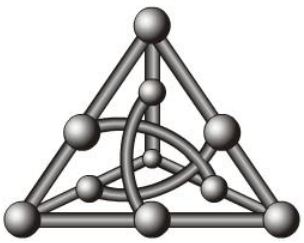
Redução dos efeitos negativos das suspeitas incorretas no algoritmo de consenso de Chandra e Toueg

Lucas Menezes Fermino

DISSERTAÇÃO APRESENTADA
À
FACULDADE DE COMPUTAÇÃO
DA
UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO



Área de concentração: Ciência da Computação
Orientador: Prof. Dr. Irineu Sotoma

Pesquisa apoiada pela CAPES
(Coordenação de Aperfeiçoamento de Pessoal de Nível Superior)

Campo Grande, Maio de 2011

Redução dos efeitos negativos das suspeitas incorretas no algoritmo de consenso de Chandra e Toueg

Este exemplar corresponde à redação final
da dissertação devidamente corrigida
e defendida por Lucas Menezes Fermino
e aprovada pela banca examinadora.

Campo Grande, Maio de 2011.

Banca examinadora:

- Prof. Dr. Irineu Sotoma (FACOM/UFMS) - orientador
- Profa. Dra. Hana Karina Salles Rubinsztejn (FACOM/UFMS)
- Prof. Dr. Edmundo Roberto Mauro Madeira (IC/UNICAMP)

Ao meu avô,
Lázaro Silvério Fermino,
eternizado em minhas lembranças.

Agradecimentos

Primeiramente agradeço a Deus e a nosso Senhor Jesus Cristo por mais uma benção concedida. Graças a Tua presença e a Tua vontade, consegui alcançar inúmeras vitórias em minha vida, e é por isso que devo tudo a Ele, Obrigado Senhor.

Gostaria de fazer um agradecimento especial ao meu orientador, professor Dr. Irineu Sotoma, que durante todo o mestrado me deu total apoio e atenção. Não tenho dúvidas que o seu incentivo, a sua paciência e seus conhecimentos foram cruciais para a conclusão desse trabalho.

Não posso me esquecer dos amigos e colegas que também participaram dessa importante trajetória. Todos, desde a turma do futebol à turma do tereré, me propiciaram momentos de muita descontração e alegria. Dois deles merecem uma atenção especial, o Ronaldo Fiorilo dos Santos (o Gordo) e o Rodrigo Mitsuo Kishi (o Zé), pelo companheirismo que me foi dado desde os tempos de graduação.

Os meus agradecimentos finais são direcionados as pessoas mais importantes da minha vida, a minha família. É óbvio que o meu pai, Valmir Oliveira Fermino, e a minha mãe, Maria Aparecida Menezes Fermino, tiveram uma participação muito mais significante nessa minha jornada. Agradeço muito a eles pelos conselhos, pelos incentivos, pelas palavras de conforto, pelos gestos de afeto e por tudo mais que ambos fizeram por mim. Agradeço também ao meu avô, Lázaro Silvério Fermino, por todos os momentos inesquecíves que passei ao seu lado, inclusive os vários momentos de prosa que tínhamos quando eu o visitava. Apesar de não estar mais entre nós, sua vida será sempre lembrada pelos bons exemplos que deixou a todos. Por último, e não menos importante, agradeço a paciência, o amor e a dedicação da minha companheira Bruna Tonon Ribeiro. Apesar da distância, ela e meus pais sempre me deram ânimo e coragem para continuar lutando por meus ideais. Todos vocês me ajudaram a superar todas as dificuldades que tive durante os meus estudos na graduação e no mestrado. Palavras são insuficientes para descrever o que vocês representam para mim. Por isso encerro dizendo, Muito Obrigado por fazerem parte da minha vida, Amo vocês!

Lucas Menezes Fermino

Resumo

Alguns protocolos na área de sistemas distribuídos, como por exemplo, *atomic broadcast* e replicação semi-passiva, se baseiam no algoritmo de consenso proposto por Chandra e Toueg. Esse algoritmo é equipado com um detector de falhas não confiável. Em sistemas distribuídos assíncronos, esse tipo de detector pode cometer erros ao suspeitar erroneamente de um processo que ainda está em execução. A presença de suspeitas incorretas degrada significativamente o desempenho do algoritmo e o desempenho de qualquer protocolo que o utiliza. Para minimizar essa degradação, nós propomos duas novas otimizações e uma adaptação da técnica *Look-Ahead* ao algoritmo. A primeira otimização, denominada *Early-Decision*, permite antecipar uma decisão para o problema de consenso. A segunda otimização, denominada *Additional-Waiting*, permite estender o tempo de espera por mensagens quando for útil. A técnica *Look-Ahead* ajuda a acelerar a execução do consenso quando existem processos em diferentes rodadas. Nós apresentamos a descrição do algoritmo que combina essas otimizações, e provamos a sua corretude.
Nós realizamos uma série de simulações para avaliar os efeitos das otimizações sobre o desempenho do algoritmo de Chandra e Toueg. Além disso, nós comparamos o desempenho de alguns algoritmos de consenso e selecionamos o melhor, o algoritmo de Paxos, para ser comparado com o algoritmo de Chandra e Toueg otimizado. Os resultados das simulações mostram que todas as otimizações são eficazes, principalmente, quando são combinadas. Na maioria das situações consideradas, o desempenho do algoritmo de Chandra e Toueg otimizado é melhor que o do algoritmo de Paxos.

Palavras-chave: Sistema distribuído assíncrono, consenso, detector de falhas, otimização, simulação, avaliação de desempenho

Abstract

Some protocols in distributed systems, such as *atomic broadcast* and semi-passive replication, are based on the consensus algorithm proposed by Chandra and Toueg. This algorithm is equipped with an unreliable failure detector. In asynchronous distributed systems, this type of detector can make mistakes by erroneously suspecting a process that is still running. The presence of wrong suspicions degrades significantly the performance of the algorithm and the performance of any protocol that uses it. To reduce this degradation, we propose two new optimizations and an adaptation of the *Look-Ahead* technique to the algorithm. The first optimization, named *Early-Decision*, allows to antecipate a decision to the consensus problem. The second optimization, named *Additional-Waiting*, allows extending the waiting time for messages when it is useful. The *Look-Ahead* technique helps speed up the execution of consensus when there are processes in different rounds. We present a description of the algorithm that combines these optimizations and prove its correctness.
We conducted some simulations to evaluate the effects of optimizations on the performance of the algorithm. In addition, we compared the performance of some consensus algorithms and selected the most efficient, the Paxos algorithm, to be compared with the Chandra and Toueg optimized algorithm. The simulation results show that all optimizations are effective, particularly, when they are combined. In most situations considered, the performance of the Chandra and Toueg optimized algorithm is better than the Paxos algorithm.

Keywords: Asynchronous distributed system, consensus, failure detectors, optimization, simulation, performance evaluation

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O consenso[1] é um dos problemas mais fundamentais na área de sistemas distribuídos, além de ser essencial na construção de protocolos distribuídos tolerantes a falhas. É impossível solucionar deterministicamente o consenso em sistemas distribuídos assíncronos propensos a falhas de processo, como mostra o resultado de impossibilidade de Fischer, Lynch e Paterson [12]. Essa impossibilidade se deve a dificuldade de se determinar se um processo falhou ou está apenas lento. Para exemplificar, suponha que existam dois processos p e q executando um protocolo de consenso. Digamos que depois de algum tempo, o processo p pare de esperar pelas mensagens do processo q, assumindo que q tenha falhado. Como o processo q não falhou, interromper a espera pelas mensagens de q não é a decisão correta a ser tomada, pois p receberia a mensagem de q em algum momento. Nesse caso, ou o processo q está lento ou a mensagem enviada por q está atrasada. Além disso, a decisão viola a propriedade *safety*[2] do consenso. Para evitar esse problema, vamos supor que o processo p não pare de esperar por uma mensagem do processo q. Nesse caso, se o processo q falhar, é fácil observar que p esperará para sempre pela mensagem de q. Essa situação causa a violação da propriedade *liveness* [6, 18]. Para contornar o resultado de impossibilidade, Chandra e Toueg [6] introduziram o conceito de detectores de falhas não confiáveis. Informalmente, um detector de falhas não confiável é um "oráculo" distribuído que fornece informações (possivelmente incorretas) sobre falhas de processo.

Chandra e Toueg [6] propuseram um algoritmo de consenso que utiliza detectores de falhas não confiáveis. O uso desse tipo de detecção não garante que as suspeitas sejam sempre corretas, pois o detector pode cometer erros ao suspeitar erroneamente de processos que ainda estão em execução [6, 18]. Um estudo realizado por P. Urbán et al. [22] revelou que a ocorrência de suspeitas incorretas degrada significativamente o desempenho desse algoritmo. Apesar de existir outros protocolos de consenso mais eficientes na literatura, a razão pela qual estamos interessados em tornar esse algoritmo mais eficiente se deve a existência de protocolos distribuídos que utilizam ou se baseiam no algoritmo de Chandra e Toueg. Por exemplo, o módulo responsável pelo consenso na replicação semi-passiva [9] é totalmente baseado no algoritmo de Chandra e Toueg. Portanto, se melhorarmos o desempenho desse algoritmo, os protocolos que o utilizam também poderão ser beneficiados.

Para minimizar a degradação de desempenho, nós propomos duas novas otimizações e uma adaptação da técnica *Look-Ahead* ao algoritmo. A primeira otimização, denomi-

[1] Do inglês: *Consensus.*

[2] Um problema pode ser definido em termos de dois tipos de propriedades: *safety* define que "nada de ruim acontece", enquanto que, *liveness* define que "alguma coisa boa vai acontecer em algum momento".

nada *Early-Decision*, permite antecipar um decisão para o consenso com base no estado atual dos processos. A segunda otimização, denominada *Additional-Waiting*, utiliza um detector de falhas $\Diamond S$ para estender o tempo de espera por mensagens quando for possível acelerar o consenso. Uma das vantagens dessa otimização é que ela pode ser aplicada a outros protocolos de consenso. A técnica *Look-Ahead* [23] ajuda a acelerar a execução de processos atrasados por meio de mensagens enviadas em rodadas futuras. Nós apresentamos a descrição do algoritmo de Chandra and Toueg que combina essas otimizações e provamos sua corretude.

Nosso estudo se concentra apenas em cenários com suspeitas incorretas. Nós realizamos uma série de simulações para avaliar os efeitos das otimizações sobre o desempenho do algoritmo de Chandra e Toueg. As otimizações são avaliadas de duas maneiras: individualmente e combinadas. Além disso, com base nos resultados de comparação de desempenho de alguns algoritmos de consenso que realizamos, nós concluímos que o algoritmo de Paxos [3, 15, 16] é o mais eficiente. Por essa razão, nós o escolhemos para ser comparado com o algoritmo de Chandra e Toueg otimizado. O objetivo dessa comparação é determinar se a nossa proposta fornece um ganho de desempenho significativo em relação a um algoritmo de consenso mais eficiente. Os resultados confirmam a eficácia de cada otimização e mostram que, na maioria das situações consideradas, o desempenho do algoritmo de Chandra e Toueg otimizado é melhor que o do algoritmo de Paxos.

O trabalho está organizado da seguinte forma. O Capítulo 2 descreve o modelo de sistema e alguns conceitos básicos para o nosso estudo. No Capítulo 3, discutimos cada um dos algoritmos de consenso que estamos considerando. No Capítulo 4, nós descrevemos cada uma das otimizações, apresentamos o algoritmo que as combina e provamos sua corretude. O Capítulo 5 apresenta o modelo de avaliação de desempenho, alguns detalhes de implementação e os resultados das simulações. Finalmente, concluímos o trabalho no Capítulo 6.

Capítulo 2

Modelo de sistema e conceitos básicos

Neste capítulo, nós apresentamos o modelo de sistema adotado pelos algoritmos de consenso e discutimos alguns conceitos fundamentais para a compreensão do nosso estudo.

2.1 Modelo de sistema

Esse trabalho assume um sistema distribuído assíncrono, ou seja, não existe limite de tempo na transmissão de mensagens e no processamento de tarefas. O sistema é equipado com detectores de falhas não confiáveis, e composto por um conjunto finito ordenado de $n > 1$ processos, $\prod = \{p_1, p_2, ..., p_n\}$. Um processo falha (falhas não Bizantinas[1]) devido a quebra. Dizemos que um processo está correto se, e somente se, ele nunca falha. O número máximo de processos que podem falhar é denotado por f. Para garantir uma maioria de processos corretos, $f < \lceil \frac{n}{2} \rceil$. Nós adotamos o modelo *crash-stop*, isto é, falhas são permanentes[2].

Os processos se comunicam por troca de mensagens através de canais de comunicação confiáveis [2]. Um canal confiável não cria, duplica ou perde mensagens [2]. As mensagens de decisão são enviadas por *Reliable Broadcast* [1], o qual é definido em termos de duas primitivas: *R-broadcast* e *R-deliver*. Essas duas primitivas satisfazem as seguintes propriedades [1, 6]:

- Validade[3]: Se um processo correto executa um *R-broadcast*(m), então ele executa um *R-deliver*(m) em algum momento.
- Integridade uniforme[4]: Para qualquer mensagem m, todo processo executa um *R-deliver*(m) no máximo uma vez, e somente se, algum outro processo executou um *R-broadcast*(m).
- Acordo[5]: Se um processo correto executa um *R-deliver*(m), então todos os processos corretos executam um *R-deliver*(m) em algum momento.

[1]Esse tipo de falha inclui omissão de mensagens, alteração de mensagens, criação ilegítimas de mensagens, entre outras.

[2]Na prática, os processos podem se recuperar, entretanto, assumem uma nova identidade.

[3]Do inglês: *Validity*.

[4]Do inglês: *Uniform integrity*.

[5]Do inglês: *Agreement*.

2.2 O problema de consenso

O consenso é um paradigma de problemas de acordo. Os processos devem concordar unanimemente em executar uma mesma ação ou tomar uma mesma decisão. Formalmente, o consenso é definido em termos de duas primitivas: *propose* e *decide*. Quando um processo executa um *propose*(v), nós dizemos que ele propôs v; similarmente, quando um processo executa um *decide*(v), nós dizemos que ele decide v. Todos os processos iniciam o protocolo de consenso com uma estimativa inicial do valor de decisão e devem decidir o mesmo valor irrevogavelmente. O problema de consenso satisfaz as seguintes propriedades [6, 18]:

- Terminação[6]: Todo processo correto, em algum momento, decide algum valor v.
- Integridade uniforme: Todo processo decide no máximo uma vez.
- Acordo uniforme[7]: Dois processos (corretos ou quebrados) não decidem diferentemente.
- Validade uniforme[8]: Se um processo decide um valor v, então v foi proposto por algum processo.

2.3 *Atomic broadcast*

O *atomic broadcast* é outro problema fundamental na área de sistemas distribuídos assíncronos, e pode ser reduzido ao problema de consenso. Informalmente, ele garante que todos os processos corretos entreguem a mesma sequência de mensagens. O algoritmo proposto por Chandra e Toueg [6] consiste na utilização de duas primitivas: *A-broadcast*(m) e *A-deliver*(m). Um processo executa um *A-broadcast*(m) para enviar uma mensagem m de *atomic broadcast* a todos os processos[9]. A mensagem recebida por um processo é armazenada até que a ordem de entrega seja definida. A decisão sobre a ordem de entrega das mensagens é feita utilizando uma sequência numerada de execuções de consenso. A estimativa inicial do valor de decisão de cada processo, para uma dada execução k do consenso, corresponde a um conjunto de identificadores de mensagem msg_k, ou seja, cada processo propõe, inicialmente, uma sequência qualquer de mensagens. Assim que a ordem de entrega é definida, o processo executa um *A-deliver*(m) para entregar cada mensagem m na ordem correta [6].

2.4 Detectores de falhas não confiáveis

Um detector de falhas não confiável é um mecanismo que fornece informações sobre o estado do processo, ou seja, se falhou ou não. Contudo, estas informações podem ser incorretas [6, 18]. Chandra e Toueg [6] caracterizaram os detectores de falhas em termos de duas propriedades: completude[10] e precisão[11]. A completude caracteriza a capacidade de suspeitar de todos os processos quebrados, enquanto que, a precisão

[6] Do inglês: *Termination.*
[7] Do inglês: *Uniform agreement.*
[8] Do inglês: *Uniform validity.*
[9] A mensagem é enviada utilizando *Reliable Broadcast.*
[10] Do inglês: *Completeness.*
[11] Do inglês: *Accuracy.*

caracteriza a capacidade de não suspeitar dos processos corretos. Essas propriedades definem várias classes de detectores de falhas. Um detector pertence a uma dada classe se satisfizer as propriedades definidas para aquela classe. Na prática, muitas implementações de detectores de falhas são baseadas em mecanismos de *time-out* [6].

Nas próximas seções, nós discutiremos algumas classes de detectores de falhas não confiáveis.

2.4.1 Detector de falhas $\Diamond S$

Um detector de falhas $\Diamond S$ (*eventually accurate*) fornece uma lista de identificadores dos processos que estão suspeitos de falha. Cada processo é equipado com um módulo local do detector. Como se trata de um detector não confiável, ele pode erroneamente incluir um processo correto em sua lista de suspeitos ou pode deixar de incluir um processo que realmente tenha falhado. Caso o detector perceba que a suspeita tenha sido incorreta, ele pode corrigi-la. Além disso, processos distintos podem possuir diferentes listas de suspeitos [6, 18].

A lista de processos suspeitos fornecida por um detector de falhas $D \in \Diamond S$ a um dado processo p, é denotada por $D.suspected_p$. Para que um detector de falhas pertença à classe $\Diamond S$, ele deve satisfazer as seguintes propriedades [6, 18]:

- Completude forte[12]: Em algum momento, todo processo que falha é permanentemente suspeito por todo processo correto.
- Precisão terminal fraca[13]: Existe um momento em que algum processo correto nunca é suspeito por qualquer processo correto.

2.4.2 Detector de falhas Ω

Toda vez que um detector de falhas Ω é consultado, ele retorna a um processo p, um único processo q considerado estar correto. Dizemos que p confia no processo líder q. Cada processo possui um módulo do detector. Como se trata de um mecanismo não confiável, pode haver vários líderes ao mesmo tempo. Os processos confiáveis fornecidos por um detector de falhas $D \in \Omega$ a um dado processo p, são obtidos por meio da invocação da função *leader*(). Um detector dessa classe deve satisfazer a seguinte propriedade [5, 14, 17]:

Propriedade 1. Em algum momento, todo processo correto confia, permanentemente, no mesmo processo correto (líder).

Essa classe pode ser vista como um mecanismo de eleição de líder, já que ela garante que todos os processos corretos confiarão no mesmo líder em algum momento.

2.4.3 Detector de falhas $\Diamond C$

Mikel Larrea, Antonio Fernández e Sergio Arévalo [17] introduziram uma classe de detectores de falhas denominada *eventually consistent*. Essa classe combina as características das classes de detectores Ω e $\Diamond S$.

[12]Do inglês: *Strong completeness.*
[13]Do inglês: *Eventual weak accuracy.*

Um detector de falhas $\Diamond C$ atende a dois tipos de requisição: uma para requisitar o conjunto de processos suspeitos e a outra para requisitar um processo confiável [17]. Para que um detector de falhas D pertença à classe $\Diamond C$, ele deve fornecer a todo processo p, um conjunto de processos suspeitos e um processo confiável, de modo que [17]:

- o conjunto de processos suspeitos satisfaz as propriedades definidas pela classe $\Diamond S$;
- os processos confiáveis satisfazem a propriedade 1 definida pela classe Ω; e
- existe um momento em que os processos confiáveis não são suspeitos.

2.5 Algoritmos indulgentes

Um algoritmo indulgente é um algoritmo distribuído que, além de tolerar falhas de processo, tolera também informações não confiáveis [13]. Algoritmos que utilizam detectores de falhas não confiáveis são indulgentes. De modo informal, esses algoritmos são indulgentes em relação aos seus oráculos, pois estes fornecem informações não confiáveis sobre as falhas que ocorrem no sistema. Embora as propriedades *safety* e *liveness* do consenso sejam mantidas, a propriedade de terminação do consenso (propriedade *liveness*) pode levar um pouco mais de tempo para ser satisfeita, já que depende do oráculo [11, 13, 14].

2.5.1 Propriedades indulgentes

Neste trabalho, todos os algoritmos de consenso considerados são indulgentes. O uso de detectores de falhas não confiáveis impede a definição de um limite superior no número de passos de comunicação[14] necessário para alcançar consenso, pois são mecanismos que podem cometer erros [11]. R. Guerraoui e M. Raynal [14] definiram algumas propriedades que limitam o número de passos de comunicação em execuções estáveis (falhas são iniciais[15] e os oráculos se comportam de forma perfeita). As propriedades definidas são as seguintes [14]:

- *Oracle-efficiency.* Em execuções de consenso onde não existem falhas e o oráculo (Ω ou $\Diamond S$) se comporta de forma perfeita, são necessários apenas dois passos de comunicação para se alcançar uma decisão.
- *Zero-degradation.* Esta propriedade estende a propriedade *oracle-efficiency.* Ao invés de execuções sem falhas, tem-se execuções com falhas iniciais. Algoritmos de consenso que satisfazem essa propriedade necessitam de apenas dois passos de comunicação para alcançar consenso.
- *One-step-decision.* Existem aplicações em que um valor α aparece com mais frequência do que outros, logo, ele possui maiores chances de ser proposto. Assim, em qualquer execução do consenso, onde todos os processos (que não falharam inicialmente) propõem o valor α e o oráculo se comporta de forma perfeita, um único passo de comunicação é suficiente para alcançar consenso. Além disso, existem casos em que um conjunto pré-determinado de processos S é inicialmente

[14] Instante onde ocorre troca de mensagens entre os processos.
[15] São falhas que ocorrem antes de iniciar a execução de uma instância do consenso.

conhecido por cada processo. Quando os processos em S não falham e propõem o mesmo valor, um único passo de comunicação também é suficiente para alcançar consenso.

- *Configuration-efficiency.* Quando todos os processos (que não falharam inicialmente) propõem o mesmo valor inicial, uma decisão é alcançada sem a necessidade de um oráculo e com apenas dois passos de comunicação.

Além dessas propriedades, W. Wu et al. [23] definiram também a seguinte propriedade indulgente:

- *Round-zero-degradation.* Quando o oráculo se comporta de forma perfeita em uma rodada r e as falhas ocorrem antes de iniciar essa rodada, apenas dois passos de comunicação são necessários para se alcançar uma decisão em r.

Capítulo 3

Algoritmos de consenso

Nas próximas seções, discutiremos uma variedade de algoritmos de consenso que utilizam diferentes mecanismos de eleição de líder e de detecção de falhas. Esse capítulo se resume apenas em apresentar e explicar os algoritmos estudados. A análise comparativa das características de cada um é apresentada na Seção 5.3.

Os algoritmos de consenso assumem o modelo de sistema apresentado na Seção 2.1. Cada rodada dos algoritmos é composta por duas tarefas principais. A primeira tarefa (Tarefa 1) corresponde ao módulo principal do consenso, enquanto que, a segunda tarefa (Tarefa 2) é responsável pela entrega das mensagens de decisão. Ambas são executadas concorrentemente.

Todo processo p executa uma instância k do algoritmo de consenso. Em cada um deles, o consenso é inicializado por meio da chamada do método *propose*(v) (Tarefa 1), onde v corresponde a estimativa inicial do valor de decisão. Um processo alcança uma decisão quando executa a instrução *decide*(v) (Tarefa 2).

