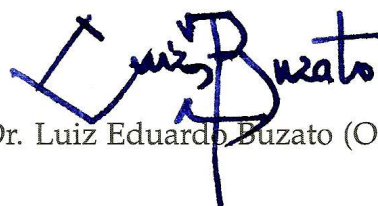


Uma Arquitetura de Software para Replicação Baseada em Consenso

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Gustavo Maciel Dias Vieira e aprovada pela Banca Examinadora.

Campinas, 17 de novembro de 2010.



Prof. Dr. Luiz Eduardo Buzato (Orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

**FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DO IMECC DA UNICAMP**

Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Vieira, Gustavo Maciel Dias

V673a Uma arquitetura de software para replicação baseada em consenso/
Gustavo Maciel Dias Vieira-- Campinas, [S.P. : s.n.], 2010.

Orientador : Luiz Eduardo Buzato.

Tese (doutorado) - Universidade Estadual de Campinas, Instituto de
Computação.

1.Sistemas distribuídos. 2.Algoritmos distribuídos. 3.Middleware.
4.Redes de computação - Protocolos. 5.Serviços na Web. I. Buzato,
Luiz Eduardo. II. Universidade Estadual de Campinas. Instituto de
Computação. III. Título.

Título em inglês: A software architecture for consensus based replication

Palavras-chave em inglês (Keywords): 1. Distributed systems. 2. Distributed algorithms.
3.Middleware. 4. Computer network protocols. 5. Web services.

Área de concentração: Sistemas de Computação

Titulação: Doutor em Ciência da Computação

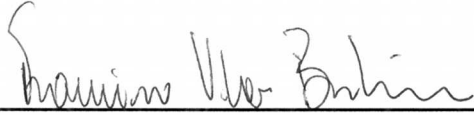
Banca examinadora: Prof. Dr. Luiz Eduardo Buzato (IC – UNICAMP)
Prof. Dr. Francisco Vilar Brasileiro (CEEI – UFCG)
Prof. Dr. Jonida Silva Fraga (DAS – UFSC)
Prof. Dr. Fernando Pedone (Faculty of Informatics – USI)
Prof. Dr. Ricardo de Oliveira Anido (IC – UNICAMP)

Data da defesa: 17/11/2010

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 17 de novembro de 2010, pela Banca examinadora composta pelos Professores Doutores:



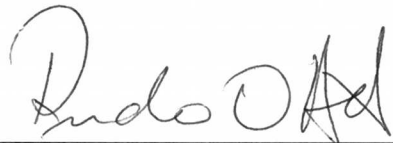
Prof. Dr. Francisco Vilar Brasileiro
Centro de Engenharia Elétrica e Informática / UFCG



Prof. Dr. Joni da Silva Fraga
Departamento de Automação e Sistemas / UFSC



Prof. Dr. Fernando Pedone
Faculty of Informatics / USI



Prof. Dr. Ricardo de Oliveira Anido
IC / UNICAMP



Prof. Dr. Luiz Eduardo Buzato
IC / UNICAMP

Uma Arquitetura de Software para Replicação Baseada em Consenso

Gustavo Maciel Dias Vieira¹

Novembro de 2010

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (Orientador)
- Prof. Dr. Francisco Vilar Brasileiro
Centro de Engenharia Elétrica e Informática — UFCCG
- Prof. Dr. Joni da Silva Fraga
Departamento de Automação e Sistemas — UFSC
- Prof. Dr. Fernando Pedone
Faculty of Informatics — USI
- Prof. Dr. Ricardo de Oliveira Anido
Instituto de Computação — Unicamp
- Profa. Dra. Ingrid Eleonora Schreiber Jansch Pôrto (Suplente)
Instituto de Informática — UFRGS
- Prof. Dr. Edmundo Roberto Mauro Madeira (Suplente)
Instituto de Computação — Unicamp
- Prof. Dr. Arnaldo Vieira Moura (Suplente)
Instituto de Computação — Unicamp

¹Apoio financeiro do CNPq, processo número 142638/2005-6.

Resumo

Esta tese explora uma das ferramentas fundamentais para construção de sistemas distribuídos: a replicação de componentes de software. Especificamente, procuramos resolver o problema de como simplificar a construção de aplicações replicadas que combinem alto grau de disponibilidade e desempenho. Como ferramenta principal para alcançar o objetivo deste trabalho de pesquisa desenvolvemos Treplica, uma biblioteca de replicação voltada para construção de aplicações distribuídas, porém com semântica de aplicações centralizadas. Treplica apresenta ao programador uma interface simples baseada em uma especificação orientada a objetos de replicação ativa.

A conclusão que defendemos nesta tese é que é possível desenvolver um suporte modular e de uso simples para replicação que exhibe alto desempenho, baixa latência e que permite recuperação eficiente em caso de falhas. Acreditamos que a arquitetura de software proposta tem aplicabilidade em qualquer sistema distribuído, mas é de especial interesse para sistemas que não são distribuídos pela ausência de uma forma simples, eficiente e confiável de replicá-los.

Abstract

This thesis explores one of the fundamental tools for the construction of distributed systems: the replication of software components. Specifically, we attempted to solve the problem of simplifying the construction of high-performance and high-availability replicated applications. We have developed Treplica, a replication library, as the main tool to reach this research objective. Treplica allows the construction of distributed applications that behave as centralized applications, presenting the programmer a simple interface based on an object-oriented specification for active replication.

The conclusion we reach in this thesis is that it is possible to create a modular and simple to use support for replication, providing high performance, low latency and fast recovery in the presence of failures. We believe our proposed software architecture is applicable to any distributed system, but it is particularly interesting to systems that remain centralized due to the lack of a simple, efficient and reliable replication mechanism.

Agradecimentos

Uma tese, mesmo sendo feita por uma só pessoa, envolve muito mais trabalho do que qualquer pessoa pode dar conta. Gostaria de agradecer a todos os companheiros de trabalho que deram a sua contribuição para a construção deste trabalho, direta ou indiretamente. Dentre estes, não posso deixar de agradecer por nome o meu orientador Luiz E. Buzato por ter encarado comigo este desafio, sempre com uma confiança inabalável em mim e em meu trabalho.

Outros me ajudaram sem me conhecer. Não vou parafrasear Newton pois não acredito que cheguei a enxergar tão longe assim, mas sem dúvida me apoiei no ombro de gigantes. Durante este trabalho tive a oportunidade de estudar o trabalho de grandes pesquisadores e ser influenciado por eles. Esta foi a melhor parte desta experiência e gostaria de ser capaz de retribuir esta ajuda deixando um pequeno apoio aos que virão depois de mim, começando nesta tese e continuando nos meus outros projetos de pesquisa.

Por fim, o mais importante, a minha família de perto e de longe. Eles foram o meu suporte e sempre me ajudaram em momentos cruciais, mesmo sem entender exatamente o que estava acontecendo. Agradeço à Candi pelo carinho, compreensão e paciência, em uma intensidade que somente uma pessoa genuinamente especial poderia dar. Agradeço também à ela e ao Vítor por criarem meu porto seguro, onde os problemas não existiam e que estava sempre me esperando quando as coisas não iam muito bem (e quando iam bem também). Agradeço aos meus pais e irmãos pelo carinho e pelas demonstrações frequentes de apoio e confiança, as coisas mais importantes para alguém que almeja terminar um projeto como este.

Conteúdo

Resumo	vii
Abstract	ix
Agradecimentos	xi
1 Introdução	1
1.1 Modelo Computacional	2
1.2 Replicação Síncrona	3
1.3 Replicação Ativa e Consenso	5
1.4 Paxos e Fast Paxos	7
1.5 Contribuições e Organização da Tese	11
1.6 Trabalhos Relacionados	13
2 An Object-Oriented Specification for Active Replication Using Consensus	19
2.1 Introduction	19
2.2 Treplica	21
2.2.1 Motivation	21
2.2.2 Overview	23
2.2.3 System Specification	24
2.3 An Object-Oriented Abstraction for Replication	26
2.3.1 Asynchronous Persistent Queue	27
2.3.2 Replicated State Machine	29
2.4 Treplica by Example	31
2.5 Treplica Implementation	36
2.5.1 Replicated State Machine Implementation	36
2.5.2 Paxos Persistent Queue	37
2.5.3 The Paxos Algorithm	38
2.5.4 Paxos and Replication	40
2.5.5 Treplica Software Architecture	42

2.5.6	Support Modules	43
2.5.7	Paxos Agents Modules	49
2.6	Applications	59
2.7	Performance	60
2.8	Related Work	61
2.9	Conclusion	64
3	Dynamic Content Web Applications: Crash, Failover, and Recovery Analysis	65
3.1	Introduction	65
3.2	Treplica	66
3.3	The TPC-W Benchmark	68
3.4	RobustStore	69
3.5	Evaluation	70
3.5.1	Method	70
3.5.2	Speedup	73
3.5.3	Scaleup	74
3.5.4	One crash, one autonomous recovery	75
3.5.5	Two crashes, autonomous recoveries	79
3.5.6	Two crashes, one autonomous, one delayed recovery	81
3.5.7	Discussion	83
3.6	Related Work	83
3.7	Conclusion	85
4	The Performance of Paxos and Fast Paxos	87
4.1	Introduction	87
4.2	Theory	88
4.2.1	Paxos and Fast Paxos	88
4.2.2	Performance Expectations	91
4.3	Practice	93
4.3.1	Treplica	93
4.3.2	Experimental Setup	94
4.3.3	Experiments	95
4.3.4	Scale Up	96
4.3.5	Speed Up	97
4.3.6	Quorum Sizes	98
4.3.7	Retries and Collisions	99
4.3.8	Failures	100
4.4	Related Work	102
4.5	Conclusion	102

5	On the Coordinator’s Rule for Fast Paxos	105
5.1	Introduction	105
5.1.1	Fast Paxos	106
5.2	Choosing Quorums	108
5.3	Coordinator’s Rule	109
5.4	Simplified Coordinator’s Rule	110
5.5	Conclusion	111
6	A Recovery Efficient Solution for the Replacement of Paxos Coordinators	113
6.1	Introduction	113
6.2	Paxos	115
6.2.1	Core Algorithm	116
6.2.2	Stable Memory Requirements	117
6.2.3	Liveness	118
6.3	Original Coordinator Validation	119
6.4	Seamless Coordinator Validation	120
6.4.1	Activation Procedure	121
6.4.2	Correctness	123
6.5	Experimental Evaluation	124
6.5.1	Method	124
6.5.2	Induced Failures	125
6.5.3	Intrinsic Failures	129
6.6	Fast Paxos	132
6.7	Related Work	133
6.8	Conclusion	134
7	Conclusão	137
7.1	Contribuições	137
7.2	Trabalhos Futuros	138
	Bibliografia	140

Lista de Tabelas

2.1	Parameters for congestion and flow control	54
3.1	One failure: performability	77
3.2	One failure: accuracy	79
3.3	Two overl. crashes: performability	79
3.4	Two overlapped crashes: accuracy	80
3.5	Delayed recovery: performability	82
3.6	Delayed recovery: accuracy	82
3.7	Paxos and Application Availability.	84
6.1	Average Performance of the Application under Induced Failures	129

Lista de Figuras

1.1	Paxos	9
1.2	Fast Paxos	10
2.1	Cluster Configurations for Replication	25
2.2	Software Architecture of an Application	26
2.3	Active Replication Components	26
2.4	The Hash Table Application	31
2.5	Paxos Persistent Queue	42
2.6	Speedup	61
2.7	One Failure: 5 Replicas	62
3.1	RobustStore components	69
3.2	Experimental setup	71
3.3	Speedup	74
3.4	Scaleup for 1000 WIPS	75
3.5	One failure: 5 replicas	77
3.6	One failure: recovery times	78
3.7	Two overlapped crashes	80
3.8	Delayed recovery	82
4.1	Scale up (2000 op/s)	96
4.2	Speedup (4 Replicas)	97
4.3	Speedup (8 Replicas)	98
4.4	Paxos with Large Quorums	99
4.5	Retries and Collisions	99
4.6	Single Failure (8 replicas, 2000 op/s)	101
6.1	Local View of an Agent	117
6.2	Global View as Observed by a Coordinator	122
6.3	Process Faultload	128
6.4	Network Faultload	129

6.5 Speedup (9 replicas) and Scaleup (3000 op/s) 131

Capítulo 1

Introdução

A construção de sistemas confiáveis a partir de componentes não-confiáveis é o grande objetivo de tolerância a falhas por software. O tema desta tese é a ferramenta fundamental para se conseguir este objetivo: a replicação de componentes de software. Especificamente, procuramos resolver o problema de como simplificar a construção de aplicações replicadas que combinem alto grau de disponibilidade e desempenho.

Disponibilidade e desempenho são objetivos que não necessitam de justificativa. Idealmente, gostaríamos que as aplicações nunca falhassem e pudessem atender a um número infinito de usuários. Mais difícil de caracterizar é o problema da simplificação da construção de aplicações disponíveis. A simplicidade de uma aplicação é uma propriedade de difícil quantização e, por consequência, de difícil verificação. No entanto, se o programador não consegue entender o ambiente de programação com qual trabalha, poucas são as chances de que a aplicação resultante seja altamente disponível.

Como ferramenta principal para alcançar o objetivo deste trabalho de pesquisa desenvolvemos Treplica, uma biblioteca de replicação voltada para construção de aplicações distribuídas, porém com semântica de aplicações centralizadas. Treplica apresenta ao programador uma interface simples baseada em uma especificação orientada a objetos de replicação ativa. Isolados atrás desta especificação residem vários mecanismos complexos que queremos esconder do programador, mas que são fundamentais para a eficiência da solução.

A conclusão que defendemos nesta tese é que é possível desenvolver um suporte modular e de uso simples para replicação que exibe alto desempenho, baixa latência e que permite recuperação eficiente em caso de falhas. Esta abordagem pode ser utilizada tanto em subsistemas específicos de grandes aplicações distribuídas [19] quanto em aplicações inteiras [20]. Aplicações personalizadas usadas internamente em empresas e instituições não possuem o mesmo porte de grandes aplicações de internet,

mas possuem requisitos de disponibilidade semelhantes. Na prática, estes sistemas são centralizados devido ao seu porte e à restrição de custos. Acreditamos que a arquitetura de software proposta tem aplicabilidade em qualquer sistema distribuído, mas é de especial interesse para estes sistemas que não são distribuídos pela ausência de uma forma simples, eficiente e confiável de replicá-los.

Esta tese está organizada na forma de uma coletânea de artigos, descrevendo os principais resultados da pesquisa. Nesta introdução fazemos um breve resumo sobre replicação em sistemas distribuídos e introduzimos a terminologia básica utilizada. Na Seção 1.1 definimos o modelo computacional adotado por Treplica. Nas Seções 1.2 e 1.3 apresentamos os conceitos básicos de replicação. Na Seção 1.5 listamos as contribuições desta tese e fornecemos um guia de leitura para o restante do texto. A Seção 1.6 discute sucintamente um conjunto de trabalhos relacionado a esta pesquisa.

1.1 Modelo Computacional

Todos os resultados desta tese supõem o modelo *assíncrono* de computação distribuída incrementado com detectores de falhas. Neste modelo um sistema distribuído é composto por processos que se comunicam exclusivamente por meio de mensagens enviadas por canais de comunicação. Vários processos podem residir em uma mesma máquina, mas eles são considerados autônomos e somente se comunicam através de troca de mensagens. Não existem memória compartilhada, um relógio global compartilhado ou um limite superior para o tempo de entrega de uma mensagem.

Este sistema assíncrono é sujeito a falhas. Os canais de comunicação podem falhar, atrasando, perdendo ou mudando a ordem das mensagens. Porém, as mensagens não são corrompidas e os canais de comunicação são justos. Ou seja, uma mensagem que seja enviada infinitas vezes será certamente recebida em algum momento. Os processos falham de acordo com o modelo *falha-e-recuperação*. Neste modelo processos falham apenas por colapso, se recuperam e voltam a operação normal. Durante estas falhas não são executadas ações que não estejam especificadas nos algoritmos e os dados armazenados em memória persistente sobrevivem às sucessivas falhas e recuperações. Estes dados estão disponíveis para o processo quando da sua volta a operação, mas todas as outras informações são perdidas.

