* é máximo (ts_{max}), o *overhead* é mínimo (zero).

Por fim, a ação de controle calcula a partir da variação calculada para o time-silence (ts_var), o valor da ação de controle (*action*), linha 5 do Algoritmo 5.12 – utilizando uma constante K_p que define o ganho do controlador e o tamanho *h* da janela de tempo medido entre duas ativações seguidas do controlador. Esta ação então é aplicada ao valor do *ts*, mantendo-o entre o mínimo (zero) e o máximo (ts_{max}), linhas 6–12 do Algoritmo 5.12.

Observa-se ainda que a estimativa dos parâmetros ts_{max}, δ_{max} e δ_{min}, utilizados na determinação dos limites temporais para completude e estabilidade, torna o algoritmo adaptativo às variações de carga do ambiente ao longo da execução, prevenindo mudanças de visão desnecessárias por falsas suspeitas e o uso de limites temporais extremamente conservadores que retardam a convergência do protocolo.

5.1.3.3 Avaliação de desempenho

Modelo de Simulação

É simulado um cenário com processos conectados em sítios que se comunicam entre si sobre a Internet, assumindo que há uma relação entre o atraso das mensagens e o uso de recursos da rede, ou seja, o atraso aumenta de acordo com a quantidade de tráfego transmitido. De modo a prover tal modelo, nós simulamos os efeitos de uma infraestrutura subjacente de comunicação com roteamento central, e modelamos o efeito do enfileiramento sobre o atraso fim-a-fim baseado em duas partes: atraso do roteamento central e atrados desde um sítio da borda até o roteador central, e deste, após o roteamento até outro sítio. Para esta última parte, uma distribuição log-normal é utilizada de modo a expressar o comportamento de atraso inter-sítios: média de $10ms$ e desvio-padrão de $5ms$[3].

Métricas de Desempenho

São utilizadas as seguintes métricas de desempenho: o *overhead* do protocolo, dado pela carga adicional imposta a rede; e a *latência na entrega de mensagens*, também denominada de tempo de bloqueio, medido da recepção da mensagem pelo processo à entrega correspondente pela camada de aplicação.

Deve-se observar que o *overhead* guarda relação com o consumo de recursos pelo protocolo, o qual deve ser auto-regulado pela abordagem proposta, sendo esta métrica a de maior interesse na presente avaliação. O *overhead* (ovh) é dado por $ovh = n_{ct}/(n_{ap} + n_{ct})$, onde n_{ct} e n_{ap} são, respectivamente, o número total de mensagens de controle e o número total de mensagens da aplicação. Pode-se tornar o protocolo mais rápido (i.e. produzir entrega de mensagens mais rápida, reduzindo o tempo de bloqueio), se reduzimos o valor do período do time-silence, forçando maior atividade do protocolo e mais mensagens de controle – isto contudo, incrementa ovh e o consumo de recursos (i.e. uso da rede). O efeito do *time-silence* no tempo de bloqueio (BT) depende da carga de trabalho atual, o que nos sugere que a avaliação deve considerar diferentes condições de carga.

Descrição dos Experimentos

A versão auto-gerenciável do protocolo de comunicação proposto é avaliada de forma comparativa à versão base, considerando um conjunto representativo de cenários em que variamos: o tamanho n do grupo de processos, o perfil de carga de trabalho, e a possibilidade de ocorrência de falhas.

Por exemplo, o perfil de carga de trabalho é caracterizado na simulação através de uma função de probabilidade de Bernoulli associada a cada processo que decide quando enviar uma mensagem, determinando três níveis de carga possível: 100, 150 e 50 mensagens por segundo. Denominamos tais cargas de trabalho de *média*, *alta* e *baixa*, respectivamente.

Nos experimentos, mensura-se o tempo de bloqueio e o *overhead* para cada configuração. Cada experimento simulou a execução em 60 segundos, de modo que o número de mensagens

[3]Nós mensuramos os atrasos entre dois sítios distintos da Rede Nacional de Ensino e Pesquisa (RNP), e computados máximo de $221,855ms$, média de $10,21ms$ e desvio-padrão de $4,796ms$ para 2.000 pacotes de 15KB.

por processo varia de 3.000, perfil de *baixa* carga, a 9.000, perfil de *alta* carga. Cada mensagem de aplicação gerada possui $4KB$. Os valores foram determinados de forma empírica. Cada experimento é replicado três vezes, calculando-se média, desvio-padrão e intervalo de confiança em 95%.

Para a versão auto-gerenciável, utiliza-se os seguintes parâmetros de configuração:

- fator de amortecimento $\alpha = 0,99$, considerando uma janela $w_1 = 100$ mensagens – i.e., $\alpha = \frac{w_1 - 1}{w_1}$;
- margem de segurança $\beta = 0,1$;
- fator de esquecimento $\phi = 0,99999$, considerando uma janela $w_2 = 100.000$ mensagens – i.e., $\phi = \frac{w_2 - 1}{w_2}$;
- ganho proporcional $K_P = 1.000$.

São avaliados seis conjuntos de experimentos distintos, descritos a seguir.

Conjunto de Experimentos **distinct-fixed-workloads**:

Neste cenário, o protocolo base (i.e. *time-silence* fixo) e o auto-gerenciável são avaliados comparativamente, sendo os fatores o número de processos (10, 20, 30 e 40), o período do *time-silence* ts e o perfil da carga de trabalho (*média*, *alta* e *baixa*). O mecanismo de blocos causais com *time-silence* fixo ($ts = 20ms$ e $ts = 200ms$) é comparado com o auto-gerenciável, no qual o *time-silence* é ajustado de acordo com um dado valor de referência de consumo de recursos também escolhido empiricamente ($rc_D = 25\%$).

Conjunto de Experimentos **variable-workload**:

Neste cenário, consideramos que a carga de trabalho varia dinamicamente. O protocolo base (i.e. *time-silence* fixo) e o auto-gerenciável são avaliados comparativamente, considerando como fator o número de processos (10, 20, 30 e 40) e a carga de trabalho varia na execução entre os perfis *alta* e *baixa*, mantendo o mesmo valor de referência de consumo de recursos ($rc_D = 25\%$).

Conjunto de Experimentos **changing set-point**:

Neste cenário, experimentamos como o protocolo auto-gerenciável se adapta a uma alteração no seu *set-point* de operação e o impacto desta na velocidade de entrega de mensagens. O valor de referência de consumo de recursos (rc_D) é alterado de 25% para 40%, para uma configuração com um número fixo de processos ($n = 20$) e o perfil de carga de trabalho *média*.

Conjunto de Experimentos **faulty scenario**:

Simulamos a ocorrência de falhas de processos em um grupo. Consideramos a versão auto-gerenciável e o protocolo base com *time-silence* fixo ($ts = 20ms$ e $ts = 200ms$) com um número $n = 5$ de processos, dos quais 2 processos falham por *crash* no instante $t = 1.000ms$, o que causa a execução do procedimento de mudança de visão e observamos como o protocolo se adapta, mantendo o *set-point* de operação.

Comparamos o protocolo base com *time-silence* fixo ($ts = 10ms$ e $ts = 500ms$) e a versão auto-gerenciável, considerando um consumo de recursos desejado de $rc_D = 40\%$. O perfil de trafégo utilizado foi em rajadas: cada processo simula uma aplicação que continuamente alterna no envio de uma rajada de mensagens e em inatividade. O número de mensagens de cada rajada é definido por uma distribuição uniforme com média de 100 e desvio-padrão de 10 mensagens. O tempo de inatividade é definido por outra distribuição uniforme com média de $14ms$ e desvio-padrão de $4ms$. O perfil de tráfego dos serviços distribuídos que executam de forma concorrente não é conhecido. Em cada experimento, cada processo transmite ao menos 10.000 mensagens (cerca de 150.000 mensagens no total).

Resultados e Discussão

Conjunto de Experimentos **distinct-fixed-workloads**:

A Tabela 5.1 apresenta os valores de média e intervalo de confiança (i.c. – intervalo de confiança de 95%) para o *overhead* do protocolo, e média e desvio padrão para o tempo de bloqueio na entrega das mensagens. De modo a melhor visualizar as variações do *overhead* de acordo com os diferentes perfis de carga, mensuramos a diferença percentual relativa entre as cargas de trabalho alta e baixa para cada configuração (a diferença percentual relativa, d.p.r., é calculada pela divisão da diferença absoluta de dois valores pela média dos mesmos).

Para os diferentes tamanhos de grupo (de 10 a 40 nós) para valores fixos do *time-silence*, observamos uma alta variação no overhead (de acordo com a d.p.r. calculada). Por outro lado, a abordagem auto-gerenciável permite obter uma variação muito menor (de 0.45% a 9.00%), mantendo-se próximo ao *overhead* correspondente ao consumo de recursos desejado. Deve-se observar que este *overhead* ovh_P é calculado na forma $ovh_P = rc_D * ovh_{max}$, e dado que $rc_D = 25\%$, o valor de ovh_P é 22.5%, 23.75%, 24.167% e 24.375% para configurações de 10, 20, 30 e 40 processos, respectivamente. Isto resulta do fato que o componente controlador mantém a relação de compromisso entre o tempo de bloqueio e o overhead.

Conjunto de Experimentos **variable-workload**:

Em um cenário com carga de trabalho variável, observa-se que o ponto de operação varia de forma dinâmica ao longo da operação. Os resultados apresentados na Tabela 5.2 indicam que, mesmo em cenários dinâmicos, a versão auto-gerenciável mantém o ponto de operação para o *overhead* desejado, mesmo no cenário de mudança dinâmica de carga para diferentes tamanhos de grupo. Os resultados são similares aos da Tabela 5.1.

Conjunto de Experimentos **changing set-point**:

A Figura 5.9 apresenta um traço do overhead e do tempo de bloqueio neste experimento em que o ponto de referência do consumo de recursos é alterado durante a execução: o protocolo auto-gerenciável inicialmente se ajusta ao consumo de recursos inicial de 25% ($rc_D = 0,25$) – equivalente a 23.75% de *overhead* (i.e., $rc_D * ovh_{max} = 0,25 * 19/20 = 0,2375$); depois disto, no instante $t = 10.000ms$, o percentual de consumo de recursos desejado é aumentado para 40%, o qual implica em *overhead* desejado de 38% (i.e., $rc_D * ovh_{max} = 0,40 * 19/20 = 0,38$); então, a abordagem auto-gerenciável provê o ajuste de seu ponto de operação: incrementando

Tabela 5.1. Tabela dos resultados de simulação do protocolo auto-gerenciável de comunicação em grupo em comparação com o protocolo base, considerando 3 cenários distintos de carga de trabalho e 4 diferentes tamanhos de grupo (conjunto de experimentos **distinct-fixed-workloads**).

Fatores		tempo de bloqueio (ms)			*overhead* (%)					
n	carga	$ts = 20ms$	$ts = 200ms$	auto-gerenciável	$ts = 20ms$		$ts = 200ms$		auto-gerenciável	
		média ± desvio-padrão	média ± desvio-padrão	média ± desvio-padrão	média ± i.c.	d.p.r.	média ± i.c.	d.p.r.	média ± i.c.	d.p.r.
10	baixa	46,67 ± 09,87	111,69 ± 43,66	49,47 ± 18,55	32,02 ± 0,28	1,35	5,30 ± 0,03	15,37	21,33 ± 0,01	0,45
	média	46,60 ± 09,86	111,14 ± 43,59	46,22 ± 13,30	31,59 ± 0,05		5,18 ± 0,02		21,36 ± 0,01	
	alta	46,88 ± 09,80	107,08 ± 40,45	45,31 ± 13,33	31,96 ± 0,07		4,54 ± 0,02		21,43 ± 0,01	
20	baixa	47,96 ± 09,32	118,06 ± 42,77	53,18 ± 15,72	33,49 ± 0,15	11,78	5,56 ± 0,04	15,99	21,90 ± 0,01	2,70
	média	46,95 ± 09,04	109,63 ± 38,00	52,93 ± 15,29	29,76 ± 0,04		4,73 ± 0,01		21,59 ± 0,02	
	high	46,39 ± 10,07	110,76 ± 43,52	49,80 ± 15,21	32,09 ± 0,16		5,17 ± 0,03		22,18 ± 0,01	
30	low	50,07 ± 10,06	111,95 ± 42,25	46,89 ± 11,28	38,88 ± 0,51	32,57	4,91 ± 0,01	9,15	21,98 ± 0,03	1,61
	medium	46,32 ± 09,11	106,79 ± 38,46	46,79 ± 10,83	28,53 ± 0,06		4,48 ± 0,01		21,94 ± 0,01	
	alta	45,79 ± 09,20	108,64 ± 41,34	45,60 ± 09,84	27,99 ± 0,08		4,65 ± 0,01		21,63 ± 0,01	
40	baixa	48,21 ± 08,94	116,67 ± 42,40	68,75 ± 38,57	30,52 ± 0,01	32,12	4,74 ± 0,01	38,42	24,17 ± 0,14	9,00
	média	46,84 ± 08,24	105,66 ± 31,94	56,12 ± 17,09	25,69 ± 0,01		3,83 ± 0,01		22,09 ± 0,01	
	alta	45,22 ± 07,80	096,66 ± 27,63	46,77 ± 09,25	22,07 ± 0,01		3,21 ± 0,01		22,69 ± 0,06	

o overhead de mensagens e obtendo entrega mais rápida.

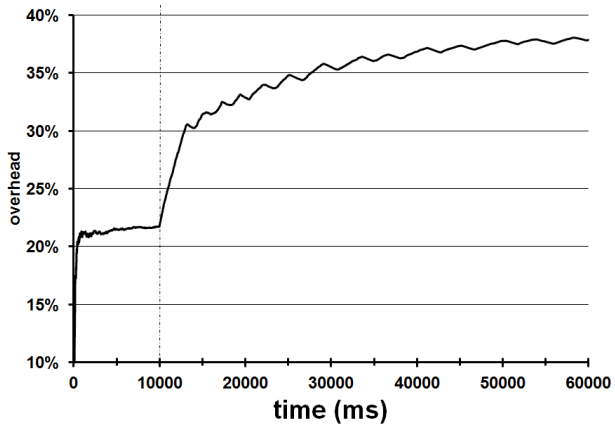
Conjunto de Experimentos **faulty scenario**:

É simulada a ocorrência de falhas por *crash* de 2 dos 5 processos de um grupo no instante $t = 1.000ms$. Como podemos observar na Figura 5.10, a partir do instante de ocorrência da falha, há aumento do tempo de bloqueio e do *overhead* devido à troca de mensagens para a nova visão. Contudo, a abordagem auto-gerenciável produz convergência mais rápida para um novo ponto de operação, graças ao ajuste dinâmico do *time-silence*, em comparação ao *time-silence* fixo, com leve aumento no tempo de bloqueio, conforme segue.

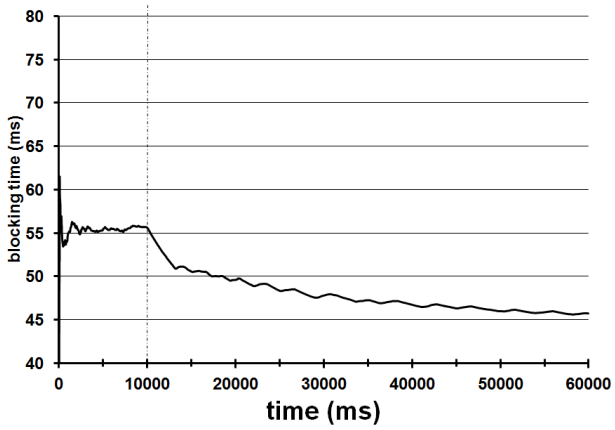
Quando os processos falham por crash, não há geração de mensagens para completude dos

Tabela 5.2. Tabela dos resultados de simulação do protocolo auto-gerenciável de comunicação em grupo, considerando carga de trabalho variável e 4 diferentes tamanhos de grupo (conjunto de experimentos **variable-workload**).

factors		resultados	
n	carga	tempo de bloqueio (ms)	overhead (%)
		média ± desvio-padrão	média ± i.c.
10	VARYING	49,76 ± 18,69	21,22% ± 0,02%
20	VARYING	53,63 ± 16,13	21,63% ± 0,01%
30	VARYING	56,74 ± 17,00	20,81% ± 0,10%
40	VARYING	66,26 ± 34,85	23,21% ± 0,10%



(a) overhead

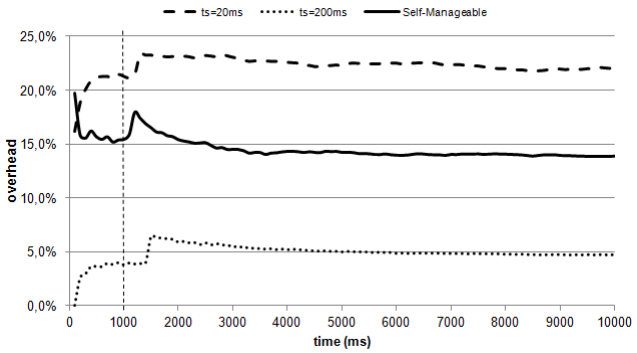


(b) tempo de bloqueio

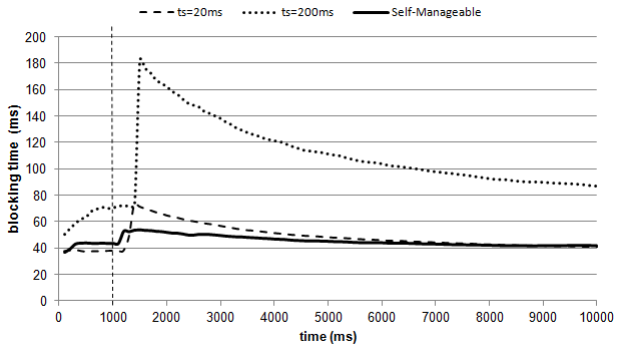
Figura 5.9. Gráfico da resposta do protocolo auto-gerenciável de comunicação em grupo a mudança do *set-point* em $t = 10.000ms$.

blocos pelos processos falhos, bloqueando a entrega dos blocos, o que explica o pico do bloqueio após $t = 1.000ms$ no gráfico da Figura 5.10(b). Por outro lado, uma vez suspeitada a falha, as mensagens geradas na troca de visão causam o aumento do *overhead*, o que explica o pico no gráfico da Figura 5.10(a).