3.1 Algoritmo de Chandra e Toueg

O algoritmo proposto por Chandra e Toueg [6] (mostrado na Figura 3.1), utiliza um detector de falhas $\Diamond S$ e determina o coordenador de uma rodada por meio do paradigma de rotação de coordenadores[1]. Um processo p gerencia as seguintes variáveis:

- n: número de processos.
- r_p: rodada do consenso.
- c_p: coordenador da rodada.
- est_p: estimativa do valor de decisão.
- ts_p: estampa de tempo[2]. Última rodada em que a estimativa foi atualizada.
- $state_p$: estado do consenso.

Cada rodada é composta por 4 fases. A fase 1 não precisa ser executada na primeira rodada [8, 22]. As fases do algoritmo são descritas da seguinte forma [6]:

[1]Para cada rodada, rotaciona a lista de processos $\prod = \{p_1, p_2,...,p_n\}$ e seleciona o primeiro processo da lista como coordenador.

[2]Do inglês: *Timestamp.*

```
1: Tarefa 1:
2: procedimento propose(v_p)
3:     est_p ← v_p
4:     state_p ← undecided
5:     r_p ← 0
6:     ts_p ← 0
7:
8:     enquanto state_p = undecided faça
9:         c_p ← (r_p mod n) + 1
10:        r_p ← r_p + 1
11:
12:        Fase 1:
13:        se r_p > 1 então
14:            send (p, r_p, est_p, ts_p) para c_p
15:
16:        Fase 2:
17:        se p = c_p então
18:            se r_p > 1 então
19:                espere até [receber (q, r_p, est_q, ts_q) de ⌈(n+1)/2⌉ processos]
20:                msgs_p[r_p] ← {(q, r_p, est_q, ts_q) | p recebeu (q, r_p, est_q, ts_q) de q}
21:                t ← maior ts_q tal que (q, r_p, est_q, ts_q) ∈ msgs_p[r_p]
22:                est_p ← selecione um est_q tal que (q, r_p, est_q, t) ∈ msgs_p[r_p]
23:            send (p, r_p, est_p) para todos
24:
25:        Fase 3:
26:        espere até [receber (c_p, r_p, est_{c_p}) de c_p ou c_p ∈ D.suspected_p]
27:        se [recebeu (c_p, r_p, est_{c_p}) de c_p] então
28:            est_p ← est_{c_p}
29:            ts_p ← r_p
30:            send (p, r_p, ack) para c_p
31:        senão
32:            send (p, r_p, nack) para c_p
33:
34:        Fase 4:
35:        se p = c_p então
36:            espere até [receber (q, r_p, ack) ou (q, r_p, nack) de ⌈(n+1)/2⌉ processos]
37:            se [recebeu (q, r_p, ack) de ⌈(n+1)/2⌉ processos] então
38:                R-broadcast (p, r_p, est_p, decide)
39:
40: Tarefa 2:
41:  quando R-deliver (q, r_q, est_q, decide) faça
42:     se state_p = undecided então
43:         state_p ← decide
44:         decide (est_q)
```

Figura 3.1: Algoritmo de Chandra e Toueg.

- Na fase 1, todo processo em uma rodada $r > 1$, envia sua estimativa e sua estampa de tempo ao coordenador c ($[r \bmod n] + 1$) da rodada.
- Na fase 2, o coordenador da rodada $r = 1$ sempre propõe sua própria estimativa aos processos. O coordenador de uma rodada $r > 1$, espera até receber uma maioria de estimativas. Após a espera, ele seleciona a estimativa com maior estampa de tempo e a envia a todos os processos como sua nova estimativa.
- Na fase 3, existem duas possibilidades para cada processo p:

  - p adota a estimativa de c, atualiza sua estampa de tempo com a rodada atual e envia um *ack* para c para indicar que ele adotou a estimativa; ou
  - p suspeita de c e o envia um *nack*.

- Na fase 4, o coordenador espera até receber uma maioria de mensagens *ack*. Se todas as mensagens recebidas são *ack*, então o valor proposto se torna o valor de decisão. O coordenador envia a mensagem de decisão a todos os processos usando *Reliable Broadcast*. Por outro lado, se uma mensagem *nack* for recebida dentre a maioria de mensagens, então o coordenador procede para a primeira fase da próxima rodada.

3.2 Algoritmo de Mostefaoui e Raynal

O algoritmo proposto por Mostefaoui e Raynal [18] (mostrado na Figura 3.2), utiliza os mesmos mecanismos de eleição de líder e de detecção de falhas do algoritmo de Chandra e Toueg. Cada rodada do algoritmo é dividida em duas fases. O objetivo da fase 1 é fornecer a todos os processos a mesma estimativa do valor de decisão, para que na fase 2, os processos possam alcançar uma decisão. Cada processo p gerencia as seguintes variáveis:

- n: número de processos.
- r_p: rodada do consenso.
- c_p: coordenador da rodada.
- $est1_p$: estimativa do valor de decisão no início da fase 1.
- $est2_p$: estimativa do valor de decisão no início da fase 2.
- $state_p$: estado do consenso.

Na primeira fase, todo processo determina um coordenador c para a rodada atual. Feito isso, o coordenador envia sua rodada e sua estimativa a todos os processos. Após a espera na fase 1, temos as seguintes possibilidades:

- p recebe a proposta do coordenador. Nesse caso, p adota est_c como valor de $est2$; ou
- p suspeita do seu coordenador. Nesse caso, p adota $\perp$[3] como valor de $est2$.

A fase 2 é iniciada com uma troca de estimativas $est2$, onde $est2$ pode ser v ou $\perp$. Após a espera na fase 2, temos as seguintes possibilidades [18]:

- p recebe v da maioria de processos. Nesse caso, p seleciona v como valor de decisão e o envia a todos os processos usando *Reliable Broadcast*; ou
- p recebe v e $\perp$ da maioria de processos. Nesse caso, p define $est1$ com v; ou
- p recebe $\perp$ da maioria de processos. Nesse caso, p mantém o mesmo valor de quando iniciou o consenso e procede para a próxima rodada.

[3]O símbolo $\perp$ é uma forma comum de denotar a ausência do valor. Ele é conhecido como *nil* ou *null* nas várias linguagens de programação.

```
1: Tarefa 1:
2: procedimento propose(v_p)
3:     r_p ← 0
4:     est1_p ← v_p
5:     state_p ← undecided
6:
7:     enquanto state_p = undecided faça
8:         c_p ← (r_p mod n) + 1
9:         r_p ← r_p + 1
10:
11:        Fase 1:
12:        se p = c_p então
13:            send Fase1(r_p, est1_p) para todos
14:        espere até [receber Fase1(r_p, est_{c_p}) de c_p ou c_p ∈ D.suspected_p]
15:        se [recebeu Fase1(r_p, est_{c_p}) de c_p] então
16:            est2_p ← est_{c_p}
17:        senão
18:            est2_p ←⊥
19:
20:        Fase 2:
21:        send Fase2(r_p, est2_p) para todos
22:        espere até [receber Fase2(r_p, ∗) de ⌈(n+1)/2⌉ processos]
23:        se [recebeu Fase2(r_p, v) de ⌈(n+1)/2⌉ processos] então
24:            est1_p ← v
25:            R-broadcast (p, r_p, est1_p, decide)
26:        senão se [recebeu Fase2(r_p, v) de algum processo] então
27:            est1_p ← v
28:
29: Tarefa 2:
30: quando R-deliver (q, r_q, est_q, decide) faça
31:     se state_p = undecided então
32:         state_p ← decide
33:         decide (est_q)
```

Figura 3.2: Algoritmo de Mostefaoui e Raynal.

3.3 Algoritmo de Paxos

O algoritmo de Paxos (mostrado na Figura 3.3) foi introduzido por Lamport [15, 16] e proposto originalmente para o modelo *crash-recovery*. No trabalho de R. Boichat et al. [3], são apresentadas duas descrições do algoritmo de Paxos, uma para o modelo *crash-recovery* e outra para o modelo *crash-stop*. Como o modelo de falhas adotado nesse trabalho é *crash-stop*, esta seção apresenta o algoritmo de Paxos correspondente.

O algoritmo conta com duas abstrações: $\Diamond$*Register* e $\Diamond$*Leader*. A abstração $\Diamond$*Register* encapsula o algoritmo responsável por "armazenar" e "travar" o valor de decisão. Por outro lado, a abstração $\Diamond$*Leader* encapsula o algoritmo usado para eleição terminal de líder. Nosso estudo assume que essa abstração corresponde a um detector de falhas Ω.

A abstração $\Diamond$*Register* fornece um armazenamento distribuído por meio da semântica "escreve uma vez"[4]. A ação de "armazenar" um valor pode falhar se muitos processos tentarem armazenar algum valor no registrador ($\Diamond$*Register*) concorrentemente. Nesse caso, nenhum deles terá sucesso [3].

A abstração $\Diamond$*Register* é composta por dois procedimentos, $\Diamond$*Register*() e *propose*().

[4]Do inglês: *Write-once.*

O primeiro é usado apenas para inicializar as variáveis usadas pelo procedimento principal, o *propose*(). Esse procedimento é invocado pelos processos com um único argumento $v \in Values$, onde $Values$ é o conjunto de possíveis valores que podem ser armazenados no registrador. Um processo p invoca o *propose*(v) para propor um valor v. Se o *propose*() retornar $v' \neq abort$, então p decide v', caso contrário, se o *propose*() retornar *abort*, então p aborta a rodada atual [3]. A abstração $\Diamond Register$ deve satisfazer as seguintes propriedades [3]:

- Terminação: (1) Cada processo correto que propõe, em algum momento, decide ou aborta. (2) Se um único processo correto propõe infinitamente, em algum momento, ele decide.
- Acordo uniforme: Dois processos não decidem diferentemente.
- Validade uniforme: Se um processo decide um valor v, então v foi proposto por algum processo.

As duas abstrações são usadas para descrever o algoritmo de consenso de Paxos [3]. O módulo principal do consenso (Tarefa 1) é dividido em dois procedimentos, um para sua inicialização e outro para sua execução.

```
 1: Tarefa 1:
 2: procedimento INICIALIZAÇÃO
 3:     register_p ← new ◊Register
 4:     ld_p ← new ◊Leader
 5:     decision_p ← abort
 6:
 7: procedimento propose(v)
 8:     enquanto decision_p = abort faça
 9:         se ld_p.leader() = p então
10:             decision_p ← register_p.propose(v)
11:     R-broadcast (decision_p)
12:
13: Tarefa 2:
14:  quando R-deliver (v) faça
15:     decide (v)
```

Figura 3.3: Algoritmo de Paxos.

Qualquer processo eleito como líder realiza invocações *propose*(v) até deixar de ser o líder ou decidir (i.e, a invocação *propose*(v) retorna um valor $v' \neq abort$ ou o processo recebe a decisão de algum outro processo) [3].

A Figura 3.4 apresenta o pseudo-código responsável pela abstração $\Diamond Register$. Ele possui duas etapas: proposição e recebimento de proposta. A primeira é aplicada quando o processo tem algum valor a ser proposto. A segunda é executada pelos processos que agem como testemunhas (não se consideram líder) para tratar o recebimento de alguma proposta. A etapa de recebimento é executada por duas tarefas que geram respostas às invocações do método *propose*(). A tarefa de recebimento de mensagens mantém em cada processo p_i uma estimativa do valor do registrador denotada por val_i, e inicializada com $\perp$. A rodada r inicial do consenso é igual ao identificador i do processo ($i = 1, 2, ..., n$, onde n corresponde ao número de processos). Para todas as invocações subsequentes, a rodada é incrementada de n [3].

```
1:  procedimento ◊Register()
2:      read_i ← 0
3:      write_i ← 0
4:      val_i ←⊥
5:      v* ←⊥
6:      r_i ← i
7:
8:  procedimento propose(v)
9:      v* ← v
10:     se r_i > 1 então
11:         send (READ, r_i) para todos processos
12:         espere até [receber (ackREAD, r, ∗, ∗) ou (nackREAD, r) de ⌈(n+1)/2⌉ processos]
13:         se [recebeu pelo menos um (nackREAD, r)] então
14:             r_i ← r_i + n
15:             retorna (abort)
16:         senão
17:             selecione um (ackREAD, r, r', val') com maior r'
18:             se val' ≠⊥ então
19:                 v* ← val'
20:
21:     send (WRITE, r_i, v*) para todos processos
22:     r_i ← r_i + n
23:     espere até [receber (ackWRITE, r) ou (nackWRITE, r) de ⌈(n+1)/2⌉ processos]
24:     se [recebeu pelo menos um (nackWRITE, r)] então
25:         retorna (abort)
26:     senão
27:         retorna (v*)
28:
29:      espere até [receber (READ, r_i) de p_i] faça
30:         se write_i ≥ r_i ou read_i ≥ r_i então
31:             send (nackREAD, r_i) para p_i
32:         senão
33:             read_i ← r_i
34:             send (ackREAD, r_i, write_i, val_i) para p_i
35:
36:       espere até [receber (WRITE, r_i, v') de p_i] faça
37:         se write_i > r_i ou read_i > r_i então
38:             send (nackWRITE, r_i) para p_i
39:         senão
40:             write_i ← r_i
41:             val_i ← v'
42:             send (ackWRITE, r_i) para p_i
```

Figura 3.4: Pseudo-código da abstração ◊*Register*.

O método *propose*() é composto por duas fases: leitura[5] e escrita[6]. Cada uma delas pode abortar ou ter sucesso. Tanto a fase de leitura, como a fase de escrita, podem receber mensagens específicas do tipo *ack* ou *nack*. Elas são utilizadas para detectar conflito de proposições, ou seja, se existe mais de um processo propondo uma estimativa.

O objetivo da fase de leitura é detectar qualquer valor que já tenha sido escrito no registrador com sucesso, enquanto que, o objetivo da fase de escrita é fazer com que a estimativa do líder, com maior rodada, seja escrita em uma maioria de processos. Além disso, ambas as fases possuem um objetivo comum, que é obter uma garantia

[5]Do inglês: *Read*.
[6]Do inglês: *Write*.

das testemunhas de que nenhuma leitura ou escrita subsequente terá sucesso com uma rodada inferior.

A fase de leitura não precisa ser realizada na primeira rodada, pois todos os processos possuem suas estimativas val_i iniciadas com $\perp$. Nesse caso, a fase de escrita é iniciada diretamente. Para as demais rodadas, a fase de escrita iniciará apenas quando a fase de leitura obtiver êxito. Caso as duas fases tenham sucesso, uma decisão é alcançada [3].

O método *propose*() executa as seguintes etapas:

- A estimativa de escrita v^* é definida, inicialmente, com o valor v. Logo em seguida, p_i verifica se $r > 1$. Em caso afirmativo, p_i inicia a fase de leitura para determinar se existe algum valor escrito no registrador. Caso contrário, p_i mantém o valor em v^* e inicia a fase de escrita. Na fase de leitura, p_i envia uma mensagem de leitura (READ, r_i) a todos os processos. Se algum processo p_j recebeu uma mensagem de leitura ou de escrita com $r \geq r_i$ antes da mensagem de leitura de p_i, então p_j envia um *nack*READ a p_i. Caso contrário, p_j define $read_j$ com r_i e envia um *ack*READ contendo val_j (estimativa do valor de escrita) e $write_j$ (rodada em que val_j foi atualizado pela última vez). Se uma maioria de mensagens *ack*READ for recebida, então p_i seleciona o valor val ($\neq \perp$) com maior $write$ e avança para a fase de escrita. Caso $val = \perp$, então p_i mantém o valor em v^*. Por outro lado, se p_i receber pelo menos um *nack*READ dentre a maioria de mensagens, então p_i aborta a fase de leitura.

- Na fase de escrita, p_i envia uma mensagem de escrita (WRITE, r_i, v^*) a todos os processos. Se algum processo p_j recebeu uma mensagem de leitura ou de escrita com $r > r_i$ antes da mensagem de escrita de p_i, então p_j envia um *nack*WRITE a p_i. Caso contrário, p_j atualiza val_j com v^* e $write_j$ com r_i e envia um *ack*WRITE para p_i. Se uma maioria de mensagens *ack*WRITE for recebida, então p_i retorna o valor proposto v^* como valor de decisão. Por outro lado, se p_i receber pelo menos um *nack*WRITE dentre a maioria de mensagens, então p_i aborta a fase de escrita.

3.4 Algoritmo baseado na classe $\Diamond C$

M. Larrea, A. Fernández e S. Arévalo [17] propuseram um algoritmo de consenso (mostrado nas Figuras 3.5 e 3.6) baseado na classe de detectores $\Diamond C$. O algoritmo é uma variação do algoritmo proposto por Chandra e Toueg, porém, ao invés de utilizar o paradigma de rotação de coordenadores, o algoritmo utiliza um detector de falhas Ω para eleger os coordenadores de cada rodada.

Toda rodada do algoritmo é dividida em 5 fases, e pode ter mais de um coordenador. A tarefa adicional (Tarefa 3), executada concorrentemente com as outras duas, garante que nenhum coordenador bloqueie durante a execução do consenso [17]. Um processo p gerencia as seguintes variáveis [17]:

- n: número de processos.
- r_p: rodada do consenso.
- est_p: estimativa do valor de decisão.
- ts_p: estampa de tempo. Última rodada em que a estimativa foi atualizada.

- $state_p$: estado do consenso.
- $chosen_p$: indica se o coordenador da rodada foi ou não escolhido.
- $replied_p$: indica se o processo enviou ou não uma resposta para algum coordenador da rodada.

As fases do algoritmo são as seguintes:

- Na fase 0, todo processo determina o seu coordenador para a rodada atual. Um processo se torna seu próprio coordenador quando a primeira cláusula da linha 13 é satisfeita. Quando isso ocorre, o coordenador anuncia aos demais processos que ele é o coordenador. Um processo se torna um participante quando recebe uma mensagem de um coordenador, ou seja, quando a segunda cláusula da linha 13 é satisfeita. Logo após a fase 0 e concorrentemente com o algoritmo principal (Tarefa 3), se um processo p receber uma mensagem $(q, r_q, coordinator)$ de q, com $r_q \leq r_p$, então p lhe enviará uma estimativa $\perp$.
- Na fase 1, todo processo p envia sua estimativa e sua estampa de tempo ao coordenador c da rodada atual.
- Na fase 2, todo coordenador c espera até receber uma maioria de mensagens. Após a espera, temos as seguintes possibilidades:
  - c recebe uma maioria de estimativas $\neq \perp$. Nesse caso, c adota a estimativa com maior estampa de tempo e a propõe a todos os processos (isso indica que pelo menos uma maioria de processos possui o mesmo coordenador); ou
  - c não recebe uma maioria de estimativas $\neq \perp$. Nesse caso, c propõe $\perp$.
- Na fase 3, existem três possibilidades para cada processo p:
  - p recebe uma proposta $\neq \perp$ de algum coordenador e o envia um *ack* para indicar que sua estimativa foi adotada; ou
  - p recebe uma proposta $= \perp$ do seu coordenador, descarta a mensagem e procede para a próxima rodada; ou
  - p suspeita do seu coordenador e procede para a próxima rodada.

  Logo após essa fase e concorrentemente com o algoritmo principal (Tarefa 3), se um processo p receber uma proposta (q, r_q, est_q) de algum coordenador, com $r_q \leq r_p$, então p lhe enviará um *nack*.
- Na fase 4, o coordenador que obteve sucesso na fase 2, espera até receber uma maioria de mensagens *ack* ou *nack*. Após a espera, temos as seguintes possibilidades:
  - c recebe uma maioria de mensagens *ack*. Nesse caso, o valor proposto se torna o valor de decisão. O coordenador envia a mensagem de decisão a todos os processos usando *Reliable Broadcast*; ou
  - c não recebe uma maioria de mensagens *ack*. Nesse caso, o coordenador procede para a primeira fase da próxima rodada.

```
1: Tarefa 1:
2: procedimento propose(v_p)
3:     est_p ← v_p
4:     state_p ← undecided
5:     r_p ← 0; ts_p ← 0
6:
7:     enquanto state_p = undecided faça
8:         chosen_p ← false
9:         replied_p ← false
10:        r_p ← r_p + 1
11:
12:        Fase 0:
13:        espere até [p = leader() ou receber (q, r_q, coord) de algum processo tal que (r_q ≥ r_p)]
14:        se [recebeu (q, r_q, coord) de algum processo tal que (r_q ≥ r_p)] então
15:            c_p ← q; r_p ← r_q
16:        senão
17:            c_p ← p
18:            send (p, r_p, coord) para os demais processos
19:        chosen_p ← true
20:
21:        Fase 1: send (p, r_p, est_p, ts_p) para c_p
22:
23:        Fase 2:
24:        se p = c_p então
25:            espere até [receber (q, r_p, est_q, ts_q) ou (q, r_p, ⊥, 0) de ⌈(n+1)/2⌉ processos]
26:            se [recebeu (q, r_p, est_q, ts_q) de ⌈(n+1)/2⌉ processos] então
27:                decidible_p ← true
28:                est_p ← est_q com maior estampa de tempo
29:                send (p, r_p, est_p) para todos
30:            senão
31:                decidible_p ← false
32:                send (p, r_p, ⊥) para todos
33:
34:        Fase 3:
35:        espere até [receber (q, r_p, est_q) de algum processo q ou (c_p, r_p, ⊥) de c_p
           ou c_p ∈ D.suspected_p]
36:        se [recebeu (q, r_p, est_q) de algum processo q] então
37:            est_p ← est_q; ts_p ← r_p
38:            send (p, r_p, ack) para q
39:        senão se [recebeu (c_p, r_p, ⊥) de c_p] então
40:            descarte a mensagem
41:        replied_p ← true
42:
43:        Fase 4:
44:        se p = c_p e decidible_p então
45:            espere até [receber (q, r_p, ack) ou (q, r_p, nack) de ⌈(n+1)/2⌉ processos]
46:            se  [recebeu (q, r_p, ack) de ⌈(n+1)/2⌉ processos] então
47:                R-broadcast (p, r_p, est_p, decide)
```

Figura 3.5: Tarefa 1 do algoritmo baseado na classe $\Diamond C$.

A descrição do algoritmo desta seção não é a mesma apresentada em [17]. Primeiramente, nós desconsideramos a melhoria proposta ao algoritmo por motivos que serão esclarecidos na Seção 5.3. Em segundo lugar, nós observamos que no algoritmo original, existem situações onde um determinado processo p pode enviar mais do que uma mensagem *nack* para o seu coordenador. Isso acontece quando um processo suspeita

48: **Tarefa 2:**
49: **quando** *R-deliver* (q, r_q, est_q, *decide*) **faça**
50: **se** $state_p = undecided$ **então**
51: $state_p \leftarrow decide$
52: *decide* (est_q)
53:
54: **Tarefa 3:**
55: **quando** receber (q, r_q, *coord*) de q tal que ($r_q < r_p$ **ou** ($r_q = r_p$ **e** $chosen_p = true$)) **faça**
56: *send* (p, r_q, $\perp$, 0) para q
57: **quando** receber (q, r_q, est_q) de q tal que ($r_q < r_p$ **ou** ($r_q = r_p$ **e** $replied_p = true$)) **faça**
58: *send* (p, r_q, *nack*) para q

Figura 3.6: Tarefas 2 e 3 do algoritmo baseado na classe $\Diamond C$.

incorretamente do seu coordenador e, em seguida, recebe a sua proposta (verificar a fase 3 do algoritmo original). Esse tipo de situação prejudica o desempenho do algoritmo. Uma solução simples para contornar esse problema é não enviar a mensagem *nack* na fase 3. Nós provamos que essa modificação não bloqueia nenhum coordenador na fase 4.