É possível ter uma ideia aproximada do estado de um processo através de um serviço de detecção de falhas não confiável. Este serviço pode ser usado para descobrir se um processo está em operação normal ou se ele falhou e ainda não conseguiu se recuperar. Porém, as respostas do serviço de detecção de falhas não são perfeitas e o mesmo pode cometer vários erros. Ou seja, o detector de falhas pode indicar que

um processo em operação falhou e vice versa. Mesmo assim, este serviço é fundamental para a existência de um algoritmo de replicação correto em sistemas distribuídos assíncronos como veremos adiante.

Um modelo de falhas de processos alternativo é o modelo *falha-sem-recuperação*. Neste modelo, o processos falham apenas por colapso e nunca mais se recuperam durante a execução da computação. Não empregamos este modelo nesta tese, mas a grande maioria dos resultados teóricos em replicação usa este modelo como base.

1.2 Replicação Síncrona

Replicação é uma funcionalidade central para sistemas distribuídos tolerantes a falhas. A disponibilidade de dados replicados em vários processos aumenta o paralelismo e a confiabilidade do acesso a estes dados. Cada cópia da informação, chamada de *réplica*, pode potencialmente ser acessada concorrentemente e a falha do processador onde esta cópia reside não deve interromper o funcionamento das outras réplicas. Manter a consistência destas réplicas na presença de falhas nos processadores ou na rede é um problema difícil e intrincado.

Nesta tese estamos interessados no problema de replicação de dados usando apenas mecanismos de software implementados em hardware de prateleira. Os dados replicados sempre existem no contexto de uma aplicação que os utiliza. Mais precisamente, estamos interessados em replicar o *estado* de um processo de aplicação. Estes processos aceitam requisições de clientes e atendem a estas requisições consultando e/ou alterando o seu estado. Estas requisições são chamadas *operações*. As operações são procedimentos atômicos que executam completamente ou não são executados. Durante a execução de uma operação, o cliente que a requisitou fica bloqueado a espera de uma resposta. Por outro lado, o processo que a está executando pode executar várias operações ao mesmo tempo se possuir mais de uma *thread* de execução.

Existem duas formas principais de se classificar estratégias de replicação de acordo com a consistência dos dados replicados: síncrona e assíncrona [44]. Na replicação síncrona o cliente só observa a execução de sua operação quando possíveis atualizações foram ou serão garantidamente propagadas para todas as réplicas. Este modelo de replicação garante consistência forte dos dados, mas implica em tempos maiores para completar uma operação. Na replicação assíncrona o cliente pode observar o término de uma operação mesmo que nem todas as possíveis atualizações tenham sido propagadas. Neste modelo a consistência não é garantida, mas a operação se completa mais rapidamente.

Várias estratégias foram propostas para implementar replicação síncrona, mas as duas estratégias principais são *primário-backup* (*primary-backup*) e replicação ativa [45,

92]. No modelo primário-*backup*, também conhecido como mestre-escravo, uma réplica primária processa todos os pedidos de atualização dos dados replicados. Este processo primário executa a atualização localmente e propaga o estado resultante para o conjunto de réplicas *backup* passivas. Desta forma, a consistência da atualização dos dados é garantida pela unicidade da réplica primária. Caso o primário falhe, uma das réplicas escravas pode tomar o seu lugar. Na replicação ativa todas as réplicas executam as atualizações dos dados replicados. Neste modelo, supõe-se que cada réplica opera como uma máquina de estados determinista de forma que as réplicas se mantenham idênticas ao executar as mesmas operações. A consistência de atualização de dados é mantida por algum protocolo, executado pelas réplicas, que permita ordenar totalmente as requisições.

A replicação primário-*backup* tem como grande vantagem a simplicidade do conceito. Como apenas uma réplica executa as operações de atualização, o programador do servidor e do cliente pode considerar o sistema replicado como idêntico a um sistema centralizado. Desta forma, não é necessário cuidado especial ao se projetar a aplicação, permitindo inclusive o uso de operações não-deterministas. Esta simplicidade, no entanto, não se mantém quando é considerada a possibilidade de falha da réplica primária. Neste caso, será necessário detectar e tratar este evento o que aumenta consideravelmente a complexidade de implementação tanto do servidor quanto do cliente. Na replicação ativa, a complexidade de detecção e tratamento de falhas é embutida no protocolo de ordenação empregado, não trazendo complexidade extra ao programador. Desta forma, falhas são transparentes e o conjunto das réplicas continua provendo serviço enquanto um número mínimo destas continue funcionando. Por outro lado, o programador da aplicação deve garantir que as suas operações são deterministas, de forma a poderem ser re-executadas em todas as réplicas.

Em ambos os modelos de replicação síncrona existe um custo inerente à replicação. Na replicação primário-*backup* este custo é menor, correspondendo à transmissão de informações aos escravos e ao fato que estes não podem atender às requisições. Na replicação ativa este custo corresponde ao fato que todas as operações devem ser re-executadas em todas as réplicas. Independente da estratégia adotada, implementar replicação síncrona na presença de falhas exige cuidados. No caso de replicação primário-*backup* a falha do primário pode levar a uma situação onde apenas parte das réplicas escravas recebeu a atualização. No caso de replicação ativa, a falha de uma réplica pode fazer com que a mesma perca algumas atualizações de estado e fique inconsistente em relação às demais. A redundância embutida nos algoritmos de replicação que conseguem executar corretamente mesmo na presença de falhas de rede e de processos é responsável pela complexidade e custo destas soluções.

Uma forma interessante de se entender as implicações do custo da replicação síncrona em relação à replicação assíncrona é através do Teorema CAP [43]. Este teorema afirma que não é possível obter mais do que duas das seguintes propriedades ao mesmo tempo: consistência, disponibilidade ou tolerância a partições. Como a ocorrência de partições é uma realidade inevitável na prática devido a falhas de rede e de processos, temos que escolher entre consistência e disponibilidade.

Replicação síncrona faz a escolha por consistência, abrindo mão da disponibilidade em situações onde as réplicas não conseguem se coordenar. Replicação assíncrona escolhe a disponibilidade, com suporte a modelos de consistência relaxados que exigem reconciliação de dados. Desta forma, em sistemas modernos replicação síncrona é usada para guardar meta-dados [62], bloqueios (*locks*) [19] e outros dados cruciais ao funcionamento do sistema. Por sua vez, replicação assíncrona é usada para guardar o volume principal de dados [41]. Nesta tese estamos interessados apenas em replicação síncrona, pois este modelo de replicação mais se aproxima da semântica usual de uma aplicação centralizada.

1.3 Replicação Ativa e Consenso

Se o projetista do sistema deseja privilegiar a consistência dos dados usando replicação síncrona, a estratégia de replicação ativa possui uma série de vantagens. Este tipo de replicação permite uma consistência de dados equivalente a de uma única cópia centralizada enquanto garante que várias réplicas tenham acesso ativo aos dados. Por esta razão, esta estratégia tem sido utilizada com frequência em sistemas modernos [19, 48, 49, 62]. A replicação ativa é o tema principal desta tese e nesta seção detalharemos os seus conceitos fundamentais.

Replicação ativa (ou abordagem de máquina de estados) é uma estratégia de replicação onde a aplicação é modelada como uma máquina de estados determinista. As operações executadas pela aplicação correspondem a transições desta máquina de estados e os eventos que geram estas transições são difundidos, na mesma ordem, para todas as réplicas. O determinismo da máquina de estados garante que todas as réplicas, se possuírem o mesmo estado inicial, permanecerão idênticas à medida que as operações forem executadas. Replicação ativa foi proposta pela primeira vez por Lamport [55] e foi detalhadamente descrita por Schneider [79].

Temos então duas tarefas principais relacionadas à construção de um sistema que empregue replicação ativa: criação da máquina de estados e a difusão totalmente ordenada de operações. A modelagem de uma aplicação como uma máquina de estados determinista é um problema eminentemente centralizado, que tem como principal dificuldade a remoção de não-determinismo da aplicação. A difusão de operações

(mensagens) totalmente ordenadas para um grupo de réplicas é um problema bem mais complicado, especialmente em sistemas distribuídos assíncronos. Em particular, a difusão totalmente ordenada é equivalente ao problema de *consenso* nestes sistemas [28, 34] e este problema, por sua vez, é impossível de ser resolvido mesmo que somente um processo falhe [38].

Na prática, uma forma de se evitar a impossibilidade de resolução do problema de consenso consiste em mudarmos o modelo de sistema de assíncrono para parcialmente síncrono [34]. Vários protocolos de difusão ordenada foram propostos para diferentes modelos de sistema, que conseqüentemente oferecem diferentes garantias de confiabilidade. Défago et al [33] apresentam um estudo amplo sobre estes protocolos, com uma taxonomia dos mecanismos fundamentais usados em sua especificação.

Dentre esses mecanismos, os protocolos de consenso representam uma das abstrações mais interessantes. Consenso serve como base para a solução do problema da difusão totalmente ordenada devido a equivalência entre os dois problemas [28]. O que torna a abordagem de redução a consenso interessante é a sólida base teórica do problema de consenso, que inclui provas de correção, resultados de impossibilidade e algoritmos bem eficientes, mesmo com garantias fortes de entrega [33]. Um exemplo do tipo de fundação teórica que podemos encontrar no problema de consenso é o trabalho seminal de Chandra e Toueg em detectores de falhas [28]. Entre exemplos de protocolos de difusão total eficientes baseados em consenso podemos citar [18, 28, 58, 71, 72, 73].

Grande parte da literatura sobre consenso e difusão totalmente ordenada considera o modelo de falhas falha-sem-recuperação, onde um processo falha e nunca mais retorna à operação [33]. Dentro deste modelo, uma das soluções mais usadas para resolver estes problemas de replicação é usar um mecanismo de comunicação em grupo baseado em sincronia virtual. No modelo de sincronia virtual, um serviço de pertinência ao grupo (*group membership service*) define grupos dinâmicos onde processos podem entrar e sair do sistema, tanto explicitamente ou devido a uma falha [14]. Todos os processos de um grupo mantêm uma cópia local da lista de membros deste grupo, esta cópia local é a visão que o processo tem do grupo. A entrega de mensagens é regida por estas visões.

Em sistemas de comunicação de grupo, o serviço de pertinência ao grupo age como o mecanismo fundamental de tolerância a falhas, escondendo do programador a necessidade de monitorar e tratar a ocorrência de falhas. Todas as mensagens enviadas durante a duração de uma visão são recebidas por todos os processos incluídos nesta visão. Este serviço de entrega também provê, dentro de uma visão, várias garantias de entrega, incluindo ordenação total. No entanto, a pertinência ao grupo em sincronia virtual possui um procedimento de detecção e recuperação de falhas muito

rígido, como consequência do uso do modelo de falhas falha-sem-recuperação. Se um processo falha, ele é removido da computação e só pode retornar a mesma após a instalação de uma nova visão de processos ativos. Se o processo na verdade não falhou, mas foi erroneamente considerado falho, ele é forçado a desligar-se e a retornar ao grupo para garantir a consistência [14].

Esta rigidez na detecção e tratamento das falhas não é inerente à interface de programação de sincronia virtual, mas apenas às implementações atuais que usam o serviço de pertinência ao grupo como base de tolerância a falhas. Um conjunto de especificações para comunicação em grupo, com semântica idêntica a sincronia virtual, mas implementadas com base em uma redução para o problema de consenso são propostas por Schiper [78]. Uma grande vantagem desta abordagem é que a solução resultante é mais resistente a falhas do subsistema de detecção de falhas [83]. Uma solução baseada em pertinência ao grupo força processos a saírem e retornarem ao grupo quando sua falha é erroneamente detectada, diminuindo o desempenho. Uma solução baseada em consenso é capaz de distinguir falhas transientes do sistema de comunicação de falhas mais duradouras envolvendo subsistemas e processos, tomando a decisão de exclusão de um processo de um grupo com base na gerência de recursos do sistema [30]. Por fim, soluções baseadas em consenso possuem uma maior robustez, sendo capazes de progredir mesmo sob uma carga intensa e várias falhas de temporização [82].

Em comparação com o modelo de falha falha-sem-recuperação adotado em sincronia virtual, um modelo mais realista seria falha-e-recuperação. Aguilera et al. [4] mostram diferentes detectores de falhas e algoritmos de consenso neste modelo, com requisitos distintos de memória estável. Rodrigues e Raynal [76] apresentam um algoritmo de difusão totalmente ordenada no modelo falha-e-recuperação que trata consenso como um componente caixa preta e pode ser implementado com qualquer algoritmo de consenso falha-e-recuperação.

1.4 Paxos e Fast Paxos

O algoritmo Paxos é uma solução completa para replicação ativa usando consenso no modelo falha-e-recuperação [56]. Este algoritmo foi extensivamente estudado [16, 17, 59] e possui bom desempenho teórico [75]. Fast Paxos é uma variante de Paxos que melhora a latência de obtenção de consenso para apenas dois passos de comunicação [57]. Este é um algoritmo ótimo para o problema de consenso no modelo falha-e-recuperação [58]. Nesta seção faremos uma breve descrição destes dois algoritmos, focando no funcionamento de seus componentes principais. Descrições completas de ambos podem ser encontradas em [57].

Processos no sistema são agentes reativos que podem assumir vários papéis: um *proponente* (*proposer*) que pode propor valores, um *receptor* (*acceptor*) que escolhe um único valor ou um *aprendiz* (*learner*) que aprende o valor escolhido. Para resolver o consenso, agentes do Paxos executam várias *rodadas*, onde cada rodada possui um *coordenador* e é unicamente identificada por inteiro positivo, o *número de rodada*. Proponentes enviam a sua *proposta* para o coordenador que tenta alcançar consenso sobre ela em uma rodada. O coordenador é responsável por essa rodada e é capaz de decidir, após aplicar uma regra local, se outras rodadas tiveram sucesso ou não. A regra local do coordenador é baseada em quóruns de receptores e exige que pelo menos $\lfloor N/2 \rfloor + 1$ receptores façam parte de uma rodada, onde N é o número total de receptores no sistema [57].

Cada rodada acontece em duas fases, com dois passos cada, como ilustrado na Figura 1.1:

- Na Fase 1a o coordenador envia uma mensagem convidando todos os receptores a participar de uma rodada r . Um receptor aceita o convite apenas se ele não aceitou participar de uma rodada $s \geq r$, caso contrário ele ignora o convite.
- Na Fase 1b todo receptor que aceitou o convite responde ao coordenador a última proposta votada por este receptor e a rodada em que este voto ocorreu, ou *null* se ele nunca votou.
- Na Fase 2a, se o coordenador da rodada r recebeu respostas de um quórum de receptores, ele analisa o conjunto de respostas recebidas e escolhe uma proposta p que foi ou poderia ter sido decidida em rodadas com número menor que r . Ele então pede a estes receptores para votar nesta proposta, ou caso ela seja *null*, para votar em uma das propostas feitas pelos proponentes.
- Na Fase 2b, após receber um pedido para votar do coordenador, receptores votam na proposta sugerida se eles não votaram em nenhuma rodada $s \geq r$. Os receptores votam enviando o número de rodada e a proposta aos aprendizes.
- Finalmente, um aprendiz descobre que uma proposta p foi escolhida se ele recebe mensagens da Fase 2b de um quórum de receptores, todos votando em p na mesma rodada r .