Depois da instalação da nova visão, a abordagem autonômica percebe a mudança no número de membros e executa duas ações associadas. Primeiro, como o número de processos reduz de 5 para 3, o *overhead* máximo estimado pelo controlador decresce de 80% para $66,7\%$, ou seja, de $4/5$ para $2/3$, isto força a redução do ponto de operação em termos do *overhead* do protocolo. Segundo, a redução no número de membros do grupo reduz o consumo de recursos pelo protocolo, isto permite aumentar o *overhead*, uma vez que há mais recursos disponíveis no ambiente computacional: 3 processos consumem menos recursos que 5. O algoritmo considera ambos os efeitos e promove um ajuste moderado no ponto de operação dado pelo *overhead* de 15% para 14%, o que explica o comportamento do overhead médio observado para o protocolo auto-gerenciável na Figura 5.10(b). Adicionalmente, o decréscimo do consumo de recursos devido a redução do número de membros do grupo faz que o controlador diminua o período do



(a) overhead



(b) tempo de bloqueio

Figura 5.10. Gráfico da resposta do protocolo auto-gerenciável de comunicação em grupo a falhas de dois processos em $t = 1.000ms$.

time-silence, provendo entrega mais rápida, devido ao menor tempo de bloqueio.

5.2 REPLICAÇÃO BIZANTINA

Serviços computacionais distribuídos, baseados no paradigma cliente/servidor, permitem a clientes de diferentes plataformas submeterem requisições e obterem o resultado destas após a execução da operação computacional correspondente. Em muitos destes cenários, se o processamento destas requisições for realizado por um único componente servidor, o nível de serviço obtido pelas aplicações clientes pode ser insatisfatório perante a possibilidade de falhas do servidor. É possível elevar este nível de serviço através de replicação.

Uma das técnicas de replicação é a replicação de máquina de estados [48, 62]: cada um de N servidores implementa a máquina de estados que caracteriza o serviço computacional – isto é, o conjunto de estados internos que caracterizam o serviço provido e as transições realizadas de um estado a outro na ocorrência de uma dada operação. De modo a reproduzir o mesmo comportamento, todos os servidores replicados iniciam em um mesmo estado e as máquinas de estados são determinísticas, ou seja: em face de uma dada operação em um dado estado, sempre executam a mesma transição de estados. O conjunto de servidores deve concordar, ou chegar ao consenso, em relação à ordem de execução das requisições dos clientes, e baseado nestas premissas, todos servidores devem executar o mesmo conjunto de operações e apresentar

o mesmo conjunto de resultados.

O protocolo de gestão das réplicas é implementado de modo que seja possível tolerar a falha de até f réplicas, elevando o nível de serviço. Em ambientes benignos e com hardware e software corretos (ou com alta probabilidade disto), as falhas são em geral por *crash* (parada silenciosa) do servidor devido a causas naturais, como, por exemplo, falhas elétricas.

Contudo, em ambientes de computação distribuída abertos, como a Internet, os nós servidores podem estar imersos em um ambiente hostil, em que os componentes do sistema podem falhar por causas naturais ou maliciosas, como, por exemplo, ataques de intrusão, ou mesmo podem existir falhas latentes do ambiente computacional, as quais induzem falhas por comportamento arbitrário – no qual, além da possibilidade de *crash*, os resultados podem ser omitidos ou mesmo alterados. Falhas arbitrárias são muito mais difíceis de serem tratadas que falhas mais benignas, como crash ou omissões.

A obtenção de consenso é um requisito para estabelecer a ordem de requisições numa máquina de estados replicada. Em 1982, o clássico problema dos *Generais Bizantinos* [22] apresentou as condições para o consenso distribuído diante de falhas arbitrárias, a partir de então denominadas falhas bizantinas. Neste mesmo artigo foi provado que o consenso bizantino precisa de $N = 3f + 1$ para tolerar o máximo de f processos falhos, em um sistema distribuído síncrono (limites superiores conhecidos para tempos de transmissão de mensagens e execução de passos computacionais).

Alguns protocolos para consenso bizantino (ou o equivalente *multicast* bizantino atômico) foram propostos desde então, por exemplo: [73], [74], [75], [76]. Contudo, em geral, esses protocolos se revelaram caros para o caso comum (sem falhas) e por isso suscitaram pouco interesse para sua utilização. Foi a partir do trabalho de Castro e Liskov [18] sobre um protocolo de replicação tolerante a falhas bizantinas, dito PBFT (*Practical Byzantine Fault Tolerance*), que a replicação bizantina ganhou novo interesse. Tal trabalho propôs uma série de otimizações, tais como técnicas de rejuvenescimento de réplicas para recuperação proativa, uso de *batching* para otimizar o processamento de conjunto de requisições, mudanças de visões periódicas para rotacionar o papel do coordenador, uso de *checkpoint* periódico para *garbage collection*. Desta forma, há um conjunto de parâmetros relacionados às técnicas de otimização (como tamanho da janela de *batch*, período de *checkpoint*, período de resincronização de estado), que devem ser ajustados de forma apropriada para garantir o desempenho desejado.

Este capítulo apresenta de forma breve a replicação bizantina a partir do protocolo PBFT de [18], e de trabalhos posteriores desenvolvidos a partir dos conceitos apresentados por este. Uma versão adaptativa, apresentada em [77–79], baseada na auto-configuração dos parâmetros operacionais do protocolo PBFT foi desenvolvida como esforço conjunto desta tese e da tese de [17], sendo apresentada como contribuição da tese uma avaliação de desempenho comparativa entre a versão clássica do PBFT e a versão adaptativa ora apresentada.

5.2.1 Modelo de sistema para replicação bizantina

O modelo de sistema é o parcialmente síncrono, baseado na hipótese GST [74]. Processos podem fazer o papel de clientes ou de servidores replicados. É assumido que réplicas implementam máquinas determinísticas. Processos podem falhar de forma benigna, como, por exemplo, por *crash ou parada silenciosa*, ou de forma arbitrária, caso das falhas bizantinas. É assumido que no máximo f processos falham num total de $N = 3f + 1$ réplicas da máquina de estados. Os canais de comunicação não são confiáveis: mensagens podem ser atrasadas, duplicadas, perdidas ou entregues fora de ordem de envio. Também é assumido que processos falham de forma independente.

Assume-se que existe uma função *hash D* que calcula um sumário ou $digestD(m)$ para cada mensagem m de forma única, com alta probabilidade. Cada par de processos p_i e p_j de Π compartilha um par de chaves simétricas $k_{i,j}$ e $k_{j,i}$ utilizados para computar os códigos de autenticação de mensagem (MAC) $\mu_{i,j}$ e $\mu_{j,i}$ que autentica uma mensagem enviada de p_i para p_j e uma mensagem de p_j para p_i, respectivamente.

A autenticação de uma mensagem m é realizada através do *digest* $D(m)$ e do código de autenticação de mensagem (MAC) computado. A mensagem m de p_i para p_j é autenticada na mensagem $< m >_{\mu_{i,j}}$. Uma mensagem difundida por uma réplica p_i em *multicast* para todas as réplicas é autenticada na mensagem $< m >_{\alpha_i}$, onde α é um autenticador, um vetor com o conjunto de MACs $\mu_{i,j}$, para todas as réplicas p_j. Assumimos com alto grau de probabilidade que nós maliciosos não podem assumir o papel de outros nós ao autenticarem mensagens.

5.2.2 PBFT e trabalhos relacionados

A execução do PBFT é baseada em visões. Em cada visão, uma réplica é designada primária, responsável pelo sub-protocolo de acordo ou consenso. O sub-protocolo de acordo é a base do PBFT, e define em que ordem as requisições dos clientes são executadas pelas réplicas. A comunicação entre as réplicas é autenticada por meio de chaves criptográficas, conforme definido no modelo do sistema (ver Seção 5.2.1).

O sub-protocolo de acordo atua como um protocolo de efetivação (*commit*) de três fases adaptado a falhas bizantinas. O cliente envia a requisição ao grupo de réplicas. Então, inicia-se a fase de PRE-PREPARE, na qual o primário assinala um número de sequência e envia a requisição ao grupo de réplicas. As réplicas respondem a esta requisição com uma mensagem de PREPARE. Se ao menos $2f$ mensagens de PREPARE para uma requisição são recebidas por uma réplica, esta envia COMMIT para a requisição. Uma réplica executa uma requisição se recebe $2f + 1$ mensagens de COMMIT desta requisição. Por fim, o resultado é enviado ao cliente, o qual aceitará o resultado, se recebe ao menos $f + 1$ respostas equivalentes.

De modo a maximizar o desempenho, é utilizada criptografia com chaves públicas para a troca periódica de chaves simétricas de sessão, sendo estas utilizadas na comunicação em pares das réplicas. Devido ao custo da autenticação, o primário pode agrupar um conjunto de re-

quisições recebidas em uma única mensagem com um lote de requisições a serem processadas (*batching*). Nesse mecanismo de *batching*, cada requisição recebida é armazenada em *buffer* e seu respectivo resumo (*digest*) é gerado. Quando existem requisições suficientes para preencher um lote, uma mensagem de PRE-PREPARE é enviada contendo, entre outros campos, o conjunto de *digests* calculados para cada requisição do lote. Para prevenir que o primário fique bloqueado esperando indefinidamente que o lote de requisições seja preenchido, quando a primeira requisição do lote é recebida, um temporizador é iniciado, e caso o lote não seja preenchido dentro do *timeout* de *batching*, então a mensagem de PRE-PREPARE é enviada com as requisições recebidas até então. Uma vez que o número de requisições que chegam dentro de um *timeout* pode variar, diferentes lotes podem agrupar diferentes números de requisições – por conta disto, alguns autores, como [80], afirmam que o mecanismo de *batching* é adaptativo, apesar do mesmo não observar quaisquer aspectos referentes ao desempenho do protocolo ou da carga das aplicações para determinar o tamanho do lote.

Em caso de suspeita de falha do primário, o sub-protocolo de mudança de visão permite a eleição de uma nova réplica primária. De modo a evitar falhas intermitentes (de maior probabilidade com o envelhecimento), ou mesmo permitir restaurar uma réplica eventualmente comprometida, o sub-protocolo de recuperação pró-ativa reinicia uma réplica de forma periódica – esta técnica é dita rejuvenescimento de réplica.

Em diferentes casos, como na eleição de uma nova réplica ou no de rejuvenescimento da réplica, é necessário que uma réplica recupere o estado da máquina de estados. Ao longo da execução, a recuperação do estado pode ser custosa, o sub-protocolo de *checkpoint* permite sincronizar réplicas, limitando o estado que deve ser mantido em *log*.

Outros protocolos foram publicados com o objetivo de otimizar aspectos do PBFT. Por exemplo, o Zyzzyva [81] modifica o sub-protocolo de acordo por meio da execução especulativa, onde o acordo somente é executado em caso de suspeitas de falha. A versão original do PBFT considera execução especulativa somente para operações que não alteram os estados das réplicas, como, por exemplo, operações de leitura. Recentemente, o PBFT foi especializado em [80] para permitir execução especulativa para operações que alteram o estado das réplicas. Entretanto, diferente do Zyzzyva, esta versão do PBFT permite que o cliente execute de forma especulativa, mas não externaliza o estado da máquina replicada antes que o acordo seja executado.

Os diferentes protocolos apresentados na literatura diferem com relação a estratégia utilizada para cada sub-protocolo que compõe a replicação tolerante a falhas bizantinas, em especial na escolha do sub-protocolo de acordo, desde uma abordagem especulativa, como em [81], a uma abordagem de adaptação pré-definida em tempo de projeto, como em [82], em que diferentes estratégias para o sub-protocolo de acordo podem ser escalonadas em face de diferentes condições de gatilho, pré-definidas na configuração do protocolo. Ao nosso saber, nenhuma abordagem se baseia na configuração dinâmica dos parâmetros que definem os pontos de operação dos sub-protocolos. Neste aspecto, a abordagem proposta é genérica o suficiente para ser

aplicada não somente ao PBFT, mas em outros protocolos derivados do PBFT.

Alguns protocolos de replicação bizantina baseados em quóruns atuam de forma adaptativa, mas considerando a adaptação a falhas, sem considerar, por exemplo, as mudanças dinâmicas na carga de trabalho do ambiente distribuído. Em [83], por exemplo, é considerada uma abordagem baseada em quoruns, em que o limiar de falhas é inicialmente configurado em um valor otimista. Na medida em que os servidores do sistema falhem, este limiar é incrementado. Tal limiar é usado na definição do número mínimo de servidores nos quóruns, por conta disso, os autores nomeiam a abordagem de sistema de quoruns dinâmicos. A abordagem de [83] é especializada em [84], em que um detector de falhas bizantinas é usado para permitir que servidores defeituosos sejam removidos, possibilitando assim que o número total de servidores considerados seja reavaliado em tempo de execução – que promove alterações nas composições dos quoruns. Por conta disto, os autores a denominam de abordagem re-configurável de quoruns bizantinos. Estes trabalhos consideram apenas a reconfiguração de parâmetros mediante às falhas no sistema, enquanto que a abordagem proposta realiza a adaptação considerando as condições de carga no ambiente – buscando assim por configurações que permitam um melhor desempenho em face de mudanças nas condições de carga. Em outros trabalhos não diretamente relacionados à presente proposta, o tratamento de falhas bizantinas também tem sido estudado no contexto de armazenamento de dados [85, 86].

5.2.3 aPBFT: uma versão adaptativa do PBFT

Conceber estratégias de adaptação de parâmetros para um protocolo de replicação bizantina não é uma tarefa simples, uma vez que a implementação destes protocolos é realizada considerando diferentes aspectos – como, por exemplo, segurança, gestão de memória, gerenciamento de réplicas, sincronização de estados. Estes diferentes aspectos, relacionados ao protocolo de replicação, tornam a análise extremamente complexa e dificulta a obtenção de estratégias adequadas de adaptação de parâmetros operacionais. Nesse sentido, a estratégia de ajuste dinâmico ora apresentada foi concebida abstraindo os detalhes internos de implementação. Para tanto, introduzimos aqui um modelo abstrato no qual os diferentes mecanismos, sub-protocolos e procedimentos do protocolo são vistos como caixas pretas organizadas de forma a compor um *pipeline* abstrato. Os estágios desse *pipeline* abstrato são selecionados considerando as etapas de interesse da adaptação de parâmetros. A partir do *pipeline* abstrato, propomos o aPBFT (*Adaptive* PBFT), uma extensão adaptativa do PBFT, a qual baseia a sua adaptação no ajuste dinâmico do tamanho da janela de *batch* (*BS*, *Batch Size*) e do *timeout* de *batching* (BT, *Batch Timeout*). A escolha dos parâmetros de adaptação se deve ao fato do mecanismo de *batching* representar um ponto importante para conciliar a carga das aplicações com os custos computacionais associados ao fluxo de execução do protocolo de replicação.

O aPBFT realiza estimativas a respeito da carga imposta pelos clientes e sobre o desempenho do protocolo básico (PBFT). Para isto, a estratégia proposta implementa um laço de controle

que usa informações obtidas a partir dos eventos associados ao envio, recebimento e processamento de mensagens do PBFT e determina os novos parâmetros do *batching*. As subseções a seguir apresentam em maiores detalhes os principais aspectos relacionados à implementação do aPBFT, como apresentado na Figura 5.11.

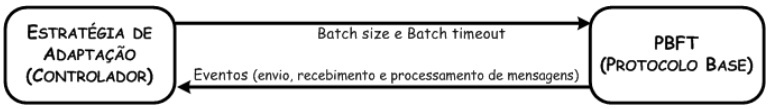


Figura 5.11. Laço de controle para a versão adaptativa do PBFT.

5.2.3.1 Descrição do pipeline abstrato para o atendimento de requisições

O aPBFT abstrai o fluxo de atendimento de requisições do PBFT considerando um *pipeline* abstrato com quatro estágios, denominados: *checking*; *buffering* and *batching*; *ordering*; e *processing* (ver Figura 5.12).

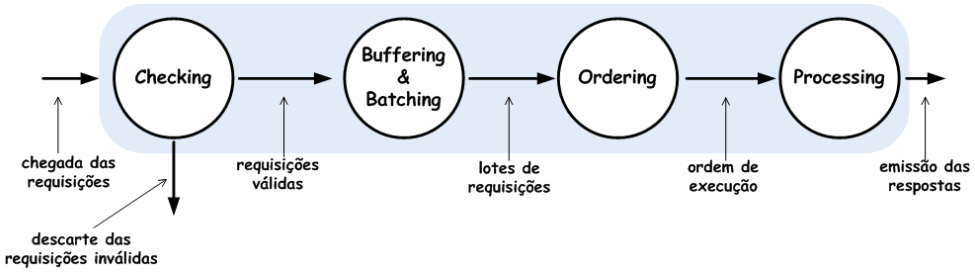


Figura 5.12. *Pipeline* abstrato para representar o processamento de requisições no PBFT.

O estágio de *checking* corresponde aos procedimentos realizados pelo PBFT para verificar a integridade, autenticidade e validade das requisições dos clientes. Requisições válidas são repassadas para o mecanismo de *batching*, enquanto que requisições inválidas são descartadas. No estágio de *buffering and batching*, as requisições aceitas são armazenadas em um *buffer*. Quando o número de requisições é suficiente para completar um lote (ou *batch*) de requisições, ou seja, o número de requisições é maior ou igual ao *batch size*, ou quando o *timeout* de *batch* expira, a réplica primária assinala este lote com um número seqüencial e, então, se inicia a fase de *ordering*. No estágio de *ordering*, as réplicas executam o acordo sobre a ordem das requisições. No estágio de *processing*, os lotes efetivados são processados e, então, cada réplica envia as respostas das requisições aos respectivos clientes solicitantes.

Evidentemente, esta descrição baseada no *pipeline* abstrai uma série de nuances do protocolo de replicação. Durante a fase de ordenação, por exemplo, é possível que várias instâncias do sub-protocolo de acordo sejam executadas em paralelo. Todavia, o uso do *pipeline* abstrato coloca o protocolo de replicação em um nível de detalhe adequado para a implementação da estratégia de adaptação.

5.2.3.2 Detalhes de implementação da abordagem adaptativa

A estimativa do timeout de *batching* se baseia no intervalo médio de tempo para ordenação e execução das requisições, como apresentado na Figura 5.13. Neste texto, este intervalo de tempo é dito MTE, *Mean Time To Ordering and Execute*.

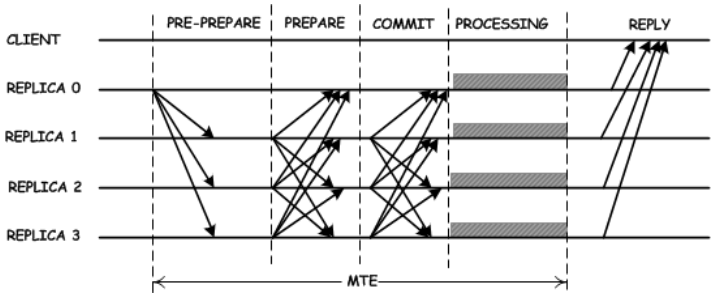


Figura 5.13. Tempo médio para ordenação e execução das requisições (MTE) no PBFT.