Prova por contradição. Seja r a menor rodada onde um coordenador c bloqueia para sempre na fase 4. Portanto, c enviou uma estimativa $\neq \perp$ na fase 2. Nesse caso, na fase 3 da rodada r, existem três possibilidades para um processo correto p:

- p recebe a estimativa de c e o envia uma mensagem *ack*; ou
- p recebe uma estimativa $= \perp$ do seu coordenador, descarta a mensagem e avança para a próxima rodada; ou
- p suspeita do seu coordenador e avança para a próxima rodada.

Se p receber a estimativa do coordenador c, então p lhe enviará uma resposta. Por outro lado, se qualquer uma das outras duas possibilidades ocorrer, então nenhuma resposta será enviada para c. Porém, devido a Tarefa 3 (linhas 57 e 58) e a propriedade *No Loss* do canal de comunicação, em algum momento, p receberá a estimativa de c e lhe enviará uma mensagem *nack*. Portanto, como todo processo correto envia uma resposta ao coordenador c e existe uma maioria de processos corretos (pela hipótese de maioria), então c não bloqueia para sempre na fase 4, o que é uma contradição.

3.5 Algoritmo *Fast Indulgent*

Esta seção apresenta o algoritmo de consenso proposto por P. Dutta e R. Guerraoui [11]. O algoritmo (mostrado na Figura 3.7) é equipado com um detector de falhas Ω e satisfaz as propriedades indulgentes *oracle-efficiency*, *zero-degradation* e *round-zero-degradation*. Cada processo p gerencia as seguintes variáveis:

- r_p: rodada do consenso.
- $est1_p$: estimativa do valor de decisão na fase 1.
- $est2_p$: estimativa do valor de decisão na fase 2.
- $leader_p$: processo confiável de p.

- $state_p$: estado do consenso.

Na fase 1, todo processo p invoca a função *leader*(). Ela retorna a identificação de algum processo considerado estar correto (i.e, um líder). Feito isso, p envia uma mensagem Fase1 com sua estimativa, sua rodada atual e seu líder a todos os processos. Após a espera na fase 1, temos as seguintes possibilidades [11, 14]:

- p recebe uma maioria de mensagens com mesmo líder l, incluindo a mensagem de l. Nesse caso, p adota est_l como valor de $est2$; ou
- p recebe uma maioria de mensagens com líderes diferentes. Nesse caso, p adota $\perp$ como valor de $est2$; ou
- p suspeita do seu líder. Nesse caso, p também adota $\perp$ como valor de $est2$.

Na fase 2, cada processo envia uma mensagem com sua rodada atual e sua estimativa $est2$ (v ou $\perp$) a todos os processos. Após a espera na fase 2, temos as seguintes possibilidades [11, 23]:

```
 1: Tarefa 1:
 2: procedimento propose(v_p)
 3:     est1_p ← v
 4:     r_p ← 0; leader_p ←⊥
 5:     state_p ← undecided
 6:
 7:     enquanto state_p = undecided faça
 8:         r_p ← r_p + 1
 9:
10:         Fase 1:
11:         leader_p ← leader()
12:         send Fase1(r_p, est_p, leader_p) para todos
13:         espere até [(receber Fase1(r_p, ∗, l) do leader_p e de ⌈(n+1)/2⌉ - 1 processos)
            ou (leader_p ≠ leader())]
14:         se [recebeu Fase1(r_p, ∗, l) de ⌈(n+1)/2⌉ processos e Fase1(r_p, est_l, ∗) de p_l]  então
15:             est2_p ← est_l
16:         senão
17:             est2_p ←⊥
18:
19:         Fase 2:
20:         send Fase2(r_p, est2_p) para todos
21:         espere até [receber Fase2(r_p, ∗) de ⌈(n+1)/2⌉ processos]
22:         se [recebeu Fase2(r_p, v) de ⌈(n+1)/2⌉ processos] então
23:             est1_p ← v
24:             R-broadcast (p, r_p, est1_p, decide)
25:         senão se [recebeu Fase2(r_p, v) de qualquer processo] então
26:             est1_p ← v
27:
28: Tarefa 2:
29:  quando R-deliver (q, r_q, est_q, decide) faça
30:     se state_p = undecided então
31:         state_p ← decide
32:         decide (est_q)
```

Figura 3.7: Algoritmo *Fast Indulgent*.

- p recebe v da maioria de processos. Nesse caso, p seleciona v como valor de decisão e o envia a todos os processos usando *Reliable Broadcast*; ou
- p recebe v e $\perp$ da maioria de processos. Nesse caso, p define $est1$ com v; ou
- p recebe $\perp$ da maioria de processos. Nesse caso, p mantém o mesmo valor de quando iniciou o consenso e procede para a próxima rodada.

Capítulo 4

Otimização do algoritmo de Chandra e Toueg

Existem algumas otimizações na literatura projetadas para acelerar a execução dos protocolos de consenso [8, 9, 17, 23]. Com base no nosso estudo, verificamos que as otimizações propostas em [17] e [23] são as mais efetivas no combate aos efeitos negativos causados pelas suspeitas incorretas no algoritmo de Chandra e Toueg.

A primeira é uma melhoria proposta por M. Larrea et al. [17] para acelerar o acordo do protocolo de consenso deles. A idéia básica da melhoria é utilizar um detector de falhas $\Diamond S$ para fazer com que o coordenador da rodada espere pelas mensagens de todos os processos não suspeitos de falha. A vantagem dessa melhoria é que ela aumenta as chances do coordenador alcançar consenso na rodada corrente. Por outro lado, a desvantagem é que ela pode realizar esperas desnecessárias. Para contornar esse problema, nós propomos a otimização *Additional-Waiting*. Na Seção 4.2, discutimos em detalhes como essa otimização funciona.

A segunda é a técnica de otimização *Look-Ahead* proposta por W. Wu et al. [23], que consiste na utilização de mensagens futuras para reduzir tempo de espera desnecessário. Devido ao assincronismo do sistema, mensagens enviadas em rodadas futuras podem ser recebidas antecipadamente por processos mais lentos. Baseado nas informações contidas em tais mensagens, um processo mais lento pode acelerar sua execução. Antes do *Look-Ahead*, D. Dolev et al. [10] utilizou mensagens futuras para ajudar a tratar falhas por omissão de mensagens. Em seu protocolo, todo processo que recebe uma mensagem de uma rodada futura imediatamente salta para a rodada da mensagem. Esse mecanismo é interessante porque permite que o processo se adapte rapidamente ao estado atual do consenso. Entretanto, ele possui algumas desvantagens: (1) ele é projetado especificamente para o protocolo deles, logo, não pode ser facilmente aplicado a outros protocolos; e (2) possíveis decisões podem ser perdidas. Ao contrário desse mecanismo, a técnica *Look-Ahead* é mais eficiente porque pode ser facilmente aplicada a outros protocolos de consenso e pode evitar a perda de possíveis decisões.

Outra maneira de tornar os protocolos de consenso mais eficientes é por meio da utilização de propriedades indulgentes [11, 14]. Para o nosso estudo, não é interessante aplicar essas propriedades ao algoritmo, pois elas são definidas para execuções estáveis. No entanto, a propriedade indulgente *one-step-decision*, definida por R. Guerraoui e M. Raynal [14], nos ajudou a projetar a otimização *Early-Decision*. A propriedade *one-step-decision* foi definida com base no estudo realizado por F. Brasileiro et al. [4]. Ela garante uma decisão para o consenso em um único passo de comunicação em execuções estáveis, quando todos os processos que não falharam inicialmente propõem o mesmo valor. Isso é possível se os processos explorarem um conhecimento inicial sobre um valor

privilegiado. Ao contrário da propriedade, a nossa otimização adquire conhecimento sobre esse valor durante a execução do consenso. Assim, podemos alcançar consenso com um único passo de comunicação em uma determinada rodada $r > 1$.

4.1 Otimização *Early-Decision*

Para alcançar consenso no algoritmo de Chandra e Toueg, o coordenador deve receber uma maioria de mensagens *ack*, ou seja, pelos menos uma maioria de processos precisa ter a mesma estimativa e a mesma estampa de tempo. Se o coordenador receber uma única mensagem *nack*, a rodada atual é abortada. Existem situações onde o coordenador recebe uma mensagem *nack* antes da maioria de mensagens *ack*. Nesse caso, embora não seja possível alcançar uma decisão, observe que existe uma maioria de processos com a mesma estimativa e a mesma estampa de tempo. Assim, o estado atual desses processos corresponde à condição necessária para alcançar consenso. Se o coordenador da próxima rodada receber as mensagens desse processos na fase 2, ele não precisará esperar pela fase 4 para tomar uma decisão. Isso evita que as mensagens nas fases 2 e 3 sejam transmitidas, logo, apenas um passo de comunicação é necessário para alcançar consenso.

A idéia básica da otimização é verificar se uma maioria de processos adotou uma estimativa na mesma rodada anterior. Não basta que os processos tenham a mesma estimativa, eles também devem ter a mesma estampa de tempo para garantir a convergência das estimativas para o valor de decisão. Nós dizemos que uma decisão é alcançada na fase 2 de uma rodada $r > 1$, quando o coordenador recebe uma maioria de mensagens com a mesma estimativa e a mesma estampa de tempo.

A otimização foi projetada, principalmente, para melhorar o desempenho do algoritmo em caso de suspeitas incorretas. Contudo, ela também é útil em situações onde o coordenador falha após propor sua estimativa na fase 2. Nesse caso, nenhuma decisão é alcançada na rodada em que o coordenador falha, porém, existe a possibilidade de uma maioria de processos adotar sua estimativa. Suponha que não exista uma maioria de processos corretos com mesma estimativa e mesma estampa de tempo em uma rodada r. Se o coordenador dessa rodada falhar após propor sua estimativa, então é possível que uma maioria de processos corretos receba a proposta do coordenador. Se isso ocorrer, o coordenador de uma rodada futura pode alcançar consenso na fase 2. Por outro lado, se o coordenador falhar antes de enviar sua proposta, ele será suspeito pelos demais processos, e assim, o consenso não poderá ser alcançado na fase 2 da próxima rodada.

As Figuras 4.1 e 4.2 ilustram, respectivamente, uma execução simples do algoritmo de Chandra e Toueg e uma execução otimizada com *Early-Decision*. Na Figura 4.1, p_1 é o coordenador da rodada $r = 1$. A primeira fase não é necessária nesta rodada. Na fase 2, p_1 seleciona sua própria estimativa e a envia a todos os processos. Na fase 3, p_3 suspeita de seu coordenador e p_1 e p_2 recebem a estimativa do coordenador. Assim, p_3 envia uma mensagem *nack* para p_1, enquanto que, p_1 e p_2 adotam a estimativa do coordenador, atualizam suas estampas de tempo com a rodada atual ($ts_1 = ts_2 = 1$) e enviam uma mensagem *ack* para p_1. Na fase 4, p_1 recebe um *ack* e, em seguida, um *nack*. Nesse caso, o coordenador p_1 avança para a próxima rodada. Na rodada $r = 2$, p_2 é o novo coordenador. Na fase 1, todos os processos enviam suas estimativas e estampas de tempo a p_2. Na fase 2, p_2 recebe a estimativa e estampa de tempo $(est_2, 1)$ e, em seguida, $(est_1, 1)$, e propõe a estimativa com maior estampa de tempo. Na fase 3, todos os processos recebem a estimativa do coordenador e enviam uma mensagem *ack* para p_2. Na fase 4, p_2 recebe uma maioria de mensagens *ack* e alcança uma decisão.

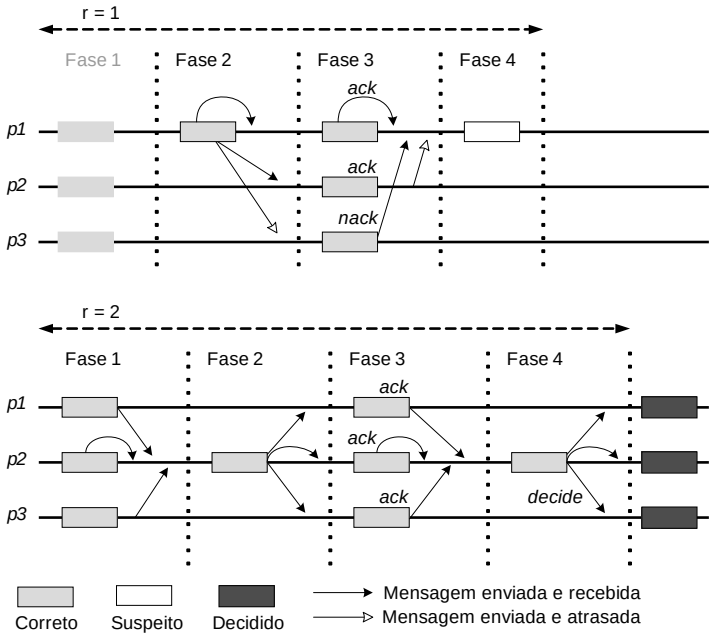


Figura 4.1: Execução simples do algoritmo de Chandra e Toueg.

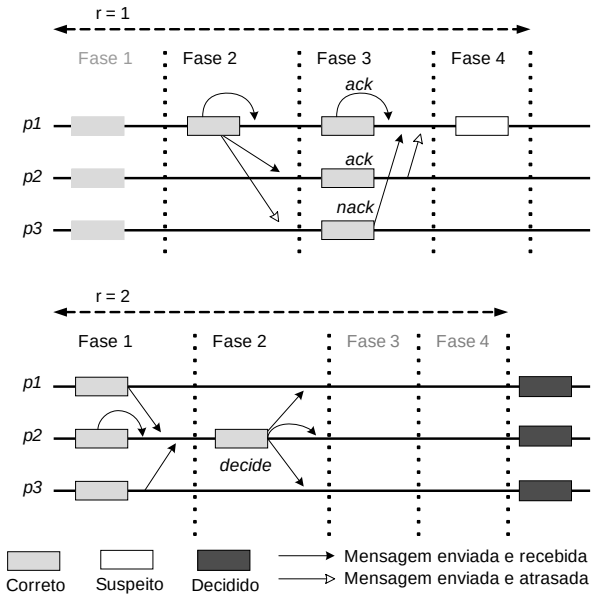


Figura 4.2: Otimização *Early-Decision*.

Na Figura 4.2, a rodada $r = 1$ é exatamente igual a da Figura 4.1. Na rodada $r = 2$, p_2 é o novo coordenador. Na fase 1, todos os processos enviam suas estimativas para p_2. Na fase 2, p_2 recebe a estimativa e a estampa de tempo $(est_2, 1)$ e, em seguida, $(est_1, 1)$. Devido a otimização, como uma maioria de processos possui a mesma estimativa e a mesma estampa de tempo, o processo p_2 alcança uma decisão.

Comparando as Figuras 4.1 e 4.2, nós claramente observamos que o número de passos de comunicação para alcançar consenso na rodada $r = 2$ é reduzido de três para um. A vantagem dessa otimização é que ela é simples e não gera qualquer envio de mensagem adicional.

4.2 Otimização *Additional-Waiting*

A melhoria proposta em [17] se baseia na suposição de que existe mais que uma maioria de processos corretos. Basicamente, ela requer apenas o uso de um detector de falhas da classe $\Diamond S$. A idéia é utilizar as propriedades de completude e precisão do detector para fazer com que o coordenador da rodada espere pelas mensagens de todos os processos não suspeitos de falha. Desse modo, o coordenador pode receber mais que uma maioria de mensagens. Por exemplo, no algoritmo de Chandra e Toueg, o coordenador propõe sua estimativa na fase 2 e espera pelas respostas de uma maioria de processos na fase 4. Se qualquer uma das respostas é uma mensagem *nack*, a decisão não é alcançada. Com a melhoria, o coordenador pode receber mensagens adicionais suficientes para completar a maioria de mensagens *ack*. Assim, ele pode decidir na rodada corrente, mesmo tendo recebido uma mensagem *nack*.

A melhoria aumenta a probabilidade de um coordenador alcançar uma decisão na rodada atual. No entanto, em algumas situações, ela pode causar perda de desempenho devido à realização de esperas desnecessárias. Em uma execução do consenso sem falhas e sem suspeitas, o coordenador precisa receber uma maioria de mensagens *ack* para chegar a um consenso, porém a melhoria faz com que o coordenador espere pelas mensagens de todos os processos. Neste caso, podemos ver que a espera não é útil. Para contornar esse problema, nós propomos a otimização *Additional-Waiting*. A idéia é manter o efeito positivo causado pela melhoria e evitar esperas desnecessárias. Um aspecto importante desta otimização é que ela pode ser aplicada a outros protocolos de consenso, desde que eles também assumam um detector de falhas da classe $\Diamond S$.

Nós aplicamos a otimização nas fases 2 e 4 do algoritmo de Chandra e Toueg. Na fase 2, nós utilizamos as otimizações *Additional-Waiting* e *Early-Decision* juntas. Assim, a otimização *Additional-Waiting* aumenta a chance de satisfazer a otimização *Early-Decision*. Por outro lado, sem a otimização *Early-Decision*, a otimização *Additional-Waiting* na fase 2 não é útil, porque a extensão do tempo de espera não impede a transmissão de mensagens nas fases 2 e 3.

A otimização, em ambas as fases, funciona da seguinte forma: primeiramente, o coordenador espera por uma maioria de mensagens. Se as mensagens não forem suficientes para alcançar uma decisão na fase atual, o coordenador verifica se existem processos não suspeitos que ainda não tiveram suas mensagens recebidas. Nós dizemos que estes processos estão **ativos**. Se o número de processos ativos for suficiente para tentar satisfazer a condição de consenso, então o coordenador realiza a espera adicional. Em outras palavras, a otimização é acionada apenas quando há processos ativos suficientes para tentar completar a maioria de mensagens necessárias para alcançar consenso.

A Figura 4.3 ilustra a combinação das otimizações *Additional-Waiting* e *Early-Decision*. A rodada $r = 1$ é exatamente igual a da Figura 4.1. Na fase 1 da rodada $r = 2$, todos os processos enviam suas estimativas e estampas de tempo ao coordenador p_2 da rodada. Ele recebe a estimativa e a estampa de tempo $(est_2, 1)$ e, em seguida, $(est_3, 0)$. A maioria de estimativas e estampas de tempo recebidas são diferentes, e isso impede o coordenador de alcançar consenso na fase 2. Entretanto, como p_1 está ativo, isto é, não está suspeito e sua mensagem ainda não foi recebida, p_2 realiza a espera adicional. Assim, p_2 recebe uma mensagem adicional com estimativa e estampa de tempo $(est_1, 1)$. Neste caso, como $est_1 = est_2$ e $ts_1 = ts_2$, a condição de consenso pode ser satisfeita na fase 2.

A Figura 4.4 ilustra o comportamento da otimização *Additional-Waiting* na fase 4. O processo p_1 é o coordenador da rodada $r = 1$. Na fase 2, p_1 seleciona sua própria estimativa e envia a todos os processos. Na fase 3, p_3 suspeita de seu coordenador e p_1

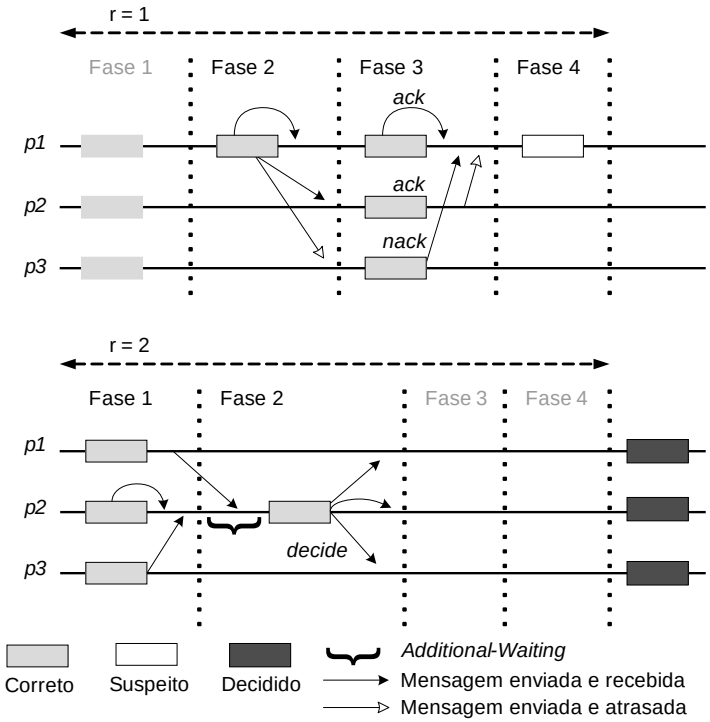


Figura 4.3: Combinação das otimizações *Early-Decision* e *Additional-Waiting*.

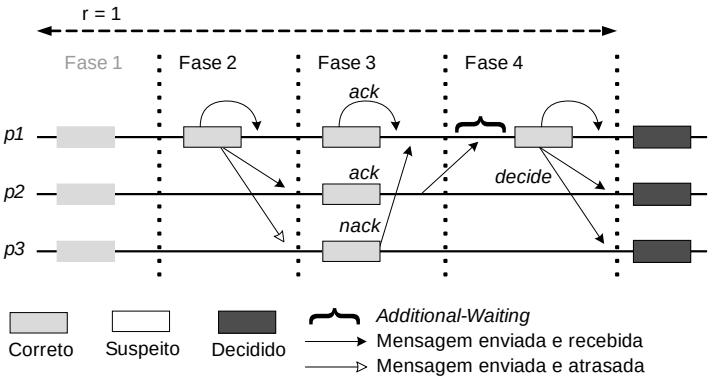


Figura 4.4: Otimização *Additional-Waiting* na fase 4.

e p_2 recebem a estimativa do coordenador. Assim, p_3 envia uma mensagem *nack* a p_1, enquanto que, p_1 e p_2 adotam a estimativa de p_1, atualizam suas estampas de tempo e enviam uma mensagem *ack* para p_1. Na fase 4, p_1 recebe um *ack* e, em seguida, um *nack*. Como p_2 está ativo, p_1 realiza a espera adicional. Graças a otimização, p_1 recebe uma mensagem adicional do tipo *ack*, e, portanto, pode tomar uma decisão.