Fast Paxos muda Paxos permitindo que os proponentes enviem as suas propostas diretamente aos receptores. Para conseguir isto, as rodadas são divididas em rodadas *rápidas* e rodadas *clássicas*. Os quóruns usados por Fast Paxos são maiores do que aqueles usados por Paxos e devem ter tamanhos apropriados para satisfazer os

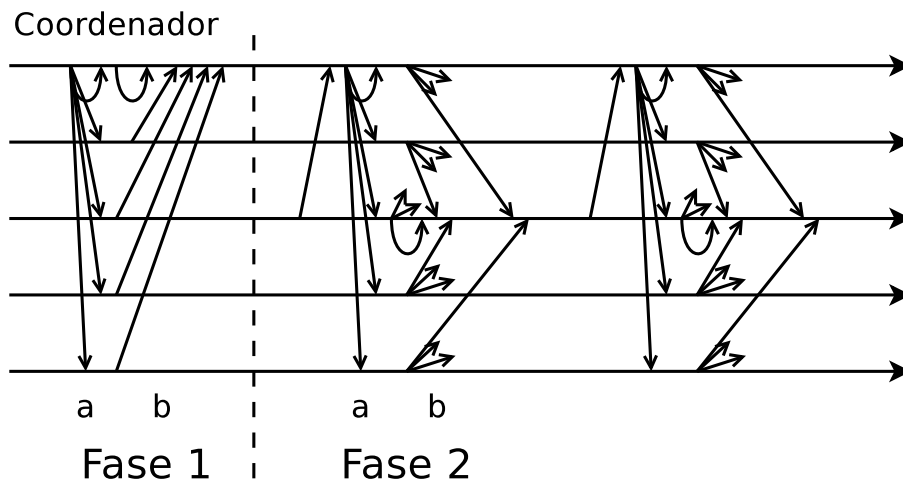


Figura 1.1: Paxos

requisitos da regra local. Especificamente, é possível minimizar o número de receptores em uma rodada rápida estipulando que tanto quóruns rápidos quanto clássicos contêm $\lfloor 2N/3 \rfloor + 1$ receptores [57, 86].

Uma rodada de Fast Paxos acontece de forma similar a uma rodada de Paxos, exceto que a Fase 2 é mudada, como ilustrado na Figura 1.2:

- Na Fase 2a, se o coordenador recebeu respostas de um quórum rápido de receptores indicando que nenhum deles já votou, ele instrui os proponentes a pedir diretamente aos receptores que votem em uma proposta de sua escolha.
- Na Fase 2b, após receber um pedido para votar feito por um dos proponentes, os receptores votam em uma proposta.

Esta descrição de ambos os algoritmos considera apenas uma única instância de consenso. No entanto, Paxos também define uma forma de entregar um conjunto de mensagens totalmente ordenadas. A ordem de entrega destas mensagens é determinada por uma sequência de inteiros positivos, tal que a cada inteiro corresponde uma *instância* de consenso. Cada instância i terá um valor decidido, que corresponde a i -ésima mensagem (ou conjunto ordenado de mensagens) a ser entregue na sequência de mensagens. Cada instância de consenso é independente das demais e várias instâncias podem estar em curso ao mesmo tempo. Para suportar o modelo de falhas falha-e-recuperação, ambos os algoritmos exigem que os agentes armazenem estado em memória não volátil [57]. Este estado é composto por um registro das instâncias iniciadas, os números de rodadas usados e as propostas feitas e votadas, entre outros dados.

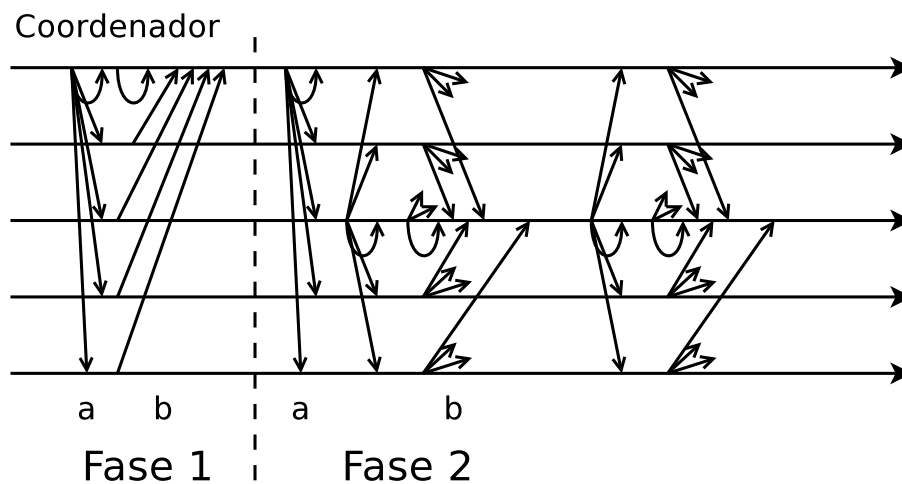


Figura 1.2: Fast Paxos

Em Paxos e Fast Paxos qualquer processo pode agir como o coordenador de uma rodada enquanto ele seguir a regra para escolher uma proposta coerente com o resultado das rodadas anteriores na Fase 2a. A escolha de coordenador e a decisão de iniciar uma nova rodada de consenso são feitas com base em algum mecanismo de temporização, uma vez que Paxos supõe um modelo computacional parcialmente síncrono para garantir *liveness*. Especificamente, a todo momento deve existir apenas um coordenador ativo para garantir que o algoritmo progrida. Se dois ou mais processos iniciam agentes coordenadores, o algoritmo pode travar enquanto estes múltiplos coordenadores competem pela atenção dos receptores com números de rodada que crescem rapidamente. Por esta razão, a *liveness* do algoritmo depende de um procedimento de seleção de coordenador. Este procedimento não precisa ser perfeito. A correção do algoritmo nunca é comprometida se zero ou mais coordenadores estiverem ativos ao mesmo tempo. Porém, o procedimento de seleção de coordenador deve ser robusto o suficiente para garantir que apenas um único coordenador esteja ativo a maior parte do tempo. Chamamos o processo de colocar um novo coordenador em operação de *validação de coordenador*.

Considerando a natureza concorrente das instâncias de consenso, uma otimização comum é feita durante a validação de um novo coordenador. A Fase 1 em Paxos e as Fases 1 e 2a em Fast Paxos são executadas apenas uma vez para todas as infinitas instâncias de consenso ainda não usadas. O coordenador de Paxos “guarda” estas instâncias para serem usadas no futuro, ou no caso de Fast Paxos, ele autoriza os proponentes a usarem estas instâncias. A melhoria gerada por esta fatoração de operações permite que Paxos alcance o consenso em apenas três rodadas de comunicação e Fast Paxos em apenas duas rodadas de comunicação, como ilustrados nas

Figuras 1.1 e 1.2. Infelizmente, Fast Paxos nem sempre pode ser rápido. Os proponentes podem propor valores diferentes de forma concorrente, causando uma colisão de suas propostas. Adicionalmente, falhas de processos e de rede podem impedir que uma rodada termine com sucesso. Vários mecanismos de recuperação podem ser empregados para tratar colisões e falhas, mas a intervenção do coordenador será necessária para iniciar mais uma rodada clássica [57].

Como Paxos oferece uma visão combinada de consenso e replicação ativa, ele fornece um bom desempenho por minimizar o número de abstrações necessárias. Esta simplicidade conceitual o coloca como solução ideal de replicação para os sistemas onde replicação consistente é necessária, porém sem perder desempenho. Esta é, por exemplo, a situação encontrada por sistemas replicados críticos dentro de aplicações distribuídas maiores [26].

1.5 Contribuições e Organização da Tese

Esta tese está organizada como uma coletânea de artigos, refletindo as principais contribuições do trabalho. Estas contribuições podem ser divididas em duas grandes áreas: a biblioteca Treplica e contribuições ao conhecimento do algoritmo Paxos e Fast Paxos.

A primeira parte trata da proposta de uma especificação de replicação ativa e a sua implementação na forma da biblioteca Treplica. Nesta parte, descrevemos uma forma simples de se construir aplicações replicadas e como este mecanismo foi implementado usando um algoritmo de consenso. Caracterizamos também o desempenho e a confiabilidade desta abordagem, justificando a sua aplicabilidade. Esta parte é composta pelos seguintes capítulos:

Capítulo 2: Este capítulo é composto pelo artigo *“Implementation of an Object-Oriented Specification for Active Replication Using Consensus”* [89]. Este artigo descreve a nossa proposta de uma especificação orientada a objetos para replicação e a sua implementação na forma da biblioteca Treplica. Neste trabalho é descrita a arquitetura de software de Treplica, as suas principais decisões de projeto e os problemas que as motivaram. Por completude, é feita uma breve análise de desempenho de Treplica que corresponde a um sub-conjunto dos dados apresentados no Capítulo 3. O leitor pode pular esta análise (Seção 2.7) em uma leitura completa da tese. Uma primeira versão deste artigo foi publicado nos anais do 26º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2008), Rio de Janeiro, Brasil [87]. A versão presente nesta tese é uma versão substancialmente estendida, publicada como Relatório Técnico IC-

10-26 do Instituto de Computação da Unicamp, que está atualmente em fase final de preparo para submissão para avaliação no periódico *Software: Practice and Experience*.

Capítulo 3: Este capítulo é composto pelo artigo “*Dynamic Content Web Applications: Crash, Failover, and Recovery Analysis*” [20]. Este artigo mostra como falhas e recuperações afetam o desempenho de um aplicação Web implementada com Treplica. O desempenho é medido usando-se o *benchmark* TPC-W, aumentado com medidas de disponibilidade. Os resultados obtidos mostraram um bom desempenho, excelente escalabilidade e disponibilidade ininterrupta. Este artigo foi publicado nos anais da *39th International Conference on Dependable Systems and Networks (DSN 2009)*, Estoril, Portugal (doi:10.1109/DSN.2009.5270331).

A segunda parte trata das contribuições que fizemos ao conhecimento do algoritmo Paxos e Fast Paxos. Nesta parte descrevemos três contribuições principais, que incluem a caracterização do desempenho de Paxos e Fast Paxos, uma regra de consistência simplificada para Fast Paxos e um procedimento otimizado de substituição de coordenador para Paxos. Esta parte é composta pelos seguintes capítulos:

Capítulo 4: Este capítulo é composto pelo artigo “*The Performance of Paxos and Fast Paxos*” [88]. Este artigo faz a caracterização do desempenho dos algoritmos Paxos e Fast Paxos, os comparando em situações com e sem falhas. A avaliação de desempenho foi realizada em uma ambiente de LAN e pudemos observar que Paxos teve um melhor desempenho que Fast Paxos. Mais interessante foi a observação que a violação da suposição otimista do Fast Paxos não foi a causa desta diferença, mas sim violações de temporização. Este artigo foi publicado nos anais do 27º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2009), Recife, Brasil.

Capítulo 5: Este capítulo é composto pelo artigo “*On the Coordinator’s Rule for Fast Paxos*” [86]. Este artigo mostra uma análise da implementação da regra de consistência do algoritmo Fast Paxos como implementada pelo processo coordenador em função do número de processos no quórum. Com base nesta análise, é proposta uma regra simplificada de consistência interessante para implementação. Este artigo foi publicado no periódico *Information Processing Letters*, volume 107, 2008 (doi:10.1016/j.ipl.2008.03.001).

Capítulo 6: Este capítulo é composto pelo artigo “*A Recovery Efficient Solution for the Replacement of Paxos Coordinators*” [90]. Este artigo mostra uma otimização do

procedimento de substituição de um coordenador no algoritmo Paxos. O coordenador é um agente fundamental para o funcionamento deste algoritmo e a sua substituição pode levar o sistema a parar. Este trabalho apresenta um procedimento de substituição de coordenador que provoca o mínimo de perturbação nas operações do sistema. Uma observação interessante é que mesmo na ausência de falhas de processos, a coordenação troca constantemente em sistemas sobrecarregados o que torna esta otimização interessante. O artigo presente nesta tese é uma versão revisada do Relatório Técnico IC-10-13 do Instituto de Computação da Unicamp, submetida para avaliação no periódico *Transactions on Parallel and Distributed Systems*.

Além destes artigos, podemos citar um outro trabalho desenvolvido durante a pesquisa de tese mas que não se encaixa na presente organização da mesma. O artigo "*Evaluation of a Read-Optimized Database for Dynamic Web Applications*" [80] investiga o uso de um banco de dados especializado para sistemas de *data warehousing* como suporte para construção de aplicações Web. Este artigo foi publicado nos anais da *Fourth International Conference on Web Information Systems and Technologies (WEBIST 2008)*, Volume 1, Funchal, Portugal.

O Capítulo final da tese resume as contribuições apresentadas e mostra alguns trabalhos futuros.

1.6 Trabalhos Relacionados

A idéia de se armazenar os dados na memória principal, utilizando um registro persistente de operações como um mecanismo de tolerância a falhas, é descrita por Birrell et al. [15]. A API atual do Treplica foi influenciada por Prevayler [95], especificamente pelo uso que faz de características de linguagens de programação modernas como Java e C# para simplificar a implementação de mecanismos complexo e fornecer uma API simples. Em comparação a estes dois sistemas centralizados, Treplica estende esta abordagem de persistência baseada em registro de operações como base para replicação.

Replicação de dados com alto nível de consistência e desempenho tem sido usada frequentemente em mecanismos de controle de sistemas distribuídos de grande escala [19, 48, 49]. Estes sistemas exigem alguma forma de controlar a operação de um número muito grande de processadores e serviços da forma mais autônoma possível. Estes mecanismos apresentam uma abstração de programação baseada em bloqueios, além de servir como repositório consistente de configurações. Como o Treplica, muitos destes sistemas utilizam o algoritmo Paxos para implementar replicação.

O sistema de bloqueios Chubby é usado pela Google para controlar vários de seus serviços [19]. Apesar de usar uma abstração de programação baseada em bloqueios, Chubby possui várias características estruturais similares ao Treplica, incluindo um “registro tolerante a falhas” replicado usando Paxos, bem similar a uma fila persistente [26]. Os autores argumentam que esta abstração é interessante e que pretendem usar esta abstração para criar outros sistemas replicados no Google [26]. Chubby é uma aplicação dedicada à gerência de bloqueios distribuídos e não exporta o seu estado replicado como um serviço para outras aplicações. O Treplica, por sua vez, fornece apenas o serviço de estado replicado, persistente e disponível, sendo uma ferramenta que pode ser usada para a construção de serviços de bloqueio similares ou de outras aplicações especializadas. Chubby usa o algoritmo Paxos clássico para implementar replicação, enquanto Treplica usa os algoritmos Paxos e Fast Paxos.

Um outro sistema de bloqueio distribuído é o FaTLease [48], parte do sistema de arquivos orientado a objetos XtremFS [47]. Como este sistema de arquivos utiliza bloqueios com uma granularidade bem pequena, o FaTLease deve ser bastante eficiente. Para alcançar este objetivo o sistema emprega uma variante do algoritmo Paxos otimizada para manter bloqueios com prazo de validade (*leases*). A otimização consiste em operar apenas em memória principal, sem fazer uso da custosa memória secundária. O algoritmo Paxos exige memória persistente para garantir a consistência, mas FaTLease se esquivava desta exigência utilizando a validade dos bloqueios por ele gerenciados. Como todo bloqueio tem um prazo de validade, existe um momento no futuro onde todos os bloqueios não são mais relevantes. No FaTLease uma réplica que falha só retorna ao sistema após esperar tempo suficiente para que todos os bloqueios que ela poderia ter tomado conhecimento percam a validade. Desta forma, esta réplica não precisa preservar estado algum. Como Chubby, FaTLease é uma aplicação específica para o problema de coordenação através de bloqueios de sistemas distribuídos. Treplica por sua vez tem maior aplicabilidade, pois é uma biblioteca de uso geral que pode construir outros tipos de aplicações. Treplica exige também dois acessos a memória secundária para cada operação ordenada pelo sistema, o que implica em um custo maior que o FaTLease. No entanto, a estratégia de recuperação do FaTLease possui um impacto na disponibilidade do sistema devido ao período que uma réplica deve esperar para se recuperar. Treplica, por sua vez, garante operação continuada, com perda mínima de desempenho em caso de falha.