A ideia central, usada pela estratégia de adaptação, é que o *timeout* de *batching* deve ser longo o suficiente para permitir o acúmulo de requisições para a composição do lote, mas, por outro lado, o mesmo deve ser curto o suficiente para permitir que pelo menos um lote de requisições esteja no estágio de *ordering* – de modo a manter tal estágio sempre ocupado, na medida em que existem requisições a serem ordenadas e processadas. Entretanto, se o intervalo médio entre chegadas das requisições dos clientes (dito, MTA – *Mean Time Between Arrivals*) é maior que o MTE, então o *timeout* de *batching* pode ser zero, uma vez que a carga imposta pelos clientes é muito baixa e os estágios de *ordering* e *processing* do *pipeline* estão operando em um ritmo menor que suas capacidades de execução.

A métrica MTE é calculada usando a média móvel das últimas W amostras dos intervalos de tempo para ordenação e execução (TE), isto é:

$$MTE_k = (1-\alpha) * MTE_{k-1} + \alpha * TE_k \tag{5.1}$$

em que $\alpha = \frac{1}{W}$ e $TE = T_{exec} - T_{ord}$, sendo T_{ord} e T_{exec}, respectivamente, os instantes imediatamente antes da emissão de um PRE-PREPARE pp e imediatamente após a execução do lote de requisições associado ao pp.

A métrica MTA, por sua vez, é calculada a partir da relação entre o intervalo de tempo no qual a réplica primária está em execução (em seu papel como primária) e o número de requisições recebidas, isto é:

$$MTA = \frac{t - t_0}{n_{req}} \tag{5.2}$$

onde: t é o instante de tempo atual, obtido a partir do relógio local da réplica primária; t_0 é o instante de tempo no qual a réplica assumiu o papel de réplica primária; e n_{req} representa o número de requisições recebidas até o instante t.

O tamanho do *batch* (BS) é estimado usando a relação entre o MTE e MTA:

$$BS = \left\lceil \frac{MTE}{MTA} \right\rceil \tag{5.3}$$

A ideia central para o ajuste de *BS* é a mesma usada na adaptação do *timeout* de *batching*, isto é, *BS* deve ser definido de tal forma que permita ao menos um lote de requisições esteja nos estágios *ordering* ou *processing*. Assim, se os estágios *ordering* e *processing* levam em média MTE unidades de tempo para realizar em conjunto suas atividades, então o lote deve conter o número de requisições que foram possíveis de ser acumuladas neste intervalo de tempo.

5.2.3.3 Algoritmos usados pela estratégia de adaptação do aPBFT

O aPBFT é implementado a partir de três procedimentos: (i) sensoriamento dos eventos do PBFT; (ii) ajuste do *timeout* de *batching*; e (iii) ajuste do tamanho do *batch*.

O procedimento de sensoriamento é apresentado no Algoritmo 5.13. Neste algoritmo, antes do envio de um PRE-PREPARE, a réplica primária armazena o valor de seu relógio local em T_{ord} e, então, para cada requisição relacionada ao PRE-PREPARE a ser enviado, armazena em *buffer* uma tupla contendo a identificação do cliente, o *timestamp* da requisição e T_{ord} (l. 2–4). Após a execução de uma requisição, a tupla armazenada para tal requisição é lida, e em seguida a replica primária armazena o valor de seu relógio local em T_{exec} (l. 6–7). Os valores do campo T_{ord} da tupla e da variável T_{exec} são usados para o cálculo de intervalo MTE e, em seguida, a tupla é removida definitivamente do *buffer* (l. 8–9).

Os procedimentos de ajuste do *timeout* e do tamanho de *batch* são acionados, pelo procedimento de sensoriamento, a cada nova requisição aceita e a cada lote de requisições executado, respectivamente (l. 10–14).

Algoritmo 5.13: Mecanismo adaptativo proposto para o PBFT: sensoreamento dos eventos relacionados a execução do PBFT.

```
1  on event ( before sending PRE-PREPARE PP ) do
2      assign the local clock timestamp to T_ord;
3      foreach REQUEST r related to PP do
4          buffer tuple <r.client, r.timestamp, T_ord>;
5  on event ( after processing REQUEST r ) do
6      read from buffer the tuple TP where there is a match with r.client and r.timestamp;
7      assign the local clock timestamp to T_exec;
8      use TP.T_ord and T_exec to compute the new MTE;
9      remove from buffer the tuple which match with TP;
10 on event ( after accepting REQUEST R ) do
11     if (is the primary replica) then
12         call procedure AdaptBatchTimeout();
13 on event ( after the execution of the last batch of requests ) do
14     call procedure AdaptBatchSize();
```

O Algoritmo 5.14 apresenta o procedimento para adaptação do *timeout* de *batching*. Este procedimento consiste na estimativa da carga das aplicações (usando MTA, l. 1) e na atribuição

do valor do intervalo médio para ordenação e execução como *timeout* de *batching* (l. 2 e 4). Contudo, antes de definir o valor do *timeout* de *batching*, o procedimento verifica se a carga das aplicações é inferior à capacidade dos estágios de *ordering* e *processing*, e em caso afirmativo o *timeout* de *batching* é definido com valor zero (l. 3).

Algoritmo 5.14: Mecanismo adaptativo proposto para o PBFT: adaptação do *timeout* de *batching*.

```
1 compute MTA;
2 assign MTE to BT;
3 if (MTA ≥ MTE) then assign 0 to BT;
4 if (is the primary replica) then assign BT to batching timeout;
```

Algoritmo 5.15: Mecanismo adaptativo proposto para o PBFT: adaptação do tamanho do *batch*.

```
1 assign 1 to BS;
2 if (MTE have been previously estimated) then
3     compute MTA;
4     estimate BS = ⌈MTE / MTA⌉;
5 assign BS to batching size;
```

O Algoritmo 5.15 apresenta o procedimento para adaptação do tamanho do *batch*. Este procedimento calcula o valor de MTA e usa o valor de MTE previamente calculado para definir o valor do novo tamanho de *batch* (l. 2–4). Caso MTE não tenha sido calculado ainda (por conta de uma mudança de visão, por exemplo), o valor de *BS* é definido como 1 (l. 1). Por fim, o procedimento usa o valor de *BS* para redefinir o tamanho do *batch* usado pelo PBFT (l. 5).

5.2.4 Avaliação de desempenho

O PBFT foi implementado nos principais aspectos relevantes para esta avaliação de desempenho. No lado servidor, a implementação inclui gestão de memória, mudança de visão, *checkpoint* de estados, procedimentos de troca de chaves, transferência de estado entre réplicas, detecção de falhas de réplica primária e mecanismo de *batching*. No lado do cliente, a estratégia de retransmissão inclui o mecanismo de *backoff* exponencial usado para retransmissão.

O serviço implementado considera um sistema de arquivos simulado. Apesar da transferência de estado implementada permitir que uma nova réplica seja completamente integrada e sincronizada a partir de réplicas em execução, o procedimento usado no rejuvenescimento de réplicas não foi implementado. Além deste, a execução especulativa de operações de leitura considerada na implementação original (proposta em [18]) também não foi considerada. A ausência destes procedimentos, entretanto, não influencia na avaliação de desempenho, uma vez que a versão adaptativa foi concebida sobre esta implementação do PBFT e a mesma também não considera tais procedimentos.

Modelo de Simulação

De modo a caracterizar um grupo de processos distribuídos sobre a Internet, o modelo simulado considera canais com atrasos fim-a-fim variados segundo uma distribuição *Lognormal*, o qual caracteriza de forma adequada o tempo de transferência de protocolos HTTP e FTP de acordo com análise de traços em [50]. Tal distribuição de atrasos foi parametrizada com média de $5ms$ e desvio padrão de $15ms$ [4]. Além disso, neste modelo simulado, são considerados pontos de roteamento principal na estrutura subjacente, de modo que há um efeito de enfileiramento que pode alterar o atraso fim-a-fim provido pela função de densidade de probabilidade utilizada. Para tanto, o comutador usado possui capacidade de 10Mbps e esta conectado em enlaces com MTU de $64KB$.

Cada processo executa sobre uma estação simulada com capacidade de processamento de $9Gbps$[5] e o custo de troca de contexto equivalente a $0,001 * 10^{-9}ms$. Cada réplica executa a versão do lado servidor do PBFT, com parâmetros configurados conforme originalmente proposto pelos autores.

Métricas de Desempenho

Para avaliar o desempenho são observados o tempo médio de resposta (RT) e a vazão média no atendimento de requisições (TH). A métrica RT é medida considerando o intervalo médio (em milissegundos) entre o instante de chegada da requisição na réplica e o instante de envio da resposta da mesma ao cliente. A métrica TH é medida considerando o número de atendimentos por segundo.

Além disso, é utilizada uma medida da potência de atendimento (PW) [10], a qual é definida como $PW = \frac{TH}{RT}$. Esta métrica mede a relação de compromisso entre latência de atendimento e vazão, uma vez que, quão menor é o tempo de bloqueio e quão maior é a vazão, maior será o valor PW.

Descrição dos Experimentos

Os experimentos contrapõem o desempenho do PBFT original com a versão adaptativa proposta (aPBFT). Os fatores de simulação são o número de processos clientes (1, 25, 50 e 100), o número de réplicas (4 e 7, tolerando-se, respectivamente, 1 e 2 processos falhos simultaneamente) e a janela de *batching*: tamanho (BS, 1 e 50 no PBFT original) e *timeout* (BT, $10ms$ e $1.000ms$ no PBFT original) – os valores da janela de *batching* são considerados para maximizar a potência do PBFT original em cenários extremos de carga e são definidos de forma dinâmica no aPBFT.

De forma a simular a geração de requisições destes processos clientes, cada cliente utiliza uma função de densidade de probabilidade de Bernoulli para decidir quando enviar uma mensagem, seguindo a taxa de 100 requisições por segundo. Cada série de experimentos simula a execução do protocolo durante um tempo de $20min$ e é repetida três vezes. As métricas de desempenho consideradas são expressas a partir de valores médios e intervalos de confiança

[4] Valores extraídos de medidas de atraso fim-a-fim entre dois sítios conectados a Rede Nacional de Ensino e Pesquisa em Salvador-BA, em que se obteve atraso máximo de $97ms$, mínimo de $1ms$, médio de $5,23ms$ e desvio-padrão de $14,79ms$ para a transmissão de 2000 pacotes de $15KB$ cada.

[5] Estimado considerando uma estação Intel i7, 3.3GHz e 64 bits/instrução.

(com nível de confiança de 95%).

Por fim, as séries de experimentos formam dois grupos: FIX-PAYLOAD, VAR-PAYLOAD. No grupo FIX-PAYLOAD, para um número fixo de 4 réplicas, os tamanhos das requisições e respostas são fixadas em $8KB$. No grupo VAR-PAYLOAD, para um número fixo de 100 clientes, os tamanhos das requisições/respostas na execução variam uniformemente entre de $256B$ a $8KB$, sendo analisado também cenários com números distintos de réplicas (4 e 7). Observa-se que 4 réplicas é o cenário típico de aplicações reais, dificilmente extrapolando para 7 réplicas, tolerando a falha de 1 e 2 réplicas, respectivamente. O número de clientes avaliado neste grupo foi fixado em 100.

Resultados e Discussão

Experimentos do Grupo FIX-PAYLOAD

A Tabela 5.3 apresenta os resultados da avaliação para o grupo FIX-PAYLOAD. Por exemplo, para $BS = 1$, pode-se observar que a variação de BT não afeta de forma substancial o desempenho mensurado por TH e RT – isto pode ser verificado por meio do emparelhamento dos desempenhos médios[6] para cada métrica em cada cenário, ou seja, 1, 25, 50 e 100 clientes. Nestes casos, TH e RT do PBFT aumentam apenas com o aumento do número de clientes: TH aumenta para um maior número de requisições submetidas à máquina de estados, enquanto que RT aumenta por causa do respectivo aumento dos custos (de processamento e troca de mensagens) oriundos do aumento da carga.

Para as configurações de PBFT com $BS = 50$, os efeitos dos valores atribuídos a BT se tornam mais perceptíveis. Valores maiores de BT implicam em piores desempenhos médios em termos de TH e RT. Quando BS é maior que o número de clientes ativos, o lote de requisições só será enviado ao final do timeout. Deste modo, para $BT = 1.000ms$, ou seja, $BT = 1s$, tem-se $PW = 1$ e 25 requisições processadas por segundo, – exatamente o número de clientes ativos em cada caso, conforme apresentado nas Tabelas 5.3(a) e 5.3(b).

Por outro lado, quando BS é menor ou igual ao número de clientes ativos, as diferenças médias de desempenho entre as configurações de PBFT com $BS = 50$ e $BT = 10ms$ ou $1.000ms$ se devem ao fato de que um maior BT é influenciado pela variabilidade dos atrasos de processamento e comunicação induzidos pelo grande número de clientes, conforme pode-se ver nas Tabelas 5.3(c) e 5.3(d). Quando $BT = 10ms$, os lotes são preenchidos em $10ms$ independente de BS, o que garante que sempre existem lotes sendo processados pela máquina de estados, na presença de pelo menos uma requisição em *batching*. Para $BT = 1.000ms$, por outro lado, implica em uma redução no número de lotes processados, de fato reduzindo TH. Este efeito é mais evidente para um cenário com 50 clientes ativos e PBFT configurado com $BS = 50$ e $BT = 1.000ms$, conforme pode-se ver nas Tabelas 5.3(c).

Os resultados obtidos para o PBFT original demonstram o compromisso com o desempenho

[6]Na comparação por emparelhamento, dois desempenhos médios m_1 e m_2 (com $m_1 > m_2$) são comparados considerando os seus respectivos intervalos de confiança (c_1 e c_2). Se $m_1 - c_1 \leq m_2$ ou $m_2 + c_2 \geq m_1$, então não se pode afirmar que existe diferença em termos de desempenho, ver [10].

(a) 1 cliente

BS	$BT(ms)$	$TH(req/s)$	$RT(ms)$	PW
1	10	$39,9 \pm 0,1$	$10,3 \pm 1,6$	3,9
1	1000	$40,1 \pm 0,1$	$10,4 \pm 1,6$	3,9
50	10	$28,6 \pm 0,1$	$15,8 \pm 3,4$	1,8
50	1000	$1,0 \pm 0,1$	$20,5 \pm 27,0$	0,1
	aPBFT	$40,0 \pm 0,1$	$10,4 \pm 2,0$	3,9

(b) 25 clientes

BS	$BT(ms)$	$TH(req/s)$	$RT(ms)$	PW
1	10	$497,0 \pm 2,5$	$73,9 \pm 124,8$	6,7
1	1000	$494,0 \pm 1,3$	$55,1 \pm 79,7$	9,0
50	10	$809,0 \pm 0,1$	$16,0 \pm 3,4$	50,6
50	1000	$24,5 \pm 0,4$	$25,0 \pm 99,0$	1,0
	aPBFT	$837,5 \pm 0,1$	$15,1 \pm 5,2$	55,6

(c) 50 clientes

BS	$BT(ms)$	$TH(req/s)$	$RT(ms)$	PW
1	10	$495,8 \pm 1,8$	$72,4 \pm 126,1$	6,9
1	1000	$496,8 \pm 2,4$	$63,4 \pm 123,6$	7,8
50	10	$1634,9 \pm 0,1$	$46,2 \pm 94,9$	35,4
50	1000	$845,5 \pm 0,1$	$30,0 \pm 86,5$	28,2
	aPBFT	$1667,6 \pm 0,1$	$15,2 \pm 4,2$	110,1

(d) 100 clientes

BS	$BT(ms)$	$TH(req/s)$	$RT(ms)$	PW
1	10	$496,2 \pm 2,0$	$264,1 \pm 164,9$	1,9
1	1000	$496,4 \pm 2,2$	$217,6 \pm 147,2$	2,3
50	10	$3273,8 \pm 0,1$	$15,8 \pm 3,4$	208,1
50	1000	$2883,7 \pm 0,1$	$20,1 \pm 19,6$	143,3
	aPBFT	$3325,9 \pm 0,1$	$15,3 \pm 3,4$	217,9

Tabela 5.3. Resultados das simulações do grupo de experimentos FIX-PAYLOAD.

da configuração para os parâmetros do mecanismo de *batching*. Por exemplo, no cenário com $BS = 50$ e $BT = 10ms$, pode-se observar que o desempenho do protocolo é melhor que a configuração com $BS = 1$ (a qual é similar ao protocolo sem mecanismos de *batching*). A versão adaptativa proposta (aPBFT), por sua vez, consegue um desempenho sempre equivalente ou superior ao PBFT original, para quaisquer dos cenários analisados – observe a variação de *PW* para as diferentes configurações na Figura 5.14.

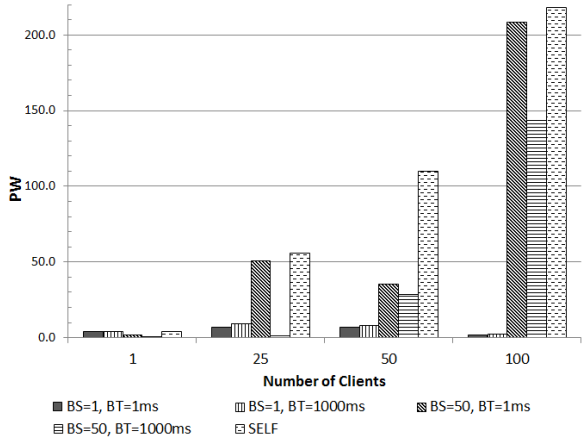


Figura 5.14. Gráfico da resposta de *PW* da versão adaptativa do PBFT – grupo de experimentos FIX-PAYLOAD.

Por fim, observa-se que os melhores resultados do aPBFT devem-se ao fato de que o ajuste dinâmico de parâmetros amortiza os efeitos de alta carga de trabalho com maiores valores para *BT* e *BS* e, ao mesmo tempo, o protocolo aPBFT se ajusta para rápido processamento dos lotes em condições de baixa carga – isto é, aPBFT trabalha como um mecanismo de controle de fluxo que auto-ajusta automaticamente o fluxo da máquina de estados. Tais benefícios podem ser observados comparando-se os resultados obtidos em diferentes configurações entre o clássico

PBFT e a versão aPBFT – conforme apresentado na Tabela 5.3.

Experimentos do Grupo VAR-PAYLOAD

A Tabela 5.4 apresenta os resultados da série de experimentos do grupo VAR-PAYLOAD para 100 clientes, com configurações com 4 e 7 réplicas. Podemos observar que os resultados para as métricas observadas são similares aos obtidos no grupo FIX-PAYLOAD para as configurações equivalentes.