4.3 Técnica *Look-Ahead*

A técnica *Look-Ahead* utiliza mensagens de rodadas futuras para acelerar a execução do consenso. Devido ao assincronismo do sistema, alguns processos podem ser mais rápidos do que outros e, em um mesmo instante de tempo, processos distintos podem estar em diferentes fases ou rodadas. Assim, uma mensagem que contém informações sobre o "futuro" pode ser utilizada por um processo mais lento para interromper a espera de alguma mensagem atrasada e acelerar sua execução. No entanto, nem todas as mensagens futuras são úteis, algumas delas podem impedir o processo de tomar uma

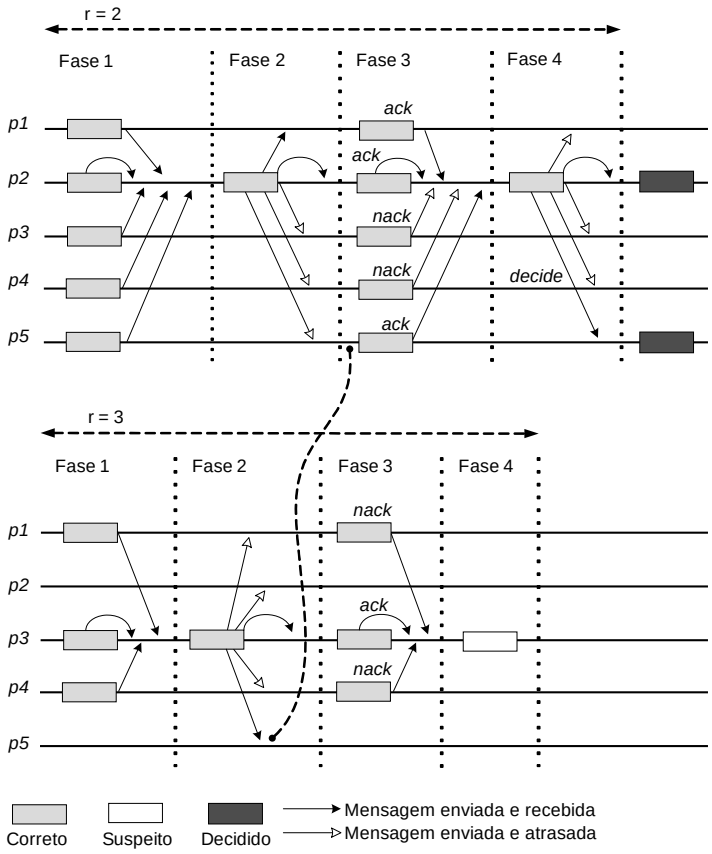


Figura 4.5: Técnica de otimização *Look-Ahead*.

decisão na rodada corrente [23]. W. Wu et al. [23] mostram que a técnica pode ter um impacto positivo ou negativo sobre o desempenho do consenso. Isso depende da escolha das mensagens e do modo como elas são utilizadas.

Em geral, a técnica *Look-Ahead* permite a um processo "olhar" o conteúdo de uma mensagem futura para antecipar a execução de alguma ação, como por exemplo, a interrupção antecipada de uma espera. Entretanto, nós ressaltamos que toda mensagem futura não é descartada, mas sim, armazenada para posteriormente ser processada em sua rodada correspondente. Somente quando isso ocorre é que a mensagem é descartada.

Nós aplicamos a técnica na fase 3 do algoritmo de Chandra e Toueg. Se um processo p, em uma rodada r, recebe a estimativa de um coordenador c' de uma rodada futura $r' > r$, então p interrompe a espera da fase 3, adota a estimativa recebida, atualiza a estampa de tempo com a rodada atual r e envia uma mensagem *ack* para o coordenador c da rodada r. Além de reduzir o tempo de espera de uma mensagem atrasada, a nossa adaptação aumenta a chance de um processo enviar uma mensagem *ack* ao coordenador da rodada e não causa perda de desempenho.

A Figura 4.5 ilustra um cenário onde a técnica melhora o desempenho do algoritmo. Na fase 2, o coordenador p_2 da rodada $r = 2$ propõe a estimativa com maior estampa de tempo a todos os processos. Os processos p_3 e p_4 suspeitam do coordenador p_2 e o enviam uma mensagem *nack*, enquanto o processo p_1 adota a estimativa do coordenador, atualiza sua estampa de tempo ($ts_1 = 2$) e o envia uma mensagem *ack*. Os processos p_1, p_3 e p_4 avançam para a próxima rodada, enquanto que, p_2 e p_5 permanecem na rodada $r = 2$. Na fase 2 da rodada $r = 3$, o coordenador p_3 recebe uma maioria de estimativas e propõe aquela com maior estampa de tempo (est_1) como sua nova estimativa. Em seguida, o coordenador p_3 é suspeito pelos processos p_1 e p_4, o que impede o alcance de

uma decisão. Na rodada $r = 2$, devido ao assincronismo e ao *Look-Ahead*, p_5 recebe a estimativa do processo p_3. Assim, p_5 interrompe a espera pela mensagem atrasada de p_2, atualiza sua estimativa e sua estampa de tempo e envia uma mensagem *ack* a p_2. As mensagens *ack* são recebidas primeiramente por p_2, o que permite o alcance de uma decisão na rodada $r = 2$. Neste exemplo, o tempo de espera do processo p_5 na fase 3 é reduzido graças a técnica *Look-Ahead*. Além disso, embora o processo p_5 não tenha recebido a estimativa do coordenador p_2, a técnica permitiu que o processo enviasse uma mensagem *ack* para p_2, o que garantiu uma decisão para o consenso.

4.4 Descrição do algoritmo

Nesta seção, nós apresentamos o algoritmo de Chandra e Toueg equipado com as otimizações discutidas anteriormente. O algoritmo é mostrado na Figura 4.6. A fase 1 não foi modificada. Na fase 2, o coordenador da rodada $r = 1$ sempre propõe sua própria estimativa a todos os processos. O coordenador c de uma rodada $r > 1$ espera até receber uma maioria de estimativas. Após a espera, temos as seguintes possibilidades:

- c recebe uma maioria de mensagens com a mesma estimativa v e a mesma estampa de tempo t. Portanto, c seleciona v como valor de decisão e o envia a todos os processos usando *Reliable Broadcast*;
- c recebe uma maioria de mensagens $(q, r_p, *, *)$. Nesse caso, existem duas possibilidades:
  - Existem processos ativos suficientes para tentar satisfazer a condição de consenso. Desse modo, para cada processo ativo q, c recebe a estimativa de q ou suspeita dele. Em seguida, c retorna à linha 17; ou
  - Não existem processos ativos suficientes para tentar satisfazer a condição de consenso. Desse modo, c seleciona a estimativa com maior estampa de tempo e a envia a todos os processos como sua nova estimativa.

Na fase 3, existem três possibilidades para cada processo p:

- p adota a estimativa de c, atualiza sua estampa de tempo e envia uma mensagem *ack* para c; ou
- p adota a estimativa proposta por um coordenador c' de uma rodada futura, atualiza sua estampa de tempo e envia uma mensagem *ack* para c. Nesse ponto, a estimativa de c' é armazenada até que o processo p alcance a rodada de c' e o envie uma mensagem *ack*; ou
- p suspeita de c e o envia uma mensagem *nack*.

Na fase 4, o coordenador espera até receber uma maioria de respostas. Após a espera, temos as seguintes situações:

- c recebe uma maioria de mensagens *ack*. Portanto, o valor proposto se torna o valor de decisão. Em seguida, o coordenador envia o valor de decisão a todos os processos usando *Reliable Broadcast*;

```
1:  procedimento propose(v_p)
2:      est_p ← v_p
3:      state_p ← undecided
4:      r_p ← 0; ts_p ← 0
5:      enquanto state_p = undecided faça
6:          c_p ← (r_p mod n) + 1
7:          r_p ← r_p + 1
8:
9:          Fase 1:
10:         se r_p > 1 então
11:             send (p, r_p, est_p, ts_p) para c_p
12:
13:         Fase 2:
14:         se p = c_p então
15:             se r_p > 1 então
16:                 espere até [receber (p, r_p, est_p, ts_p) de ⌈(n+1)/2⌉ processos]
17:                 se [recebeu (p, r_p, v, t) de ⌈(n+1)/2⌉ processos] então        ▷ Early-Decision
18:                     est_p ← v; R-broadcast (p, r_p, est_p, decide)
19:                 senão se [∃ processos ativos suficientes] então                 ▷ Additional-Waiting
20:                     espere até [para cada processo ativo q: receber (q, r_p, ∗, ∗) ou q ∈ D.suspected_p]
21:                     retorne à linha 17
22:                 senão
23:                     est_p ← est_q com maior ts_q
24:                     send (p, r_p, est_p) para todos
25:             senão
26:                 send (p, r_p, est_p) para todos
27:
28:         Fase 3:
29:         espere até [receber (c_p, r_p, est_{c_p}) de c_p ou c_p ∈ D.suspected_p
            ou (∗, > r_p, est_{c'}, ∗) de algum coordenador c'] ▷ Look-Ahead
30:         se [c_p ∈ D.suspected_p] então
31:             send (p, r_p, nack) para c_p
32:         senão
33:             se [recebeu (c_p, r_p, est_{c_p}) de c_p] então
34:                 est_p ← est_{c_p}
35:             senão
36:                 est_p ← est_{c'}                                               ▷ Look-Ahead
37:             ts_p ← r_p
38:             send (p, r_p, ack) para c_p
39:
40:         Fase 4:
41:         se p = c_p então
42:             espere até [recebeu (q, r_p, ack) ou (q, r_p, nack) de ⌈(n+1)/2⌉ processos]
43:             se [recebeu (q, r_p, ack) de ⌈(n+1)/2⌉ processos] então
44:                 R-broadcast (p, r_p, est_p, decide)
45:             senão se [∃ processos ativos suficientes] então                    ▷ Additional-Waiting
46:                 espere até [para cada processo ativo q: receber (q, r_p, ∗) ou q ∈ D.suspected_p]
47:                 retorne à linha 43
48:
49: quando R-deliver (q, r_q, est_q, decide)
50: se state_p = undecided então
51:     state_p ← decide
52:     decide (est_q)
```

Figura 4.6: Algoritmo de Chandra e Toueg otimizado.

- c não recebe uma maioria de mensagens *ack*. Nesse caso, existem duas possibilidades:
  - Existem processos ativos suficientes para tentar satisfazer a condição de consenso. Desse modo, para cada processo ativo q, c recebe uma mensagem *ack* ou *nack* de q ou suspeita dele. Em seguida, c retorna à linha 43; ou
  - Não existem processos ativos suficientes para tentar satisfazer a condição de consenso. Desse modo, c avança para a primeira fase da próxima rodada.

A qualquer momento, se um processo executa um *R-Deliver* de uma decisão (linhas 49-52), então ele decide.

4.5 Prova de corretude do algoritmo

Nesta seção, nós provamos que as propriedades do consenso não são violadas pelas otimizações propostas.

Lema 1 *Se nenhum processo decide em qualquer rodada $r' \leq r$, então todos os processos corretos iniciam a rodada $r + 1$.*

Prova. Prova por contradição. Seja r' a menor rodada onde alguns processos corretos bloqueiam para sempre em alguma sentença de espera. Na fase 2, nós devemos considerar dois casos:

(1) Se $r' = 1$, então o coordenador atual c' não espera na fase 2, logo, ele não bloqueia para sempre em qualquer sentença de espera na fase 2.

(2) Se $r' > 1$, então todos os processos corretos alcançam o final da fase 1 da rodada r' e enviam uma mensagem do tipo $(*, r', est, *)$ para c'. Já que uma maioria de processos estão corretos, pelo menos $\lceil (n+1)/2 \rceil$ mensagens são enviadas para c'. Devemos considerar dois casos:

   (a) Em algum momento, c' recebe $\lceil (n+1)/2 \rceil$ estimativas e envia sua estimativa $(c', r', est_{c'})$ a todos os processos. Se o coordenador c' esperar pelas estimativas de todo processo ativo q, então, como os conjuntos $D.suspected_p$ satisfazem a propriedade de completude forte, c' receberá a estimativa de q ou suspeitará de q em algum momento. Portanto, c' não bloqueia para sempre em qualquer sentença de espera na fase 2.

   (b) c' falha.

   No primeiro caso, todo processo correto recebe $(c', r', est_{c'})$ em algum momento. No segundo caso, como os conjuntos $D.suspected_p$ satisfazem a propriedade de completude forte, todo processo correto p suspeita, permanentemente, de c' após um certo período de tempo. Assim, em ambos os casos, nenhum processo correto bloqueia para sempre em qualquer sentença de espera na fase 3. Logo, todo processo correto envia uma mensagem do tipo $(*, r', ack)$ ou $(*, r', nack)$ para c' na fase 3. Como existe pelo menos um maioria de processos corretos então, em algum momento, c' receberá $\lceil (n+1)/2 \rceil$ mensagens *ack* ou *nack*. Se c' esperar pelas mensagens de todo processo ativo q, então, c' receberá a mensagem de q ou suspeitará de q em algum momento. Portanto, c' não bloqueia para sempre em qualquer sentença de espera na fase 4. Isto mostra que todos os processos corretos completam a rodada r' − uma contradição que completa a prova do Lema 1.

Teorema 1 (Terminação) *Todo processo correto decide algum valor v em algum momento.*

Prova. Temos que analisar dois possíveis casos:

(1) Algum processo correto decide. Logo, ele deve ter executado um *R-deliver* de alguma mensagem do tipo $(*, *, *, decide)$. Pela propriedade de acordo do *Reliable Broadcast*, em algum momento, todos os processos corretos executam um *R-deliver* desta mensagem e decidem.

(2) Nenhum processo correto decide. Prova por contradição.

Pela propriedade de precisão terminal fraca, existe um processo correto q e um instante t, tal que, nenhum processo correto suspeita de q após t. Seja $t' \geq t$ o instante de tempo onde todos os processos defeituosos falham. Observe que nenhum processo suspeita de q após o instante t'. A partir disso e do Lema 1, existe uma rodada r onde:

(a) Todo processo correto alcança a rodada r após o instante t' (quando nenhum processo suspeita de q).

(b) q é o coordenador da rodada r.

Na fase 1 da rodada r, todos os processos corretos enviam suas estimativas para q. Na fase 2, q recebe $\lceil (n+1)/2 \rceil$ estimativas. Se q não receber uma maioria de mensagens $(*, *, v, t)$, então q envia (q, r, est_q) para os processos. Na fase 3, como q não é suspeito por qualquer processo correto após o instante t, todo processo correto espera pela estimativa de q e, em algum momento, a recebe e envia uma mensagem *ack* para q. Além disso, nenhum processo envia um *nack* para q (isso acontece apenas quando um processo suspeita de q). Assim, na fase 4, q recebe $\lceil (n+1)/2 \rceil$ mensagens do tipo $(*, r, ack)$ e nenhuma mensagem do tipo $(*, r, nack)$. Portanto, q executa um *R-broadcast*$(q, r, est_q, decide)$. Pelas propriedades de validade e acordo do *Reliable Broadcast*, em algum momento, todos os processos corretos executam um *R-deliver* da mensagem de q e decidem, o que é uma contradição. Assim, o caso (2) é impossível e isto conclui a prova do Teorema 1.

Teorema 2 (Integridade uniforme) *Todo processo decide no máximo uma vez.*

Prova. Segue diretamente do algoritmo (mostrado na Figura 4.6) nas linhas (49-52), onde nenhum processo decide mais que uma vez.

Teorema 3 (Acordo uniforme) *Dois processos não decidem diferentemente.*

Prova. Se nenhum processo decide, o teorema é trivialmente verdadeiro. Se algum processo decide, ele deve ter executado um *R-deliver* de uma mensagem do tipo $(*, *, *, decide)$. Pela propriedade de integridade uniforme do *Reliable Broadcast* e pelo algoritmo, um coordenador executou um *R-broadcast* desta mensagem anteriormente. Desse modo, $\lceil (n+1)/2 \rceil$ mensagens do tipo $(*, *, ack)$ foram enviadas em alguma rodada. Seja r a menor rodada onde $\lceil (n+1)/2 \rceil$ mensagens do tipo $(*, r, ack)$ são enviadas a um coordenador. Seja c o coordenador da rodada r e est_c a estimativa de c no fim da fase 2 da rodada r. Nós afirmamos que para toda rodada $r' \geq r$, se um coordenador c' (1) propor ou (2) decidir $est_{c'}$ na fase 2 da rodada r', então $est_{c'} = est_c$.

Prova por indução no número da rodada. A afirmação é trivialmente verdadeira para $r' = r$. Nós assumimos que a afirmação se mantém para toda rodada r', $r \leq r' < k$. Seja c_k o coordenador da rodada k. Nós mostraremos que a afirmação se mantém para $r' = k$, ou seja, se c_k (1) propor ou (2) decidir est_{c_k} na fase 2 da rodada k, então $est_{c_k} = est_c$.

Pelo algoritmo (mostrado na Figura 4.6), se c_k (1) propor ou (2) decidir est_{c_k} na fase 2 da rodada k, então ele deve ter recebido estimativas de pelo menos $\lceil (n+1)/2 \rceil$ processos. Assim, no máximo $\lfloor (n-1)/2 \rfloor$ processos podem estar na rodada r. Como r é a menor rodada onde $\lceil (n+1)/2 \rceil$ mensagens do tipo $(*, r, ack)$ são enviadas ao coordenador c, então existe algum processo p, tal que, (1) p enviou uma mensagem do tipo $(*, r, ack)$ para c na fase 3 da rodada r e (2) (p, k, est_p, ts_p) é recebido por c_k na fase 2 da rodada k.

Como p enviou $(p,\ r,\ ack)$ para c na fase 3 da rodada r, $ts_p = r$ ao fim da fase 3 da rodada r. Como ts_p é crescente, $ts_p \geq r$ na fase 1 da rodada k. Assim, na fase 2 da rodada k, (p, k, est_p, ts_p) é recebido por c_k com $ts_p \geq r$. É fácil observar que c_k não recebe nenhuma mensagem $(q,\ k,\ est_q,\ ts_q)$ com $ts_q \geq k$. Seja t o maior ts_q recebido por c_k na fase 2. Logo, $r \leq t < k$.

(1) Se c_k envia est_{c_k} então c_k executa $est_{c_k} \leftarrow est_q$, pois (q, k, est_q, t) foi recebido por c_k. Como $r \leq t < k$, pela hipótese de indução, $est_q = est_c$. Assim, c_k define $est_{c_k} \leftarrow est_c$ na fase 2 da rodada k.

(2) Se c_k decide est_{c_k} então c_k executa $est_{c_k} \leftarrow est_q$. Como t é o maior ts_q recebido por c_k, então a maioria de estimativas e estampas de tempo recebidas devem ser iguais a est_q e t. Como $r \leq t < k$, pela hipótese de indução, $est_q = est_c$. Assim, c_k define $est_{c_k} \leftarrow est_c$ na fase 2 da rodada k.

Isto conclui a prova da afirmação.

Agora nós mostramos que se um processo decide um valor, então ele decide est_c. Suponha que algum processo p execute um *R-deliver* da mensagem $(q, r_q, est_q, decide)$ e, assim, decida est_q. Pela propriedade de integridade uniforme do *Reliable Broadcast* e pelo algoritmo, o processos q deve ter executado um *R-broadcast* da mensagem $(q, r_q, est_q, decide)$ nas fases 2 ou 4 da rodada r_q.

(1) Se a decisão é enviada na fase 2, então, pela definição de r, $r \leq r_q$, e da afirmação acima, $est_q = est_c$.

(2) Se a decisão é enviada na fase 4, então q deve ter recebido $\lceil (n+1)/2 \rceil$ mensagens do tipo $(*, r_q, ack)$ na fase 4 da rodada r_q. Neste caso, q envia est_q na fase 2 da rodada r_q. Pela definição de r, $r \leq r_q$, e da afirmação acima, $est_q = est_c$.

Teorema 4 (Validade uniforme) *Se um processo decide um valor v, então v foi proposto por algum processo.*

Prova. Do algoritmo (mostrado na Figura 4.6), todas as estimativas que um coordenador recebe na fase 2 são valores propostos. Portanto, o valor de decisão que um coordenador seleciona, certamente, foi proposto por algum processo. Assim, a validade uniforme do consenso também é satisfeita.

Capítulo 5

Avaliação de desempenho

Neste capítulo, nós discutimos o modelo de avaliação adotado na realização de nossos experimentos e, também, alguns detalhes a respeito da implementação dos algoritmos. Finalmente, apresentamos os resultados obtidos de todas as simulações realizadas.

5.1 Modelo de avaliação de desempenho

Os algoritmos são avaliados em um sistema onde os processos enviam mensagens de *atomic broadcast* uns aos outros. Nós estudamos o sistema utilizando simulação. A vantagem de se utilizar um sistema simulado é que podemos avaliar o desempenho dos algoritmos em diversos cenários que podem ocorrer em sistemas reais.

Nosso estudo se baseia na avaliação de desempenho realizada por P. Urbán et al. [22], o qual é composta por: métrica de desempenho, carga de trabalho[1], carga de falhas[2], ambiente de execução, modelagem dos detectores de falhas e modelagem do oráculo de eleição de líder. Os experimentos foram realizados utilizando o *framework* Neko [21] em uma estação de trabalho com Sistema Operacional Ubuntu 9.10 (Kernel Linux 2.6.31-22-generic), processador Intel(R) Xeon(R) série 7500 com 8-núcleos de processamento (2.00GHz) e 16 GB de memória RAM.

Nós realizamos uma série de simulações para comparar o desempenho de alguns algoritmos de consenso, e para avaliar os efeitos das otimizações sobre o algoritmo de Chandra e Toueg. Nós avaliamos o ganho de desempenho de cada otimização, comparando os resultados com e sem otimização. Assim, foi possível verificar em quais situações cada uma é melhor. Em seguida, analisamos a combinação das otimizações e comparamos o algoritmo otimizado (mostrado na Figura 4.6) com o algoritmo de melhor desempenho obtido de nossa avaliação, o algoritmo de Paxos [3, 15, 16].

5.1.1 Métrica de desempenho

A métrica de avaliação usada é a latência mínima[3] do *atomic broadcast*. Para cada mensagem de *atomic broadcast* é definida uma latência L da seguinte forma: suponha que um processo p_i execute um $A\text{-}broadcast(m)$ em um determinado instante t_0 e que p_{i+1} execute um $A\text{-}deliver(m)$ em um determinado instante t_i, onde $i = 1, ..., n$. A latência é definida como sendo o tempo entre um $A\text{-}broadcast(m)$ e o primeiro $A\text{-}deliver(m)$, ou seja, $L_{min} = (\min_{i=1,...,n} t_i)$ - t_0. Cada ponto nos gráficos (mostrado na

[1]Do inglês: *Workload*.
[2]Do inglês: *Faultload*.
[3]Do inglês: *Early latency*.

Seção 5.3) corresponde a média da latência mínima, computada sobre várias execuções de *atomic broadcast* [22]. O número de execuções de *atomic broadcast* e de consenso é determinado pelo tempo de duração da simulação, que é igual a 10^5 ms. No geral, foram realizadas aproximadamente de 100 a 5000 execuções de consenso em cada simulação.

5.1.2 Carga de trabalho e carga de falhas

A latência é sempre medida sob uma certa carga de trabalho. A carga de trabalho escolhida é simples: (1) para todos os destinos, um processo envia mensagens de *atomic broadcast* a uma taxa constante e (2) o evento *A-broadcast*(m) é executado por um processo estocástico *Poisson* [19]. A taxa global de mensagens de *atomic broadcast* é conhecida como vazão[4] e denotada por T [22]. A vazão é utilizada para impor diferentes cargas de trabalho ao sistema.

A carga de falhas define os eventos relacionados à quebra de processos que ocorrem durante a simulação. Em [22] são descritas quatro diferentes cargas de falhas. Nosso estudo visa analisar apenas a carga responsável pela geração de suspeitas incorretas. Nesse cenário, nenhuma falha ocorre, porém, os detectores de falhas geram suspeitas incorretas, e o oráculo de eleição de líder retorna líderes distintos para cada processo. Os parâmetros que influenciam a latência são o algoritmo A, a quantidade de processos n, a vazão T e outros dois parâmetros que definem, respectivamente, a frequência com que as suspeitas incorretas ocorrem e a duração de cada uma delas. Esses parâmetros são discutidos na próxima seção.

5.1.3 Modelagem dos detectores falhas e do oráculo de eleição de líder

Os eventos relacionados aos detectores de falhas são simulados. Essa simulação se baseia no conceito de qualidade de serviço[5] dos detectores de falhas, introduzido em [7]. Os autores identificaram duas métricas primárias para modelar a ocorrência de suspeitas incorretas, e uma métrica adicional para definir como essas métricas se relacionam:

- Tempo de recorrência de erros[6] (T_{MR}): período de tempo entre dois erros consecutivos, calculado com base na diferença entre o início do segundo e do primeiro erro (ver Figura 5.1).
- Tempo de duração dos erros[7] (T_M): período de tempo que um detector de falhas leva para corrigir uma suspeita incorreta (ver Figura 5.1).
- Duração do período bom[8] (T_G): período de tempo em que o detector considera um processo confiável (ver Figura 5.1).