Zookeeper¹ e Autopilot [49] são sistemas completos de coordenação de aglomerado, semelhantes ao Chubby mas com mais funcionalidades. Zookeeper possui um conjunto simples de primitivas para sincronização, configuração, manutenção e resolução de nomes. Ele fornece também um serviço de monitoramento deste sistema

¹<http://hadoop.apache.org/zookeeper/>

de arquivos que pode ser usado como forma de comunicação e notificação. Autopilot controla todos os aspectos de um aglomerado, como provisionamento de novos serviços, instalação de aplicações, monitoramento e correção de erros e coordenação. Ambos os sistemas são usados como ponto de controle central das aplicações distribuídas em um aglomerado. Logo, eles são replicados para garantir operação ininterrupta destas aplicações em caso de falhas. Estes sistemas, de forma geral, provêm um serviço mais especializado que o Treplica. Mesmo assim, é interessante observar que os projetistas destes sistemas, devido a sua centralidade no aglomerado, decidiram usar um mecanismo baseado em consenso para replicar os dados cruciais, e decidiram manter estes dados em memória principal usando a memória secundária para tolerância a falhas. Estas são algumas das decisões de projeto adotadas no Treplica e podemos afirmar que o Treplica poderia ser uma ferramenta adequada a construção de sistemas de coordenação similares.

Uma alternativa ao uso de bloqueios distribuídos são as transações distribuídas. Camargos et al. [23] apresentam um sistema de terminação de transações distribuídas baseado em um registro persistente dos votos para abortar ou concluir a transação. A especificação proposta abstrai detalhes sobre a implementação do serviço de registro distribuído. Além disso, cada participante do sistema pode contar com o registro para preservar os dados de suas transações e não precisa se preocupar com a persistência de seus dados locais. Os autores fornecem duas implementações deste serviço, ambas baseadas em consenso. A abstração de serviço de registro é similar a abstração de filas persistentes usada pelo Treplica. Ambas soluções fornecem um registro persistente e replicado de dados ordenados, permitindo que as aplicações usem este registro para manter a sua consistência. Treplica tem aplicação mais geral, já que não define semântica dos dados transportados e não é restrito à gerência de transações. O serviço de registro persistente por sua vez roda exclusivamente um algoritmo de terminação de transações.

Boxwood [62] é um arcabouço para a construção de aplicações de armazenamento de dados distribuídas. Os criadores deste arcabouço defendem o uso de estruturas de dados genéricas e de mais alto nível como fundação para a construção de sistemas distribuídos complexos. Uma das abstrações propostas consiste em um serviço de consenso genérico usando Paxos. Este módulo é usado por vários componentes do Boxwood, incluindo o seu gerente de bloqueios distribuídos. No entanto, Boxwood é centrado em um domínio específico de aplicação (armazenagem de dados distribuídos) e provê uma interface de baixo nível aos seus serviços, enquanto o Treplica oferece uma interface de programação de mais alto nível.

Na literatura há vários trabalhos que apontam de um lado a simplicidade conceitual de Paxos e de outro lado a sua complexidade de implementação; o algoritmo

exige que o projetista defina vários aspectos que foram propositadamente deixados de lado em sua especificação teórica [16, 26, 59]. Como parte do trabalho de construção do Treplica alguns destes aspectos são levantadas e definidos. Um outro trabalho direcionando exclusivamente a descrição detalhada de uma implementação de Paxos pode ser encontrado em [7]. Este artigo descreve todos os aspectos de um sistema completo de replicação ativa usando Paxos, incluindo um estudo bem completo do desempenho desta implementação. Nesta descrição são descritos mecanismos de controle de fluxo e de congestionamento, procedimento de eleição de coordenador e outros aspectos de implementação que não são usualmente detalhados. Os autores observaram que estes mecanismos são considerados como apenas questões de engenharia, mas podem afetar o algoritmo, especialmente seus requisitos de liveness. Nesta tese propomos uma abstração para programação de replicação ativa que inclui requisitos de persistência e apresentamos uma implementação desta proposta. Desta forma, Paxos possui uma posição central no desenvolvimento deste texto, mas o trabalho cobre uma gama de assuntos mais ampla.

Sistemas de comunicação em grupo fornecem um conjunto de primitivas que podem ser usadas na construção de aplicações replicadas. Entre as ferramentas de maior sucesso podemos citar Isis [12], Totem [68] e Transis [6]. Isis introduziu muitas das idéias que influenciaram intensamente as ferramentas que o sucederam, incluindo o modelo de programação baseado em sincronia virtual [13, 14, 39]. Horus [85] foi o sucessor de Isis e introduziu uma arquitetura de software altamente modular e uma versão melhorada de sincronia virtual. Ensemble [84] e JGroups [9] são reimplementações de Horus em ML e Java, respectivamente, e ambos são ativamente mantidos e usados. Outras ferramentas de comunicação em grupo mais recentes, também ativamente mantidas e usadas, são Spread [5] e Appia [67]. Devido à sua intenção de combinar replicação e persistência, Treplica se distancia da organização tradicional destes sistemas ao procurar adotar consenso como modulo fundamental para construir aplicações replicadas.

A abstração de filas persistentes é bem similar ao padrão *publish/subscribe* de comunicação de grupos de processos implementado em *middleware* orientados a mensagens (*message oriented middleware*; MOM) [10]. A troca de mensagens em MOM é assíncrona, com a garantia que até processos defeituosos podem esperar receber todas as mensagens enviadas, na mesma ordem vista pelos outros processos. Além de difusão de mensagens, MOM permitem a construção de grafos elaborados para o fluxo de mensagens e muitos executam conversões de formato destas mensagens enquanto as mesmas são transportadas neste grafo. Como exemplos de MOM podemos citar

os produtos IBM WebSphere MQ² e Apache ActiveMQ³. Estes sistemas são bem mais pesados comparados com Treplica e são usualmente implementado sobre sistemas de banco de dados centralizados, herdando destes sistemas o seu comportamento de falhas. Treplica também é projetado para processos mais fortemente acoplados e não provê fluxo explícito de mensagens nem conversões de formato.

Um domínio de aplicação que usa replicação de forma muito intensa é o de banco de dados. O uso de replicação em banco de dados possui uma história longa, marcada por técnicas e terminologia próprias [92]. No contexto desta tese, estas estratégias de replicação são relevantes porque existe uma significativa sobreposição de conceitos entre a visão transacional típica de bancos de dados replicados e a invocação remota de objetos distribuídos replicados [92].

Em banco de dados, replicação é usada primariamente como técnica de tolerância a falhas. Servidores replicados são amplamente usados para proteger contra falhas catastróficas, preservando o oferecimento de serviço. Raramente dados replicados são usados como ferramenta para aumentar o desempenho do sistema, pelo contrário, replicação é sempre vista como uma fonte de sobrecarga, necessária para a tolerância a falhas [54]. Estudos abrangentes das estratégias de replicação adotadas em bancos de dados podem ser encontradas nos trabalhos de Wiesmann et al. [91, 92].

As primeiras iniciativas de pesquisa em replicação de dados síncrona em bancos de dados foram baseadas em bloqueios distribuídos [11] ou algoritmos de quórum [2, 42]. Trabalhos posteriores propuseram o uso de difusão totalmente ordenada como uma alternativa a bloqueios para a implementação de replicação síncrona [3, 46, 54, 70]. O estudo de Wiesmann e Schiper [93] descreve estes protocolos e analisa o seu desempenho. Uma das técnicas mais interessantes utiliza o conceito de certificação de transações [70]. Nesta abordagem todas as escritas de uma transação são adiadas até o momento do *commit*. Neste ponto, o conjunto de escritas é difundido para as outras réplicas que validam a possibilidade de efetuar o *commit* usando o princípio de isolamento de *snapshot*. Todas as réplicas recebem os conjuntos de escritas na mesma ordem e o processo de certificação é determinista, logo todas tomarão as mesmas decisões sem necessidade de coordenação.

Implementações que usam difusão totalmente ordenada para replicar banco de dados podem ser encontradas nos sistemas de pesquisa Postgres-R [53], Tashkent [36] e Tashkent+ [37]. No entanto, pouco destas e outras pesquisas está sendo efetivamente aplicada em sistemas gerenciadores de banco de dados comerciais. Uma razão para isto é que adaptar um sistema de banco de dados existente para suportar replicação é uma tarefa muito complexa, levando-se em conta a quantidade de recursos já in-

²<http://www-306.ibm.com/software/integration/wmq/>

³<http://activemq.apache.org/>

cluída em tais sistemas. Alguns pesquisadores propuseram uma solução para este problema empregando *middleware*. Nestas abordagens os banco de dados convencionais são executados sem alterações e são coordenados por uma camada de replicação implementada como um adaptador de serviço. Dois exemplos desta abordagem são Sequoia/C-JDBC [25] e Ganymed [74]. Ambos implementam a camada de abstração de conectividade de banco de dados JDBC.

Uma forma comum de se fazer persistência e replicação de dados consiste em usar bancos de dados replicados como repositório de dados e acessá-los usando os mecanismos de consultas usuais, como o SQL. De forma oposta ao Treplica, estes sistemas provêem uma solução pesada e custosa para o problema de replicação. Desta forma, não são muito úteis como blocos fundamentais para a construção de sistemas distribuídos confiáveis. Consideramos o Treplica como uma solução superior nestes casos por oferecer uma interface mais enxuta de programação e por apresentar alto desempenho, como argumentaremos nesta tese.

Capítulo 2

An Object-Oriented Specification for Active Replication Using Consensus

Most of the software tools created so far to aid in the construction of distributed applications addressed how to replicate data consistently in the presence of failures, but without offering much relief for the problem of building a dependable and long-running application. This paper describes our experience building Treplica, a replication toolkit offering application developers a very simple programming model based on an object-oriented specification for replication of durable applications. Treplica simplifies the development of high-available applications by making transparent the complexities of dealing with replication of data that must survive process crashes and recoveries. These complexities are not negligible, and we believe we have found a compelling way to address this problem under a simple-to-understand object-oriented interface. We have used Treplica successfully to add fault tolerance to a implementation of the TPC-W benchmark and we have obtained very good performance, even in the presence of failures.

2.1 Introduction

For more than three decades system developers have pursued the goal of connecting off-the-shelf computers together using standard network resources to obtain a system with better availability than any of its individual parts. The system obtained this way has greater availability because it contains sub-systems to spare. For example, each computer of the system can contain a copy of some critical process so that partial computer failures are guaranteed not to make the system or the application it hosts unavailable to its users. Unfortunately, the full potential of redundancy and replication can only be successfully harnessed if three main obstacles are overcome: (i)

performance, (ii) availability despite component failures and (iii) programming simplicity.

These three seemingly simple goals are very hard to reach in practice due to asynchronous nature of distributed systems. Nonetheless, many solutions exist for replicating data and services with many different suppositions regarding system models, replication guarantees and application behavior [45]. However, with the exception of data intensive solutions for relational databases, few solutions tackle the problem of managing replication of applications whose services and data must be always available for long periods of time. The designer of this class of distributed applications faces the daunting task of maintaining consistency in the presence of unpredictable failures and concurrency.

This paper discusses our experience in building and using Treplica, a replication library that overcomes (i) by implementing consensus-based active replication and (ii) by offering the application developers a very simple programming model based on an object-oriented specification for replication. Treplica has been designed to be resilient, transparent and efficient. Resiliency means that Treplica implements at its core a replication protocol that gives applications the ability of tolerating crashes and recoveries of a subset of their replicated components without having to worry about the consistency of the replicated state. Treplica guarantees resiliency through consensus-based active replication, specifically the Paxos [56] algorithm. Transparency guarantees that programmers can develop replicated distributed applications without having to be concerned about how replication is actually implemented. In fact, application programmers can program their stateful applications as a set of stateless objects. Concerning efficiency, experimental results show that Treplica can provide the necessary processing capacity to guarantee very good application response times.

In summary, Treplica simplifies the development of high-available applications by making transparent many of the complexities related to consistent replication and recovery in the presence of failures. These complexities are not negligible; care must be taken to correctly implement the replication algorithms, detect and manage failure, perform recovery, among other issues [7, 26]. We believe we have found a compelling way of factoring out these concerns under a simple-to-understand programming interface. Treplica stands in the middle ground between the low-level flexibility of message-based group communication toolkits and the extensive data processing capabilities of databases. The main contributions of this work are:

- The design and implementation of an object-oriented abstraction for replication as a way to simplify the construction of dependable and long-lived applications.
- The use of consensus as a foundation for the implementation of this modular

abstraction.

- The description of the software architecture of *Treplica* accompanied by a detailed discussion of our design choices and the problems that motivated them.
- The experimental validation of *Treplica*'s performance and availability. *Treplica* shows good performance and uninterrupted service, even with multiple failures.

The remaining of this paper is structured as follows. Section 2.2 gives an overview of *Treplica*, goes in more depth in the rationale for its creation and outlines its software architecture. Section 2.3 describes the object-oriented specification for replication while Section 2.4 gives an example of how this specification is used in *Treplica* to build a complete application. Section 2.5 goes into more detail on the internal structure of *Treplica*, describing our implementation of the Paxos protocol. Section 2.6 briefly describes the typical profile of applications built with *Treplica* and Section 2.7 shows the performance attained by one such application, the TPC-W benchmark. Section 2.8 discusses related work and Section 2.9 makes some concluding remarks.

2.2 *Treplica*

2.2.1 Motivation

Replication is a crucial mechanism used in distributed systems to increase the system reliability and performance. Active replication is a general technique to replicate the internal state of processes that prioritizes consistency [79]. In active replication all processes sharing the same state, called *replicas*, behave as deterministic state machines. All replicas share the same source of events that trigger transitions in their underlying state machine. As a consequence, all replicas stay the same as long as they process the same sequence of events.

There are many forms of implementing active replication. One of the more common is to employ a total order broadcast primitive to propagate events orderly among the replicas [33]. As an example, take the usual situation of a set of servers providing service to a set of clients. In this scenario, a client can broadcast its request to the set of servers using the total order broadcast. All the servers will observe the request at the same position in their events sequence and will perform the same operation, yielding the same result. The client picks the first answer it receives. The conceptual simplicity of active replication over a total order broadcast primitive makes it a very used solution in practice.

However, it is necessary to consider the relationship between the properties of the total order broadcast primitive and of the application being developed. In particular, it is necessary to establish how the state of the total order broadcast relates to any local state maintained by the application, specially in the presence of failures. By definition persistent data survives failures, thus any information transmitted by the total order broadcast primitive that do not survive failures is a potential source of inconsistency for the application. To illustrate this point we take as an example the more mature way to implement total order broadcast: virtual synchrony-based group communication.

In the virtual synchrony model, message delivery is constrained by views of operational processes maintained by a group membership service [14]. This group membership service supports dynamic groups where processes join and leave the system, either explicitly or due to a failure. This service acts as the basic fault tolerance mechanism, hiding from the programmer the need to monitor the occurrence of failures. All messages sent during the lifetime of a view are received by all processes encompassed by it, and all message delivery guarantees such as total order are enforced. However, group membership in virtual synchrony assumes a crash-no-recovery failure model. If a process fails, it can only rejoin the computation when a new view is instated. If the failure of a process is wrongly detected, the process is forced to shutdown and rejoin the group to guarantee view consistency [14].

Whenever a process joins a group, creating a new view, it is assumed this is the first time this process is seen by the group. That is, there is no explicitly defined rejoin operation. Processes are assumed to be stateless and they must catch up with the group state by means of a state transfer from another process in the group. This behavior directly affects the type of failures supported by the application. Take for example a distributed application where all replicas reside in the same cluster. If it is possible to guarantee the whole set of replicas never crashes completely, one can use the shared state maintained in the main-memory of these replicas to preserve the application state. However, if one must tolerate whole cluster failures, some stable storage must be used.

Specifically, each replica must store and update its complete state in disk to account for the situation it is the last one to fail in the cluster. During recovery of a failed replica, it runs a protocol to determine if it is joining an existing group or creating a new group. If it is joining, it should discard all its local state and restart from the state currently held by the replicas in the group. Thus, all processes use costly stable memory, but it is only necessary by a single process in the less likely event of a total crash, instead of the more common occurrence of a partial crash. One can circumvent this basic behavior by creating an application specific protocol that

makes the state transfer more efficient. This can be accomplished by using as much as possible the local persistent state held by a replica to complete the state held by a view of processes. However, it rests on the programmer the hard task of designing and implementing this protocol. Moreover, if a replica is *wrongly* suspected of having failed, it still must restart its operation and discard all its local state.