(a) 4 réplicas e 100 clientes

BS	$BT(ms)$	$TH(req/s)$	$RT(ms)$	PW
1	10	$496,6 \pm 2,3$	$78,5 \pm 152,5$	6,3
1	1000	$496,9 \pm 2,6$	$52,18 \pm 124,7$	9,5
50	10	$3277,75 \pm 0,1$	$15,7 \pm 3,4$	209,2
50	1000	$2882,6 \pm 0,1$	$19,8 \pm 5,9$	145,7
	aPBFT	$3325,3 \pm 0,1$	$15,3 \pm 3,4$	217,9

(b) 7 réplicas e 100 clientes

BS	$BT(ms)$	$TH(req/s)$	$RT(ms)$	PW
1	10	$494,0 \pm 0,4$	$44,2 \pm 71,6$	11,18
1	1000	$494,7 \pm 1,1$	$33,00 \pm 60,977$	14,9
50	10	$3275,91 \pm 0,1$	$17,6 \pm 5,2$	185,9
50	1000	$2878,6 \pm 0,1$	$19,8 \pm 5,9$	145,1
	aPBFT	$3331,28 \pm 0,2$	$15,2 \pm 3,5$	218,7

Tabela 5.4. Resultados das simulações do grupo de experimentos VAR-PAYLOAD.

Pode-se observar que em cenários de 4 e 7 réplicas há um leve incremento no desempenho em termos de *RT* (menores valores) – uma vez que a presença de mensagens de menor tamanho neste cenário implica em menores custos de processamento criptográfico e de transmissão. As vantagens da versão adaptativa podem ser melhor observadas pela variação da métrica *PW* para as diferentes configurações.

Pode-se observar os efeitos do auto-ajuste da janela de *batching* por meio do traço dos valores de *BS* e *BT*: observamos que estes valores rapidamente convergem a uma pequena região de variação. Os traços das Figuras 5.15 e 5.16 mostram, respectivamente, para um cenário de 100 clientes e 4 réplicas em uma janela de observação de 12.000*ms* que *BT* variou de 8*ms* a 14*ms*, e *BS* variou de 24 a 47 requisições por lote.

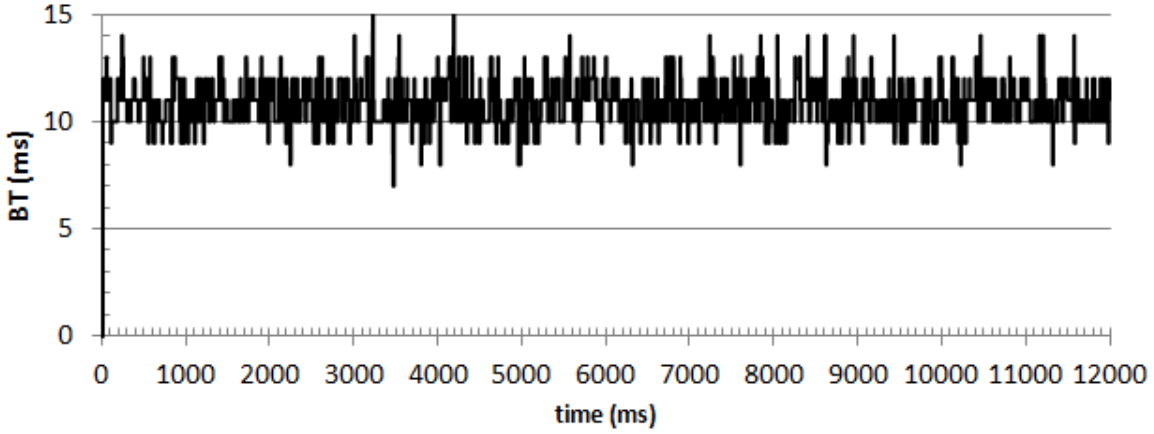


Figura 5.15. Gráfico da resposta de *BT* da versão adaptativa do PBFT (100 clientes) – grupo de experimentos VAR-PAYLOAD.

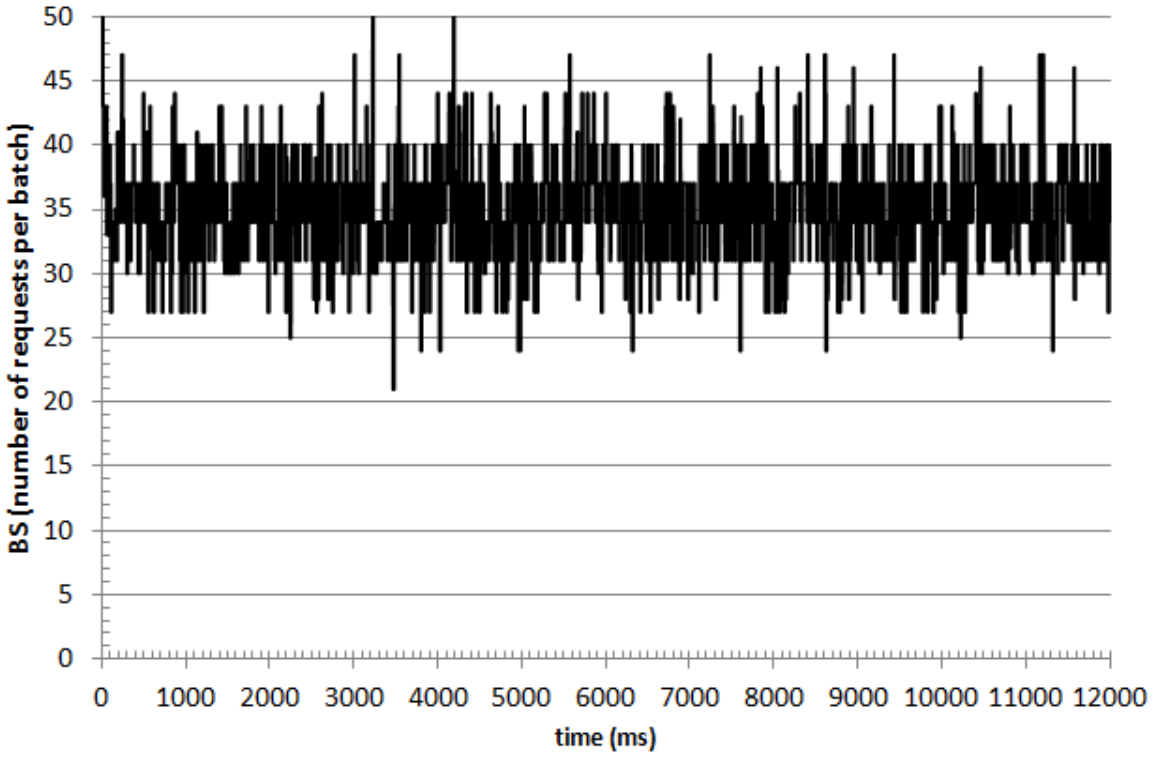


Figura 5.16. Gráfico da resposta de *BS* da versão adaptativa do PBFT (100 clientes) – grupo de experimentos VAR-PAYLOAD.

5.3 CONSENSO EM CENÁRIOS HÍBRIDOS E DINÂMICOS

Esta seção apresenta a avaliação de desempenho de um protocolo de consenso distribuído apresentado em [6], especialmente desenhado para adaptar o quórum a um cenário híbrido. Canais de comunicação podem variar sua qualidade de serviço (entre *timely* e *untimely*) em tempo de execução e processos podem falhar por *crash*; ainda, por meio do subsistema de monitoramento e de detectores de defeitos, processos são classificados nos conjuntos $live_i$, $down_i$ e $uncertain_i$.

Como dito anteriormente, a percepção de tais conjuntos ($live_i$, $down_i$ e $uncertain_i$) pode variar ao longo do tempo de execução, conforme os dados coletados pelo monitoramento e as regras de transição: processos podem degradar sua qualidade de serviço ao longo da execução do protocolo (e.g. mover de *live* para *uncertain*), mas não pode ocorrer *upgrade* (e.g. um processo em *uncertain* não move-se para *live* ou *down*). O modelo de sistema permite desenvolver protocolos adaptativos a partir da evolução da informação acerca dos conjuntos de processos.

Em especial, o consenso adaptativo apresentado em [6] utiliza um quórum adaptativo, a partir da qualidade de serviço de processos e canais expressa pelos conjuntos. No cenário síncrono (i.e. todos processos e canais de comunicação são *timely*), *uncertain* é vazio e até $n-1$ processos podem falhar por *crash*.; no caso assíncrono, todos processos pertencem a *uncertain* e no máximo $\frac{n}{2}-1$ processos podem falhar por *crash*.

5.3.1 Avaliação de desempenho

Modelo de Simulação

O progresso do consenso é avaliado no grupo de experimentos FED-CLOUD: *clusters* conectados por meio da Internet. Cada *cluster* representa um grupo de computadores interconectados por uma rede local, onde, pode-se assumir um comportamento síncrono, contudo a nuvem não é em si um sistema síncrono. Este ambiente é representado por um modelo de sistema distribuído híbrido, onde cada cluster local é um sub-sistema síncrono interconectado por canais de comunicação assíncronos com os *clusters* remotos.

De forma similar aos experimentos de replicação bizantina, o comportamento dos canais de comunicação entre os *clusters* (*untimely*) foi modelado por meio de distribuição *Lognormal*, parametrizada com média de $5ms$ e desvio padrão de $15ms$. Os canais de comunicação entre os processos de um mesmo *cluster* são determinísticos (*timely*) com latência máxima de $2ms$.

O Apêndice B apresenta a documentação dos códigos-fonte associados ao cenário de avaliação do grupo de experimentos FED-CLOUD.

Em um outro grupo de experimentos, denominado QOS-DEGRAD, é mensurado como um processo p_i percebe e adapta o seu quórum à modificações dinâmicas na qualisdade de serviço. O cenário inicia-se em um ambiente com reserva de recursos que mantém configuração de canais de comunicação *timely* entre todos processos, os quais gradativamente degradarão para canais *untimely*.

Deve-se notar que em ambos cenários de execução, processos possuem acesso a uma infraestrutura local de gerenciamento da qualidade de serviço de modo a poder inferir a qualidade de serviço dos canais de comunicação.

Métricas de Desempenho

De forma similar à avaliação de desempenho da replicação bizantina, para avaliar o desempenho são observados o tempo médio de resposta (RT) e a vazão média no atendimento de requisições (TH). A métrica RT é medida considerando o intervalo médio (em milissegundos) entre o instante de início e o instante de término de um consenso. A métrica TH é medida considerando o número de consensos por segundo. Além disso, também é utilizada a medida da potência de atendimento (PW) [10], $PW = \frac{TH}{RT}$.

Descrição dos Experimentos

O grupo de experimentos FED-CLOUD avalia como o consenso progride em um cenário híbrido. Os fatores de simulação são: número de processos (40, 80 e 120) e percentual de processos que falham ao longo da execução (0, 30% e 60%).

É gerada uma sequencia de requisições de consenso, onde um novo consenso inicia-se após

o término do anterior. Cada experimento simula a execução do protocolo por um período de 20 minutos, sendo repetido três vezes. As métricas de desempenho consideradas são expressas a partir de valores médios e intervalos de confiança (com nível de confiança de 95%).

No grupo de experimentos QOS-DEGRAD, a classe **Scenario** foi estendida para produzir a degradação da qualidade de serviço em tempo de execução. Ou seja, ao longo da execução, os canais de comunicação inicialmente *timely* degradam para *untimely* até que todos canais de comunicação perdem o comportamento *timely*.

Resultados e Discussão

A tabela 5.5 apresenta os dados coletados para o grupo de experimentos FED-CLOUD. Como pode-se observar, o algoritmo de consenso adaptativo utiliza do monitoramento da qualidade de serviço de processos e canais de comunicação, e o consenso progride mesmo na presença de uma maioria de processos falhos (e.g. 60%). Como pode-se observar, do tempo de resposta (RT) e da potência (PW), não há diferença significativa para um mesmo número de processos entre o cenário sem falhas e os cenários com falhas, embora o RT aumente de acordo com o percentual de processos falhos, em face da latência da exclusão dos processos falhos do quórum do consenso.

Tabela 5.5. Resultados das simulações do grupo de experimentos FED-CLOUD.

Número de Processos (n)	Percentual de Processos Falhos (f)	$RT(ms)$	$TH(req/ms)$	PW
40	0	$26,68 \pm 0,08$	$30,67 \pm 0,10$	1,15
40	30%	$27,07 \pm 0,09$	$30,67 \pm 0,10$	1,13
40	60%	$27,22 \pm 0,11$	$30,67 \pm 0,10$	1,13
80	0	$25,61 \pm 0,02$	$24,00 \pm 0,01$	0,94
80	30%	$25,61 \pm 0,10$	$24,00 \pm 0,02$	0,94
80	60%	$26,56 \pm 0,06$	$24,00 \pm 0,01$	0,90
120	0	$27,83 \pm 0,03$	$16,00 \pm 0,01$	0,57
120	30%	$31,50 \pm 0,29$	$17,33 \pm 0,10$	0,55
120	60%	$32,60 \pm 0,26$	$18,67 \pm 0,10$	0,57

Na Figura 5.17, podemos observar no cenário QOS-DEGRAD que a medida que o processo p_0 percebe a degradação da qualidade de serviço, o seu conjunto *uncertain* aumenta, mudando o quórum do consenso adaptativo. Inicialmente todos canais de comunicação são *timely* e nenhum processo pertence a *uncertain*. Este conjunto então gradativamente aumenta de acordo com modificações na qualidade de serviço. Quando nenhum outro processo *timely* é conectado a p_0, este exclui a si mesmo de *live*, movendo-se para *uncertain*.

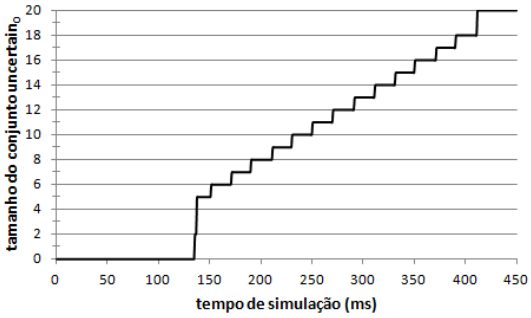


Figura 5.17. Gráfico do consenso adaptativo no grupo de experimentos QOS-DEGRAD: tamanho de $uncertain_0$.

5.4 CONCLUSÕES

Este capítulo apresentou a avaliação de desempenho de diferentes propostas para cenários de comunicação em grupo e de replicação tolerante a falhas bizantinas, utilizando o *framework* de simulação apresentado e comparando as propostas com abordagens clássicas existentes que apresentam deficiências nos cenários estudados.

Protocolos de comunicação em grupo combinam um conjunto de mecanismos, como, por exemplo, a instalação de visões, consenso e difusão atômica confiável, de modo a prover uma abstração adequada de comunicação para serviços de computação distribuída, como a replicação ativa de servidores. Diferentes implementações de protocolos de comunicação em grupo têm sido propostos, de acordo com o ambiente desejado e o conjunto de propriedades a garantir para o serviço [54, 56, 57].

A partir do mecanismo *timed causal blocks* [5], foi realizada uma avaliação preliminar deste mecanismo em [65], e implementado e avaliado em ambiente síncrono um protocolo genérico de comunicação em grupo em [8], por fim, este protocolo é estendido para suportar o acordo uniforme e avaliado de forma mais abrangente em [35]. Este protocolo de comunicação em grupo, permite o ajuste da execução ao nível de qualidade de serviço dos componentes do sistema distribuído, caracterizando um sistema *self-aware*, e o desenvolvimento e análise de desempenho deste protocolo em comparação com abordagens clássicas no caso síncrono e assíncrono.

Desta forma, o protocolo adequa-se a execução de comunicação em grupo em ambientes híbridos. Por meio deste, os mesmos algoritmos podem ser instanciados em diferentes modelos de sistema (síncrono, assíncrono, ou híbrido), simplificando o desenho do sistema. No caso síncrono, a completude lógica dos blocos provê entrega adiantada em relação aos limites dos *timeouts* associados aos blocos, e no caso de expiração destes, há um detector de defeitos perfeito integrado ao protocolo. No caso assíncrono, os *timeouts* indicam suspeita de falhas. O algoritmo pode adaptar sua execução aos diferentes níveis de qualidade de serviço existentes, otimizando o funcionamento.

Destacamos ainda o desenvolvimento também a partir do mecanismo *timed causal blocks* de um protocolo de comunicação em grupo cujo ajuste do parâmetro ts do mecanismo de *time-*

silence observa o nível acordado pelo usuário para o consumo de recursos do sistema, se auto-configurando mesmo em face a mudanças no ambiente, como, por exemplo, variações na carga de trabalho. A construção de algoritmos auto-gerenciáveis e a abordagem de ajuste proposta para este novo protocolo são contribuições de outro trabalho de tese apresentado em [17].

Desta forma, foi realizado o desenvolvimento do mecanismo básico do protocolo, a identificação de métricas e dos pontos de sensoreamento e de atuação específicos para o protocolo auto-gerenciável, e, em especial, a implementação e a análise de desempenho comparativa entre uma versão auto-gerenciável e outra clássica em que a configuração de parâmetros é fixa. Os resultados foram apresentados inicialmente em [71] e, posteriormente, de forma estendida em [72].

Em geral, protocolos de comunicação em grupo não suportam auto-configuração dinâmica dos seus parâmetros operacionais a partir dos requisitos do usuário. Esta abordagem permite o ajuste dos parâmetros de modo a obter uma solução de compromisso combinando desempenho, dado pela latência na entrega de mensagens, e custo, dado pelo *overhead*. A abordagem utiliza teoria de controle realimentado, provendo o auto-ajuste em tempo de execução.

A construção dos protocolos apresentados, baseados no mecanismo *timed causal blocks*, permitiu integrar todos os mecanismos do protocolo de comunicação em grupo, detecção de falhas, ordenação de mensagens e reconfiguração do grupo, otimizando a execução dos protocolos. Protocolos baseados em consenso [58] apresentam um alto custo de execução. No caso assíncrono, abordagens assimétricas [57] podem ser mais eficientes no número de mensagens transmiticas, contudo necessitarão de mensagens de *heartbeat* adicionais, em face da detecção de defeitos não ser integrada ao mecanismo de entrega da mesma forma que o mecanismo de *TimedCB*.

Quanto à replicação bizantina, o PBFT apresentou um marco importante, consolidando diferentes técnicas de engenharia para a implementação prática de um protocolo de replicação tolerante a intrusões. A partir do PBFT, diferentes protocolos de replicação tolerante a falhas bizantinas podem ser compostos. Tais protocolos podem ser decompostos em um conjunto de sub-protocolos com funções distintas: de acordo, de mudança de visão, de recuperação pro-ativa e de *checkpoint*. Bem como requisições podem ser processadas em lote, a partir do primário, reduzindo o custo computacional da autenticação de mensagens. De fato, Zyzzyva [81] apresenta um sub-protocolo de acordo distinto do PBFT, mais otimista, baseado em execução especulativa, e [82] propõe combinar diferentes estratégias baseadas em diferentes sub-protocolos de acordo.