Essas métricas são variáveis aleatórias e exponencialmente distribuídas [7], definidas sobre um par de processos. A expressão $T_{MR} = T_M + T_G$ define como essas métricas estão relacionadas. Na prática, as suspeitas incorretas ocorrem devido a *time-outs* prematuros.

[4]Do inglês: *Throughput.*
[5]Do inglês: *Quality of Service* (QoS).
[6]Do inglês: *Mistake recurrence time.*
[7]Do inglês: *Mistake duration.*
[8]Do inglês: *Good period duration.*

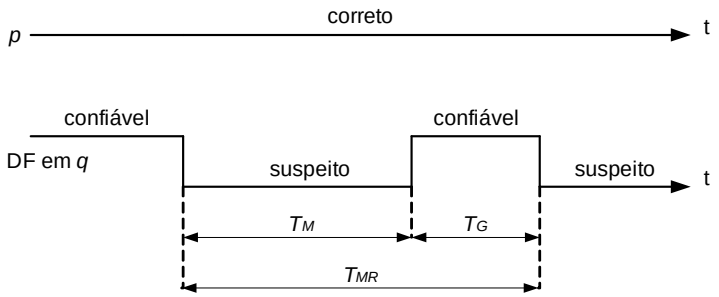


Figura 5.1: Métricas de QoS para suspeitas incorretas. Processo q monitora p.

O modelo adotado desconsidera as mensagens que são enviadas pelo detector de falhas. Essa desconsideração não influencia no desempenho dos algoritmos de consenso, visto que, todo algoritmo é afetado pelo custo causado pelas mensagens do detector.

Com relação à modelagem do oráculo com capacidade de eleição de líder, nós utilizamos uma abordagem bastante simples. O modelo corresponde a uma transformação do detector de falhas $\Diamond S$ a um detector de falhas Ω. O oráculo retorna como líder o processo com menor índice do conjunto de processos confiáveis (processos que não estão suspeitos pelo detector de falhas $\Diamond S$) [22].

5.1.4 Ambiente de execução

O ambiente de execução define quais os fatores envolvidos na transmissão de mensagens. Nosso estudo levou em consideração dois ambientes. Um deles se baseia na contenção de recursos [20]. Esse conceito é importante porque influencia diretamente no desempenho dos algoritmos distribuídos. Basicamente, existem dois tipos de recursos: o de rede (compartilhado por todos os processos), e o de CPU (um para cada processo). O recurso de rede representa o meio de transmissão de mensagens, e o recurso de CPU representa o processamento realizado pelos controladores de rede e pela camada de rede durante a emissão e a recepção de uma mensagem.

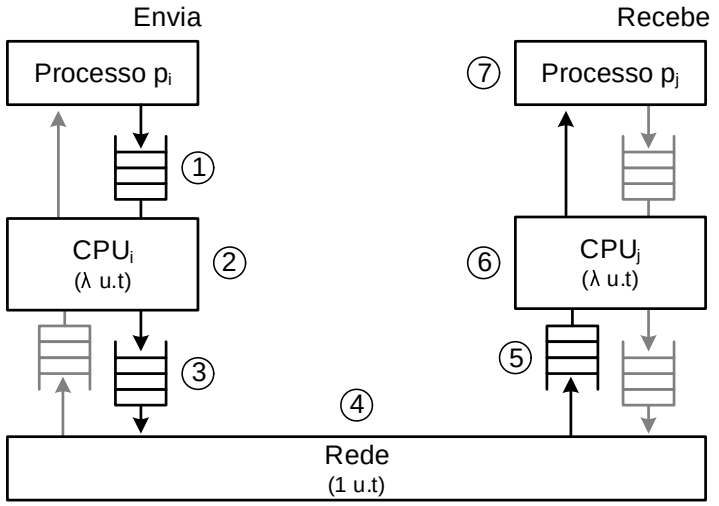


Figura 5.2: Modelo de transmissão de mensagens utilizando contenção de recursos.

O modelo de transmissão de mensagens para esse ambiente é ilustrado na Figura 5.2. O tempo gasto no recurso de rede corresponde a 1 unidade de tempo (u.t), enquanto que, o tempo gasto em cada recurso de CPU corresponde a λ unidades de tempo [20, 22].

A transmissão de uma mensagem m, de um processo p_i para um processo p_j, ocorre da seguinte forma:

1. Após ser colocada na fila de envio do processo emissor p_i, m aguarda a CPU_i estar disponível para ser processada.
2. m utiliza o recurso de CPU_i por λ unidades de tempo e, em seguida, é colocada na fila de rede do processo emissor p_i.
3. Já na fila de rede do processo emissor, m aguarda a rede estar disponível para ser transmitida.
4. m utiliza o recurso de rede por 1 unidade de tempo e, em seguida, é colocada na fila de recebimento do processo destino p_j.
5. Já na fila de recebimento do processo destino, m aguarda a CPU_j estar disponível para ser processada.
6. m utiliza o recurso de CPU_j por λ unidades de tempo e, em seguida, é entregue ao processo destino p_j.
7. p_j recebe a mensagem m.

A Figura 5.4 ilustra um exemplo de como funciona a troca de mensagens entre três processos (p_1, p_2 e p_3) usando a configuração $\lambda = 0{,}5$. O acesso à rede é modelado pela política *round-robin* [20], ilustrado pelo algoritmo da Figura 5.3. Inicialmente, o processo p_1 envia uma mensagem m_1 para os processos p_2 e p_3. A mensagem $m_{1,2}$ é a primeira a ser processada pela CPU_1, e é transmitida no instante $t = 0{,}5$. Após a transmissão, finalizada no instante $t = 1{,}5$, existem duas mensagens ($m_{1,3}$, $m_{3,1}$) prontas (já processadas por suas respectivas CPUs) para acessar o recurso de rede. De acordo com a política de acesso à rede, o processo p_3 é o próximo a usar o recurso. As mensagens $m_{1,2}$ e $m_{3,1}$ são recebidas pelos processos p_2 e p_1, respectivamente, nos instantes $t = 2$ e $t = 3$. As demais transmissões seguem as mesmas etapas.

1: $i \leftarrow 1$
2: **espere até** existir alguma fila de rede não vazia
3: **enquanto** a fila de rede da CPU_i estiver vazia **faça**
4: $\quad i \leftarrow (i \bmod n) + 1$
5: $m \leftarrow$ extrair a primeira mensagem da fila de rede da CPU_i
6: **espere** 1 unidade de tempo
7: inserir m na fila de recebimento da CPU_{dest}
8: $i \leftarrow (i \bmod n) + 1$

Figura 5.3: Política de acesso à rede (executado pela rede).

Esse ambiente foi projetado para se comportar de uma maneira mais determinística. A desvantagem é que ele gera pouco assincronismo, e isso limita a nossa capacidade de avaliar a técnica *Look-Ahead*. Por essa razão, nós também optamos em utilizar um ambiente de rede mais abstrato, fornecido pelo *framework* Neko [21], que nos permite manipular explicitamente os atrasos de mensagen usando um parâmetro β exponencialmente distribuído. Por meio desse parâmetro, nós podemos impor ao sistema períodos assíncronos suficientes para analisar a técnica. Embora seja um modelo, particularmente, não realístico em termos de desempenho real, esse ambiente "exercita" mais os

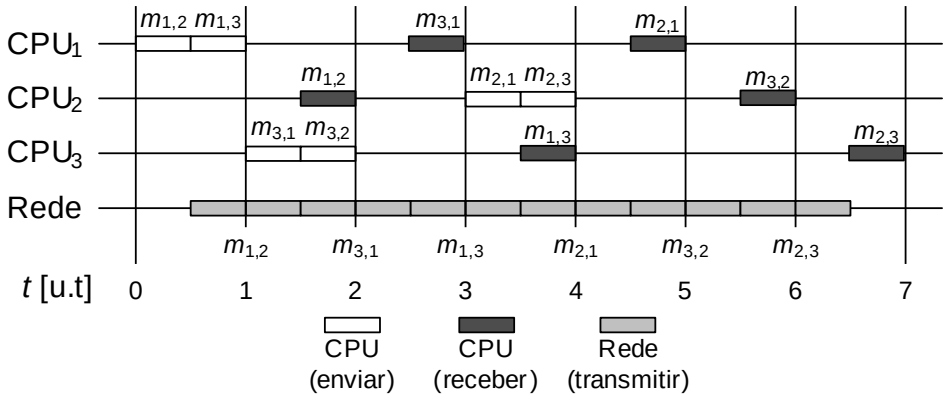


Figura 5.4: Exemplo de transmissão de mensagens para $\lambda = 0{,}5$.

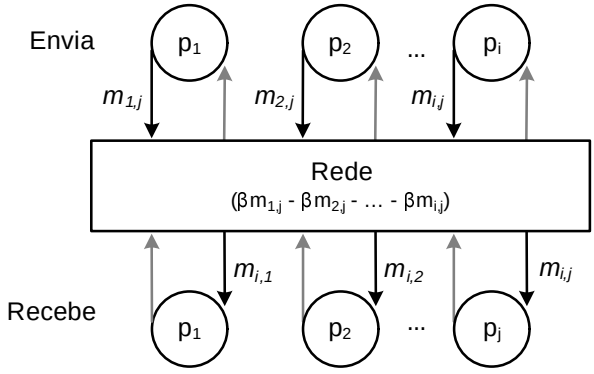


Figura 5.5: Modelo de transmissão baseado em atrasos de mensagens.

algoritmos distribuídos do que o primeiro ambiente discutido. A Figura 5.5 ilustra o modelo de transmissão.

Como podemos observar, o modelo é considerado ser mais abstrato pelo fato de englobar os tempos gastos na CPU, nas filas de espera e na transmissão de uma mensagem. Além disso, essa abstração se deve a vários outros aspectos que podem ser considerados implicitamente nesse modelo, como por exemplo, rotas distintas com diferentes fluxos de mensagens, enlaces com diferentes larguras de banda, diferentes tipos de rede, entre outros. Esses fatores não são considerados no primeiro ambiente, e são responsáveis por gerar assincronismo em uma rede real.

A Figura 5.6 ilustra um exemplo de como funciona a troca de mensagens entre três processos usando a configuração $\beta = 1{,}5$. Nesse ambiente, o tempo que um processo gasta para enviar duas mensagens consecutivas de um mesmo *broadcast* é pequeno, e por isso, nós o desconsideramos no exemplo. Cada mensagem enviada possui um determinado atraso. Por exemplo, a mensagem $m_{2,1}$ gasta 2 u.t para ser transmitida, enquanto que, a mensagem $m_{2,3}$ gasta apenas 0,5 u.t.

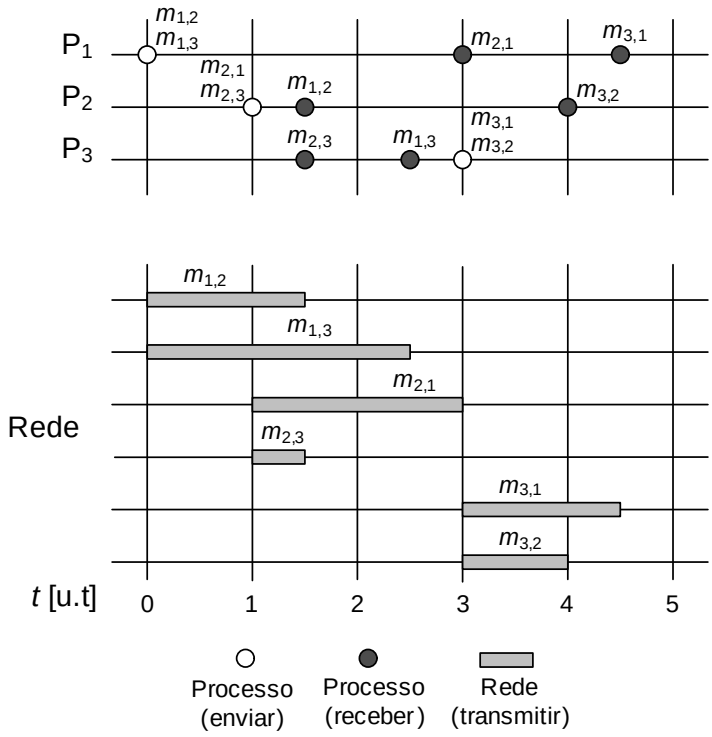


Figura 5.6: Exemplo de transmissão de mensagens para $\beta = 1{,}5$.

5.2 Detalhes de implementação

Todos os experimentos apresentados nesse trabalho foram realizados com ajuda do *framework* de simulação e prototipação Neko [21]. O Neko é uma ferramenta baseada na plataforma Java, que oferece um ambiente robusto e extensível para projetar e avaliar o desempenho de algoritmos distribuídos. Todo o modelo de avaliação de desempenho que utilizamos é fornecido pelo *framework*. Em relação aos algoritmos de consenso, apenas os mais comuns (Paxos e Chandra e Toueg) são fornecidos pelo Neko. Os demais tiveram que ser implementados.

Um aspecto importante do trabalho, e do próprio *framework*, é que nenhum novo módulo, com exceção dos algoritmos de consenso, teve que ser criado no Neko para conduzir as simulações. Entretanto, nós nos deparamos com diversos erros de implementação existentes no *framework* (versão mais atual), que felizmente puderam ser corrigidos.

5.3 Resultados

Nesta seção, nós apresentamos os resultados das simulações. Por meio dos parâmetros λ e β, nós realizamos diversas simulações com uma variedade de ambientes de rede. Os valores adotados para λ são: 0,1, 10 e 1. As configurações $\lambda = 0{,}1$ ms e $\lambda = 10$ ms correspondem, respectivamente, a sistemas onde a comunicação gera maior contenção na rede e na CPU, enquanto que, $\lambda = 1$ ms corresponde a uma configuração de rede intermediária. Nós discutimos, principalmente, os resultados referentes a configuração $\lambda = 1$ ms, visto que, o tempo gasto nos recursos de rede e CPU são iguais e, por isso, não interferem no comportamento dos algoritmos. Em relação ao parâmetro β, nós

consideramos um atraso médio igual a 5 ms (valor definido com base no estudo de W. Wu et al. [23]).

As simulações envolvem 3 e 7 processos, e são realizadas sob as seguintes cargas de trabalho: $T = 10$ (1/s) e $T = 50$ (1/s). Os resultados para a configuração $\lambda = 10$ ms e $T = 50$ (1/s) não são apresentados, pois a primeira invocação do consenso não pôde ser finalizada no tempo de duração da simulação.

Os gráficos mostram a média da latência mínima em função do tempo de recorrência de erros. O tempo de duração dos erros T_M é fixado, e assume dois possíveis valores: 10 e 100 ms. Nós discutimos apenas os resultados para a configuração $T_M = 10$ ms. Os resultados para a configuração $T_M = 100$ ms são similares[9], e podem ser consultados no Apêndice A do trabalho. O intervalo de confiança de 95% é mostrado para cada ponto no gráfico. Em alguns gráficos o intervalo é imperceptível. As Tabelas 5.1 e 5.2 mostram, respectivamente, como os algoritmos e as otimizações são representados nos gráficos. O algoritmo CTO corresponde ao algoritmo apresentado na Seção 4.4.

CT	Algoritmo de Chandra e Toueg
CTO	Algoritmo de Chandra e Toueg otimizado
MR	Algoritmo de Mostefaoui e Raynal
Paxos	Algoritmo de Paxos
DC	Algoritmo baseado na classe $\Diamond C$
FI	Algoritmo *Fast Indulgent*

Tabela 5.1: Representação dos algoritmos de consenso nos gráficos.

ED	*Early-Decision*
AW2	*Additional-Waiting* na fase 2
AW4	*Additional-Waiting* na fase 4
AW2/4	*Additional-Waiting* nas fases 2 e 4
LA	*Look-Ahead*

Tabela 5.2: Representação das otimizações nos gráficos.

5.3.1 Avaliação dos algoritmos de consenso

Nesta seção, nós avaliamos os resultados obtidos da comparação entre os algoritmos estudados nesse trabalho. O objetivo é determinar o algoritmo de consenso mais eficiente, e compará-lo com o algoritmo de Chandra e Toueg otimizado. Essa avaliação também visa identificar as características principais de cada algoritmo, de modo que possamos compreender suas diferenças. Por essa razão, os algoritmos são avaliados em sua forma clássica (sem otimizações). Antes de discutirmos os resultados, nós identificamos os fatores que mais afetam o desempenho dos algoritmos, que são:

- Mecanismo de eleição de líder;
- Número de passos de comunicação por rodada; e
- Número de mensagens enviadas por rodada.

[9] O aumento do T_M implicou também no aumento do T_{MR}.

Dos algoritmos estudados, aqueles com maior número de mensagens são os que possuem menor número de passos de comunicação. Embora eles transmitam muitas mensagens, a vantagem de se ter poucos passos de comunicação é que, dependendo do comportamento da rede, o consenso pode ser alcançado mais rapidamente. Por exemplo, o algoritmo MR precisa de apenas dois passos de comunicação para alcançar consenso. Porém, a complexidade em termos de mensagens na fase 2 é alta, $O(n^2)$. Apesar disso, nem todas as mensagens precisam ser transmitidas para que um processo decida na fase 2.

Com base nesses fatores, nós avaliamos a latência dos algoritmos em dois momentos: no estado estável e no estado variável. O foco do nosso estudo é analisar o estado variável da latência, que é onde ocorre o maior número de suspeitas incorretas. Entretanto, nós também avaliamos o estado onde o número de suspeitas incorretas é baixo.

Para ajudar a nossa análise, a Tabela 5.3 apresenta a complexidade de cada rodada dos algoritmos. A constante k corresponde a um custo adicional, relativamente baixo, gasto pelo algoritmo devido ao envio de mensagens extras. Na tabela, o paradigma de rotação de coordenadores é denotado por PRC.

	Algoritmos				
	CT	MR	Paxos	DC	FI
Mecanismo de Eleição de Líder	PRC	PRC	Ω	Ω	Ω
Número de Passos de Comunicação	3	2	4	4	2
Número de Mensagens	$3.n$	$n + n^2$	$4.n$	$4.n + k$	$2.n^2$

Tabela 5.3: Complexidade dos algoritmos.

Avaliando os resultados no ambiente definido por λ

Os resultados mostrados na Figura 5.7 foram obtidos utilizando a seguinte configuração, $\lambda = 1$ ms e $T_M = 10$ ms. Primeiramente, nós avaliamos o estado estável da latência. Ela alcança esse estado devido ao crescimento do T_{MR}. A partir do momento que o tempo entre suspeitas incorretas se torna suficientemente grande, a probabilidade dos protocolos de consenso terminarem em apenas uma rodada é maior. As diferenças de desempenho observadas no estado estável da latência podem ser explicadas, principalmente, pelo número de passos de comunicação e pelo número de mensagens. Nesse estado, o mecanismo de eleição de líder influencia muito pouco o desempenho dos algoritmos pelo fato de quase não ocorrer suspeitas incorretas.

Para $n = 3$ e $T = 10$ (1/s), a latência dos algoritmos se estabiliza quando $T_{MR} \geq 200$ ms. O algoritmo MR possui a latência com a melhor estabilização. Nós podemos perceber que as diferenças são causadas, principalmente, pelo número de passos de comunicação, já que o algoritmo MR possui um alto número de mensagens. Os algoritmos Paxos e DC tendem a se estabilizar sobre os mesmos valores pelo fato de possuírem a mesma complexidade na primeira rodada. O desempenho do algoritmo CT é melhor que o dos algoritmos Paxos e DC devido ao menor número de passos de comunicação. As latências dos algoritmos MR e FI não se estabilizam sobre os mesmos valores, porque, apesar de possuírem o mesmo número de passos de comunicação, eles não possuem o mesmo número de mensagens.

Para $n = 7$ e $T = 10$ (1/s), os algoritmos MR e FI possuem o pior desempenho. Além disso, enquanto os demais algoritmos alcançam o estado estável da latência quando $T_{MR} \geq 300$ ms, os algoritmos MR e FI alcançam esse estado apenas quando $T_{MR} \geq 2000$

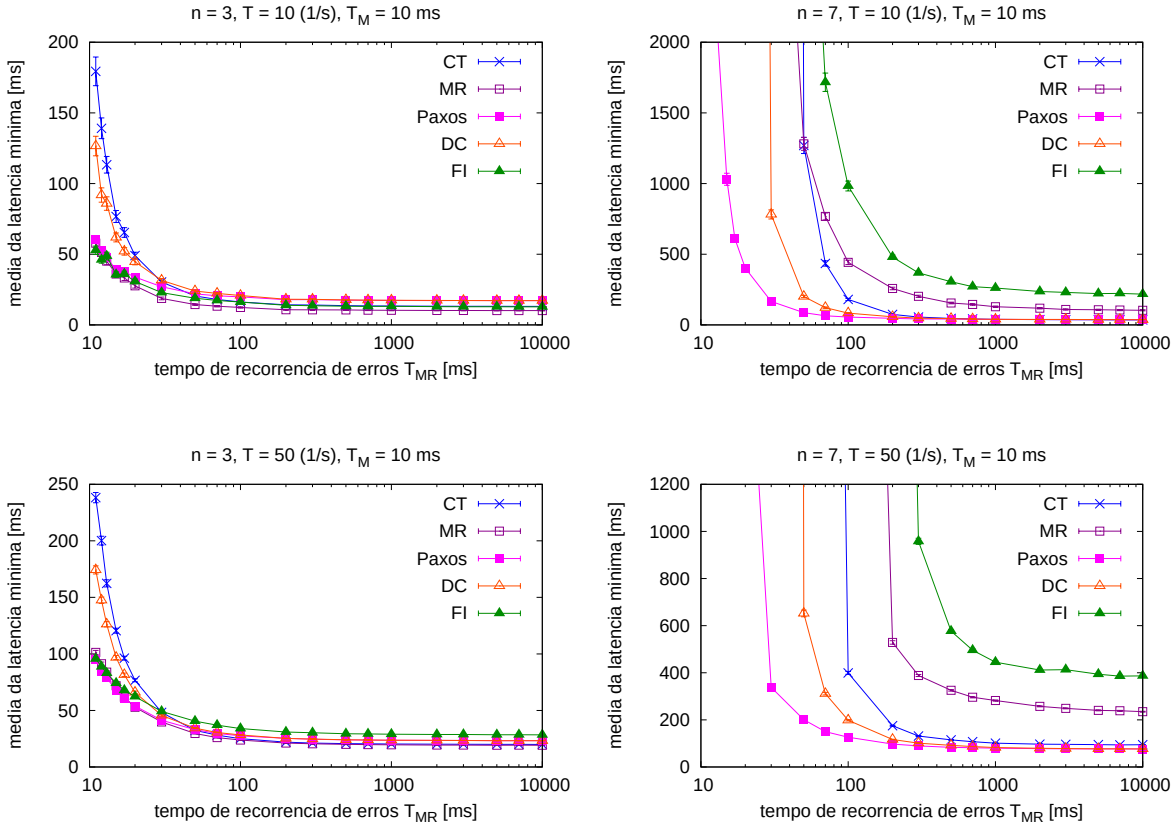


Figura 5.7: Comparação dos algoritmos. Latência vs. T_{MR} para $\lambda = 1$ ms.

ms. Isso mostra o quanto eles são sensíveis ao aumento do número de processos. O baixo desempenho em relação aos demais pode ser explicado pela alta complexidade em termos de mensagens, $O(n^2)$. À medida que o número de processos aumenta, o número de mensagens cresce exponencialmente.