Although we have used virtual synchrony as an example, the problem just described comes from the necessity of synchronizing total order delivery state and application state. In fact, all properties of the total order primitive being used and of the resulting application must be cautiously matched, considering consistency suppositions, failure models and other aspects. This way, the lower level details contaminates all the upper layers including the programming abstraction, making the task of the application developer much harder.

2.2.2 Overview

Treplica is a replication library designed to provide a simple and object-oriented way to build highly available applications. These applications can encompass the entire system or be restricted to crucial sub-systems where performance, consistency and reliability are central. To reach this goal, we decomposed the problem of implementing replication in components with simple and clearly defined interfaces. So, a developer who wants to implement a replicated distributed application does not reason in terms of messages, processes, failures or data items. Instead, he reasons about the execution of the application operations, transitions of a *replicated state machine*, that are triggered by events that are made available through an *asynchronous persistent queue*. Treplica is an implementation of this object-oriented specification for replication.

We decided to expose the state machine component to the developer as a programming tool using the reflection facilities of modern languages to encode and execute state and state transitions. Using state machines as a concrete programming interface is desirable because states and transitions are easily implemented as objects. Treplica is implemented in Java, and in this language the application state is represented by serializable objects and actions as runnable, serializable objects. The object-oriented specification embodied by Treplica can easily be implemented in any other dynamic language and, with some extra programming effort, in more traditional languages such as C.

The main design decision underlying Treplica is to allow the programmer to consider the application as being stateless, leaving the actual durability of the application to the library. This decision is supported by the observation that the same requirements of active replication can be used to provide a simple but powerful persistence

mechanism. Active replication requires the application to perform actions that change its state in a deterministic way. These actions are then broadcast, in the same order, to all replicas that locally replay them. Within this same framework, we consider that the actions aren't only sent to the other replicas but logged to stable storage [15]; this way it is possible to recover from failures by replaying the log. Determinism ensures that after each recovery the application will restart in the same state it was before the failure. For efficiency and ease of implementation, we require that the application fit in main-memory, as we do not provide any means of selectively unloading parts of the application state to secondary memory. With the current size and cost of main-memory, we don't consider this limitation to be a problem for the class of applications that can benefit from using Treplica.

To support active replication in Treplica, we have decided to concentrate on consensus-based total order algorithms for the crash-recovery failure model. The Paxos algorithm and its variants are examples of specially suited algorithm of this class, as it was created with active replication in mind. These algorithms are particularly interesting because they provide the continuous delivery of messages to a replica even in the presence of failures and recoveries. This allows Treplica to have a simpler software architecture and increases its potential for good responsiveness in the presence of partial failures. Moreover, these extra guarantees allow Treplica to avoid expensive coordination of the local application state and the shared state during recovery. Obviously, relying on stronger guarantees implies a larger cost to deliver messages. However, for this class of algorithms, this cost is related to writes to stable memory and these writes are already required to ensure the application can survive catastrophic failures. By combining the stable memory requirements of the application and the total order primitive we were able to obtain a good failure-free performance that is minimally affected by the occurrence of faults.

2.2.3 System Specification

The target platform for Treplica are commodity clusters. The main characteristic of such clusters is that the nodes are connected by a high bandwidth and low latency interconnect network that supports broadcast. The replicas exchange messages through this network to keep the shared state consistent while potentially serving client requests. Throughout this paper, *client* is any process that does not have a copy of the replicated state. Only replicas hold the replicated state and only them are able to use Treplica services to query and change this state. Clients depend on the replicas, that act as servers, to perform these actions in their behalf. In fact, clients often interact with a higher level abstraction provided by the replicas and are unaware of the exist-

tence of the replicated state. We call this higher level view of the set of replicas an *application*.

Treplica does not restrict how the clients access the application or how their access is load balanced among the replicas. The application is free to implement its service in many ways, as long as the guarantees provided by Treplica are sufficient. For example, it can serve remote clients using a RPC mechanism, it can implement a web service, it can serve local clients through sockets, etc. Treplica does not dictate or implement any such mechanism, leaving the designer free to choose the more appropriate solution for a particular application. Figure 2.1 shows two examples of possible cluster configurations. Figure 2.1(a) shows a setup where remote clients connect to replicas in a cluster mediated by a load balancer acting as a reverse proxy. Figure 2.1(b) shows a group of clients that share the cluster with the replicas and access an application elected master.

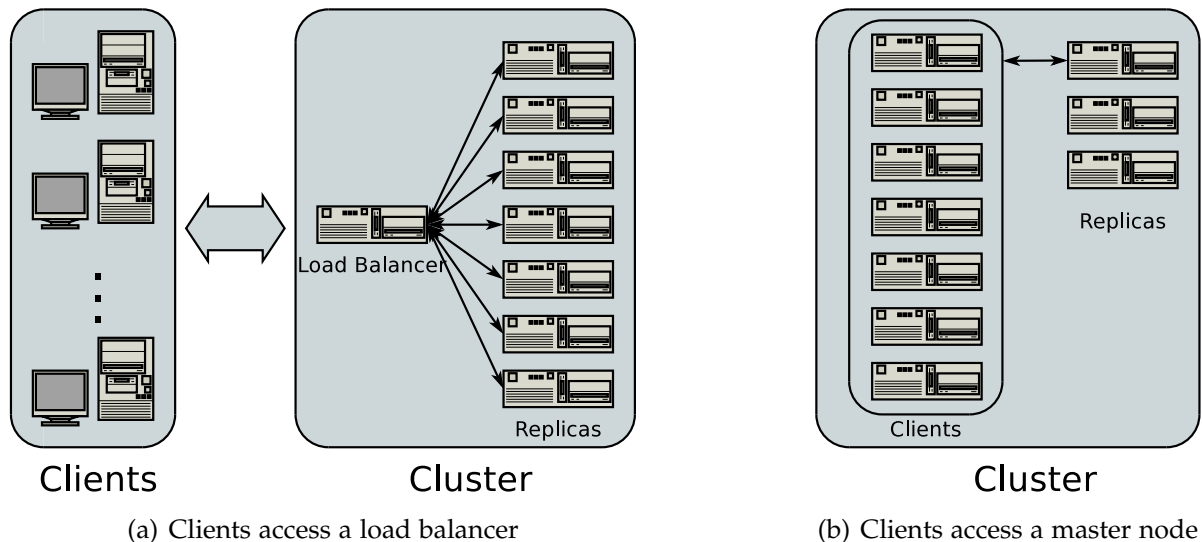


Figure 2.1: Cluster Configurations for Replication

The replicated application state is left under the control of Treplica. This way the application programmer should not be concerned with replica management or fault-tolerance implementation details, as shown in Figure 2.2. For simplicity of implementation and performance, the entire replicated state must fit in main-memory. However, this isn't an intrinsic property of the design, only a characteristic of the current implementation. More importantly, the application state must change only in a deterministic and controlled way to accommodate active replication.

The architectural restrictions imposed by Treplica affect only the replicated state. Usually, information kept by the application that only regards the local status of the

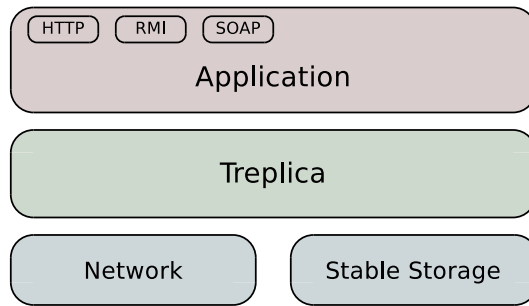


Figura 2.2: Software Architecture of an Application

connections with its clients are not replicated and are kept in local volatile memory. Moreover, any data kept by the application that does not require replication or persistence can be stored in any way required by the application designer.

2.3 An Object-Oriented Abstraction for Replication

Our proposal of an object-oriented abstraction for replication is based on two main components: *replicated state machine* and *asynchronous persistent queue*. Figure 2.3 shows the interface of these components and their relation to the application and to each other.

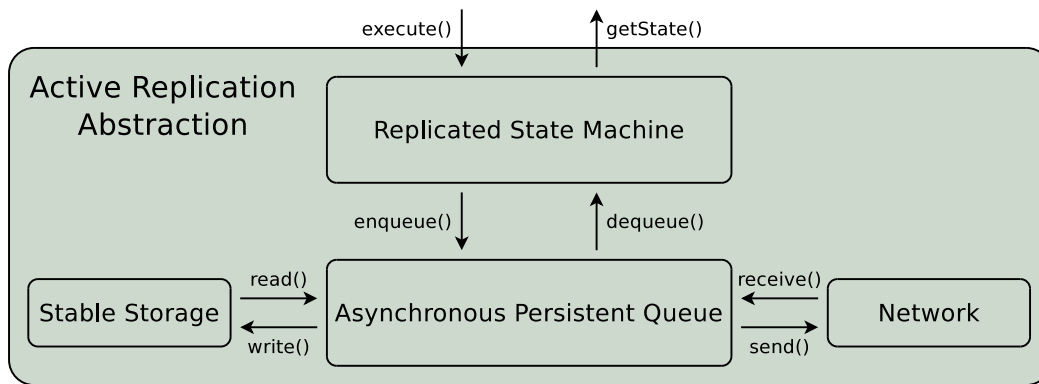


Figura 2.3: Active Replication Components

The replicated state machine provides an abstraction to the operation of any deterministic application. It allows the maintenance of application state by stipulating a simple interface for querying and modifying such state. This component is accessed directly by the application that uses its services to hold, replicate and persist its state. All these operations are performed transparently, and require no intervention from

the user of this component. The asynchronous persistent queue is an abstraction for a persistent and fault tolerant object queue. It represents an ordered record of objects sent to a group of processes, that is guaranteed to be available even if all processes in this group fail. It can be used as a persistent log of events triggering transitions in the distributed state machine. In fact, this component is more general and could be used in other settings as it represents an abstraction for a consensus service, useful in other contexts besides replication.

In the rest of this section we describe the specification of these two components. Our description follows a bottom-up approach, starting with the asynchronous persistent queue and then moving up to the replicated state machine. This way it is easier to isolate the services provided by each component, how these services can be used and the design decisions related to their provision.

2.3.1 Asynchronous Persistent Queue

Asynchronous persistent queues are a way for a group of processes to exchange objects. These objects are sent by any process connected to the queue and broadcast to the others, totally ordered and with guaranteed delivery regardless of failures. This behavior can be more precisely described by the following three properties:

- Objects are delivered in the same order to all processes.
- Objects are delivered to all processes, even if a process crashes and later recovers.
- Objects are persistent and survive crashes of all processes.

These properties are very similar to the properties of total order broadcast [33], but state explicitly that a failed process that eventually *recovers* must also receive all ordered objects. Each process that interacts with the queue component does so through a *queue endpoint*, bound to a specific queue. The primitives of the asynchronous persistent queue component are very simple:

`enqueue(object)`: Adds an object to the end of this queue, making it available to all other processes.

`dequeue()`: Removes the next object from this queue.

The `enqueue()` method changes the state shared by all processes, the queue itself. The contents of the queue should be consistently managed ensuring that all calls to `enqueue()` in every process generate only a single ordering of all objects. Correspondingly, every call to `dequeue()` made by the processes sharing a queue will reflect this

same order. Each queue endpoint has associated with it the object delivery history. For instance, a new process joining a queue, using a new queue endpoint, will receive all objects ever sent to the queue. These objects are both local queued objects or objects queued by other processes, and they may be stored locally or fetched from the network. From the point of view of the client process, this distinction is irrelevant. Thus, by relying on the total order guaranteed by the queue and in the fact that queues are persistent, individual processes can become replicas of each other using active replication, while remaining in their perspective completely stateless.

To efficiently provide this high level abstraction to the client process, it is necessary to define some mechanism to limit the number of objects in the queue. Suppose a process fails after having executed for a considerable time and then recovers. It is the responsibility of the queue to provide it with its recovery state in the form of an object log that, in this case, can be very large. To reduce the size of this log one might periodically take snapshots of the queue, save them to stable storage and rollback the process to one of such snapshots when necessary. The problem with this approach is that the persistent queue abstraction promises the application it will receive all objects in a queue, regardless of failures. Our solution to this dilemma is what we call *queue controlled persistence*, where a snapshot of the application is stored alongside with a snapshot of the queue to stable storage. The queue handles the coordination of local snapshots among all replicas and guarantees that each replica always sees a sequence of objects *consistent with its state*. This means a process, remaining stateless, never misses a single object even when just a subset of the objects are re-delivered in the presence of failures and recoveries. This requires the process to put its state under control of the persistent queue, by being instrumented with get and set state procedures that are callable by the persistent queue implementation. Two extra primitives are added to the persistent queue component to bind it with the entity responsible for storing the application state and to control the checkpointing process:

`bind(stateManager)`: Binds a process state, represented by its state manager, to a queue endpoint. The state manager is any application component capable of implementing the `getState()` and `setState(state)` primitives.

`checkpoint()`: Instructs the queue to save a current snapshot of its state, including the process state.

At any time, but specially during the call to `bind()`, the state manager must guarantee that it is able to take a meaningful snapshot of the process state and it is able to replace the state with a snapshot provided by the queue. By correctly choosing an appropriate snapshot, the local state of the client will be always consistent with the

next object to be received from the queue. This may require, if a process fails and falls behind the others, that upon recovery the queue replace its local state with any suitable snapshot obtained from the other replicas. This snapshot can either be in the logical past or future of the state the process had when it crashed. Similarly, even if a process just falls behind the other but does not fail, its state still can be changed by the queue, but in this case only to a forward state. Thus, to support the strong guarantee of queue persistence the application not only can be stateless, but it is required to be stateless. Some control over the process of snapshot creation is provided by the `checkpoint()` operation. This method is provided so the client process of the persistent queue can influence the time a snapshot is taken, but the queue implementation is free to implement its own checkpointing policy.

2.3.2 Replicated State Machine

Using the guarantees provided by the asynchronous persistent queues it is straightforward to build a set of replicas using active replication. It is possible to use the ordered sequence of objects provided by the queue component to implement the replicas. This would require the application programmer to build some type of deterministic state machine to use active replication, to convert operations on this state machine to data, to build a monitoring subsystem to service the client requests synchronously, and to create a state manager to handle the set and get state operations required by the queue component. These are exactly the functions performed by the replicated state machine component. This component provides a higher level abstraction that supports the construction of replicated state machines with minimum effort.

The state machine component is a very simplified version of a finite state machine. It does not concern itself with the definition of all states, transitions, conditions and actions. It just treats the set of all states as a black box, and routes all external generated events to this black box. The state machine component is simply a framework to event logging, where events generate changes in a deterministic state. If the application requires a more complete implementation of a state machine, it is free to do so by using the persistent queue component directly.

The replicated state machine component allows the state it manages to be changed only by executing *actions*. An action is a data item that represents an operation to be performed on the stored state and its parameters. The existence of an action represents the occurrence of an event that may trigger transitions in the underlying state machine. The component doesn't care how transitions are implemented, thus an action must encode the conditions and operations that should be performed. Locally, each replica stores all its state in the replicated state machine and only changes

it using actions passed to the `execute()` method. The primitives provided by the replicated state machine component are listed below:

`create(initialState, queue)`: Creates a new state machine bound to a queue. An initial state should be provided, because the replica that calls this method can be the first one to bind to this queue.

`getState()`: Returns the current state of the state machine. A process can query this state at will, but cannot change it.

`execute(action)`: Executes an action on the distributed state, performing all necessary steps to coordinate this change with the other replicas.

Once a state machine is created with a template initial state, the actual state of the application is unknown and can change at any time. If the application wants to consult its state, it should first obtain an updated version by calling `getState()`. The state can be queried at will, but changes can only be performed by creating suitable actions and passing them to the `execute()` operation. Actions applied to the state machine by the local client are only performed by the state machine after they have been submitted to the asynchronous persistent queue component. The local client of the state machine perceives the execution of the action as a call to a blocking primitive. A successful return of the call guarantees that the action submitted has been performed by this replica and the effects of such execution are visible in the local state. As the underlying queue is asynchronous, the fact an action was executed in one replica does not imply that it was performed in all replicas.