Contudo, em todos os casos, há um conjunto de parâmetros que definem o ponto de operação de cada um destes sub-protocolos e que podem ser ajustados para maximizar o desempenho de acordo com o volume de carga e demais características do ambiente. Definir tais parâmetros em tempo de projeto é possível, contudo, se o ambiente se apresentar extremamente dinâmico, tais características se alterarão bastante ao longo do tempo e será necessário reobservar o ambiente e reconfigurar tais parâmetros. O uso de uma estratégia adaptativa, baseada em uma malha

de controle fechada, em que o protocolo auto-ajusta seus parâmetros conforme o observado no ambiente permite que a reconfiguração ocorra de forma dinâmica ao longo da execução do protocolo.

Esta proposta adaptativa para o protocolo PBFT, publicada em [77–79], se baseia no ajuste paramétrico da janela de *batching* e a análise de desempenho desta versão em comparativo com o protocolo clássico PBFT demonstra que a mesma se apresenta mais eficiente em cenários que as características do ambiente (e.g. carga de trabalho) varia, ao longo do tempo.

Por fim, a avaliação de desempenho do consenso adaptativo nos permitiu exercitar o *framework* de simulação em cenários com características híbridas e dinâmicas, bem como verificar como o protocolo de consenso adaptativo proposto em [6] adapta o quórum do consenso a qualidade de serviço existente no ambiente distribuído.

Capítulo 6

CONSIDERAÇÕES FINAIS

Como já discutido, diferentes ambientes computacionais dos dias atuais representam sistemas distribuídos cujos componentes possuem características híbridas ou dinâmicas, e que não são adequadamente representados pelos modelos clássicos convencionais síncrono e assíncrono. Diferentes propostas foram apresentadas na Literatura na construção de modelos adequados a caracterizar tal comportamento [4–8]. A partir destes modelos, podemos construir algoritmos adaptáveis, os quais, por exemplo, otimizam o seu desempenho de acordo com o nível de sincronia existente no ambiente.

A avaliação destes algoritmos não é trivial: o uso de medição pode não ser escalável, dada a necessidade de reproduzir uma infraestrutura heterogênea que represente uma quantidade significativa de nós com diferentes comportamentos; e o estudo de modelos analíticos, baseados em teoria das filas ou redes de Petri, pode ser demasiadamente complexo, em face dos diferentes comportamentos e da possibilidade de alteração destes ao longo da execução.

Quanto à avaliação por meio de simulação, ambientes genéricos de eventos discretos (e.g. SMPL [11]) e simuladores de redes de computadores (e.g. NS-2 [12], NS-3 [13] e Omnet++ [14]) não possuem o nível de abstração adequado para a representação do comportamento fim-a-fim dos sistemas distribuídos.

As características desejáveis para este simulação incluem desde a composição de cenários híbridos – ou seja, com alguns componentes com restrições de tempo real e outros não; ativação de tarefas baseadas em agendamento no tempo (*time-triggered*) ou por evento (*event-triggered*), à configuração de componentes com características síncronas e assíncronas, com possibilidade de alteração do nível de serviço ao longo da execução e à representação de diferentes modelos de falhas; simulação de relógio local (não global), a partir do qual cada processo baseia sua computação, entre outras.

Já os ambientes adequados a simulação de ambientes distribuídos Neko [15] e Simmcast [16] não atendem a todas estas características. Por exemplo, o Neko não permite agendamento de tarefas de tempo real em janelas de tempo estritas, e o Simmcast não prevê a simulação de

relógio físico local a cada processo, o que permitiria representar dessincronia entre processos ao longo do tempo.

Tais fragilidades motivaram a proposta de um *framework* de simulação adequado a estes ambientes para avaliação de desempenho de sistemas distribuídos híbridos e dinâmicos. Tal proposta permite avaliar os diferentes modelos de sistemas distribuídos, desde os casos clássicos síncrono e assíncrono, até os sistemas parcialmente síncronos e sistemas ditos híbridos ou dinâmicos, em que as características dos componentes (processos e canais de comunicação) podem variar no espaço ou no tempo.

O *framework* atende às características elencadas e permite estender o comportamento de seus blocos principais, criando e derivando novos ambientes de simulação. Permite ainda balancear o nível de abstração da simulação de forma conveniente ao desenvolvedor, representando tão somente comportamentos fim-a-fim caracterizados por funções determinísticas ou probabilísticas, ou utilizar blocos que caracterizem a infraestrutura subjacente de processamento e de comunicação. A construção do *framework* utilizou um modelo de simulação de sistemas distribuídos de referência que propomos, o qual se baseia em um modelo de sistema distribuído genérico, que permite representar os diferentes modelos de sistema propostos na literatura.

Utilizamos este ambiente no desenho e avaliação de uma proposta de protocolo de comunicação em grupo que avançou no desenho de sistemas distribuídos *self-aware*: dado um sistema distribuído com características híbridas e dinâmicas, assumimos uma infraestrutura subjacente de monitoramento, que nos permite obter informações sobre a qualidade de serviço do ambiente e adaptar o comportamento do algoritmo proposto.

Ainda, este ambiente foi utilizado na avaliação de algoritmos baseados em teoria de controle[17, 72, 79], que adaptam o seu comportamento em tempo de execução de acordo com as condições do ambiente e de requisitos de serviço especificados pelo usuário. Estes algoritmos são um protocolo de comunicação em grupo, que adequa seus parâmetros a carga de trabalho e ao consumo de recursos especificado pelo usuário e uma versão adaptativa do conhecido protocolo de replicação tolerante a falhas bizantinas PBFT, que adequa a janela de *batching* de modo a otimizar a sua vazão e tempo de resposta de acordo com as condições de carga do ambiente.

6.1 PUBLICAÇÕES E TRABALHOS FUTUROS

Os resultados desta tese foram apresentados nas seguintes publicações:

i) Freitas, Allan Edgard Silva; Macêdo, R. J. A. Um ambiente para testes e simulações de protocolos confiáveis em sistemas distribuídos híbridos e dinâmicos. In: **Anais do XXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)**. Porto Alegre, Brazil: SBC, 2009. v. 1, p. 826–839.

ii) Macêdo, R. J. A.; Freitas, Allan Edgard Silva. A generic group communication approach for hybrid distributed systems. In: **Proceedings of the 9th IFIP International Confe-**

rence on Distributed Applications and Interoperable Systems (DAIS). Berlim, Germany: Springer-Verlag, 2009. (Lecture Notes on Computer Science, 4), p. 102–115.

iii) Macêdo, Raimundo J. A.; Freitas, Allan E. S. Group communication for self-aware distributed systems. In: **Proceedings of the Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)**. Porto Alegre, Brazil: SBC, 2010. p. 915–928.

iv) Macêdo, R.J.A.; Freitas, A.E.S.; Sá, A.S. de. A self-manageable group communication protocol for partially synchronous distributed systems. In: **Proceedings of the 5th Latin-American Symposium on Dependable Computing (LADC)**. Palo Alto, CA, USA: IEEE, 2011. p. 146–155.

v) Sá, A. S. de; Freitas, A. E. S.; Macêdo, R. J. A. Auto-configuração de máquina de estados tolerante a falhas bizantinas. In: **Proceedings of the 2nd Workshop on Autonomic Distributed Systems (WoSiDA)**. Porto Alegre, Brazil: SBC, 2012. p. 35–38.

vi) Sá, A. S. de; Freitas, A. E. S.; Macêdo, R. J. A. Request batching self-configuration in byzantine fault-tolerant replication. In: **Proceedings of the 2nd Brazilian Symposium on Computing System Engineering: Operating Systems Workshop (SBESC-WSO)**. Porto Alegre, Brazil: SBC, 2012. p. 1–6. (**melhor artigo da conferência**)

vii) Sá, A. S. de; Freitas, A. E. S.; Macêdo, R. J. A. Adaptive request batching for byzantine replication. **ACM SIGOPS Operating Systems Review**, ACM, v. 47, n. 1, p. 35–42, January 2013.

viii) Macêdo, R.J.A.; Freitas, A.E.S.; Sá, A.S. de. Enhancing group communication with self-manageable behavior. **Journal of Parallel and Distributed Computing**, Elsevier, v. 73, n. 4, p. 420–433, April 2013.

ix) Macêdo, R. J. A.; Freitas, A. E. S. Dos sistemas distribuídos self-aware. In: **Proceedings of the 3rd Workshop on Autonomic Distributed Systems (WoSiDA)**. Porto Alegre, Brazil: SBC, 2013. p. 1–4.

O artigo publicado no SBRC em 2009 apresenta a primeira versão do ambiente de simulação e uma avaliação preliminar de comunicação em grupo utilizando o mecanismo de *timed causal blocks*. Posteriormente no DAIS em 2009, apresentamos um protocolo completo de comunicação em grupo baseado no mecanismo e adaptável de acordo com as condições observadas no ambiente (*self-aware*) a um ambiente híbrido e dinâmico, realizando uma avaliação no caso síncrono. O protocolo fora estendido, atendendo a propriedade de acordo uniforme na entrega de mensagens, e uma avaliação mais detalhada considerando falhas e comparativo com abordagens clássicas no cenário síncrono e no cenário assíncrono é feita no artigo publicado no SBRC de 2010.

No artigo publicado no LADC de 2011, apresentamos uma versão do protocolo de comunicação de grupo para sistemas parcialmente síncronos com hipótese *GST*, tornando o protocolo auto-gerenciável: o mesmo se adequa ao requisito de consumo de recursos definido pelo usuário. Este artigo fora estendido, com uma avaliação mais abrangente, incluindo prototipação em cenário real, e experimento da mesma abordagem auto-gerenciável em um protocolo baseado em sequenciador, sendo submetido em 2012 e aceito para publicação em 2013 no periódico *Journal of Parallel and Distributed Computing*.

A proposta adaptativa para ajuste dos parâmetros da janela de *batch* do protocolo PBFT foi apresentada de forma preliminar no WoSida em 2012, e de forma estendida no SBESC-WSO do mesmo ano, no qual obteve premiação de melhor artigo, sendo publicada versão extendida no períodico *Operating Systems Review* da ACM.

Por fim, no WoSida em 2013, apresentamos uma discussão mais extensa sobre os sistemas distribuídos *self-aware* e como compor algoritmos adaptáveis a tais sistemas.

Como alguns dos possíveis trabalhos futuros desta Tese, destaca-se estender a investigação e desenvolvimento de protocolos distribuídos para os cenários de sistemas distribuídos híbridos e dinâmicos, bem como ampliar as funcionalidades do *framework* ora apresentado, pela integração a simuladores de redes, utilizando o mesmo como uma camada do *framework*, e pela extensão do suporte estatístico do ambiente, permitindo além da integração ao R, a integração com ambientes como MATLAB e Scilab.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] FISHER, M. J.; LYNCH, N.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. **Journal of the ACM**, v. 32, n. 2, p. 374–382, April 1985.

[2] ROCHWERGER, B. et al. The reservoir model and architecture for open federated cloud computing. **IBM Journal of Research and Development**, v. 53, n. 4, p. 4:1–4:11, July 2009.

[3] LEE, E. A. Cyber physical systems: Design challenges. In: IEEE. **Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)**. Palo Alto, CA, USA, 2008. p. 363–369.

[4] GORENDER, S.; MACÊDO, R. J. A.; RAYNAL, M. A hybrid and adaptive model for fault-tolerant distributed computing. In: **Proceedings of the IEEE/IFIP International Conference on Computer Systems and Networks (DNS)**. Palo Alto, CA, USA: IEEE/IFIP, 2005. p. 412–421.

[5] MACÊDO, R. J. A. An integrated group communication infrastructure for hybrid real-time distributed systems. In: **Proceedings of the 9th Workshop on Real-Time Systems (WTR)**. Porto Alegre, Brazil: SBC, 2007. p. 81–88.

[6] GORENDER, S.; MACÊDO, R. J. A.; RAYNAL, M. An adaptive programming model for fault-tolerant distributed computing. **IEEE Transactions on Dependable and Secure Computing**, v. 4, n. 1, p. 18–31, January 2007.

[7] MACÊDO, R. J. A.; GORENDER, S. Perfect failure detection in the partitioned synchronous distributed system model. In: **Proceedings of the 4th International Conference on Availability, Reliability and Security (ARES)**. Palo Alto, CA, USA: IEEE, 2009. p. 273–280.

[8] MACÊDO, R. J. A.; FREITAS, A. E. S. A generic group communication approach for hybrid distributed systems. In: **Proceedings of the 9th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)**. Berlim, Germany: Springer-Verlag, 2009. (Lecture Notes on Computer Science, 4), p. 102–115.

[9] MACÊDO, R. J. A.; GORENDER, S. Detectores perfeitos em sistemas distribuídos não síncronos. In: **Anais do IX Workshop de Teste e Tolerância a Falhas (WTF)**. Porto Alegre, Brazil: SBC, 2008. p. 127–140.

[10] JAIN, R. **The Art of Computer Systems Performance Analysis**. New York, NY, USA: Wiley & Sons, 1991.

[11] MACDOUGALL, M. H. **SMPL-A Simple Portable Simulation Language**. Sunny Vale, CA, USA: Amdahl, 1980.

[12] BRESLAU, L. et al. Advances in network simulation. **Computer**, v. 33, n. 5, p. 59–67, May 2000.

[13] HENDERSON, T. et al. Ns-3 project goals. In: **Proceedings of the Workshop on NS-2: the IP Network Simulator (WNS)**. New York, NY, USA: ACM, 2006. p. 1–8.

[14] VARGA, A. et al. The omnet++ discrete event simulation system. In: **Proceedings of the 15th European Simulation Multiconference (ESM)**. Ostend, Belgium: EUROSIS, 2001. v. 9.

[15] URBAN, P.; DEFAGO, X.; SCHIPER, A. Neko: a single environment to simulate and prototype distributed algorithms. In: **Proceedings of 15th International Conference on Information Networking (ICOIN)**. Palo Alto, CA, USA: IEEE, 2001. p. 503–511.

[16] BARCELLOS, M. P. et al. Beyond network simulators: Fostering novel distributed applications and protocols through extendible design. **Journal of Network and Computer Applications**, v. 35, n. 1, p. 328 – 339, January 2012.

[17] Sá, A. S. de. **Mecanismos Autonômicos de Tolerância a Falhas para Sistemas Distribuídos**. Dissertação (Tese de Doutorado em Ciência da Computação) — Departamento de Ciência da Computação, Universidade Federal da Bahia, 2011.

[18] CASTRO, M.; LISKOV, B. Practical Byzantine fault tolerance. In: **Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)**. [S.l.]: USENIX, 1999. p. 173–186.

[19] LYNCH, N. **Distributed Algorithms**. San Francisco, CA, USA: Morgan Kaufmann Publishers, Inc., 1996.

[20] BALDONI, R.; BONOMI, S.; RAYNAL, M. Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. **IEEE Transactions on Parallel and Distributed Systems**, v. 23, n. 1, p. 102–109, January 2012.

[21] AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 1, p. 11–33, january 2004.

[22] LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. **ACM Transactions on Programming Languages and Systems**, v. 4, n. 3, p. 382–401, July 1982.

[23] AGUILERA, M. K.; CHEN, W.; TOUEG, S. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In: **Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG)**. Berlin, Germany: Springer-Verlag, 1997. (Lecture Notes in Computer Science, v. 1320), p. 126–140.

[24] BASU, A.; CHARRON-BOST, B.; TOUEG, S. Simulating reliable links with unreliable links in the presence of process crashes. In: **Proceedings of 10th International Workshop on Distributed algorithms (WDAG)**. Berlin, Germany: Springer-Verlag, 1996. (Lecture Notes in Computer Science, v. 1151), p. 105–122.

[25] JOHNSON, M. Proof that timing requirements of the fddi token ring protocol are satisfied. **IEEE Transactions on Communications**, v. 35, n. 6, p. 620–625, June 1987.

[26] DOLEV, D.; DWORK, C.; STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. **Journal of the ACM**, v. 34, n. 1, p. 77–97, January 1987.

[27] GHOSH, S. **Distributed systems: an algorithmic approach**. London, UK: Chapman & Hall/CRC, 2006. (Computer & Information Science Series, v. 13).

[28] CRISTIAN, F. Agreeing on who is present and who is absent in a synchronous distributed system. In: **Digest of Papers of the 18th IEEE International Symposium on Fault-Tolerant Computing (FTCS)**. Palo Alto, CA, USA: IEEE, 1988. p. 206–211.

[29] CHANDY, K.M.; LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. **ACM Transactions on Computer Systems**, v. 3, n. 1, p. 63–75, February 1985.

[30] CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. **Journal of the ACM**, v. 43, n. 2, p. 225–267, March 1996.

[31] CRISTIAN, F.; FETZER, C. The timed asynchronous distributed system model. **IEEE Transactions on Parallel and Distributed Systems**, v. 10, n. 6, p. 642–657, June 1999.

[32] VERÍSSIMO, P.; CASIMIRO, A. The timely computing base model and architecture. **IEEE Transactions on Computers**, v. 51, n. 8, p. 916–930, August 2002.

[33] GORENDER, S.; MACÊDO, R. J. A. Um modelo para tolerância a falhas em sistemas distribuídos com qos. In: **Anais do XX Simpósio Brasileiro de Redes de Computadores (SBRC)**. Porto Alegre, Brazil: SBC, 2002. v. 1, p. 277–292.

[34] BLAKE, S. et al. **An Architecture for Differentiated Services**. Request For Comments 2475 (RFC), The Internet Engineering Task Force, December 1998.

[35] MACÊDO, R. J. A.; FREITAS, A. E. S. Group communication for self-aware distributed systems. In: **Proceedings of the Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)**. Porto Alegre, Brazil: SBC, 2010. p. 915–928.

[36] MACÊDO, R. J. A.; FREITAS, A. E. S. Dos sistemas distribuídos self-aware. In: **Proceedings of the 3rd Workshop on Autonomic Distributed Systems (WoSiDA)**. Porto Alegre, Brazil: SBC, 2013. p. 1–4.

[37] SULISTIO, A.; YEO, C.S.; BUYYA, R. A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. **Software- Practice and Experience**, v. 34, n. 7, p. 653–673, June 2004.

[38] TRINDADE, R. M. **Uso do network simulator-NS para simulação de sistemas distribuídos em cenários com defeitos**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação, 2003.

[39] BAJAJ, S. et al. **Improving simulation for network research**. Technical Report 99-702b, University of Southern California, Los Angeles, CA, USA, 1999.

[40] ZENG, X.; BAGRODIA, R.; GERLA, M. Glomosim: a library for parallel simulation of large-scale wireless networks. **ACM SIGSIM Simulation Digest**, v. 28, n. 1, p. 154–161, July 1998.