Outro fator que também influencia o desempenho dos algoritmos é a vazão. Para $n = 3$ e $n = 7$, os algoritmos MR e FI foram os mais prejudicados pelo aumento da carga de trabalho, o que pode ser explicado pelo alto número de mensagens que esses algoritmos possuem.

Em relação ao estado variável da latência, os resultados mostram que, no geral, o Paxos é o algoritmo com melhor desempenho. Para $n = 3$ e $T = 10$ (1/s), a latência varia entre 11 ms $\leq T_{MR} <$ 200 ms. A diferença de desempenho observada entre os algoritmos Paxos e CT está associada ao mecanismo de eleição de líder, já que pela Tabela 5.3, o algoritmo CT possui uma complexidade melhor que a do algoritmo Paxos. Além disso, um estudo realizado por P. Urbán et al. [22] revelou que, de fato, o mecanismo de eleição de líder é o responsável por essa diferença. De acordo com o seu estudo, um oráculo com capacidade de eleição de líder é mais eficiente do que o paradigma de rotação de coordenadores porque evita que processos suspeitos sejam eleitos como coordenadores de uma rodada. Assim, o custo de uma rodada liderada por um coordenador suspeito não é computado.

Em relação ao algoritmo DC, o Paxos é melhor devido ao número de passos de comunicação, pois eles possuem o mesmo mecanismo de eleição de líder e a mesma complexidade em termos de mensagens. Por outro lado, o algoritmo MR é melhor que o CT devido ao número de passos de comunicação. Para explicar esse resultado, basta

observar que o mecanismo de eleição de líder dos dois algoritmos são iguais, e que o número de mensagens do algoritmo MR é maior. Comparando os algoritmos DC e FI, nós observamos que eles possuem o mesmo mecanismo de eleição de líder, e que o número de mensagens enviadas pelo algoritmo DC é menor ou igual. Nesse caso, o bom desempenho do algoritmo FI se deve ao baixo número de passos de comunicação. Para $T_{MR} \leq 20$ ms, existe pouca diferença de desempenho entre os algoritmos Paxos, MR e FI. Contudo, para $20 < T_{MR} < 200$, o desempenho do algoritmo MR é o melhor. Observe que para $T_{MR} > 20$ ms, os algoritmos com menor número de passos de comunicação são os que possuem o melhor desempenho. Assim, podemos afirmar que esse fator se torna o principal responsável pelas diferenças de desempenho quando $T_{MR} > 20$ ms.

Para $n = 7$ e $T = 10$ (1/s), os algoritmos com melhor desempenho são o Paxos e o DC. Eles foram os menos afetados pelo aumento do número de processos. Esses resultados são atribuídos ao mecanismo de eleição de líder e a complexidade de mensagens de cada um. Ao contrário dos demais algoritmos, eles são os únicos que possuem um oráculo com capacidade de eleição de líder e um baixo número de mensagens por rodada, $O(n)$. O baixo desempenho dos algoritmos MR e FI é explicado pelo alto número de mensagens, assim como no estado estável da latência. Já o algoritmo CT, não obteve um bom desempenho devido ao mecanismo de eleição de líder.

Para $T = 50$ (1/s), os resultados mostram que os algoritmos MR e FI são mais sensíveis ao aumento da carga de trabalho do que os demais. Observe que os dois algoritmos sofrem perda de desempenho nos dois estados da latência. O algoritmo FI é mais prejudicado pelo fato de possuir um número maior de mensagens.

Os resultados das Figuras 5.8 e 5.9 mostram que para $n = 3$, os desempenhos dos algoritmos MR e FI foram os mais afetados pelo aumento da contenção na rede e na CPU. Isso pode ser explicado pelo alto número de mensagens que os algoritmos MR e FI enviam. Quanto maior o número de mensagens enviadas, maior é o tempo gasto nos recursos de rede e CPU. Para $\lambda = 0{,}1$ ms, a perda de desempenho dos dois algoritmos é mais perceptível com o aumento da vazão. Para $\lambda = 10$ ms e $n = 7$, o desempenho do algoritmo DC é bastante prejudicado pelo aumento da contenção na CPU.

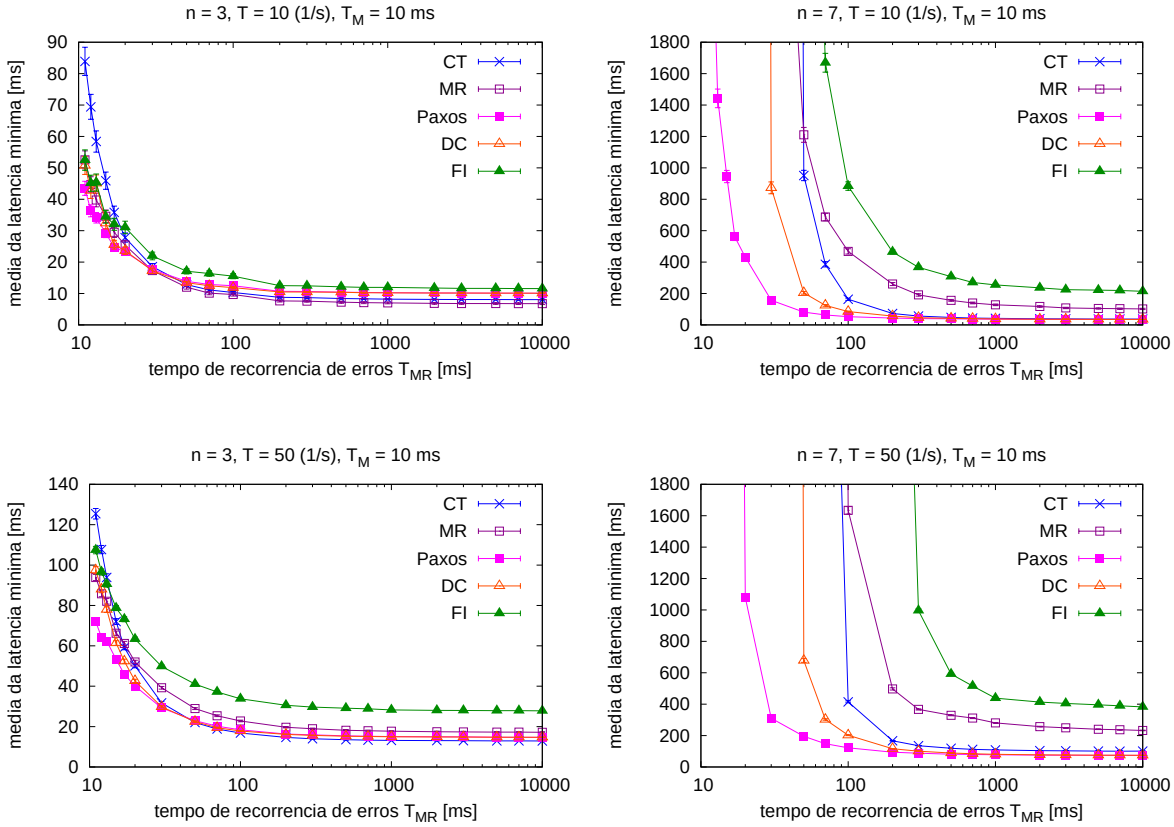


Figura 5.8: Comparação dos algoritmos. Latência vs. T_{MR} para $\lambda = 0{,}1$ ms.

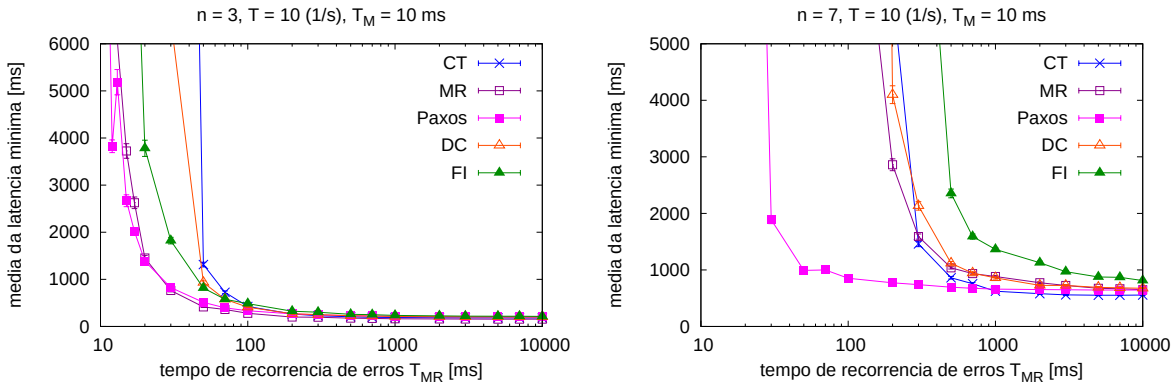


Figura 5.9: Comparação dos algoritmos. Latência vs. T_{MR} para $\lambda = 10$ ms.

Avaliando os resultados no ambiente definido por β

A Figura 5.10 apresenta os resultados para a configuração $\beta = 5$ ms e $T_M = 10$ ms. Por se tratar de um ambiente menos determinístico do que o anterior, veremos que o comportamento dos algoritmos muda bastante. Para $n = 3$, as latências dos algoritmos MR e FI possuem a melhor estabilização. Comparando os dois, o desempenho do algoritmo MR é melhor devido ao número de mensagens enviadas na fase 1. Apesar dos algoritmos possuírem uma alta complexidade em termos de mensagens, podemos observar que o número de passos de comunicação é responsável pelo bom desempenho de cada um. É por esse mesmo motivo que o algoritmo CT é mais eficiente que os

algoritmos DC e Paxos. Para $T_{MR} \geq 50$ ms, o fator que determina o desempenho de cada algoritmo é o número de passos de comunicação. Para $T_{MR} < 50$ ms, o algoritmo MR também possui o melhor desempenho. O algoritmo Paxos possui o segundo melhor desempenho para $T_{MR} < 20$ ms. Para essa taxa de suspeitas incorretas, o número de mensagens que o algoritmo FI envia na fase 1 prejudica o seu desempenho.

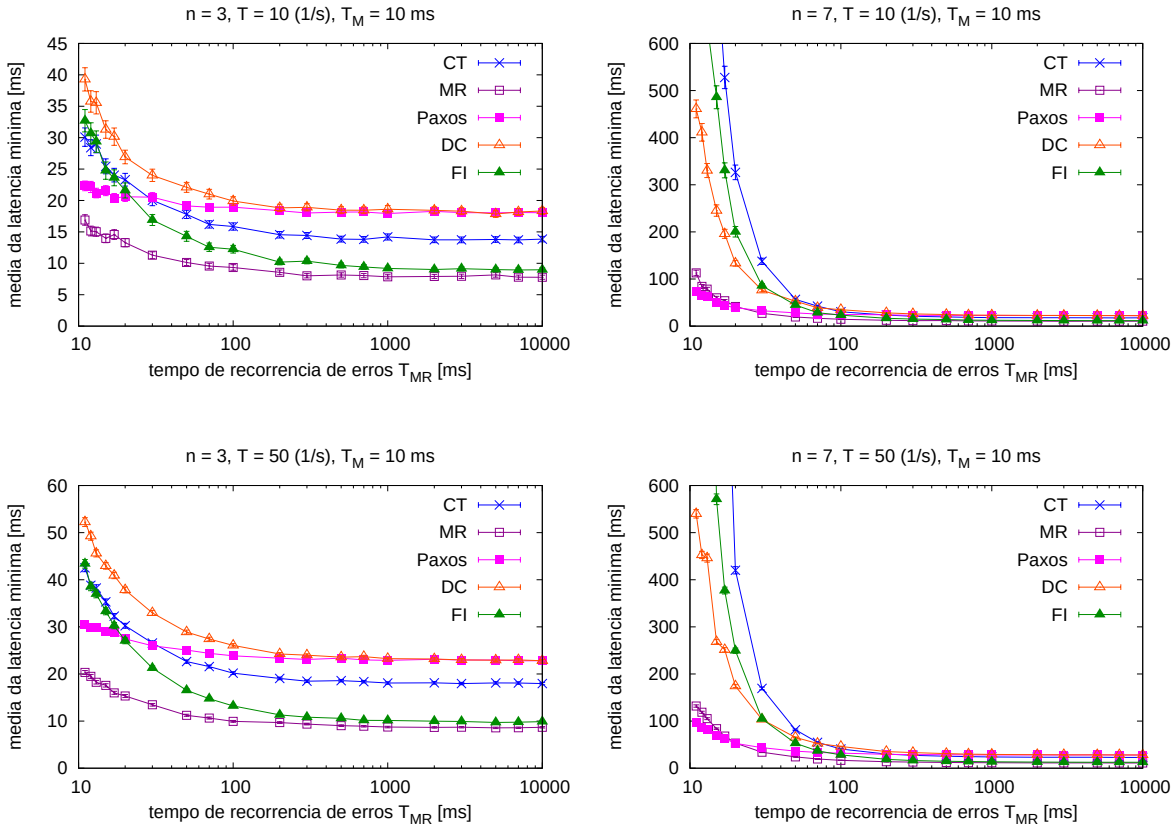


Figura 5.10: Comparação dos algoritmos. Latência vs. T_{MR} para $\beta = 5$ ms.

Para $n = 7$ e $T_{MR} < 20$ ms, o Paxos é o algoritmo com melhor desempenho. Entretanto, comparado ao algoritmo MR, a diferença de desempenho é pequena. Em relação ao estado estável da latência, o algoritmo MR possui a melhor estabilização. Os algoritmos Paxos e DC foram os que melhor reagiram ao aumento do número de processos. Por outro lado, os algoritmos MR, FI e CT foram prejudicados. A perda de desempenho do algoritmo FI está associada ao número de mensagens, visto que, ele possui o melhor mecanismo de eleição de líder e, também, o menor número de passos de comunicação. Em relação ao algoritmo MR, o número de mensagens e o mecanismo de eleição de líder são os responsáveis pela perda de desempenho. Nós atribuímos essa perda, principalmente, ao número de mensagens. Isso porque, apesar de estar equipado com o mecanismo de eleição de líder mais ineficiente, o algoritmo possui o melhor desempenho para $n = 3$. Finalmente, o algoritmo CT é prejudicado devido ao mecanismo de eleição de líder.

5.3.2 Avaliação das otimizações

Nesta seção, nós apresentamos e discutimos os resultados envolvendo as otimizações. Nessa análise, os algoritmos de Paxos e de Chandra e Toueg não executam, respectivamente, a fase 1 e a fase de leitura na primeira rodada. Nos gráficos, a combinação

das otimizações é denotada por "_". Além disso, a representação CTO corresponde ao algoritmo de Chandra e Toueg com todas as otimizações (ED, AW2/4, LA).

Avaliando os resultados no ambiente definido por λ

Os resultados apresentados na Figura 5.11 correspondem à configuração $\lambda = 1$ ms e $T_M = 10$ ms. Nós discutimos apenas os resultados para $T = 50$ (1/s), já que os resultados para $T = 10$ (1/s) são similares. Para $n = 3$, as otimizações AW4 e ED tornam o algoritmo CT mais eficiente quando submetido a altas taxas de suspeitas incorretas ($T_{MR} < 100$ ms). A otimização LA não melhora o desempenho do algoritmo devido ao pouco assincronismo existente nesse ambiente. Como discutido na Seção 4.3, a técnica LA depende do assincronismo do sistema para funcionar. Por outro lado, a otimização AW2, além de não melhorar, piora o desempenho do algoritmo. Esse resultado se deve ao aumento desnecessário do tempo de espera na fase 2.

Para $n = 7$, a otimização AW4 fornece o melhor ganho de desempenho. Para $T_{MR} \geq 1000$ ms, as otimizações AW2 e ED melhoram o estado estável da latência do algoritmo. O bom resultado fornecido pela otimização AW2 está relacionada com a detecção de falhas. Observe que o tempo adicional de espera gerado pela otimização influencia na saída dos detectores de falhas. Por exemplo, suponha que um processo p esteja em uma rodada r e seja o coordenador da rodada $r + 1$. Os detectores de falhas dos demais processos podem estar suspeitando incorretamente de p em r. Logo, na rodada $r + 1$ eles enviariam uma mensagem *nack* a p. Entretanto, a espera adicional pode fornecer tempo necessário para que esses processos corrijam suas suspeitas.

A combinação das otimizações ED, AW2 e AW4 melhora significativamente a latência do algoritmo CT, independentemente do número de processos. Podemos observar que a otimização AW2 melhora muito a eficiência da otimização ED. Isso porque o número de mensagens que podem ser recebidas pelo coordenador é maior. Por outro lado, a combinação da técnica LA não afeta em nada o desempenho do algoritmo, pois existe pouco assincronismo no ambiente.

Em relação a curva de desempenho dos algoritmos Paxos e CT, nós claramente observamos que o algoritmo CT é mais sensível a suspeitas incorretas do que o algoritmo Paxos. As otimizações nos ajudam a reduzir significativamente a diferença de desempenho entre os dois. Para poucos processos ($n = 3$) e altas taxas de suspeitas incorretas ($T_{MR} < 100$ ms), o algoritmo CTO é mais eficiente que o algoritmo Paxos. Para $n = 7$, o algoritmo Paxos é o melhor, porém nós conseguimos aumentar a tolerância à suspeitas incorretas do algoritmo CT. O ganho relativo ao T_{MR} (eixo x) foi de 56,17%. Em termos de latência (eixo y), o ganho foi de até 77,39%.

Os resultados para $\lambda = 0{,}1$ ms são similares aos da configuração $\lambda = 1$ ms e, por isso, são apresentados no Apêndice A do trabalho. Para a configuração $\lambda = 10$ ms e $T_M = 10$ ms, os resultados obtidos (mostrados na Figura 5.12) também são similares aos da configuração $\lambda = 1$ ms, porém existe uma diferença significativa na comparação entre os algoritmos CTO e Paxos. Para $n = 3$, o aumento da contenção na CPU beneficiou o algoritmo Paxos. Por outro lado, para $n = 7$ e $T_{MR} \geq 200$ ms, o aumento tornou o algoritmo CTO mais eficiente que o algoritmo Paxos.

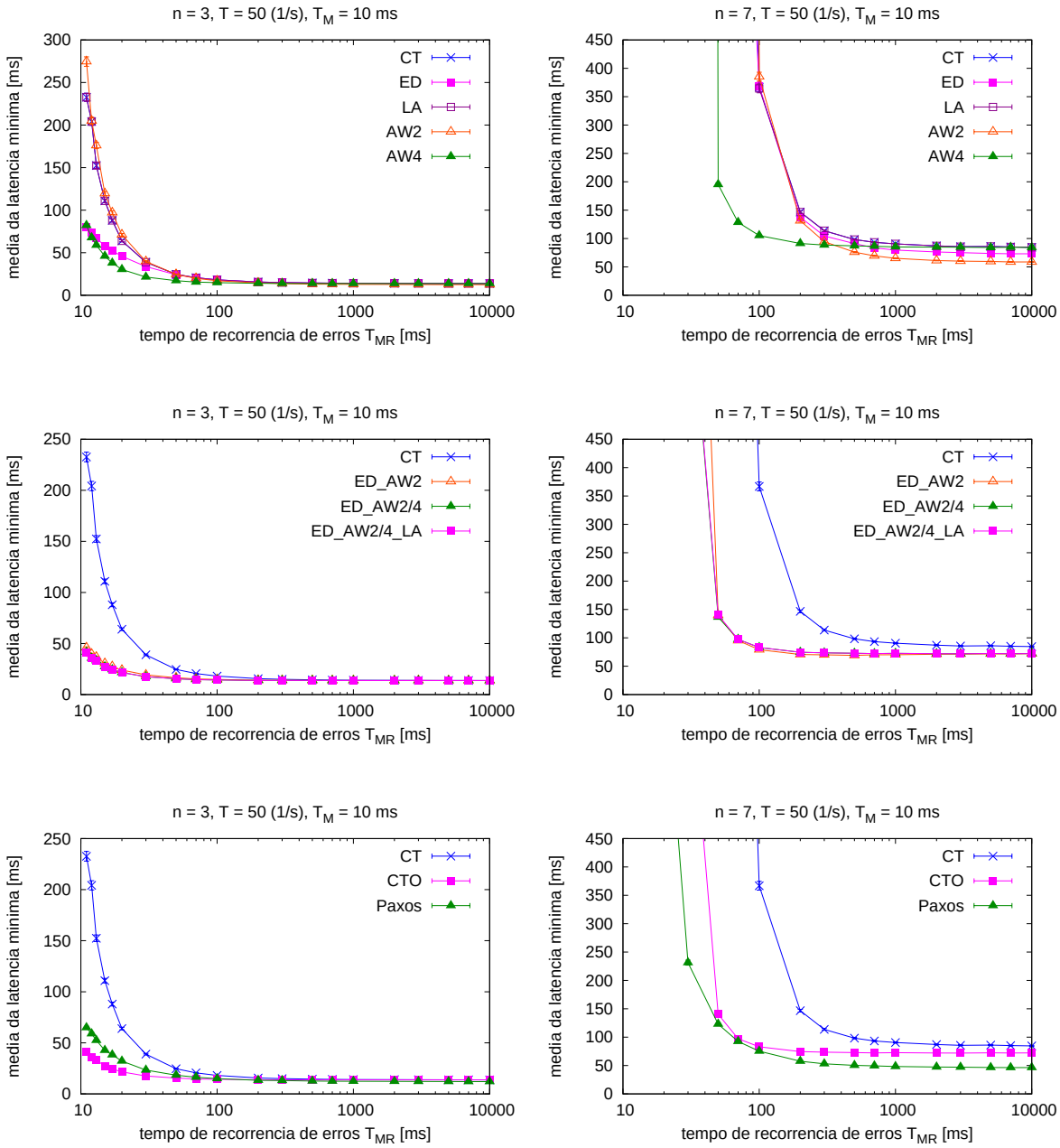


Figura 5.11: Avaliação das otimizações. Latência vs. T_{MR} para $\lambda = 1$ ms.