By its use of the asynchronous persistent queue all actions are made persistent and the state held by this abstraction is under the management of the queue. The `create()` operation can either start operating with the provided state or it can recover some state from the queue. Once a suitable state is found and installed, all pending actions in the queue are replayed and the state machine is ready to resume operations. This means that, from the point of view of the client of the state machine component, recovery is completely transparent. However, the client must be aware of this fact and avoid keeping local state associated with the replicated object, that is, it not only may be but it required to be stateless.

The replicated state machine component has only three simple primitives that implement a well-defined and easy to use programming abstraction. Thus, the major task a programmer will have to perform to use this abstraction is the definition of the application state and of the actions that modify the state, regardless of state persistence, state replication, checkpointing and recovery concerns. It is worth to note that

this step is usually carried out even for applications that do not have replicated state, so it does not add complexity to the development process.

2.4 Treplica by Example

We now describe a simple application using Treplica to make clear the service provided by the abstract components. We focus in how these services can be used to create a replicated application and how this application can be programmed using Treplica. To this end, we develop a simple hash table application that maps a string key to a value. This application exports its service to remote clients through a SOAP interface composed of two simple methods: `get(key)` and `put(key, value)`.

The software architecture of the complete application is very similar to the one depicted in Figure 2.1(a). The application is replicated using Treplica, thus providing dependable operation to its clients. The clients only know a single SOAP descriptor and are unaware of the fact the application is replicated. Each replica is organized as shown in Figure 2.4. The application interacts with Treplica using the replicated state machine component described in the last section. The replicated data is managed by Treplica, stored in the state machine component.

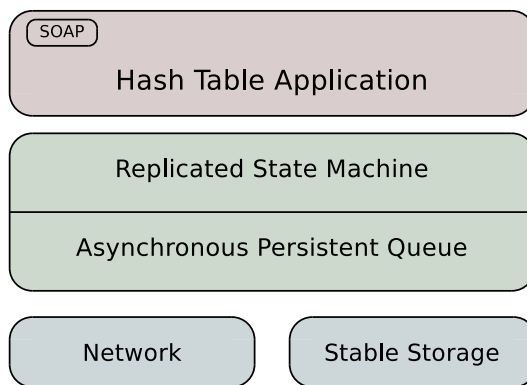


Figura 2.4: The Hash Table Application

We start the creation of the application by implementing its most basic data structure: the hash table. Using Treplica it is possible to create a distributed hash table by simply extending the hash table implementation found in the Java standard library (`HashMap`) and making it comply with the Treplica replication abstraction. The initial step of this process is to define what constitutes the application state and how it can be changed: the events and transitions. This example shows that we do not have to explicitly define states and transitions. We simply define that the state is held by the

Java hash table as a black box and that its state is changed by method calls. These calls are the events and their implementation encode the transitions.

Next, we must assert whether this component behaves as a deterministic state machine. This is done by studying the contract of the object, by analyzing its methods and, if available, by inspecting the source code. In general, objects that do not perform I/O, don't generate random numbers and don't employ date and time are safe. This is the case of the Java implementation of a hash table. If this was not the case, a simple strategy for non-determinism removal can be used. All non-repeatable operations are performed only once by a single replica and the results of these operations are encoded as constant data in the actions.

Finally, we create a proxy class (`ReplicatedMap`) that holds an instance of the original object as its state and uses the `Treplica` state machine to replicate and persist it, while at the same time presenting the same hash table interface. Example 1 shows a fragment of the proxy class, with its constructor and the two most important methods. This proxy architecture illustrates a common pattern of development using `Treplica`: we first start with an existing object that encodes the data and functionality we want to have replicated and wrap it in a `Treplica` aware layer. In this case, the object to be replicated is an off-the-shelf component that is used as a black box, but this pattern is applicable and recommended even if we have access to the object source code.

The `ReplicatedMap` class has as only attribute an instance of a replicated state machine (Line 2). This object holds the state of the application, replicating it and making it persistent. The constructor of `ReplicatedMap` initiates the state machine using one of the factory methods provided by `Treplica` (Lines 4–8). In this particular instance, a Paxos-based persistent queue is created and the state machine is bound to this queue. The Paxos persistent queue is described in Section 2.5. An empty `HashMap` is used as initial state and a path to a local directory is set as the stable memory repository. The initial state provided will most certainly be replaced if a previous instance of the `ReplicatedMap` class was created in the same local directory or if it binds to an already existing queue. Other arguments of the factory method instruct the queue to be created considering a specified maximum number of processes, an expected round-trip time and if Paxos or Fast Paxos should be used.

The state stored in the state machine can change continually and, sometimes, the actual object holding the state (the `HashMap`) may also change. So, before accessing the state it is necessary to get hold of a current object reference by calling `getState()`. The method `get(key)` is implemented by simply obtaining a reference to the current state and executing this operation directly, as it is just a query and does not change the stored data (Lines 10–12). The `put(key, value)` method is much more interesting (Lines 14–18). To implement this method we have created an object that holds the

Example 1 Proxy Class

```
01 public class ReplicatedMap<K, V> implements Map<K, V> {
02     private StateMachine stateMachine;
03
04     public ReplicatedMap(int nProcesses, String stableMedia)
05         throws TreplicaException {
06         stateMachine = StateMachine.createPaxosSM(new HashMap(), 50,
07             nProcesses, true, stableMedia);
08     }
09
10     public V get(Object key) {
11         return ((HashMap<K, V>) stateMachine.getState()).get(key);
12     }
13
14     public V put(K key, V value) {
15         try {
16             return stateMachine.execute(new PutAction<K, V>(key, value));
17         } catch (TreplicaException e) { throw new RuntimeException(e); }
18     }
19 }
```

equivalent action. This includes all data required for calling the method and the definition of which particular method should be called. Example 2 shows the action object for the put action of the hash table. This class holds the method parameters as its attributes (Line 3), initialized by the object constructor (Lines 5–7). By the contract of the Action interface, it implements the `executeOn(state)` method (Lines 9–12). The caller of this method provides the current application state as argument and the implementation performs the action using the data held in the attributes.

Example 2 Action Class

```

01 protected class PutAction<K, V> implements Action, Serializable {
02
03     private K key; private V value;
04
05     public PutAction(K key, V value) {
06         this.key = key; this.value = value;
07     }
08
09     public Object executeOn(Object state) {
10         Map map = (Map<K, V>) state;
11         return map.put(key, value);
12     }
13 }

```

The implementation of the other methods of the hash table are similar and are not shown here. If they only query the values, they are implemented like `get(key)`. If the state is changed, the methods are implemented by means of an action object as in `put(key, value)`. The complete implementation of the proxy will yield a class that can be used in place of any other map implementation in Java. The client of such class doesn't necessarily need to be aware that the object is replicated or persisted as long as its semantics match that of the state machine abstraction, as described in the last section.

Hidden in the `ReplicatedMap` lies all the integration between the application and `Treplica`. The client facing part of the application can be built using any tool desired. In this example, it is done using SOAP. A class implementing the functionality exported by SOAP is shown in Example 3. This class is just another wrapper over the hash table, restricted to the methods we want to export to the clients. The SOAP management is done by an external tool (`Axis`¹), setup with the service description

¹<http://ws.apache.org/axis/>

shown in Example 4.

Example 3 A Hash Table Application Using SOAP

```
01 public class HashTableApplication {
02     private ReplicatedMap<String, String> table;
03
04     public HashTableApplication(int maxProcesses, String stableMedia)
05         throws TreplicaException {
06         table =
07             new ReplicatedMap<String, String>(maxProcesses, stableMedia);
08     }
09
10     public String get(String key) throws TreplicaException {
11         return table.get(key);
12     }
13
14     public String put(String key, String value) {
15         return table.put(key, value);
16     }
17 }
```

Example 4 Fragment of the SOAP Service Description

```
01 <service name="HashApp" provider="java:RPC" xmlns:hash="HashApp">
02     <parameter name="allowedMethods" value="*" />
03     <parameter name="className" value="br.unicamp.HashTableApplication" />
04     <parameter name="scope" value="application" />
05 </service>
```

The SOAP framework works as an application server hosting the application and insulating it from particulars of the SOAP protocol, such as connection establishment and arguments marshaling. Thus, it keeps its internal data out of the reach of the application and of Treplica. As this bookkeeping data is constant (interface description, etc.) or volatile (connection status, etc.) this arrangement is permissible and desirable. This shows how Treplica allows the application to freely organize the aspects of its software architecture that are not related to replication.

2.5 Treplica Implementation

The asynchronous persistent queue and replicated state machine components form the base of the replication service provided by Treplica. These two abstractions are implemented as objects in the Java language, whose methods are very close to the primitives defined by the components. We have taken advantage of the object-oriented features of the language to simplify the interfaces as much as possible. The most noticeable strategy is that the objects transported by the persistent queue are serializable objects. That is, any object that can have its state automatically extracted by the Java Virtual Machine can be transported by the queue. Also, actions of the replicated state machine are simple serializable Java objects, modeled after the Command design pattern [40]. The methods encoded in the actions are built to act on the state held by the state machine using the data carried by the action as arguments.

2.5.1 Replicated State Machine Implementation

The replicated state machine component expects the higher level services provided by an asynchronous persistent queue. As described in Section 2.3.2 the replicated state machine doesn't care about explicit transitions, it just executes actions on the stored state. It is the responsibility of the client to implement, with the appropriate set of actions, meaningful states and transitions.

To support these actions the state machine provides two main services: it manages the binding of the state with the queue and it dispatches and executes actions. To keep the local view of the replicated state bound with the persistent queue it is necessary to implement a state manager. The state machine stores the replicated data for the application, providing a state manager with the required semantics. The creation of a state machine object automatically initiates the binding process, triggering any necessary recovery steps in the queue.

To change the local state, applications create actions and call the `execute()` method of the state machine. Once the action is ordered and executed on the local state, the `execute()` operation returns values or throws exceptions just like a direct invocation of the action. To effectively implement the replication, every action is sent to the queue before it is executed on the internal state. After the queue orders the actions, they are applied on the stored data by the state machine. As we assume the actions change the state deterministically, all replicas will evolve in the same way as actions are dequeued. Also, return values and exceptions are captured and routed to the corresponding calling replica.

However, the persistent queue is asynchronous. This means that the queuing of

an object does not guarantee it will be dequeued next. An arbitrary number of objects may be dequeued before a just queued object is retrieved from the queue. Actually, even objects queued locally can be dequeued in a distinct order than the one they were queued. To provide a monotonic increasing view of the stored state, the `execute()` operation of the state machine is a blocking operation. Once an application thread calls this operation it is blocked until the action created by this operation is received on the queue, it is executed and return values or exceptions are captured. The call then returns as if these operations were performed atomically.

To support this method of operation the replicated state machine has one internal thread dedicated to constantly receiving objects from the queue and to executing the associated actions on the local state. This thread consults a local data structure with action ids from all locally queued actions and decides if any local thread is blocked waiting the just executed action. If a suitable thread is found, it is woken up and return values or exceptions related to the execution of the action are routed to it. This arrangement allows the application to use multiple threads to service its clients, but Treplica guarantees that only one thread at a time executes operations on the state machine and that locally competing threads will see a consistent order of action execution.

2.5.2 Paxos Persistent Queue

The asynchronous persistent queue component depends on lower level abstractions: read and write to stable storage and send and receive messages. The service provided by these low level abstractions is not defined in the specification, and building a fault-tolerant *object* delivery system using them is far from trivial. Thus, Treplica does not assume a single implementation for the persistent queue component. It is defined as a generic interface that can be implemented in many ways that satisfy the properties outlined in Section 2.3.1.

Despite Treplica generality, we propose the use of consensus-based implementation for the persistent queue component. Specifically, our implementation in Treplica uses the Paxos algorithm. This solution to the consensus problem requires the use of stable memory in a way that allows it to be easily combined with the persistence requirements of the application. Nonetheless, it is possible to implement a persistent queue using other strategies. Besides Paxos, we have implemented prototype queues using a virtual synchrony based group communication toolkit (JGroups²) and using no replication at all for testing. Actually, the network and stable storage blocks in Figure 2.2 only reflect our current Paxos-based implementation, as queues may have

²<http://www.jgroups.org/>

their own structural requirements.

The Paxos algorithm [56] is, at the same time, a solution to the consensus problem and a mechanism for the delivery of ordered messages with the purpose of supporting active replication [79]. As such, it is a perfect semantic fit for implementing the asynchronous persistent queue component. Paxos implements uniform consensus in the crash-recovery failure model, thus it guarantees that any process that fails and later recovers will still be able to reach consensus. A consequence of this guarantee is that all processes must persist in stable memory information pertaining to the progress of the consensus instances. It is possible to directly derive all state pertaining to the queue from this state. This offers a great advantage, as the cost required to ensure strong consistency by using Paxos is the same that would be required to make the asynchronous queue persistent.

Fast Paxos [57] is an optimistic variant of Paxos that saves a communication round by assuming messages will be naturally ordered by the communication medium. Fast Paxos exhibits the characteristics that make Paxos a good choice for the implementation of a persistent queue. Treplica uses Paxos and Fast Paxos in the main persistent queue implementation (`PaxosPersistentQueue`) in such a way to selectively support both variants. In the remaining of this section we describe the Paxos and Fast Paxos algorithms, their suitability for the replication of persistent data and the implementation of a Paxos persistent queue.

2.5.3 The Paxos Algorithm

A full description of Paxos and Fast Paxos is beyond the scope of this paper, but we offer here a simple description of their main properties as they relate directly to the implementation. Full descriptions of both algorithms can be found in [57], including the computational and failure models assumed.

Processes in the system are reactive agents that can perform multiple roles: a *proposer* that can propose values, an *acceptor* that chooses a single value, or a *learner* that learns what value has been chosen. To solve consensus, Paxos agents execute multiple *rounds*, each round has a *coordinator* and is uniquely identified by a positive integer, the *round number*. Proposers send their *proposal* to the coordinator that tries to reach consensus on it in a round. The coordinator is responsible for that round and is able to decide, by applying a local rule, if other rounds were successful or not. The local rule of the coordinator is based on quorums of acceptors and requires that at least $\lfloor N/2 \rfloor + 1$ acceptors take part in a round, where N is the total number of acceptors in the system [57]. Each round progresses through two phases with two steps each:

- In Phase 1a the coordinator sends a message requesting every acceptor to participate in a round.
- In Phase 1b every acceptor that has accepted the invitation answers to the coordinator with the value and round number of the last vote it has cast.
- In Phase 2a, if the coordinator has received answers from a quorum of acceptors, it asks the acceptors to cast a vote for a suitable proposal.
- In Phase 2b, after receiving a request to cast a vote from the coordinator, acceptors cast their vote for the proposal.
- Finally, a learner learns that the proposal has been chosen if it receives Phase 2b messages from a quorum of acceptors.

Fast Paxos changes Paxos by allowing the proposers to send proposals directly to the acceptors. To achieve this, rounds are separated in *fast* rounds and *classic* rounds. The quorums used by Fast Paxos are larger than the ones used by Paxos and can assume many values that satisfy the requirements of the local rule. In particular, it is possible to minimize the number of processes in a fast quorum ensuring that both a fast and classic quorums contain $\lfloor 2N/3 \rfloor + 1$ processes [57, 86]. A Fast Paxos round progresses similarly to a Paxos round, except that Phase 2 is changed:

- In Phase 2a, if the coordinator has received answers from a fast quorum of acceptors indicating none of them has voted yet, it instructs the proposers to ask the acceptors directly to cast a vote for a proposal of their choice.
- In Phase 2b, after receiving a request to cast a vote from one of the proposers, acceptors cast a vote for a proposal.

This description of both algorithms considers only a single instance of consensus. However, Paxos also defines a way to deliver a set of totally ordered messages. The order of delivery of these messages is determined by a sequence of positive integers, such as each integer maps to a consensus *instance*. Each instance i eventually decides a proposed value, which is the message (or ordered set of messages) to be delivered as the i th message of the sequence. Each consensus instance is independent from the others and many instances can be in progress at the same time. To support the crash-recovery failure model, both algorithms require the agents to store state in stable memory [57]. The state is comprised of a record of the instances initiated, the round numbers used and proposals made or voted, among other data.