[41] BAGRODIA, R. et al. Parsec: A parallel simulation environment for complex systems. **Computer**, v. 31, n. 10, p. 77–85, October 1998.

[42] MONTRESOR, A.; JELASITY, M. Peersim: A scalable p2p simulator. In: **Proceedings of the IEEE 9th International Conference on Peer-to-Peer Computing (P2P)**. Palo Alto, CA, USA: IEEE, 2009. p. 99–100.

[43] GARCÍA, P. et al. Planetsim: A new overlay network simulation framework. In: **Proceedings of 4th International Workshop on Software Engineering and Middleware (SEM)**. Berlin, Germany: Springer-Verlag, 2005. (Lecture Notes in Computer Science, v. 3437), p. 123–136.

[44] CAVIN, D.; SASSON, Y.; SCHIPER, A. On the accuracy of manet simulators. In: **Proceedings of the second ACM international workshop on Principles of mobile computing (POMC)**. New York, NY, USA: ACM, 2002. p. 38–43.

[45] HOWELL, F.; MCNAB, R. Simjava: A discrete event simulation package for java with applications in computer systems modelling. In: **Proceedings of the 1st International Conference on Web-based Modelling and Simulation (WMC)**. San Diego, CA, USA: SCS, 1998. (Simulation Series, v. 30), p. 51–56.

[46] TYAN, H. Y.; HOU, C. J. Javasim: A component-based compositional network simulation environment. In: **Proceedings of the Western Simulation Multiconference, Communication Networks And Distributed Systems Modeling And Simulation (WMC)**. San Diego, CA, USA: SCS, 2001.

[47] RODRIGUES, L. A. **Extensão do suporte para simulação de defeitos em algoritmos distribuídos utilizando o Neko**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação, 2006.

[48] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. **ACM Computing Surveys**, ACM Press, New York, NY, USA, v. 22, n. 4, p. 299–319, December 1990. ISSN 0360-0300.

[49] ZHANG, W.; HE, J. Statistical modeling and correlation analysis of endto- end delay in wide area networks. In: **Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/ Distributed Computing (SNPD)**. Mt. Pleasant, MI, USA: ACIS, 2007. v. 3, p. 968–973.

[50] DOWNEY, A. B. Evidence for long-tailed distributions in the internet. In: **Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement (IMW)**. New York, NY, USA: ACM, 2001. p. 229–241.

[51] IHAKA, R.; GENTLEMAN, R. R: a language for data analysis and graphics. **Journal of computational and graphical statistics**, v. 5, n. 3, p. 299–314, 1996.

[52] PINGALI, S.; TOWSLEY, D.; KUROSE, J.F. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. **ACM SIGMETRICS Performance Evaluation Review**, v. 22, n. 1, p. 221–230, May 1994.

[53] BIRMAN, K. P. The process group approach to reliable distributed computing. **Communications of the ACM**, ACM Press, New York, NY, USA, v. 36, n. 12, p. 37–53, December 1993. ISSN 0001-0782.

[54] CRISTIAN, F. Synchronous and asynchronous group communication. **Communications of the ACM**, ACM Press, New York, NY, USA, v. 39, n. 4, p. 88–97, April 1996.

[55] CHANDRA, T. D. et al. On the impossibility of group membership. In: **Proc. of the 15th annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM Press, 1996. p. 322–330. ISBN 0-89791-800-2.

[56] CHOCKLER, G. V.; KEIDAR, I.; VITENBERG, R. Group communication specifications: a comprehensive study. **ACM Computing Surveys**, ACM Press, New York, NY, USA, v. 33, n. 4, p. 427–469, December 2001.

[57] DéFAGO, X.; SCHIPER, A.; URBáN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. **ACM Computing Surveys**, ACM Press, New York, NY, USA, v. 36, n. 4, p. 372–421, December 2004. ISSN 0360-0300.

[58] CHANDRA, T. D.; HADZILACOS, V.; TOUEG, S. The weakest failure detector for solving consensus. **Journal of the ACM**, ACM Press, New York, NY, USA, v. 43, n. 4, p. 685–722, 1996. ISSN 0004-5411.

[59] CRISTIAN, F. Reaching agreement on processor-group membership in synchronous distributed systems. **Distributed Computing**, Springer, v. 4, n. 4, p. 175–187, April 1991.

[60] KOPETZ, H.; GRUNSTEIDL, G. Ttp - a protocol for fault-tolerant real-time systems. **IEEE Computer**, IEEE CS Press, Los Alamitos, CA, USA, v. 27, n. 1, p. 14–23, January 1994.

[61] CRISTIAN, F. et al. Atomic broadcast: from simple message diffusion to byzantine agreement. In: **Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing (FTCS)**. Palo Alto, CA, USA: IEEE Computer Society, 1995. p. 431–437.

[62] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of ACM**, ACM Press, New York, NY, USA, v. 21, n. 7, p. 558–565, July 1978. ISSN 0001-0782.

[63] MACÊDO, R. J. A.; EZHILCHELVAN, P.; SHRIVASTAVA, S. K. Modeling group communication using causal blocks. In: **Proceedings of the 5th European Workshop on Dependable Computing (EWDC)**. New York, NY, USA: ACM Press, 1993.

[64] MACÊDO, R. J. A. Fault-tolerant group communication protocols for asynchronous systems. In: **Ph.D. Thesis, Dep. of Computing Science, U. of Newcastle upon Tyne**. [S.l.: s.n.], 1994.

[65] FREITAS, A. E. S.; MACÊDO, R. J. A. Um ambiente para testes e simulações de protocolos confiáveis em sistemas distribuídos híbridos e dinâmicos. In: **Anais do XXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)**. Porto Alegre, Brazil: SBC, 2009. v. 1, p. 826–839.

[66] KAASHOEK, M. F.; TANENBAUM, A. S. Group communication in the amoeba distributed operating system. In: **Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)**. Palo Alto, CA, USA: IEEE, 1990. p. 222–230.

[67] LUNDELIUS, J.; LYNCH, N. Upper and lower bound for clock synchronization. **Information and Control**, v. 62, n. 2, p. 190–204, August 1984.

[68] ARMBRUST, Michael et al. A view of cloud computing. **Communications of the ACM**, v. 53, n. 4, p. 50–58, April 2010.

[69] HELLERSTEIN, J. L. et al. **Feedback Control of Computing Systems**. Ontario, Canada: Wiley-Interscience, 2004.

[70] OGATA, K. **Discrete-Time Control Systems**. Upper Saddle River, NJ, USA: Prentice-Hall, 1995.

[71] MACÊDO, R. J. A.; FREITAS, A. E. S.; SÁ, A. S. de. A self-manageable group communication protocol for partially synchronous distributed systems. In: **Proceedings of the 5th Latin-American Symposium on Dependable Computing (LADC)**. Palo Alto, CA, USA: IEEE, 2011. p. 146–155.

[72] MACÊDO, R. J. A.; FREITAS, A. E. S.; SÁ, A. S. de. Enhancing group communication with self-manageable behavior. **Journal of Parallel and Distributed Computing**, Elsevier, v. 73, n. 4, p. 420–433, April 2013.

[73] FELDMAN, P.; MICALI, S. Optimal algorithms for Byzantine agreement. In: **Proceedings of the 20th ACM Symposium on Theory of Computing (STOC)**. [S.l.]: ACM, 1988. p. 148–161.

[74] DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. **Journal of the ACM**, v. 35, n. 2, p. 288–323, April 1988.

[75] GARAY, J. A.; MOSES, Y. Fully polynomial Byzantine agreement in t+ 1 rounds. In: **Proceedings of the 25th annual ACM Symposium on Theory of computing (STOC)**. [S.l.]: ACM Press, 1993. p. 31–41.

[76] MALKHI, D.; REITER, M. Unreliable intrusion detection in distributed computations. In: IEEE CS. **Proceedings of the 10th Computer Security Foundations Workshop (CSFW)**. [S.l.], 2002. p. 116–124.

[77] SÁ, A. S. de; FREITAS, A. E. S.; MACÊDO, R. J. A. Auto-configuração de máquina de estados tolerante a falhas bizantinas. In: **Proceedings of the 2nd Workshop on Autonomic Distributed Systems (WoSiDA)**. Porto Alegre, Brazil: SBC, 2012. p. 35–38.

[78] SÁ, A. S. de; FREITAS, A. E. S.; MACÊDO, R. J. A. Request batching self-configuration in byzantine fault-tolerant replication. In: **Proceedings of the 2nd Brazilian Symposium on Computing System Engineering: Operating Systems Workshop (SBESC-WSO)**. Porto Alegre, Brazil: SBC, 2012. p. 1–6.

[79] SÁ, A. S. de; FREITAS, A. E. S.; MACÊDO, R. J. A. Adaptive request batching for byzantine replication. **ACM SIGOPS Operating Systems Review**, ACM, v. 47, n. 1, p. 35–42, 2013.

[80] WESTER, B. et al. Tolerating latency in replicated state machines through client speculation. In: **Proceedings of the 6th USENIX Symposium on Networked systems design and implementation (NSDI)**. Boston, MA, USA: USENIX, 2009. p. 245–260.

[81] KOTLA, R. et al. Zyzzyva: speculative byzantine fault tolerance. In: **Proceedings of 21th ACM SIGOPS Symposium on Operating systems principles (SOSP)**. New York, NY, USA: ACM, 2007. p. 45–58.

[82] GUERRAOUI, R. et al. The next 700 BFT protocols. In: **Proceedings of the 5th European Conference on Computer systems (EuroSys)**. New York, NY, USA: ACM, 2010. p. 363–376.

[83] ALVISI, L. et al. Dynamic byzantine quorum systems. In: IEEE CS. **Proceedings of the International Conference on Dependable Systems and Networks (DSN)**. New York, NY , USA, 2000. p. 282–292.

[84] KONG, L. et al. A reconfigurable byzantine quorum approach for the agile store. In: **Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS)**. Palo Alto, CA, USA: IEEE CS, 2003. p. 219–228.

[85] ALCHIERI, E. et al. Sistemas de quóruns bizantinos pró-ativos. In: **Anais do XXVII Simposio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2009)**. Porto Alegre, Brasil: SBC, 2009. p. 245–257.

[86] GAFNI, E.; LAMPORT, L. Disk paxos. In: **Proceedings of the 14th International Conference on Distributed Computing (DISC'2000)**. London, UK: Springer-Verlag, 2000. p. 330–344.

APÊNDICES

Apêndice

A

ESTRUTURA DO HDDSS

Neste apêndice é apresentada uma visão geral de diagrama de classes base do *framework* **HDDSS**, apresentado no Capítulo 4. A Figura A.1 apresenta as classes associadas a leitura de arquivo de configuração e instanciação por meio do padrão **Factory** dos componentes associados ao cenário simulado, bem como o pacote **report** e a classe *Statistica* associados a geração de relatórios.

A Figura A.2 apresenta o conjunto de classes associado a geração de eventos aleatórios, incluindo a integração ao ambiente R, as classes geradoras de eventos em tempo de simulação, que podem, por exemplo, ser associados a distribuições de probabilidade, bem como as classes associadas à composição de canais de comunicação e à modelos de falhas de processos e canais.

Por fim, a Figura A.3 apresenta a relação entre os demais componentes da simulação: classes de infraestrutura de processamento (ambiente de execução – **RuntimeContainer** –, CPU e relógio associados) e de comunicação (rede de computadores e *buffers* de comunicação), composição de cenário (cenário, processo – **Agent** –, e canais) e classes principais de execução (**Simulator** em modo simulação e **TestBed** em modo protótipo, associado ao uso de *sockets* de comunicação).

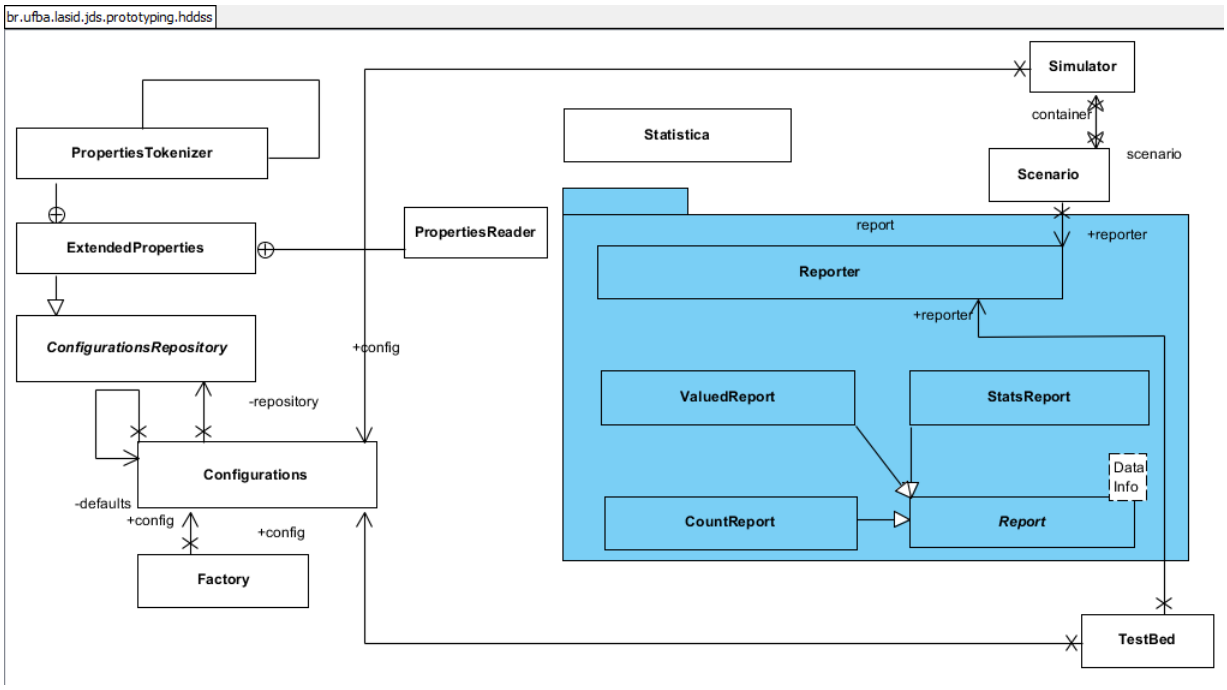


Figura A.1. Classes do **HDDSS** associadas a configuração e geração de métricas e relatórios.

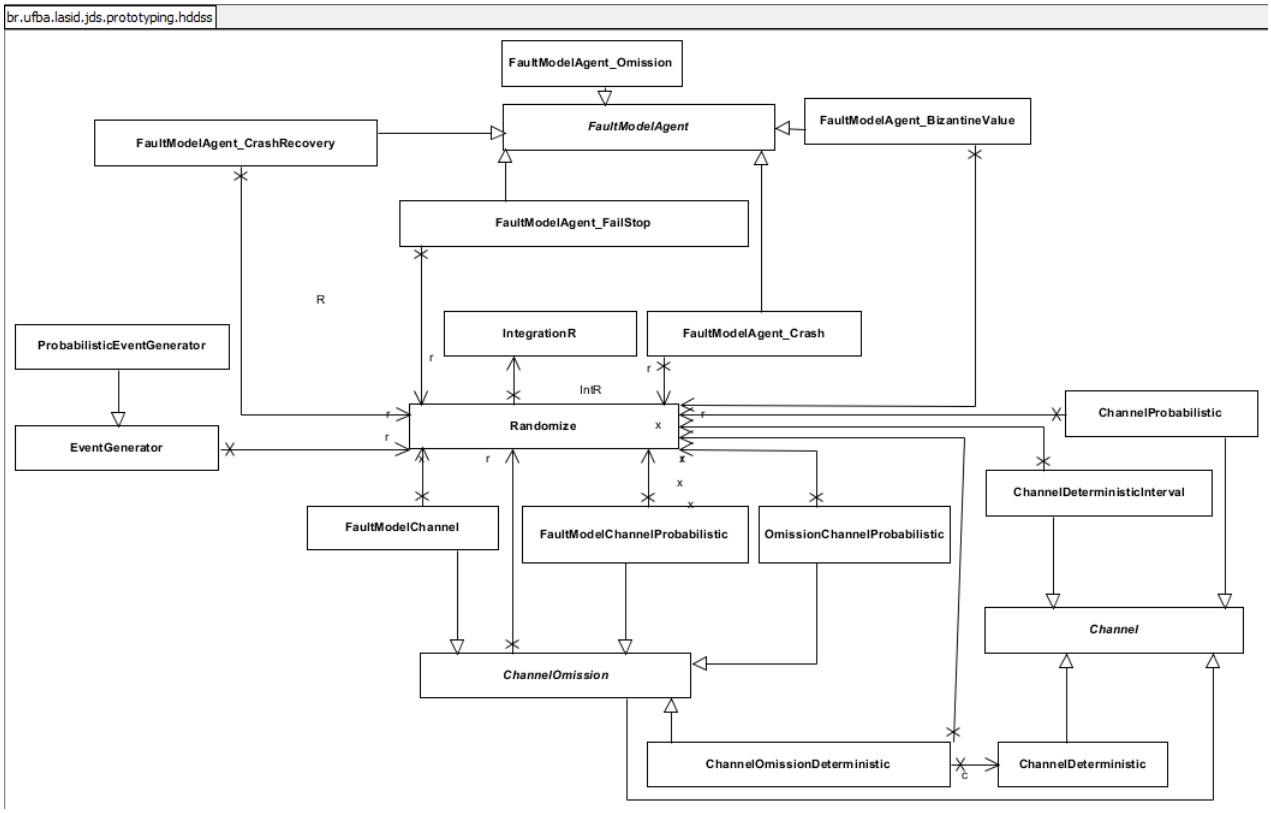


Figura A.2. Classes do **HDDSS** associadas a geração de eventos, distribuições de probabilidade e composição de canais de comunicação e modelos de falhas.