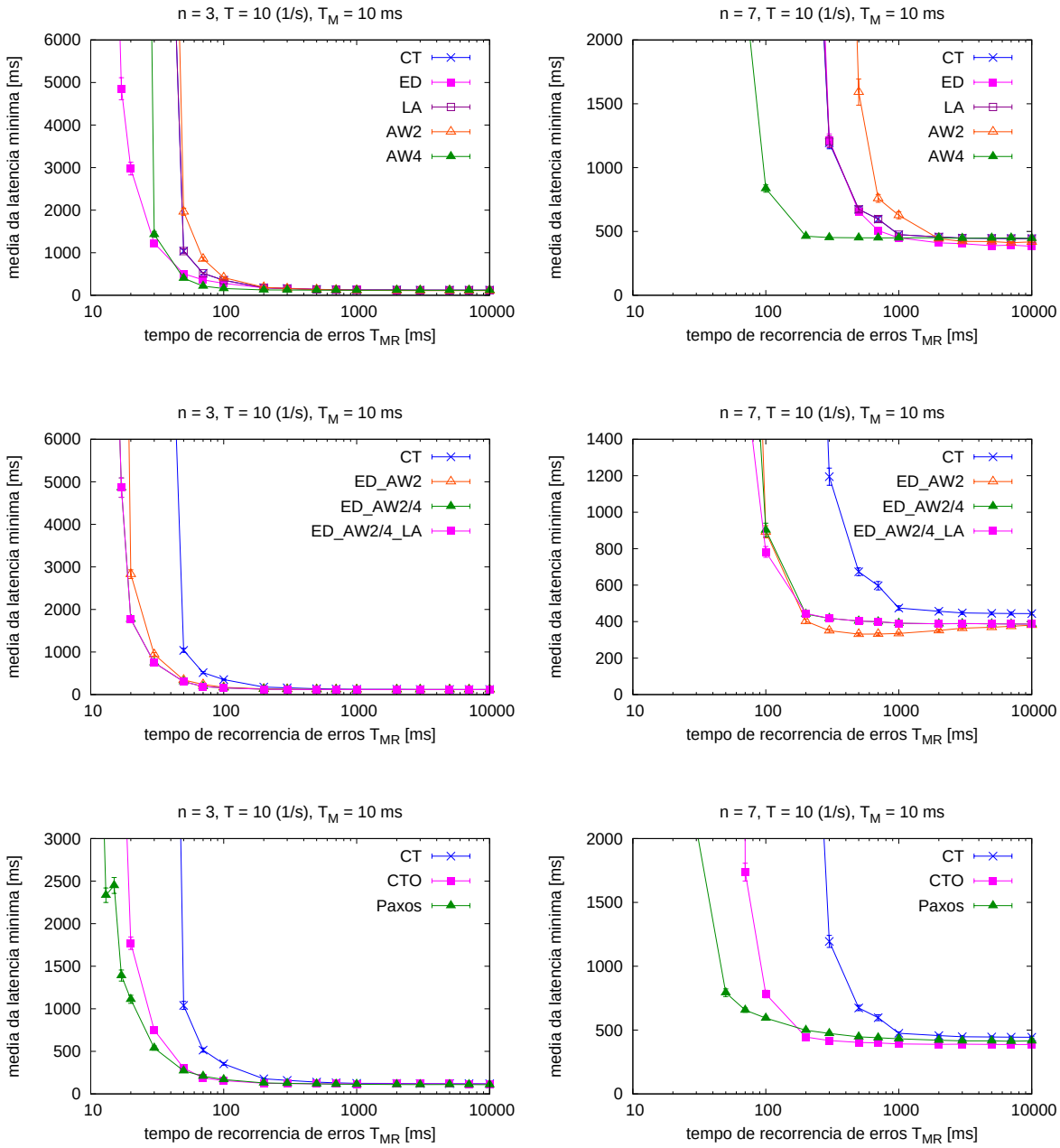


Figura 5.12: Avaliação das otimizações. Latência vs. T_{MR} para $\lambda = 10$ ms.

Avaliando os resultados no ambiente definido por β

Os resultados mostrados na Figura 5.13 correspondem a configuração $\beta = 5$ ms e $T_M = 10$ ms. Para $n = 3$, as otimizações ED, AW4 e LA beneficiam o algoritmo CT. Além disso, a otimização ED, além de reagir mais rapidamente às suspeitas incorretas, possui a latência com a melhor estabilização. A técnica LA fornece um ganho de desempenho muito pequeno comparado às demais. Entretanto, quando o número de processos aumenta para $n = 7$, esse ganho se torna mais perceptível. A razão pela qual isso ocorre está relacionada ao aumento do número de processos atrasados. A técnica LA permite que esses processos acelerem sua execução. Consequentemente, a execução do protocolo de consenso pode terminar mais rapidamente. Em relação à otimização AW4, podemos observar que ela é a melhor opção para execuções com muitos processos. Contudo, a

otimização ED também fornece um bom ganho de desempenho. A otimização AW2, pelo contrário, sempre degrada o desempenho do algoritmo.

Assim como no primeiro conjunto de resultados, a combinação das otimizações melhora significativamente a latência do algoritmo CT. Para $n = 7$, nós podemos observar que a eficiência do algoritmo CT aumenta à medida que cada otimização é aplicada. Observe que o algoritmo CTO funciona para o menor valor possível de T_{MR} (11 ms), enquanto que, o algoritmo CT funciona apenas para $T_{MR} \geq 50$ ms. Os dois últimos gráficos da Figura 5.13, ilustram o ganho de desempenho do algoritmo CTO em relação ao algoritmo Paxos. Em ambos os gráficos, o algoritmo CTO alcança decisões mais cedo do que o algoritmo Paxos na presença de altas taxas de suspeitas incorretas. Além disso, a latência do algoritmo CTO se estabiliza sobre valores menores.

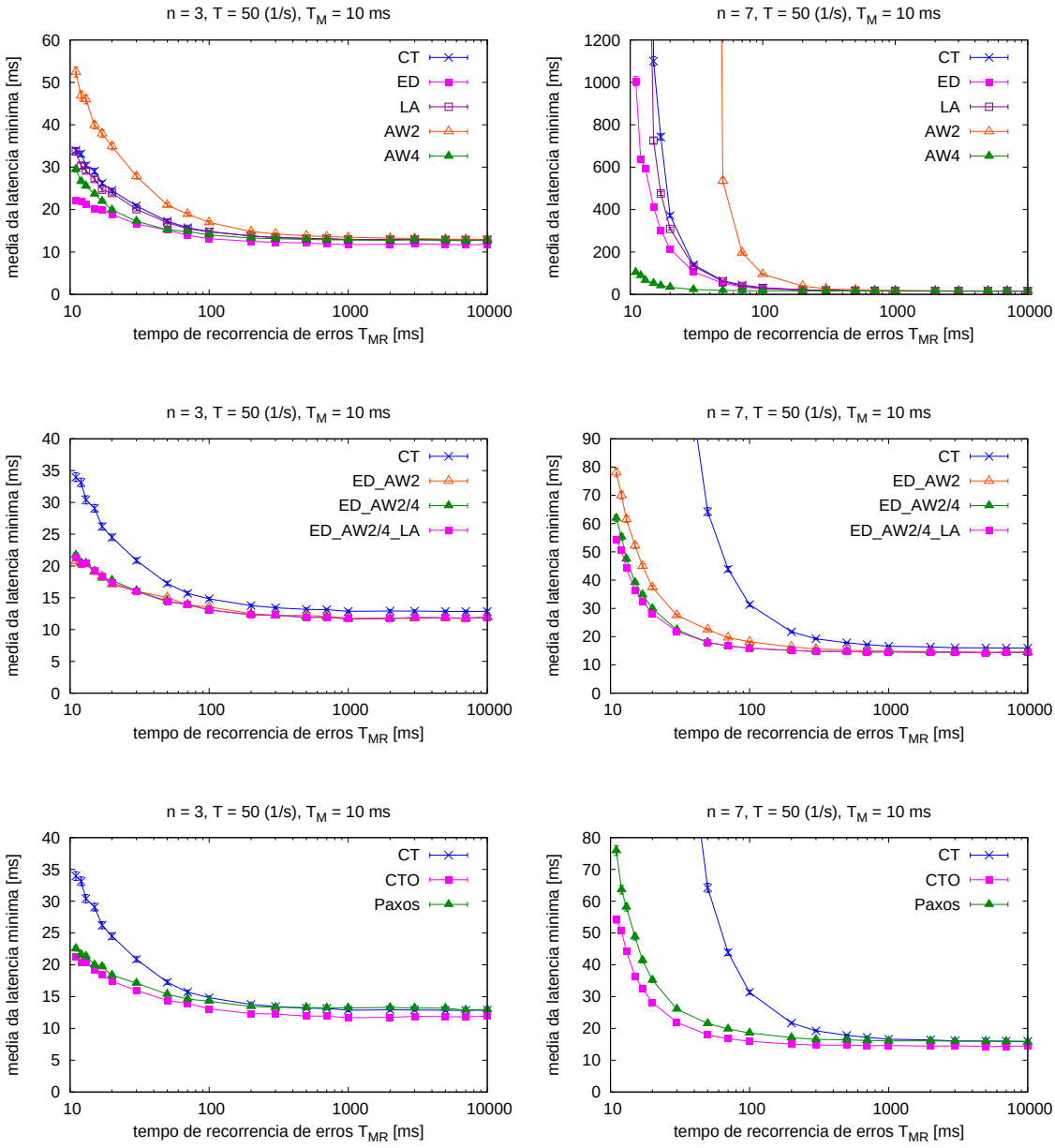


Figura 5.13: Avaliação das otimizações. Latência vs. T_{MR} para $\beta = 5$ ms.

Capítulo 6

Conclusão

Vimos que o algoritmo de consenso de Chandra e Toueg é bastante sensível às suspeitas incorretas causadas pelo mecanismo de detecção de falhas. Além de degradar o desempenho do algoritmo, as suspeitas incorretas também afetam o desempenho de todo protocolo que se baseia nele. O objetivo desse trabalho é minimizar essa degradação por meio de técnicas de otimização. A principal contribuição desse trabalho é a proposta de duas novas otimizações e uma adaptação da técnica *Look-Ahead* ao algoritmo.

A otimização *Early-Decision* permite que o consenso seja alcançado mais rapidamente quando submetido a suspeitas incorretas. Ela garante que uma decisão seja alcançada com apenas um único passo de comunicação em uma rodada particular $r > 1$, se o coordenador dessa rodada receber uma maioria de mensagens com a mesma estimativa e a mesma estampa de tempo na fase 2. A vantagem dessa otimização é que ela é simples e não gera qualquer envio de mensagem adicional. Além disso, ela não se restringe apenas a execuções com suspeitas incorretas, ela também é útil em execuções com falhas de coordenador.

A otimização *Additional Waiting* foi projetada para eliminar as esperas desnecessárias realizadas pela melhoria proposta ao protocolo de consenso baseado na classe $\Diamond C$. Assim, a otimização é acionada apenas quando for possível acelerar a tomada de uma decisão. Uma característica importante dessa otimização é que ela pode ser facilmente aplicada a outros protocolos de consenso, pois requer apenas o uso de um detector de falhas da classe $\Diamond S$.

A técnica de otimização *Look-Ahead* pode acelerar a execução dos protocolos de consenso somente se o sistema se comportar de maneira assíncrona. Quanto maior for o assincronismo, maior a sua eficácia. A idéia básica é utilizar mensagens enviadas em rodadas futuras para acelerar a execução de processos lentos. Esses processos utilizam as mensagens futuras para reduzir o tempo de espera por mensagens atrasadas. Além de reduzir o tempo de espera, nossa adaptação aumenta a probabilidade de um processo lento p enviar uma mensagem *ack* para o coordenador da rodada em que p estiver.

Além de propor essas otimizações, esse trabalho apresenta também as seguintes contribuições:

- Prova formal de corretude do algoritmo de consenso de Chandra e Toueg otimizado.
- Avaliação de desempenho de alguns algoritmos de consenso.

A prova de corretude garante que as otimizações não violam as propriedades do consenso. A avaliação de desempenho nos permite identificar e entender as principais diferenças que existem entre os algoritmos de consenso. Além disso, essa avaliação

reforça o estudo realizado por P. Urbán et al., onde eles comparam o desempenho do algoritmo de Chandra e Toueg com o do algoritmo de Paxos.

A partir dos resultados apresentados na Seção 5.3.1, nós constatamos que, no geral, o algoritmo de Paxos é o mais eficiente, principalmente, quando o número de processos aumenta. Em relação às otimizações, os resultados apresentados na Seção 5.3.2 mostram que a combinação delas fornece um ganho de desempenho significativo ao algoritmo de Chandra e Toueg, e que cada uma contribui para esse ganho. O resultado mais interessante é a comparação entre o algoritmo de Chandra e Toueg otimizado e o algoritmo de Paxos. A diferença de desempenho entre os dois foi significativamente reduzida. Na maioria das situações consideradas, o algoritmo de Chandra e Toueg otimizado é capaz de tolerar mais suspeitas incorretas do que o algoritmo de Paxos.

Durante a nossa pesquisa, surgiram algumas idéias que podem motivar trabalhos futuros, tais como:

1. A extensão da otimização *Early-Decision* a outros protocolos de consenso.

2. A extensão da técnica *Look-Ahead* para acelerar não apenas a execução de processos lentos, mas também, os processos mais adiantados.

3. Criação de um ambiente único que envolva contenção de recursos, e também, fatores que possam gerar mais ou menos determinismo na rede. Um dos grandes desafios desse projeto foi a realização de experimentos práticos. Para fazer toda a avaliação necessária, fomos obrigados a utilizar dois ambientes de simulação.

4. Criação de algoritmos híbridos, isto é, algoritmos que possam, dinamicamente, se alternar na execução do consenso dependendo do comportamento da rede. Nós observamos pelos resultados que alguns algoritmos são melhores que outros para uma determinada configuração de rede.

Referências Bibliográficas

[1] Aguilera., M. K., Chen, W., and Toueg, S. On quiescent reliable communication. *SIAM J. Comput. 29* (April 2000), 2040–2073.

[2] Basu, A., Charron-Bost, B., and Toueg, S. Solving problems in the presence of process crashes and lossy links. Tech. rep., Ithaca, NY, USA, 1996.

[3] Boichat, R., Dutta, P., Frølund, S., and Guerraoui, R. Deconstructing paxos. *SIGACT News 34* (March 2003), 47–67.

[4] Brasileiro, F. V., Greve, F., Mostéfaoui, A., and Raynal, M. Consensus in one communication step. In *Proceedings of the 6th International Conference on Parallel Computing Technologies* (London, UK, 2001), PaCT '01, Springer-Verlag, pp. 42–50.

[5] Chandra, T. D., Hadzilacos, V., and Toueg, S. The weakest failure detector for solving consensus. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1992), PODC '92, ACM, pp. 147–158.

[6] Chandra, T. D., and Toueg, S. Unreliable failure detectors for reliable distributed systems. *J. ACM 43* (March 1996), 225–267.

[7] Chen, W., Toueg, S., and Aguilera, M. K. On the quality of service of failure detectors. *IEEE Transactions on Computers 51* (2000), 561–580.

[8] Défago, X., Felber, P., and Schiper, A. Optimization techniques for replicating corba objects. In *Proceedings of the Fourth International Workshop on Object-Oriented Real-Time Dependable Systems* (Washington, DC, USA, 1999), WORDS '99, IEEE Computer Society.

[9] Défago, X., and Schiper, A. Semi-passive replication and lazy consensus. *J. Parallel Distrib. Comput. 64* (December 2004), 1380–1398.

[10] Dolev, D., Friedman, R., Keidar, I., and Malkhi, D. Failure detectors in omission failure environments. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1997), PODC '97, ACM.

[11] Dutta, P., and Guerraoui, R. Fast indulgent consensus with zero degradation. In *Proceedings of the 4th European Dependable Computing Conference on Dependable Computing* (London, UK, 2002), EDCC-4, Springer-Verlag, pp. 191–208.

[12] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32* (April 1985), 374–382.

[13] Guerraoui, R., and Lynch, N. A general characterization of indulgence. *ACM Trans. Auton. Adapt. Syst. 3* (December 2008), 20:1–20:19.

[14] Guerraoui, R., and Raynal, M. The Information Structure of Indulgent Consensus. *IEEE Transaction on Computers 53*, 4 (2004), 453–466.

[15] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst. 16* (May 1998), 133–169.

[16] Lamport, L. Paxos made simple. *ACM SIGACT News 32*, 4 (December 2001), 18–25.

[17] Larrea, M., Fernández, A., and Arévalo, S. Eventually consistent failure detectors. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 2001), SPAA '01, ACM, pp. 326–327.

[18] Raynal, M. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News 36* (March 2005), 53–70.

[19] Trivedi, K. S. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley & Sons, New York, NY, USA, 2002.

[20] Urbán, P., Défago, X., and Schiper, A. Contention-Aware Metrics for Distributed Algorithms: Comparison of Atomic Broadcast Algorithms. In *Proceedings of the 9th IEEE International Conference on Computer Communications and Networks (IC3N 2000)* (2000), pp. 582–589.

[21] Urbán, P., Défago, X., and Schiper, A. Neko: A Single Environment to Simulate and Prototype Distributed Algorithms. *Journal of Information Science and Engineering 18*, 6 (2002), 981–997.

[22] Urbán, P., Hayashibara, N., Schiper, A., and Katayama, T. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *Proc. 23nd IEEE International Symp. on Reliable Distributed Systems (SRDS)* (2004), pp. 4–17.

[23] Wu, W., Cao, J., Yang, J., and Raynal, M. Using asynchrony and zero degradation to speed up indulgent consensus protocols. *J. Parallel Distrib. Comput. 68* (July 2008), 984–996.

Apêndice A

Resultados

Os resultados apresentados neste Apêndice correspondem aos resultados omitidos no trabalho, conforme o comentário realizado na Seção 5.3.

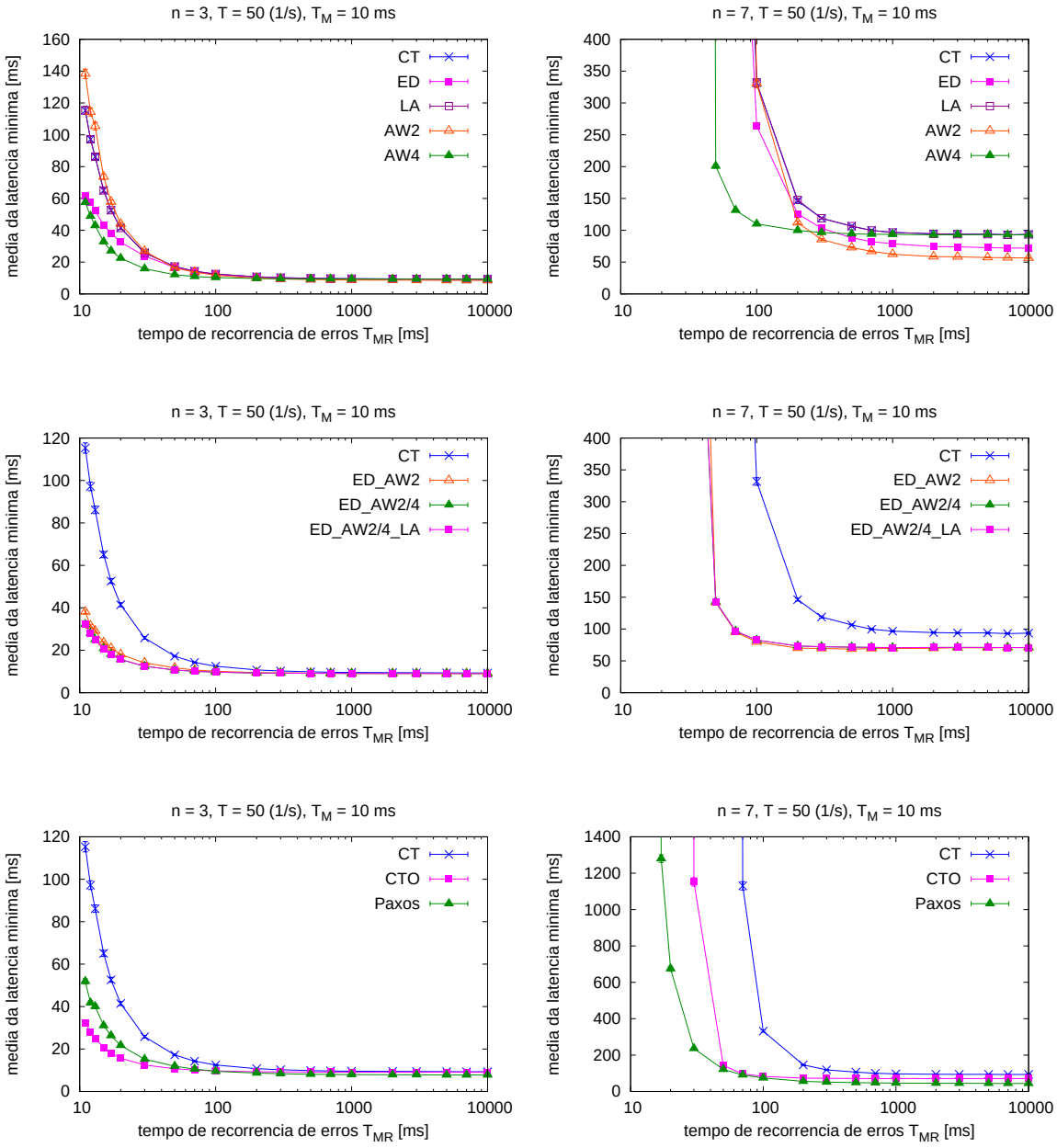


Figura A.1: Avaliação das otimizações. Latência vs. T_{MR} para $\lambda = 0{,}1$ ms.

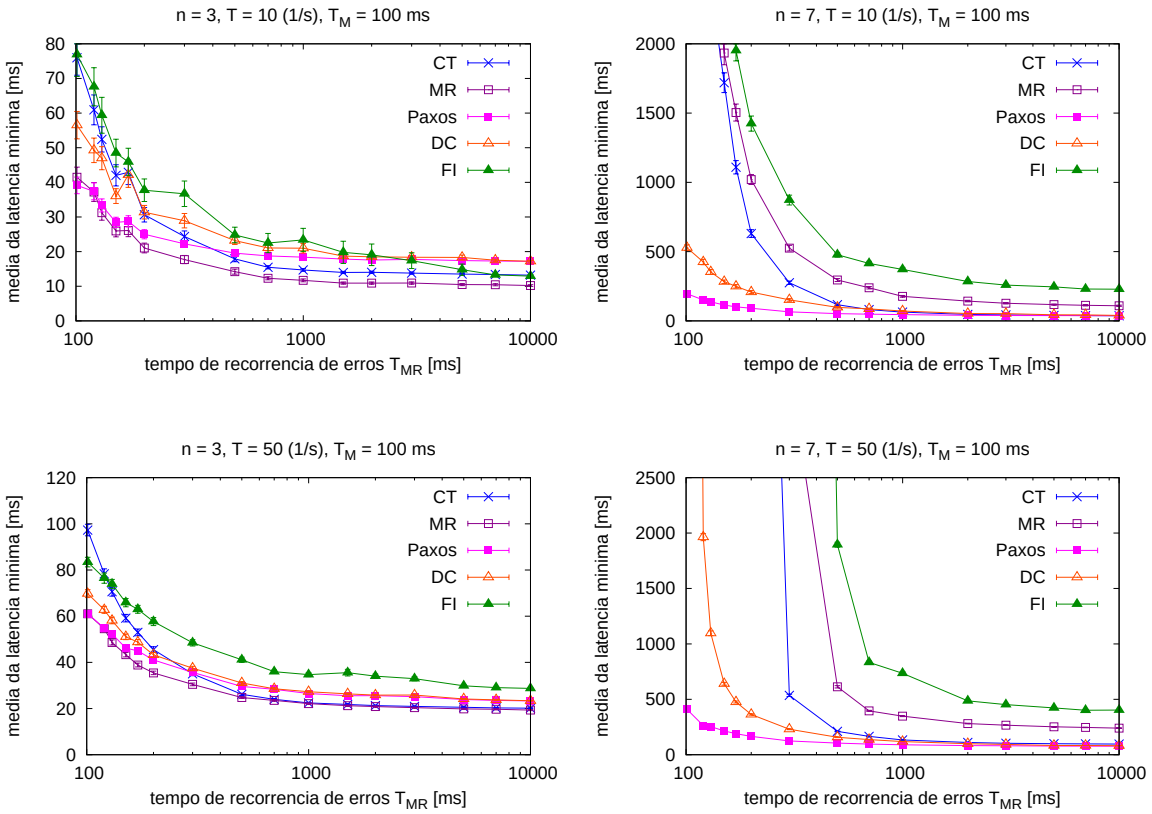


Figura A.2: Comparação dos algoritmos. Latência vs. T_{MR} para $\lambda = 1$ ms.