In Paxos and Fast Paxos, any process can act as the coordinator as long as it follows the rule for choosing a suitable proposal in Phase 2a. The choice of coordinator and

the decision to start a new round of consensus are made relying on some timeout mechanism, as Paxos assumes a partially synchronous computational model to ensure liveness. Specifically, there can be only one active coordinator at any given time to ensure progress. If two or more processes start coordinator agents, the algorithm can stall as the multiple coordinators compete for the attention of the acceptors with fast increasing round numbers. For this reason, liveness of the algorithm resides on a coordinator selection procedure. This procedure doesn't need to be perfect. Safety is never compromised if zero or more coordinators are active at any time. However, the coordinator selection needs to be robust enough to guarantee that only a single coordinator will be active most of the time. We call the creation of a coordinator agent by a process, guided by the coordinator selection procedure, *coordinator validation*.

Considering the concurrent nature of the consensus instances, a common optimization is done during coordinator validation. Phase 1 in Paxos and Phases 1 and 2a in Fast Paxos are run once for all the unused consensus instances at that time. In fact, it is always guaranteed that an infinite number of instances are in this situation. The coordinator in Paxos "saves" these instances for future use or, in Fast Paxos, it frees the proposers to use these instances. The improvement brought about by this factorization allows Paxos to achieve consensus in three communication rounds and Fast Paxos in only two communication rounds. Unfortunately, Fast Paxos cannot always be fast. Proposers can propose two different values concurrently, in this case their proposals may collide. Also, process and communication failures may block a round from succeeding. Different recovery mechanisms can be implemented to deal with collisions and failures, but eventually the coordinator intervention may be necessary to start a new classic round [57].

2.5.4 Paxos and Replication

The Paxos algorithm possesses many useful properties when used as a total order mechanism for active replication. It adheres to the crash-recovery failure model, ensuring that replicas that fail by crashing can later recover and return to the computation. Moreover, it implements uniform consensus, ensuring that even faulty replicas will see the same global order of messages. To see why these properties are invaluable, we will consider the progress of the system from the point of view of a failed replica.

In the event of a failure Paxos ensures us that the information in stable memory is sufficient for a process to recover immediately, without any coordination with the other processes. This is possible because the local stable storage of a process includes always consistent status of all consensus instances. Thus, a process that experiences a brief failure, such as a system reboot, just resumes operation normally after restoring

its local state. Neither the recovering process or the other processes notice anything unusual. This behavior is specially interesting if we consider that the process has not failed at all, but was temporarily disconnected from the remaining of the replicas. As expected, this situation isn't distinguishable from a real failure and doesn't require any type of special action or coordination from the other replicas.

If the process is unable to recover or resume communication immediately, the system will continue operation uninterrupted as long as the minimum number of correct processes is maintained. The failed process still can recover without coordination, but it may have missed a large number of messages and must catch up with the other processes. This amounts to a type of coordination with the notable exception that the system never blocks while the recovering process brings its state up to date. The process may resort to some type of state transfer with a more up to date replica, but during the execution of this process only the recovering process remains blocked. The remaining replicas operate unaffected.

This happens because Paxos does not rely on a group membership service or timeouts to decide if a process has failed. Actually, Paxos does not care about the state of any specific process to function correctly and just requires a stable coordinator and a possibly anonymous majority of working acceptors to progress. An optional group membership module may run on top of Paxos [56], but the criteria it employs to decide a process is to be excluded from the group can and should be distinct from the criteria the underlying total order algorithm uses to decide if a message is or isn't to be expected from a suspect process. This observation is fundamental to understand the high resilience to failures observed in the execution of Paxos (Section 2.7).

To support this level of resilience in the crash-recovery failure model, Paxos needs to keep information in stable memory. Usually implemented with magnetic disks, stable memory is very slow compared to volatile memory and adds significantly to the latency of operations. This time penalty could be considered against the use of Paxos for active replication. However, in *Treplica* we are interested in applications where the replicated state must be stored persistently no matter the type of failure. In particular, application state must survive a complete crash of the entire replica set. Consequently, applications would need to access stable memory anyway to keep their own state persistent. This stable memory requirement is completely independent of the requirements of the total order algorithm used. Depending on the guarantees provided by this underlying algorithm, the application would have the additional work of reconciling its persistent local state with the state of the message delivery in case of failure. Fortunately, Paxos allows us to take a different approach. Instead of reconciling total order algorithm and application state, we tie them together and manage them as a unity. This is possible because Paxos must remember the state of any con-

sensus instances, and this state includes the messages proposed and decided. From the contents of these messages it is trivial to obtain the application state. Thus, Paxos is a very desirable algorithm to implement a persistent queue because its properties combine performance that is resilient to failures and a unified view of replication and persistence.

2.5.5 Treplica Software Architecture

The software architecture of the Paxos-based asynchronous persistent queue follows very closely the agent decomposition used in the description of the algorithm, achieving a very modular design. The queue implementation is composed by internal classes performing the functionality of the four agents, assisted by generic support modules. Figure 2.5 shows the main modules of the Paxos persistent queue.

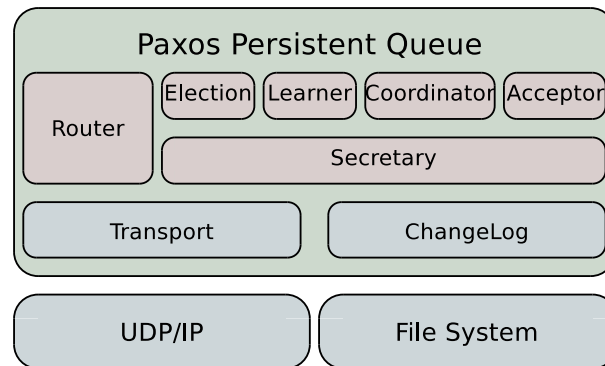


Figura 2.5: Paxos Persistent Queue

The four Paxos agents are implemented in the following classes:

Learner Combines the proposer and learner agents in a single class responsible by monitoring the flow of Paxos instances, converting them into objects and delivering them to the queue.

Acceptor Acts as an acceptor.

Coordinator Acts as a coordinator.

Election Handles the leader election algorithm used to select a single coordinator.

These classes were designed to execute independently, making it possible to create a Paxos process with only a subset of agents. Specifically, a process containing only a Learner module could propose and learn values, effectively running a complete

queue, without taking part in the consensus procedure. A process configured this way could be used to increase the scalability of the system. However, this functionality isn't supported by the queue yet.

The main classes behave in a similar way to the agents they implement as they are strictly reactive modules. They operate by processing messages addressed to them by the Router class. As a consequence of this processing, these classes may send new messages to the network, store information in stable memory or deliver ordered objects to the application. These tasks are handled by the Secretary class, that offers a uniform interface to all I/O required by the agent classes. The abstraction provided by the Secretary class gives the agent class a way to send messages encoded in a Message class and to access stable memory wrapped in a Ledger class. The Secretary on its turn relies on the services of the Transport and ChangeLog classes to access the underlying network and stable storage. These two classes provide abstractions that shield the other modules of the persistent queue from implementation details. We have implemented a Transport based on multicast over UDP/IP networks and a ChangeLog based on a simple file system.

In the following sections we describe these modules in more detail. They are presented in a bottom-up manner, starting with the support modules and then describing the Paxos agents. For each module we show its main function, how it interacts with the other modules and the implications of its structure on the Paxos implementation. To simplify this description, we name modules that depend on the a module being described as its *clients*. The original Paxos specification leaves many details unspecified, specially regarding liveness constraints and optimizations. In the following text we strive to make clear how we have implemented important open aspects of Paxos.

2.5.6 Support Modules

The support modules give an abstraction of the underlying system with clearly defined interface and service guarantees. These guarantees are very simple and can be directly mapped to many types of networks and stable storage semantics. The major motivation was to simplify the API used by the Paxos agents, hiding details about process addressing, multicast and unicast message passing, stable storage allocation and deallocation, main memory management, I/O management and error detection.

Transport The transport abstraction is defined by the Transport interface and represents a generic multicast transport. It allows its clients to send and receive unicast and multicast messages, closely matching the network properties expected in an asynchronous system. The messages are exchanged in an unreliable manner, and may be

delivered out of order, duplicated or may be lost. We have made the choice of defining the transport abstraction with so few guarantees motivated by two reasons: it matches the network requirements of many consensus algorithms for asynchronous systems, including Paxos, and it closely reflects the guarantees effectively provided by our chosen network transport, UDP/IP.

Matching the algorithm requirements is important because we avoid duplication of functionality. For example, as Paxos does not require reliable message delivery, it includes a mechanism for message buffering and retransmission. Using a reliable mechanism would duplicate this mechanism. Moreover, the reliable delivery provided by a transport such as TCP/IP only works for the crash-no-recovery failure model. In the crash-recovery failure model, the consensus algorithm still needs to check if messages were delivered even when using TCP. We understand the benefits of using reliable transports, specially regarding point-to-point bulk transfers of data [1], but we believe that algorithms like Paxos must evolve and be tuned to support multipoint delivery of ordered data with the same efficiency, but respecting the chosen failure models.

Besides message delivery properties, the transport abstraction defines a unified view of unicast and multicast messages, with a supporting addressing scheme. In brief, the same transport implementation is able to send, and more importantly, receive messages sent both to a process and to the multicast group comprised of all processes in the system. Unicast addressing is done using an opaque transport id created and managed by the transport implementation. Each process obtains its id by a call to the `getId()` method of the transport. A process may exchange unicast messages with any other process in the system as long as it knows its id, using the `sendMessage(Serializable, TransportId)` method. Ids are simply data, and may be exchanged inside regular messages. Thus, a process may announce its existence by simply multicasting its id. A message can be multicast to all processes in the system with the `sendMessage(Serializable)` method. The `receiveMessage(int)` method is used to receive a message. It is a blocking primitive that blocks until a message is received or a timeout expires.

The group of processes reached by multicasts is implicitly defined by the underlying multicast primitive used by the transport implementation. This means that the transport is required only to identify and send messages to a suitable set of processes eligible to be part of the system, this being considered the set of “all processes”. As an example of how this can be easily done, consider the UDP transport implementation. Its definition of all processes in the system is given by all processes that are listening to a predefined multicast IP and port. This IP and port are configurable parameters of the UDP transport. IP multicast routing infrastructure defines whose

processes are effectively in the system. For instance, all processes linked to the same local area network, without any intervening level 3 routers or firewalls, are in the same system. This allows to trivially consider all processes in a cluster to be in the same system and, with more elaborate routing configurations, to setup a system spanning many distinct sites.

Note that this is completely different from a group membership service that defines which processes are part of the system. If such service exists, it sits above the transport and somehow filters messages received from processes not considered to be in the group. Once again, this reflects the expected network behavior of protocols like Paxos. They do not require the precise identity of processes, but only that a minimum number of them be correct at any time. This also lays ground to build more sophisticated architectures like the one proposed in [66], where the group membership is built on top of consensus and not the other way around.

Change Log The change log abstraction shields the Paxos agents from the details regarding stable storage. Basically, the service provided is that of a persistent log of changes to an object, with support for checkpointing. In fact, the interface presented to the programmer is very similar to a simple append-only file, but with explicit support for recovery. Changes to an object can be persistently appended to the end of the log and the object can be later reconstructed by replaying these changes. Checkpointing is used to improve the performance of reconstruction by storing the changes interspersed with full copies of the object. As a file, the abstraction provides `open()` and `close()` methods to prepare a change log for use. Once open, individual changes are written with the `writeChange(Serializable)` method and checkpoints are written with the `writeCheckpoint(Serializable)` method. The similitude with a file is just an approximation to a common API for stable storage, but there is no need for an explicit backing file to support actual implementations. Our main implementation (`DiskChangeLog`) uses the local file system to implement the change log, keeping its data in several files to speed the recovery.

The change log abstraction further deviates from a simple file as it provides active support for recovery. Whenever a change log is open, all the information required to reconstruct the underlying object is transferred to the module opening the change log. This action is triggered by a call to the `open(ChangeLogClient)` method, that requires a reference to a recovery client. The recovery is very simple. The most recent checkpoint is read and passed to the recovery client. Once the client has loaded this checkpoint, all subsequent changes are passed in turn to the client that must be able to apply such changes. This way, the recovery client implements all recovery policy while the recovery mechanism is driven by the change log. Recovery consistency is

guaranteed by the change log: all changes and checkpoints writes are atomic and a failure automatically closes the change log. If a client wants to keep using the change log, it must re-open it and, at its choice, perform recovery to the point of the last successful change or detect the change whose write failed and rewrite it.

The reason we have chosen to abstract stable storage in the form of a change log is simplicity and performance, but also the desire to experiment with alternative forms of persistent storage. The append-only operation of the change log allows the use of an underlying magnetic and solid state (flash) disk in its optimal sequential access mode. Appending operations sequentially at the end of a file also simplifies recovery, as it is never necessary to reconstruct a log in case of write failure. We just create a new file and, as necessary for recovery, read the old file until the failure point. Moreover, a simpler abstraction allows multiple implementations. We already explored this possibility with a change log implementation that actually stores data in the volatile memory of other processes in the system. In this ongoing research project we are investigating the feasibility and reliability of such scheme.

Ledger The ledger is an abstraction to the stable state of the Paxos implementation. It is a common data structure, shared by all Paxos agents. The agents see main-memory oriented methods defined in the Ledger interface, while a concrete implementation has support for efficiently storing this information in stable storage. As described in Section 2.5.3, it is possible to derive the state of the replicated process from the state of the consensus instances stored in stable memory. Thus, the ledger abstraction concentrates all data that is to be held in stable memory, making it easily accessible from main-memory. The `LoggingLedger` is the object effectively made stable by the change log. To simplify the use of the change log, this implementation has support for detecting and isolating changes made to its internal state. It can export these changes and later recover its state by reapplying a set of previously exported changes. The ledger stores the complete state of each individual consensus instance, holding all data required by all types of agents. This way, it is possible for a process to create new agents, such as a coordinator, without reloading the data structure.

Secretary The secretary abstraction presents a unified view of I/O for the Paxos agents. It handles stable storage implementation using the change log and the ledger, it handles message passing using the transport, and it handles the object queue used to deliver objects to the application. The main reason this abstraction was created wasn't to isolate the agents from the underlying building blocks but to remove costly I/O operations from the thread executing the agents. Disk I/O in particular has a great potential to reduce the throughput of any Paxos implementation because of

two reasons. First, all changes written to stable storage must be flushed from any intermediary caches before algorithm execution continues, to guarantee consistency. Second, some steps of the algorithm can generate many stable memory writes. Considering that each flush operation takes around 1ms to complete and that a Paxos round demands at least two stable memory writes, we add at least a 2ms latency to all consensus instances. Moreover, all rounds must compete for access to the disk and a round must add to its latency the time required to flush the data of rounds executed before it. The secretary abstraction creates a way to solve this problem by removing from the agents the task of effectively performing I/O.

Once I/O is handled only by the secretary, it is now possible to solve the problems caused by many stable memory writes in a single operation and the lack of parallelism among multiple rounds. This is done by queuing and grouping distinct logical writes in a single physical write. This approach is advantageous because the size of the data in a complete disk write, usually performed by a `sync()` system call, has little impact on the operation latency. Making use of this observation, the secretary implementation keeps continuously writing and flushing data to the disk, in a separate thread, as long as there are requests waiting to be written. Requests that arrive in the midst of a write are queued and wait for the next flush.