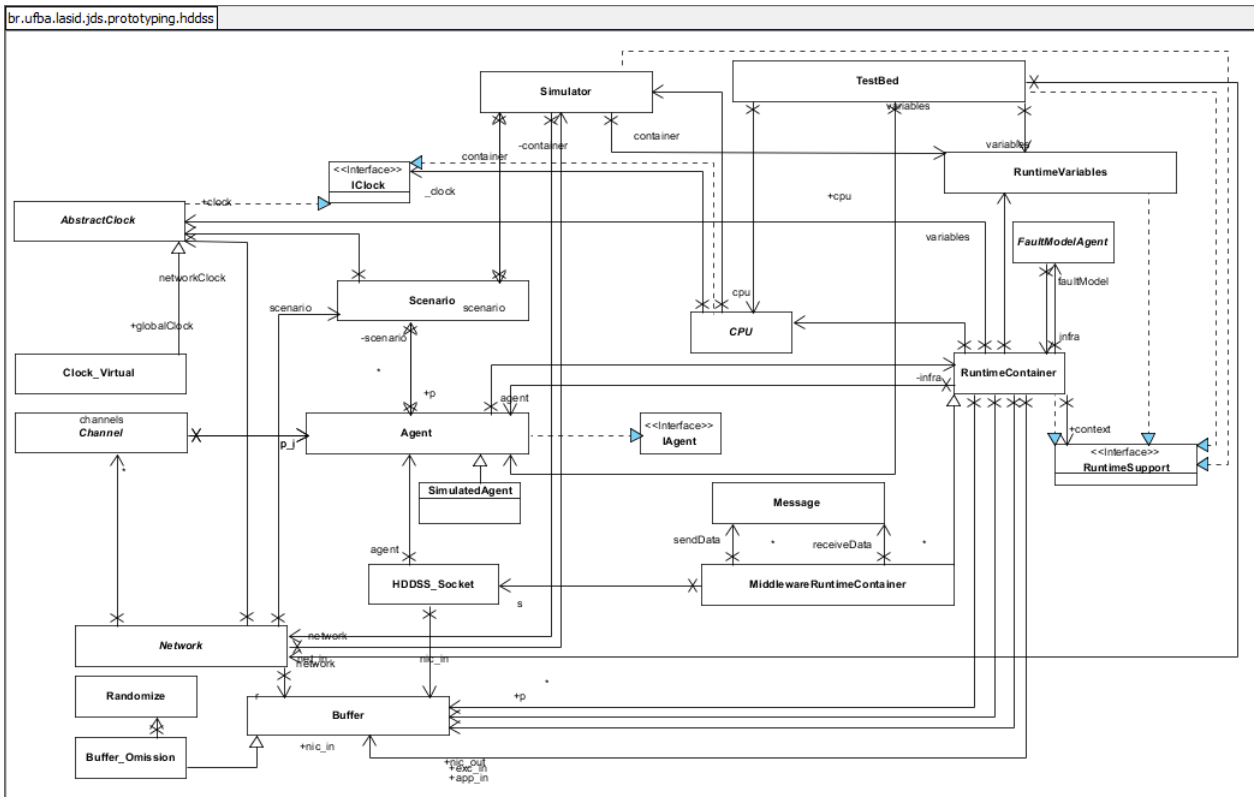


Figura A.3. Classes do **HDDSS** associadas à infraestrutura e à composição de cenários e classes do modo protótipo e do modo simulação.

Apêndice

B

CÓDIGO-FONTE DE CENÁRIO

Neste apêndice é apresentado de forma breve a construção da solução da avaliação de desempenho do consenso adaptativo [6] no cenário do grupo de experimentos FED-CLOUD. Este é um dos experimentos apresentados no Capítulo 5.

Neste grupo de experimentos, é avaliado o progresso do consenso adaptativo em um ambiente híbrido: *clusters* conectados por meio da Internet. Cada *cluster* representa um grupo de computadores interconectados por uma rede local, onde, pode-se assumir um comportamento síncrono, contudo a nuvem não é em si um sistema síncrono. Este ambiente é representado por um modelo de sistema distribuído híbrido, onde cada cluster local é um sub-sistema síncrono interconectado por canais de comunicação assíncronos com os *clusters* remotos.

No cenário em tela, para $n = 80$ processos, definimos um cenário híbrido em que os canais de comunicação tem comportamento *timely* ou *untimely*. A configuração é apresentada no Código-fonte B.1. Associamos a cada um dos elementos da simulação, agentes (processos), canais de comunicação, rede, cpu e cenário as classes específicas que implementam o comportamento desejado.

```
1  FinalTime = 1200000
2  NumberOfAgents = 80
3  MaximumDeviation = 4
4  Mode = clock
5  Debug = false
6  FormattedReport = false
7
8  clock = br.ufba.lasid.jds.prototyping.hddss.Clock_Virtual
9  clock.Mode = s
10
11 cpu = br.ufba.lasid.jds.prototyping.hddss.CPUZeroDelay
12
13 agent = br.ufba.lasid.jds.prototyping.hddss.instances.Agent_AdaptConsensus
14 agent.DeltaMin = 1
15 agent.DeltaMax = 30
```

```
16 agent.TS = 20
17 agent.PacketGenerationProb = 0.05
18
19 agentAlternative = br.ufba.lasid.jds.prototyping.hddss.instances.
       Agent_AdaptConsensus
20 agentAlternative.DeltaMin = 1
21 agentAlternative.DeltaMax = 30
22 agentAlternative.TS = 20
23 agentAlternative.PacketGenerationProb = 0.05
24 agentAlternative.FaultModel = br.ufba.lasid.jds.prototyping.hddss.
       FaultModelAgent_Crash
25
26 channel[0] = br.ufba.lasid.jds.prototyping.hddss.ChannelProbabilistic
27 channel[0].MinValue = 1
28 channel[0].Distribution = lognormal(5.0,15.0)
29
30 channel[1] = br.ufba.lasid.jds.prototyping.hddss.ChannelDeterministic
31 channel[1].Delay = 2
32
33 network = br.ufba.lasid.jds.prototyping.hddss.NetworkDeterministic
34 network.ProcessingTime = .001
35 network.FIFO = true
36 network.TripBalance = 0.5
37
38 scenario = br.ufba.lasid.jds.prototyping.hddss.
       Scenario_ProtocoloHibridoDinamico
39 scenario.DeterministicChannelType = 1
40 scenario.ProbabilisticChannelType = 0
41 scenario.InitialAlt = 0
42 scenario.EndingAlt = 39
```

Código-Fonte B.1. Arquivo de configuração de cenário de consenso adaptativo do grupo de experimentos FED-CLOUD.

O modo de simulação implementa o comportamento de relógios físicos locais (definido por **Clock_Virtual**), com $\rho_{max} = 4.10^{-3} ms$, 80 agentes e tempo total de simulação de $1200000 ms = 20 min$. O modo de depuração, que exibe mensagens extras para traço da execução, não está ativo e a saída é exibida em formato *CSV* para uso posterior em outros programas.

Para representar canais de comunicação *untimely* utilizamos a classe **ChannelProbabilistic** do *framework*, associando um comportamento definido em uma função de distribuição de probabilidade, para canais *timely*, a classe **ChannelDeterministic** é utilizada, definindo-se a latência máxima. Desta forma, $channel[0]$ está associado a configuração de um canal *untimely*, onde define-se uma latência mínima de $1ms$, e se utiliza a função Lognormal com média de $5ms$ e desvio-padrão de $15ms$,*built-in* no simulador, para definir a latência do canal; e $channel[1]$ está associado a configuração de um canal *timely* com latência de até $2ms$.

Definimos duas configurações de agentes, *agent* e *agentAlternative*, que implementam o algoritmo definido em **Agent_AdaptConsensus**, com período de inatividade $TS = 20ms$ para o detector de defeitos utilizado na computação e limite temporal (*time-out*) indicado por $DeltaMax = 30ms$. *agent* é uma configuração confiável e *agentAlternative* está sujeita a falhas por *crash* ao longo da simulação de acordo com **FaultModelAgent_Crash**.

Definimos o comportamento da infraestrutura de rede de computadores modelando efeitos de enfileiramento com tempo de serviço de $10^{-3}ms$ no roteamento por meio de **NetworkDeterministic**, assumindo comportamento *FIFO*. Como nesta simulação, atividades de processamento custoso, como, por exemplo, criptografia, não são implementadas, não definimos um modelo de CPU que implemente estes custos (uso de **CPUZeroDelay**).

Por fim, associamos na configuração do cenário escolhido (**Scenario_ProtocoloHibridoDinamico**) $channel[1]$ ao tipo determinístico e $channel[0]$ ao probabilístico. Isto será utilizado no código-fonte do cenário para montar o cenário híbrido ora descrito. Ainda, o cenário define os 48 processos iniciais (ID de 0 a 47) com a configuração de *agentAlternative* e os demais com a configuração de *agent*, ou seja 60% dos processos falham por *crash* ao longo da execução.

O Código-fonte B.2 apresenta o comportamento deste cenário, sendo utilizado o padrão de projeto Factory para criação e instanciação dos agentes e canais de comunicação, conforme o arquivo de configuração.

```
1  package br.ufba.lasid.jds.prototyping.hddss;
2  import br.ufba.lasid.jds.prototyping.hddss.RuntimeSupport.Variable;
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.StringTokenizer;
6  public class Scenario_ProtocoloHibridoDinamico extends Scenario {
7
8      ArrayList<Integer> agents;
9      ArrayList<Integer> agentsPerPartition;
10     ArrayList<ArrayList<Integer>> partitions;
11     int timely;
12     int untimely;
13     int initial, ending;
14
15     public void setDeterministicChannelType(String s)
16     {
17         timely = Integer.parseInt(s);
18     }
19
20     public void setProbabilisticChannelType(String a)
21     {
22         untimely = Integer.parseInt(a);
23     }
24
25     public void setInitialAlt(String a)
```

```
26      {
27          initial = Integer.parseInt(a);
28      }
29
30      public void setEndingAlt(String a)
31      {
32          ending = Integer.parseInt(a);
33      }
34
35
36
37      @Override
38      public void initAgents() throws Exception{
39          int n = container.get(Variable.NumberOfAgents).<Integer>value().
                intValue();
40          for(int i = 0; i < n; i++){
41                  String TAG = Agent.TAG;
42                  String TAGi = TAG + "["+i+"]";
43                  String TAGAlt = TAG+ "Alternative";
44
45                  if ( (initial ==0 ) && (ending == 0)) {
46                      if(!container.config.getString(TAGi, "null").equals("
                            null")){
47                          p[i] = (Agent) Factory.create(TAGi, Agent.class.
                                getName());
48                      }else{
49                          p[i] = (Agent) Factory.create(Agent.TAG, Agent.
                                class.getName());
50                      }
51                  } else
52                  {
53                      p[i] = (Agent) Factory.create(TAGAlt, Agent.class.
                            getName());
54                  };
55
56                  p[i].setAgentID(i);
57                  p[i].setType(container.get(Variable.Type).<String>value().
                        charAt(0));
58
59                  prepareAgent(p[i]);
60                  p[i].setScenario(this);
61                  Factory.setup(p[i], TAG);
62
63                  if(!container.config.getString(TAGi, "null").equals("null")
                        ){
64                      Factory.setup(p[i], TAGi);
65                  }
```

```
66
67              }
68          }
69
70          @Override
71          public void initChannels() {
72              int n = container.get(Variable.NumberOfAgents).<Integer>value().
                    intValue();
73              for(int i = 0; i < n; i++) {
74                      for(int j = 0; j < n; j++) {
75                          if( (i\%2) == (j\%2) )
76                              container.network.handshaking(i, j, timely);
77                          else
78                              container.network.handshaking(i, j, untimely);
79                      }
80              }
81          }
82
83
84  }
```

Código-Fonte B.2. Código-fonte de Scenario_ProtocoloHibridoDinamico.