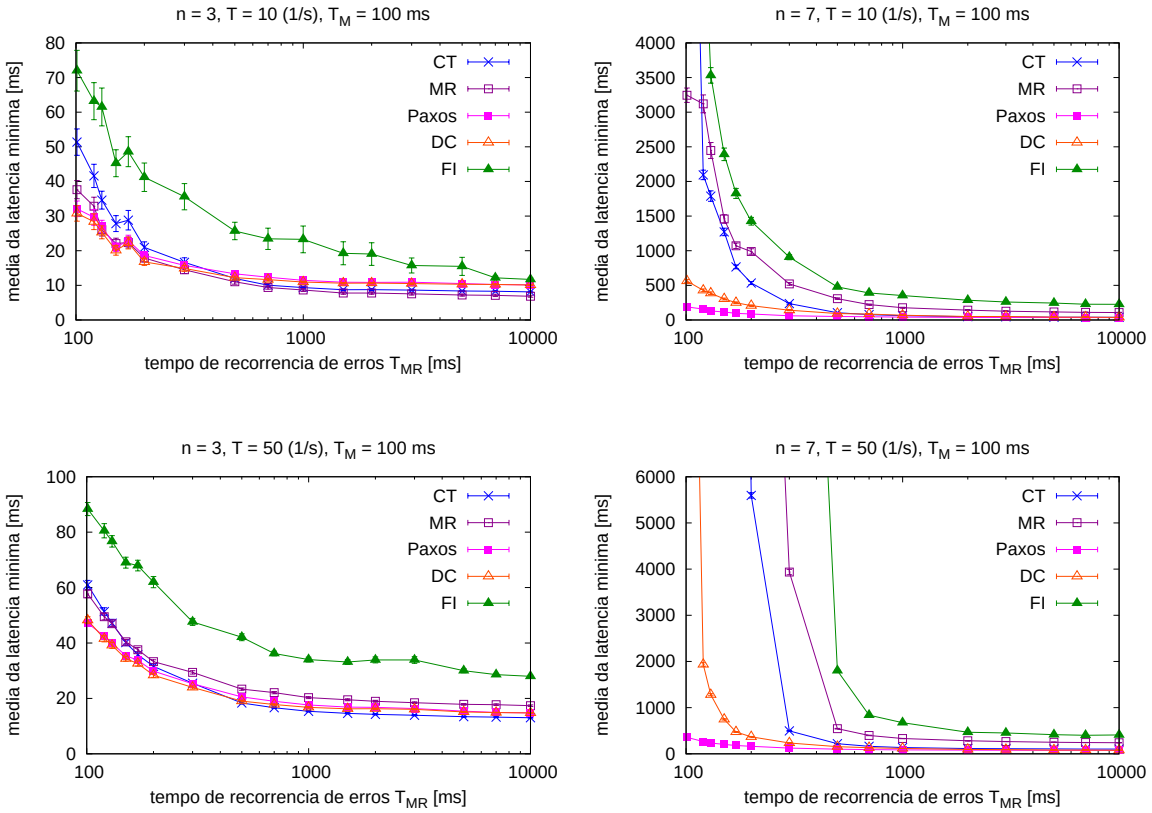


Figura A.3: Comparação dos algoritmos. Latência vs. T_{MR} para $\lambda = 0{,}1$ ms.

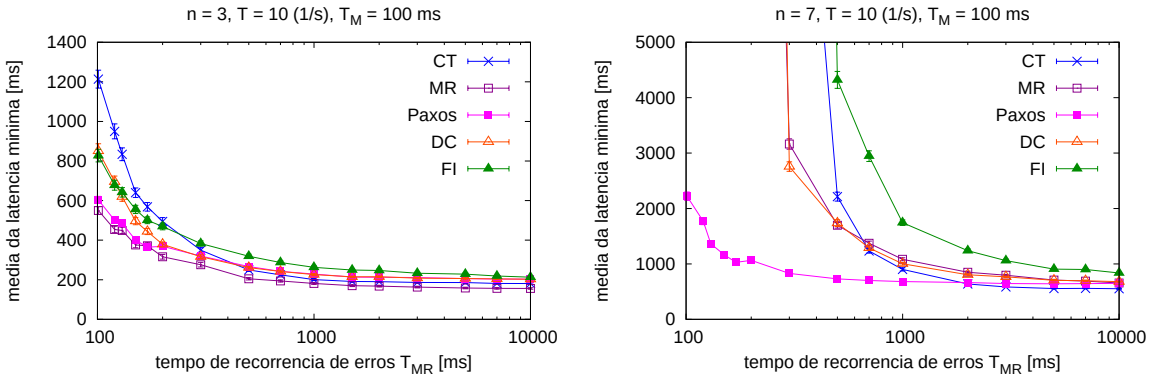


Figura A.4: Comparação dos algoritmos. Latência vs. T_{MR} para $\lambda = 10$ ms.

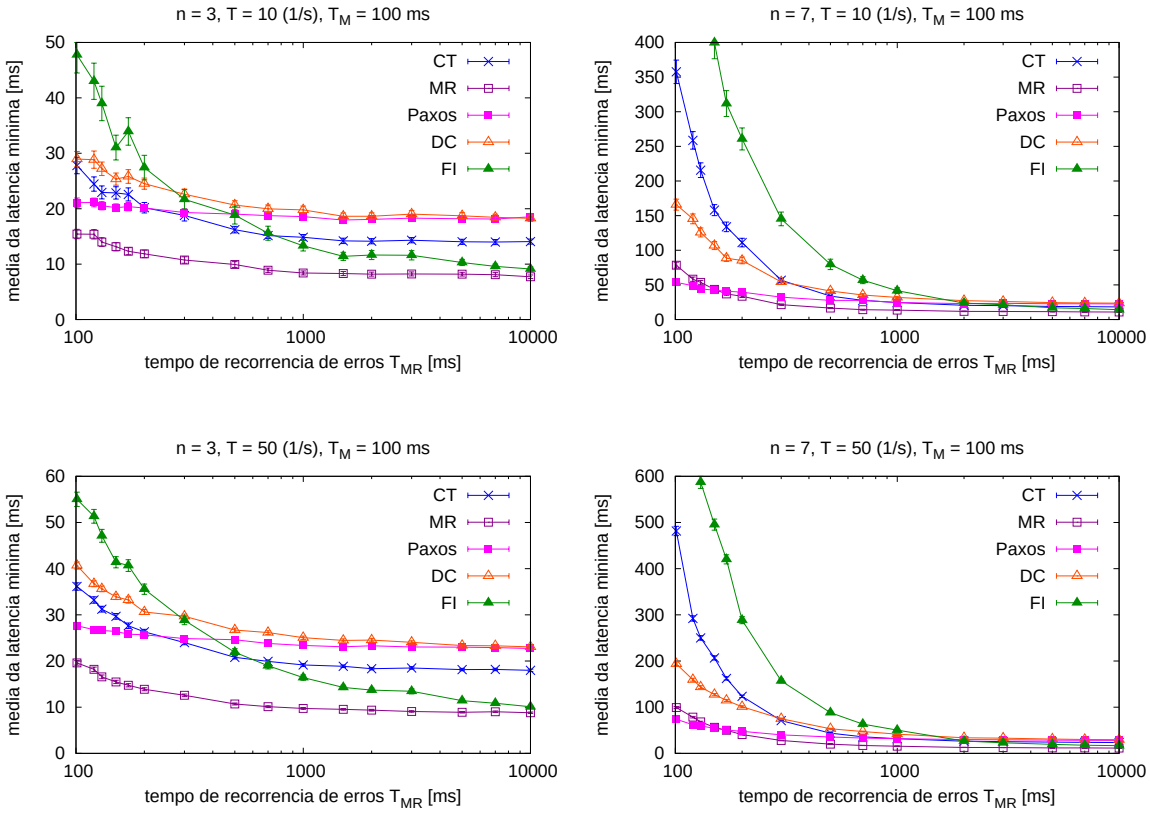


Figura A.5: Comparação dos algoritmos. Latência vs. T_{MR} para $\beta = 5$ ms.

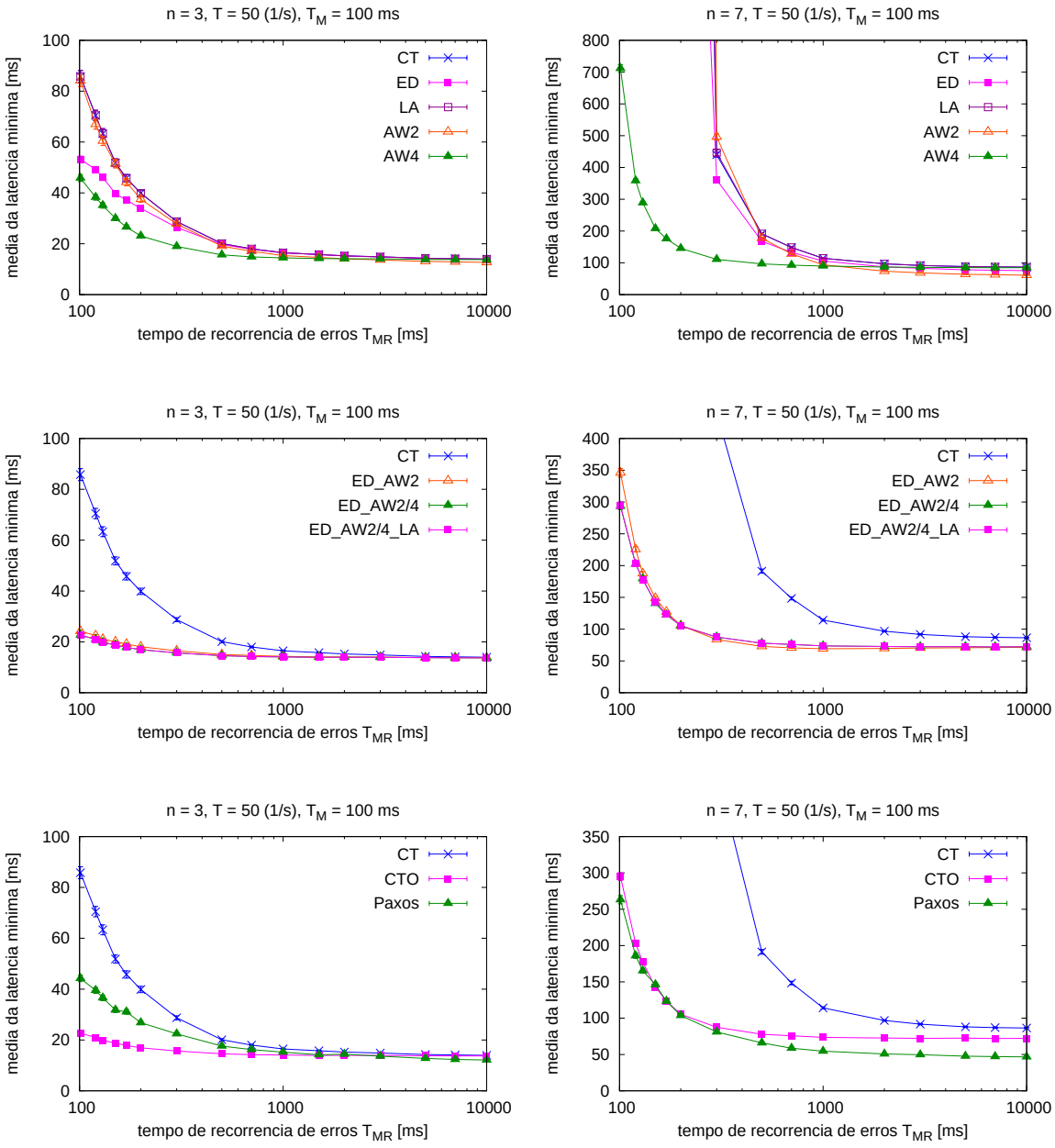


Figura A.6: Avaliação das otimizações. Latência vs. T_{MR} para $\lambda = 1$ ms.

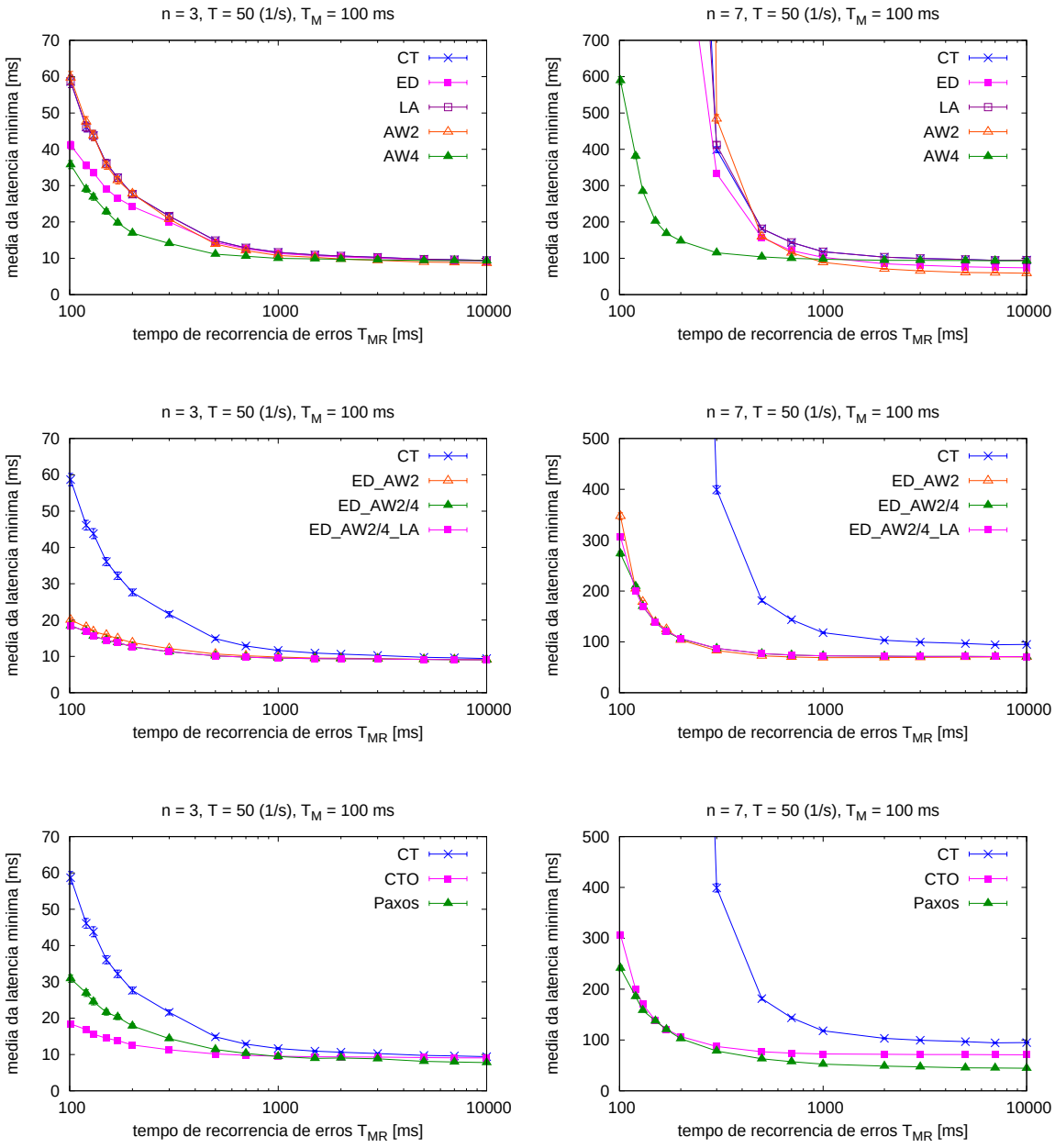


Figura A.7: Avaliação das otimizações. Latência vs. T_{MR} para $\lambda = 0{,}1$ ms.

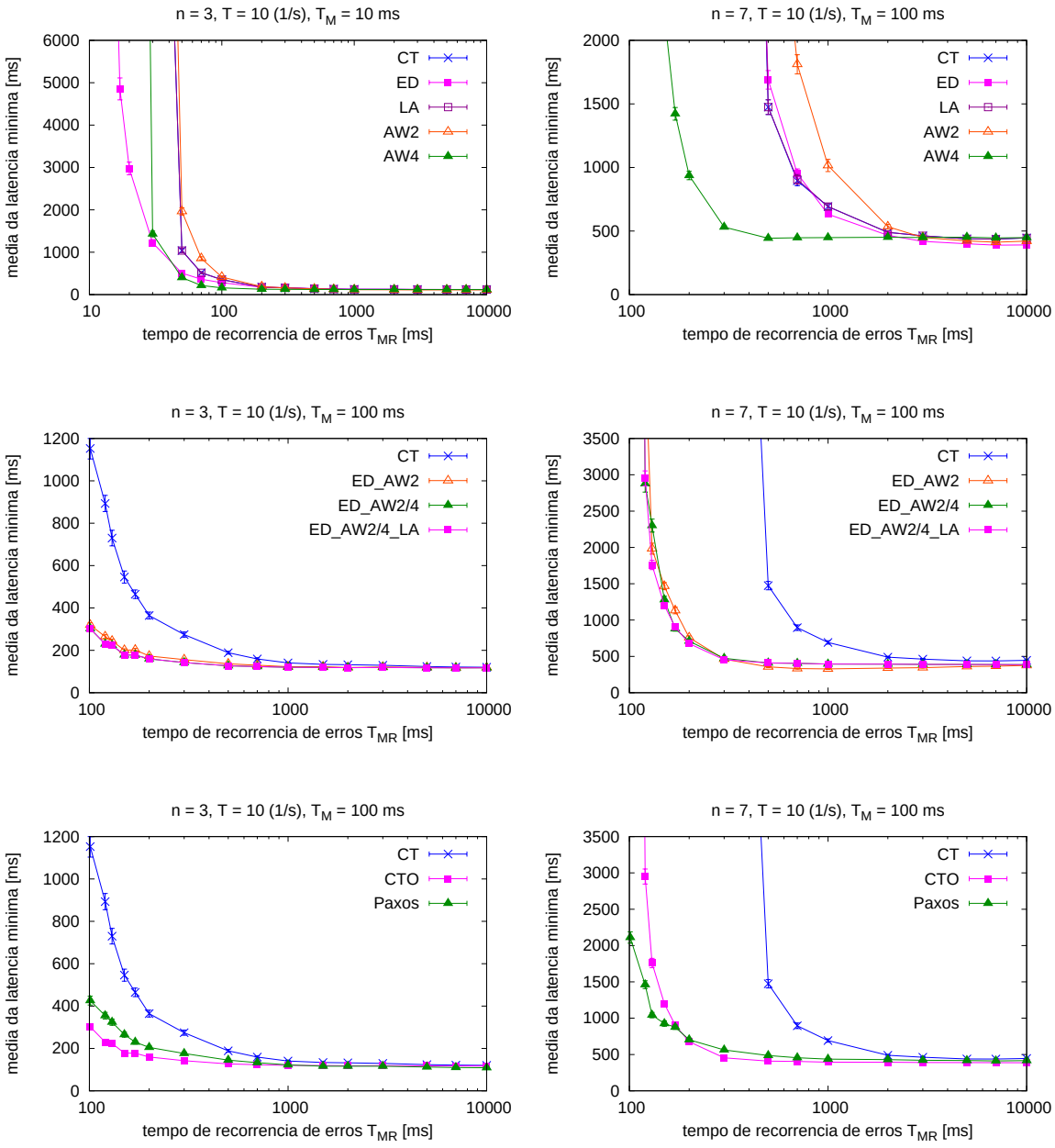


Figura A.8: Avaliação das otimizações. Latência vs. T_{MR} para $\lambda = 10$ ms.

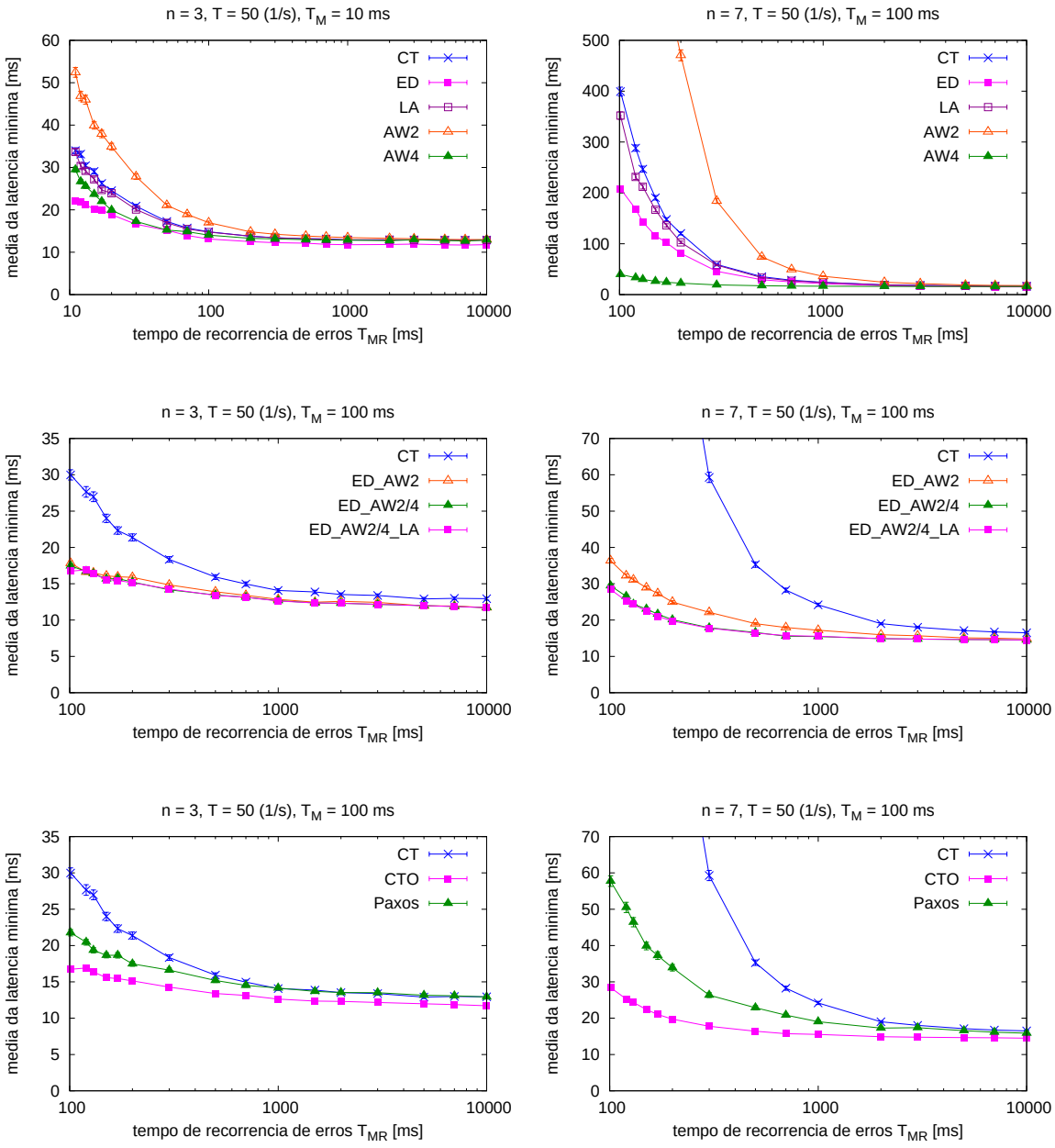


Figura A.9: Avaliação das otimizações. Latência vs. T_{MR} para $\beta = 5$ ms.