O algoritmo dos agentes definido em **Agent_AdaptConsensus** é apresentado em Código-Fonte B.3. O algoritmo é estruturado por meio de uma máquina de estados em que a cada avanço do relógio processa as mensagens existentes no seu *buffer* de recepção de acordo com o método *receive(Message msg)* e executa um conjunto de ações correspondentes ao seu estado atual por meio do método *execute()*.

```
1   package br.ufba.lasid.jds.prototyping.hddss.instances;
2
3   import br.ufba.lasid.jds.prototyping.hddss.Message;
4   import br.ufba.lasid.jds.prototyping.hddss.Randomize;
5   import br.ufba.lasid.jds.prototyping.hddss.RuntimeSupport;
6   import br.ufba.lasid.jds.prototyping.hddss.RuntimeSupport.Variable;
7   import br.ufba.lasid.jds.prototyping.hddss.SimulatedAgent;
8   import br.ufba.lasid.jds.prototyping.hddss.Simulator;
9
10  public class Agent_AdaptConsensus extends SimulatedAgent {
11
12          int DELTA;
13          int delta;
14          int ts;
15          int tsmax;
16          double prob;
17          double minProb;
18          double maxProb;
19          int logicalClock;
```

```
20          long lastTimeSent;
21          int payloadSize;
22          final int TIMEDCB_APP = 4;
23          final int TIMEDCB_TS = 5;
24          final int CONSENSUS_REQUEST = 6;
25          final int CONSENSUS_P1 = 8;
26          final int CONSENSUS_P2 = 9;
27          final int DECIDED = 10;
28          final int DOWN = 11;
29          final int UNCERTAIN = 12;
30
31          int RECV;
32          int SENT;
33
34          int stableMode;
35
36          int minExpiredBlock = Integer.MAX_VALUE;
37          int [] BM;
38          int [] LCB;
39          Consensus [] consensusArray;
40          int [] scheduler;
41          int lastBlock;
42          java.util.TreeMap blocks;
43          java.util.ArrayList msgBuffer;
44          java.util.ArrayList unstableMsgBuffer;
45          java.util.ArrayList expiredBlocks;
46          java.util.TreeMap schedule;
47          IntegerSet proposedView;
48
49          public IntegerSet down;
50          public IntegerSet live;
51          public IntegerSet uncertain;
52          public IntegerSet view;
53          Content_Acknowledge [] acks;
54
55          boolean blockingDelivery;
56          boolean consensus;
57
58          Randomize r;
59
60          public Agent_AdaptConsensus() {
61          super();
62          }
63
64          int verAgenda(int t) {
65              int B = -1;
66              if ( schedule.containsKey(t) )
```

```
67                  B = (Integer) schedule.get(t);
68              return B;
69          }
70
71          void alterarAgenda(int t, int B) {
72              if (B==-1) {
73                  schedule.remove(t);
74              }
75              else if (verAgenda(t)<B) {
76                  schedule.remove(t);
77                  schedule.put(t, B);
78
79              }
80          }
81
82
83      @Override
84          public void setup() {
85              int finalTime = getInfra().context.get(RuntimeSupport.Variable.
                    FinalTime).<Integer>value();
86              schedule = new java.util.TreeMap();
87              stableMode = 0;
88              logicalClock = 0;
89              BM = new int[getInfra().nprocess];
90              LCB = new int[getInfra().nprocess];
91              payloadSize = 0;
92
93              r = new Randomize(this.getAgentID());
94              scheduler = new int[finalTime*4];
95              for (int j = 0;j<scheduler.length ;j++) {
96                      scheduler[j]=-1;
97              }
98
99              msgBuffer = new java.util.ArrayList();
100             unstableMsgBuffer =  new java.util.ArrayList();
101             blocks = new java.util.TreeMap();
102             expiredBlocks = new java.util.ArrayList();
103
104
105             lastTimeSent = -1;
106             lastBlock = -1;
107             SENT = -1;
108             RECV = -1;
109
110             blockingDelivery = false;
111             consensus = false;
112
```

```
113
114             down = new IntegerSet();
115             live = new IntegerSet();
116             uncertain = new IntegerSet();
117             view = new IntegerSet();
118
119             proposedView = new IntegerSet();
120
121             initializeSets();
122
123
124             consensusArray = new Consensus[finalTime*2];
125             acks = new Content_Acknowledge[getInfra().nprocess];
126             for (int i = 0;i<getInfra().nprocess;i++) {
127                 BM[i]=-1;
128                 LCB[i]=-1;
129                 acks[i] = new Content_Acknowledge();
130             }
131
132         }
133
134         public void initializeSets() {
135             for (int i=0;i<getInfra().nprocess;i++) {
136                 live.add(i);
137                 view.add(i);
138             }
139         }
140
141         public void setDeltaMax(String dt) {
142             DELTA = Integer.parseInt(dt);
143         }
144
145         public void setStableMode(String dt) {
146             stableMode = Integer.parseInt(dt);
147         }
148
149         public void setPayloadSize(String dt) {
150             payloadSize = Integer.parseInt(dt);
151         }
152
153         public void setDeltaMin(String dt) {
154             delta = Integer.parseInt(dt);
155         }
156
157         public void setMaxProb(String dt) {
158             maxProb = Double.parseDouble(dt);
159         }
```

```
160
161
162         public void seMinProb(String dt) {
163             minProb = Double.parseDouble(dt);
164         }
165
166
167         public void setTimeSilence(String p) {
168             ts = Integer.parseInt(p);
169             tsmax = ts;
170         }
171
172         public void setPacketGenerationProb (String po) {
173             prob = Double.parseDouble(po);
174         }
175
176
177         public void startup(){
178
179         }
180
181        public int getMaxTS(){
182            return tsmax;
183        }
184
185        public int getDeltaMax(){
186            return DELTA;
187        }
188
189        public int getDeltaMin(){
190            return delta;
191        }
192
193        void blockRegister(int blocknumber, int sender, long ti) {
194                int TC;
195                int finalTime = getInfra().context.get(RuntimeSupport.
                       Variable.FinalTime).<Integer>value();
196                int DESVIO = getInfra().context.get(RuntimeSupport.Variable
                       .MaxDeviation).<Integer>value();
197
198                Integer block = new Integer(blocknumber);
199                long clock = getInfra().clock.value();
200
201                String mode = getInfra().context.get(RuntimeSupport.
                       Variable.Mode).<String>value();
202                double ro = getInfra().context.get(RuntimeSupport.Variable.
                       ClockDeviation).<Double>value();
```

```
203                 int maxro = getInfra().context.get(RuntimeSupport.Variable.
                        MaxClockDeviation).<Integer>value();
204
205                 if (!blocks.containsKey(block)) {
206
207                         if (block > lastBlock)
208                             lastBlock = block;
209                         blocks.put(block, clock);
210                         if (mode.equals("t")) {
211                             if (sender == getAgentID()) {
212                                 TC = (int) ( (clock+2 * getDeltaMax() +
                                        getMaxTS()) + DESVIO+1);
213                             }
214                             else
215                                 TC = (int) ( (clock+2 * getDeltaMax() +
                                        getMaxTS()-getDeltaMin()) + DESVIO+1);
216                         } else {
217                             if (sender == getAgentID()) {
218                                 TC = (int) ( (clock+2 *getDeltaMax() +
                                        getMaxTS() )* (1+ro*maxro))+1;
219                             }
220                             else
221                                 TC = (int) ( (clock+2 * getDeltaMax() +
                                        getMaxTS()-getDeltaMin())* (1+ro*maxro))
                                        +1;
222                         }
223
224                         if (TC <= finalTime)
225                                 if (scheduler[TC] < blocknumber) {
226                                     scheduler[TC] = blocknumber;
227
228                                 }
229
230                 }
231         }
232
233     int consensusNumber = 0;
234     final int CONSENSUS_MAX = 10000;
235     int timeout=0;
236     int proposedValueConsensus[] = new int[CONSENSUS_MAX];
237     int decidedValueConsensus[] = new int[CONSENSUS_MAX];
238     long startConsensus[] = new long[CONSENSUS_MAX];
239     long finalConsensus[] = new long[CONSENSUS_MAX];
240
241
242     @Override
243         public void execute() {
```

```
244             long clock = getInfra().clock.value();
245
246             //System.out.println("myID"+this.getAgentID());
247             if (this.getAgentID()==1) {
248
249                 if (getInfra().clock.value()  - SENT > 4*DELTA) {
250                     logicalClock ++;
251                     SENT = logicalClock;
252                     sendGroupMsg(clock, TIMEDCB_APP,
253                             new Content_TimedCB("payload", LCB[getAgentID()
                                ],
254                                 acks, payloadSize),
255                                     logicalClock, true );
256                     blockRegister(logicalClock, getAgentID(), clock);}
257                 }
258
259             if ( (RECV > SENT) || (blockingDelivery) ) {
260                 if (clock - lastTimeSent >= ts) {
261
262                         lastTimeSent = clock;
263                         logicalClock = max(logicalClock+1,lastBlock);
264                         SENT = logicalClock;
265
266                         sendGroupMsg(clock, TIMEDCB_TS, new Content_TimedCB
                            ("time-silent", LCB[getAgentID()], acks),
                            logicalClock );
267                         blockRegister(logicalClock, getAgentID(), clock);
268                         //System.out.println("falha");
269                         boolean isTimely=false;
270                         for (int j=0; j<getInfra().nprocess;j++){
271                             if (view.exists(j)) {
272                                 if (getScenario().
273                                     getContainer().
274                                         getNetwork().
275                                             informChannel( (int) this.
                                                getAgentID(), j)
276                                                 instanceof
277                                                     br.ufba.lasid.jds.
                                                        prototyping.hddss.
                                                        ChannelDeterministic
                                                        ) {
278                                     //System.out.println("canal
                                        deterministico "+(int) this.
                                        getAgentID()+" "+j);
279                                     isTimely=true;
280                                 }
281                             }
```

```
282                         }
283                         //System.out.println(isTimely);
284                         if (isTimely == false) {
285                             //if (live.exists((int) this.getAgentID())) {
286                                 // MUDA O PROCESSO PARA UNCERTAIN
287                                 //System.out.println("uncertain "+(int)
                                        this.getAgentID());
288                                 live.remove((int) this.getAgentID());
289                                 uncertain.add((int) this.getAgentID());
290                                 this.sendGroupMsg(clock, UNCERTAIN, (int)
                                        this.getAgentID(), 0, false);
291                             //}
292                         }
293
294
295
296                     }
297                 }
298
299                 if (scheduler[(int) clock] != -1) {
300                     if (getMin() < scheduler[(int) clock]) {
301                         for (int i = getMin()+1; i<=scheduler[(int) clock];i++)
                                {
302                             //infra.debug("p"+Integer.toString(ID)+": timeout
                                    in block "+Integer.toString(i) + " in "+Integer.
                                    toString(clock));
303                             notifyExpiredBlocks(i);
304                             IntegerSet x = retornaExpirados(i);
305                             int vetor[] = x.toVector();
306                             for (int j=0;j<vetor.length;j++) {
307
308                                 if (getScenario().
309                                         getContainer().
310                                             getNetwork().
311                                                 informChannel( (int) this.
                                                        getAgentID(), j)
312                                                 instanceof
313                                                     br.ufba.lasid.jds.
                                                         prototyping.hddss.
                                                         ChannelDeterministic) {
314                                     // RETIRA O PROCESSO DE LIVE E PASSA PARA
                                            DOWN (CERTEZA NA FALHA)
315                                     if ( live.exists(j) ) {
316                                         down.add(j);
317                                         live.remove(j);
318                                         view.remove(j);
319                                         this.sendGroupMsg(clock, DOWN, j, 0,
```

```
                                        false);
320                                 }
321
322                                 // INCLUIR INFORMAR AOS DEMAIS DA FALHA
323                                 // INCLUIR INFORMAR AOS DEMAIS QUANDO UM
                                        PROCESSO SE TORNA UNCERTAIN
324                             }
325                         }
326                     }
327                 }
328             }
329         }
330
331         public void sendGroupMsg(long clock, int type, Object value, int LC
                ) {
332             for (int j=0; j<getInfra().nprocess;j++){
333
334                 if(value instanceof Content_TimedCB){
335                     ((Content_TimedCB)value).vack[j].rsendTime = clock;
336                 }
337                 this.createMessage(clock, this.getAgentID(), j, type, value
                        , LC);
338             }
339         }
340
341         public void sendGroupMsg(long clock, int type, Object value, int LC
                , boolean pay) {
342             for (int j=0; j<getInfra().nprocess;j++){
343                 if(value instanceof Content_TimedCB){
344                     ((Content_TimedCB)value).vack[j].rsendTime = clock;
345                 }
346
347                 this.createMessage(clock, this.getAgentID(), j, type, value
                        , LC, pay);
348             }
349         }
350
351     public void notifyExpiredBlocks(int bl) {
352             expiredBlocks.add(bl);
353             if (minExpiredBlock == Integer.MAX_VALUE)
354                 minExpiredBlock = bl;
355      }
356
357      public void deliver(Message msg) {
358             if (msg.logicalClock > logicalClock) {
359                 logicalClock = msg.logicalClock+1;
360             }
```

```
361         }
362
363         public int getMin() {
364             int min = Integer.MAX_VALUE;
365             for (int i=0; i<getInfra().nprocess;i++){
366                 if (view.exists(i)) {
367                     if (BM[i] < min)
368                         min = BM[i]; }
369                 // else System.nic_out.println("O processo "+i+" não existe na
                        visao de "+ID);
370             }
371             return min;
372         }
373
374         public int getStableMin() {
375             int min = Integer.MAX_VALUE;
376             for (int i=0; i<getInfra().nprocess;i++){
377                 if (view.exists(i))
378                     if (LCB[i] < min)
379                         min = LCB[i];
380             }
381             return min;
382         }
383
384         public IntegerSet retornaExpirados(int bl) {
385             IntegerSet x = new IntegerSet();
386             for (int i=0; i<getInfra().nprocess;i++){
387                 if (view.exists(i))
388                     if (BM[i] < bl)
389                         x.add(i);
390             }
391             return x;
392         }
393
394         void checkCompletion() {
395             int n = getInfra().nprocess;
396             int min = getMin();
397             LCB[getAgentID()] = min;
398             long clock = getInfra().clock.value();
399             int minStable = getStableMin();
400             java.util.TreeSet key = new java.util.TreeSet();
401             java.util.TreeMap sorting = new java.util.TreeMap();
402             int order;
403             Integer od;
404             Message m;
405             if (blockingDelivery) return;
406
```

```
407         if (stableMode == 1)
408             min = minStable;
409
410         java.util.Collections.sort(msgBuffer, new java.util.Comparator() {
411             public int compare(Object o1, Object o2) {
412                 Message msg1 = (Message) o1;
413                 Message msg2 = (Message) o2;
414                 return      (msg1.logicalClock - msg2.logicalClock) != 0 ?
415                             (msg1.logicalClock - msg2.logicalClock) :
416                             (msg1.sender - msg2.sender);
417
418             }
419         });
420
421         int cont=0;
422         for (int i=0; i<msgBuffer.size();i++) {
423             m = (Message) msgBuffer.get(i);
424             // System.nic_out.println("p"+ID+" min="+minimo+" ts log =
                    "+m.relogioLogico);
425             if ( (m.logicalClock <= min)  ) {
426                 cont++;
427                 if (m.type == TIMEDCB_APP)
428                     this.preDeliver(m);
429             }
430         }
431
432         java.util.ArrayList remover = new java.util.ArrayList();
433         for (int i=1; i<= unstableMsgBuffer.size();i++) {
434             m = (Message) unstableMsgBuffer.get(i-1);
435             if ( (m.logicalClock <= minStable) ) {
436                 remover.add(m);
437             }
438         }
439         unstableMsgBuffer.removeAll(remover);
440
441         for (int i=1; i<=msgBuffer.size();i++) {
442             m = (Message) msgBuffer.get(i-1);
443             if ( (m.logicalClock > minStable) && !(unstableMsgBuffer.
                    contains(m))  ) {
444                 unstableMsgBuffer.add(m);
445             }
446         }
447
448         if (cont==0) return;
449
450         cont=0;
451         while (chave.iterator().hasNext()) {
```

```
452             od = (Integer) chave.first();
453             m = (Mensagem) ordenar.get(od);
454 
455             bufferDeMensagens.remove((Object) m);
456             cont++;
457             if (chave.iterator().hasNext()) od = (Integer) chave.iterator()
                    .next();
458         }
459         remover.clear();
460         for (int i=0; i< msgBuffer.size();i++) {
461             m = (Message) msgBuffer.get(i);
462             if ( (m.logicalClock <= min) ) {
463                 remover.add(m);
464                 cont++;
465             }
466         }
467         msgBuffer.removeAll(remover);
468 
469     }
470 
471         public void receive(Message msg) {
472 
473             Content_Unstable uC;
474             Consensus c;
475             int j;
476             long clock = getInfra().clock.value();
477             switch (msg.type) {
478                 case DOWN:
479                     j = ((Integer) msg.getContent()).intValue();
480                     if ( live.exists(j) ) {
481                                     down.add(j);
482                                     live.remove(j);
483                                     view.remove(j);
484                     }
485                     break;
486                 case UNCERTAIN:
487                     j = ((Integer) msg.getContent()).intValue();
488                     if ( live.exists(j) ) {
489                                     uncertain.add(j);
490                                     live.remove(j);
491                     }
492                     break;
493                 case TIMEDCB_APP:
494                 case TIMEDCB_TS:
495                     if (msg.logicalClock > BM[msg.sender]) {
496                             BM[msg.sender] = msg.logicalClock;
497                         }
```

```
498                 if ( ((Content_TimedCB) msg.content).LCB > LCB[msg.
                        sender]) {
499                     LCB[msg.sender] = ((Content_TimedCB) msg.
                            content).LCB;
500                  }
501
502                 acks[msg.sender].rrecvTime = msg.receptionTime;
503                 acks[msg.sender].lsendTime = msg.physicalClock;
504
505                 //ultimaMsgTimeStamp[msg.remetente] = msg.relogioFisico
                        ;
506
507                 RECV = max(RECV, msg.logicalClock);
508                 msgBuffer.add(msg);
509                 blockRegister(msg.logicalClock, msg.sender, msg.
                        physicalClock);
510                 checkCompletion();
511
512                 String output = "p"+getAgentID();
513                 for (int i=0;i<getInfra().nprocess;i++)
514                     output = output+" "+BM[i];
515                 output=output+" RECV "+RECV+" SENT "+SENT+ " LSB "+
                        getStableMin()+" LCB "+getMin();
516                 //infra.debug(output);
517
518                 break;
519             case CONSENSUS_REQUEST:
520                 consensusArray[msg.logicalClock] = startConsensus(msg);
521                 if (consensusArray[msg.logicalClock].getRound()%(
                        getInfra().nprocess) == getAgentID()) {
522                     sendGroupMsg(clock, CONSENSUS_P1,consensusArray[msg
                            .logicalClock], logicalClock);
523                 }
524                 break;
525             case CONSENSUS_P1:
526                 c = (Consensus) msg.content;
527                 if (consensusArray[c.number]==null)
528                     consensusArray[c.number] = c;
529
530                 if (c.getRound()==(consensusArray[c.number].getRound())
                        ) {
531                     if (msg.sender == c.getRound() % getInfra().
                            nprocess) {
532                         consensusArray[c.number].estimated = c.
                                estimated;
533                         sendGroupMsg(clock, CONSENSUS_P2,consensusArray
                                [c.number], logicalClock);
```

```
534                         }
535                     }
536                     else if  (c.getRound()<(consensusArray[c.number].
                            getRound())){
537                             c.alteraRound(consensusArray[c.number].
                                getRound() );
538                             if (msg.sender == c.getRound() % getInfra()
                                .nprocess) {
539                                consensusArray[c.number].estimated = c.
                                    estimated;
540                                 sendGroupMsg(clock, CONSENSUS_P2,
                                     consensusArray[c.number],
                                     logicalClock);
541                             }
542                     }
543
544                     break;
545                 case CONSENSUS_P2:
546                     c = (Consensus) msg.content;
547                     if (consensusArray[c.number]==null)
548                         consensusArray[c.number] = c;
549
550                     if (!consensusArray[c.number].gotQuorum) {
551                         consensusArray[c.number].quorum.add(msg.sender);
552                         // adicionaREC(c.numero, (UnstableContent) c.
                            estimado);
553                         if ( (consensusArray[c.number].rec == null) && (c!=
                            null))
554                             consensusArray[c.number].rec = c.estimated;
555                         if (gotConsensusQuorum(c.number)) {
556                             if ( c.estimated == null)
557                                 consensusArray[c.number].noneREC = true;
558                             if ( consensusArray[c.number].rec == null)
559                             {
560                                 if (consensusArray[c.number].noneREC) {
561                                     rotacionaCoordenador(c.number);
562                                 }
563                                 else {
564
565                                     rotacionaCoordenador(c.number);
566                                     consensusArray[c.number].estimated =
                                        consensusArray[c.number].rec;
567                                 }
568                             }
569                             else {
570                                 sendGroupMsg(clock, DECIDED,consensusArray[
                                    c.number], logicalClock);
```

```
571                             }
572                         }
573                     }
574
575                     break;
576
577                 case DECIDED:
578                     c = (Consensus) msg.content;
579                     // Implementa a decisao
580                     if (consensusArray[c.number].active) {
581                         consensusArray[c.number].active = false;
582                         consensusArray[c.number].estimated = c.rec;
583                         decidedValueConsensus[c.number] = (Integer) c.rec;
584                         if (consensusNumber == c.number) {
585                             finalConsensus[consensusNumber] = clock;
586                             if (this.getAgentID()==1) {
587                                 System.out.println("consenso "+
                                        consensusNumber+ " at "+clock);
588                                 this.getReporter().stats("consensus delay",
                                        finalConsensus[consensusNumber] -
                                        startConsensus[consensusNumber]);
589                                 this.getReporter().count("consensus total
                                        number");
590                             }
591                             consensusNumber++;
592
593                         }
594                         }
595                     }
596             }
597
598         void rotacionaCoordenador(int i) {
599             long clock = getInfra().clock.value();
600             consensusArray[i].alteraRound(consensusArray[i].getRound()+1);
601             if (consensusArray[i].getRound()%getInfra().nprocess ==
                    getAgentID())
602                 sendGroupMsg(clock, CONSENSUS_P1,consensusArray[i],
                        logicalClock);
603         }
604
605         Consensus startConsensus(Message m) {
606             Consensus c = new Consensus(m.logicalClock,0,m.content);
607             return c;
608         }
609
610         int max(int a, int b) {
611             if (a>b)
```

```
612                 return a;
613             else
614                 return b;
615         }
616
617         boolean gotConsensusQuorum(int i) {
618             boolean liveOk;
619             boolean uncertainOk;
620
621             liveOk = (live.size()==live.intersection(consensusArray[i].
                    quorum).size());
622
623             int contaUncertain=consensusArray[i].quorum.intersection(
                    uncertain).size();
624
625             if (uncertain.size() >0) {
626                 float perc = contaUncertain / uncertain.size();
627                 if (perc>.5) {uncertainOk = true;} else {uncertainOk =
                        false;};
628             } else
629                 uncertainOk = true;
630
631             return (liveOk && uncertainOk);
632         }
633
634 }
```

Código-Fonte B.3. Código-fonte de Agent_AdaptConsensus.

A definição de canais probabilísticos e determinísticos é simples. O Código-Fonte B.4 apresenta o comportamento de um canal baseado em uma distribuição de probabilidade (**ChannelProbabilistic**). Esta distribuição associada a classe **Randomize** é definida em configuração e utilizada pelo método *delay()* para o cálculo da latência fim-a-fim. No Código-Fonte B.5, esta latência de um canal determinístico (**ChannelDeterministic**) é determinada de forma fixa.

```
1  public class ChannelProbabilistic extends Channel {
2
3      Randomize x;
4      int minValue=0;
5
6      ChannelProbabilistic () {
7          x=new Randomize();
8      }
9
10     public void setDistribution(String dt) {
11         String at = dt.substring(1, dt.length()-1);
12         x.setDistribution(at);
13     }
```

```
14
15      public void setMinValue(String dt) {
16          minValue = Integer.parseInt(dt);
17      }
18      int delay() {
19          double d;
20          d = minValue+ x.genericDistribution();
21          return (int) d;
22      }
23
24
25  }
```

Código-Fonte B.4. Código-fonte de ChannelProbabilistic.

```
1   package br.ufba.lasid.jds.prototyping.hddss;
2
3   public class ChannelDeterministic extends Channel {
4
5       int delay;
6
7       ChannelDeterministic (int t) {
8           delay = t;
9       }
10
11      ChannelDeterministic () {
12
13      }
14
15      int delay() {
16          return delay;
17      }
18
19      public void setDelay(String dt) {
20              delay = Integer.parseInt(dt);
21       }
22
23      boolean status() {
24          return true;
25      }
26  }
```

Código-Fonte B.5. Código-fonte de ChannelDeterministic.

O comportamento de CPU sem atribuir custo diferenciado para cada operação pode ser obtido por meio da classe **CPUZeroDelay**, apresentada no Código-Fonte B.6, a qual simplesmente retorna custo zero de processamento a uma dada requisição.

```
1  package br.ufba.lasid.jds.prototyping.hddss;
2
3  public class CPUZeroDelay extends CPU{
4
5      @Override
6      public double proc(Object data) {
7          return 0.0;
8      }
9  }
```

Código-Fonte B.6. Código-fonte de CPUZeroDelay.

O comportamento de enfileiramento com taxa de serviço determinística e efeitos sobre o cálculo da latência fim-a-fim efetiva de cada mensagem obtido pela classe **NetworkDeterministic** é obtido por meio de retorno fixo do método delay(Message m) no cálculo do tempo de serviço pelo Simulador – esta classe é apresentada no Código-Fonte B.7.

```
1  package br.ufba.lasid.jds.prototyping.hddss;
2
3  public class NetworkDeterministic extends Network{
4      double processingTime;
5      public void setProcessingTime(String delay){
6          processingTime = Double.parseDouble(delay);
7      }
8      @Override
9      double delay(Message m) {
10         return processingTime;
11     }
12
13 }
```

Código-Fonte B.7. Código-fonte de NetworkDeterministic.

O código-fonte das classes-base do *framework* **HDDSS** está disponível na data de publicação desta Tese no sítio do **Projeto HDDSS** no *Google Code* em: *https://code.google.com/p/hddss/* e depositado no NIT-UFBA.