This approach streamlines access to the disk, but it doesn't change the fact that Paxos correctness is rooted in the stability of the information written to stable storage. What this means is that a thread executing an agent must block until the I/O operations it has requested to the secretary are complete. To obtain parallelism in the agent execution with this blocking behavior, we could manage many threads executing concurrently the agents. This is possible, but it requires complex concurrency control to the common data structures and is prone to lock contention. Instead, we have decided to run agents in a single thread with total control of the data structures, simplifying concurrency control. However, this approach has the problem that it serializes the execution of rounds if we were to maintain the simple blocking behavior. The solution comes from the observation that rounds are independent in Paxos. Thus, a single thread is capable of managing many rounds at a time if it can avoid to be blocked for I/O, but instead changes rounds. The secretary allows exactly this behavior by implementing asynchronous I/O operations.

These asynchronous operations work by creating a virtual barrier between the actions an agent has performed and the actions that the other agents observe it to have performed. An agent has three types of interaction with the outside world: it sends messages to the network, it delivers objects to the application and it writes to stable storage. In the Paxos algorithm messages can be lost, so a simple message send isn't binding. However, the stable write done before the message send is binding, to

allow the message to be recreated later. For example, in Phase 1b an acceptor, before sending the coordinator its last vote, must record its participation in the round chosen by the coordinator (Section 2.5.3). One way of ensuring that messages are only sent after the write to stable memory is committed is to hold the messages sent by the agent until the write is stable.

We say that a message, sent to the network or delivered to the application, is *dependent* on the last write made to stable storage but not yet actually written by the secretary. From the point of view of the other agents, the write never happened until its dependent messages are delivered. That is, a message can only be delivered after all stable memory writes that precede it causally [55] are flushed to disk. Whenever a write completes, the secretary unblocks the dependent messages and send them to the other agents. Meanwhile, the agent that created the dependent messages is free to keep processing new messages, making further changes to the stable storage and sending additional messages as long as it has work to do. If the writes are held back indefinitely, the system will eventually stop. However, there will be a steady flow of concurrent rounds to be processed to keep the non I/O bound thread of the agents busy most of the time. Put in another way, in our solution the agents do not block for I/O, but external effects of their actions do.

Another function of the secretary is to manage the creation of checkpoints for recovery. As the secretary concentrates all I/O, it is able to freeze all operations of a queue and, as a consequence, the application. This way, it can obtain a snapshot of all relevant data structures and of the application. As explained in Section 2.3, the application is accessed through its state manager, the other data structures are under control of the secretary. The secretary also generates and handles a Paxos id, uniquely identifying this process.

Router The router is a simple but vital module of the Paxos persistent queue. It binds all agents together and provides them with their main thread. Its function is to run the main loop of the Paxos implementation, receiving messages from the underlying transport and, according to their type, routing them to the appropriate agent for processing. This way, agent execution is sequential and shared data structures such as the ledger do not need concurrency control. Also, this single thread monitors a central timer and generates timer events to the agents that need it. As explained previously, the message processing code of the agents is free of long running or blocking operations. This way, agents are programmed as simple event handlers in a asynchronous event-based processing architecture. Additionally, the router is responsible for instantiating the agents and the appropriate supporting modules, handling initial configuration of the Paxos persistent queue.

2.5.7 Paxos Agents Modules

Paxos agents effectively implement the Paxos algorithm. They implement behaviors described in the algorithm specification and are responsible for its correct operation. They use the support modules described in the previous sections, adhering to the processing model of asynchronous event-based message handlers that create stable memory dependent external events. This section describes their functionality and also documents our solutions for the gaps found in the Paxos specification.

Election

This agent is responsible for the leader election protocol required by Paxos to make progress. It exposes to its clients the interface of a Ω failure detector. Briefly, this failure detector requires that any election agent trusts one process in the system as correct and that there is a time after which all election agents trust the same process [27]. If we make this trusted process run a coordinator agent, we eventually have only a single coordinator agent running as required to ensure Paxos liveness. The election agent doesn't require the clients to poll its service to notice leadership changes. Specifically, it detects when the process running the agent is the elected leader and initiates a coordinator agent in response to this event. Conversely, it detects when the process stops being a leader and stops the coordinator agent.

Stable Leader Election Any unreliable election procedure is appropriate for Paxos correctness as long as it implements Ω . This procedure must combine a mechanism to elect a single process with some type of heartbeat-based failure detector [31] as processes and communication links may fail. Moreover, as many Paxos instances are to be executed in sequence, it makes sense to avoid arbitrary leader changes and keep the same coordinator instance elected at all times. Thus, stability is an important requirement in the implementation of the election service. We have implemented an election procedure that is a variant of the algorithm proposed by Larrea et. al. [60]. To function properly, our protocol requires all links incident to a non-faulty process to be eventually timely in both directions. This effectively makes link failures to be equivalent to process failures, the most common failure situation in clusters. Besides being simple to implement, this protocol has the advantage of only requiring the regular sending of a constant size broadcast message to maintain a single leader once it is elected.

Our leader election algorithm modifies the algorithm of Larrea et. al. in two important ways: it supports an unknown set of processes and it implements leader stability. Consistently with the transport abstraction of the network provided to the Paxos

agents, the election agent assumes a completely interconnected network of anonymous participants. Anonymous means a single process does not know beforehand how many other processes there are in the system or their identity, but processes do have a unique identifier and they can discover each other by exchanging broadcast messages. Leader stability guarantees that once the system behaves synchronously long enough to elect a leader process, this process won't be demoted during synchronous operation and will always be (re)elected leader after periods of asynchronous operation as long as it does not fail.

The algorithm works as follows. All processes listen for election messages and keep a local timer. Whenever a process receives an election message that indicates a process with higher priority is requesting leadership, the process records the sending process as leader and behaves as a follower. If the timer expires and the process has not received any message from a higher priority process, it assumes it is the leader and starts sending election messages advertising its leadership. The leader process does not expect confirmation from the followers, the absence of competition indicates an implicit success of its leadership bid. In our particular implementation, the local timer is configured to expire in a time sufficient for two election messages to be received. If the network behaves synchronously and no messages are lost, after a round of election messages are sent by all contending leaders only the process with the higher priority will still consider itself a leader and only this process will keep on sending election messages.

A careful choice of process priority is required for the election protocol to function. At a minimum, it is necessary for all priorities to be unique to allow only one process to possess the higher priority from all contending processes. To this end, the process priority contains a unique Paxos id provided by the secretary. However, uniqueness isn't enough to ensure stability as a process with higher Paxos id can demote an elected leader. To achieve stability we have defined the priority to be a pair (uptime, id), where uptime is an integer counter incremented every time a leader process tries to renew its leadership. The uptime counter is initialized to 0 every time a process starts or recovers. The process with the highest counter, that is, the process that stayed most time as leader without crashing, is the process with the highest priority. In case of identical uptime values, Paxos id is used to break the tie.

Learner

The learner agent in Treplica implements the functionality of the learner and proposer agents in the Paxos algorithm. It is responsible for processing requests from the persistent queue client, creating suitable proposals to order these requests, monitoring

the proposals until they are ordered and delivering ordered proposals as objects to the persistent queue client. To understand why we have combined the functionality of these two agents in the same module, it suffices to observe the activities performed by this agent. It is possible to classify the first two tasks as pertaining to the proposer agent only and the last task to fall under the activities of the learner. Nonetheless, the third task is fundamental to the correct operation of our implementation of the Paxos persistent queue and it requires knowledge held by both proposer and learner. This happens because we support both Paxos and Fast Paxos in the same implementation and Fast Paxos removes from the coordinator agent the sole responsibility of proposing consensus values.

Stateless Proposals In Fast Paxos, at a minimum, it falls on the proposer the selection of an unused consensus instance, the proposal of a client request in this instance to the acceptors and the detection that a proposal has completed or has failed. The last step is necessary because the proposer must be able to forward a failed proposal to the coordinator, which restarts the consensus instance with a classic round number. Moreover, even in Paxos, the proposer faces the problem of making sure every request made by a client translates into exactly one ordered proposal, without repetitions. It could rely on the coordinator to ensure this, relaying to it not proposals but client requests. However, the coordinator can fail before sending a proposal in a suitable consensus instance or the message containing the request may never arrive. In this case, the proposer must resend the request to the coordinator until it is ordered. This only shifts part of the problem to the coordinator, that now must check every request it has received to see if it wasn't already ordered, without actually relieving the proposer from the task of monitoring the sequence of ordered proposals looking for its pending requests.

We solve this problem by completely shifting from the coordinator to the learners not only the task of creating a proposal from a client request and monitoring it, but also the selection of an appropriate consensus instance. A learner receives requests to be ordered from its client and queues them until it is ready to create a new proposal. When this happens, it selects from its local view of the consensus instances a non-started instance. In Fast Paxos, this means that the learner can submit the proposal to be voted by the acceptors immediately. When running Paxos, it forwards the proposal to the coordinator to be decided in the consensus instance it has selected. Either way, the acceptors broadcast their votes directly to the learners and it is their responsibility now to check if the created proposal is decided in the position specified by the selected consensus instance in a timely fashion.

This proposal monitoring is easier now, as a learner only has to observe the specific

consensus instance it has selected. If another proposal is decided in this instance, the learner selects a new non-started consensus instance and tries again. Meanwhile, the coordinator has not to monitor the proposals it manages, it just tries to decide proposals in the indicated consensus instances. Obviously, it won't violate Paxos consistency to satisfy the learner request, and informs the learner when the selected consensus instance isn't actually free for use. This behavior of the coordinator is exactly the same in Fast Paxos, but the coordinator is only called to action when a collision or timeout occurs. This way, both learner and coordinator can manage the flow of proposal requests in a stateless way. Consensus instance consistency still requires stable storage, but a request is managed only in main-memory.

Gap Detection The learner monitors only the proposals it has created, but it receives votes and computes the consensus decision for all instances. These values are ordered according to the predetermined consensus instances numbers and delivered sequentially to the client as soon as they are decided. This delivery is done by adding values to a queue, while the client removes elements from this same queue. Actually, from the point of view of the client, this queue is the asynchronous persistent queue itself. Consensus instances, however, are not decided sequentially. Due to lost messages or collisions, a gap of undecided instances may appear before a decided instance. These gap instances prevent values decided in subsequent instances of being delivered to the client. Thus, to force the decision of gaps a learner tries to pass a *null* proposal, sending it to the coordinator as usual. If these gaps are already decided but the learner is unaware of it, the coordinator will tell the learner so. If the decision isn't known to the coordinator, it will start a new round. This round, according to the consistency guarantees of Paxos, will discover if a sufficient number of acceptors have decided any proposal in this instance. In the unlikely event that nothing was decided yet, the *null* proposal will be decided. However, this proposal will be ignored by all learners and not delivered to the client.

A learner must ensure that some value is eventually decided in all consensus instances it knows to be active, be it a gap or not. This is controlled by a timeout mechanism that resends proposals to the coordinator if the particular consensus instance where it was originally sent isn't decided. As the learner is responsible for selection of the intended consensus instance for a proposal, the operation performed by coordinator upon the receipt of this message is idempotent. As an optimization the coordinator checks to see if this consensus instance is already in progress or decided and notifies the learner. This behavior is identical in Paxos and Fast Paxos, that is, the learner can always ask the coordinator to pass a proposal in a specific consensus instance if it detects a problem. This covers a special case in Fast Paxos: a collision.

Collisions In Treplica, learners detect collisions as they tally the votes cast by the acceptors. These votes can be to distinct proposals if two or more learners have concurrently initiated the same consensus instance. If the number of acceptors that have not cast a vote yet isn't enough to win a majority for the most voted proposal, a learner detects a collision. It then restarts immediately this consensus instance by sending a request to the coordinator. Lamport describes several other strategies to resolve collisions [57], but we have decided to use this simple restart procedure because of the low overhead associated with running single classic Paxos rounds and the fact that the number of collisions observed in practice is very low [88].

Congestion and Flow Control Since the learner is responsible for starting consensus rounds directly or through the coordinator, it is also responsible for any type of congestion or flow control. Intuitively, congestion and flow control have the objective of avoiding the saturation of the transmission capacity of a link or the processing capacity of a CPU, respectively. When saturated these resources tend to provide less service than when just below their maximum nominal capacity. The general mechanism to attaining this type of control for network applications is to limit the rate messages are generated by individual processes. This rate limiting can be done using explicit readings of the load on the network or CPU, or be based on indirect measures like the latency of messages or message loss. However, due to its distributed and fault-tolerant nature, Paxos presents some subtle difficulties to both flow and congestion control.

By design, a subset of all processes are allowed to fail in Paxos, and they can take an arbitrarily long time to recover. This directly impacts flow control as it is impossible for a process to distinguish if any of the other processes are slow or failed. This means that a process should never wait a slow process to avoid being blocked indefinitely by a crashed one. Congestion control seems to be immune to this effect as a process can observe and control the rate it creates its new proposals. Nevertheless, it is still necessary to ensure that the rate chosen by any single process will provide a "fair" allocation of the available link bandwidth. Again we return to the same problem: a fast process can never be sure a slow process hasn't actually failed and it can't wield to it, because a failed process may never request the released resource.

We consider adaptive congestion and flow control mechanisms that solve these problems to be very interesting research areas. We have not had the chance to research on suitable policies for congestion and flow control, but we have implemented in Treplica a rich mechanism capable of supporting many interesting policies. The mechanism defines a maximum number of pending proposals per learner, a maximum size for a proposal and a maximum number of queries for gaps in the instance

sequence. The maximum number of pending proposals is the basic tool for congestion control. It limits the rate new proposals are created and sent for vote by the learners, by forcing them to wait the complete approval of a proposal to create a new one once its local maximum was reached. This mechanism takes full advantage of the fact that learners are responsible for proposal creation. Each learner is able to control its maximum number of proposals independently and the coordinator doesn't need to monitor this quantity.

When a learner reaches its maximum number of pending proposals, subsequent client requests are queued waiting for a proposal to complete. When this happens, the learner creates a proposal containing more than one client request, ordered by arrival time. Under high client load, the size of the request queue can grow very fast, so the learner limits the number of requests that are packed in a proposal. This is controlled by the proposal maximum size. A learner creates a new proposal by concatenating client requests until the maximum size is reached or the request queue empties.

The final congestion and flow control mechanism has to do with filling gaps in the consensus instance sequence. A single learner cannot deliver the ordered requests to its client if there are undecided consensus instances before decided instances. This is particularly relevant when a crashed process tries to recover and discovers a large subset of instances whose outcome is unknown. A naive learner might try and decide *null* on all instances at once, overloading the network and the coordinator. To avoid this we define a maximum number of queries sent to the coordinator for filling gaps in the consensus instance sequence. This is similar to the maximum number of pending proposals, but can usually be larger because the gap filling process is faster than a full Paxos round for decided instances.

Currently the parameters that control maximum number of pending proposals, maximum proposal size and maximum number of gap filling queries are fixed. The values used were experimentally obtained; the current values are given in Table 2.1.

Pending Proposals	2
Proposal Size	10kB
Gap Queries	100

Tabela 2.1: Parameters for congestion and flow control

Coordinator

The coordinator is a Paxos agent responsible for ensuring the prompt conclusion of consensus instances. It receives messages from the learners asking to pass a proposal in a selected consensus instance, starts or resumes an appropriate round for this instance and monitors its conclusion. As explained in the previous section, the coordinator does not monitor individual *proposals*, but instead it ensures that a *consensus instance* reaches a single decided value. The actions of the coordinator when starting a new round for a consensus instance are central to the safety guarantees of Paxos and the timeliness of these actions is central to the liveness properties of Paxos. Thus, this agent is a very important part of any Paxos implementation.

Seamless Validation To perform these vital coordination tasks effectively, any coordinator agent created goes through a validation process. During this validation, the coordinator starts all infinite consensus instances and completes the Phase 1 of the algorithm for them. This way, in Paxos all instances that have never progressed beyond Phase 1 in any *previous round* can be started directly in Phase 2 as soon as the coordinator receives a proposal from a learner. This activation process is even more important in Fast Paxos, in which rounds can only be decided in a fast way if they have their Phase 1 previously completed by the coordinator.

The traditional specification of the validation process requires the coordinator to be brought up to date with the state of the consensus instances held by a quorum of processes. As a consequence, the coordinator blocks as it performs the required state transfer and the execution of consensus instances is interrupted. We have implemented an optimized version of the validation process that avoids tying up the coordinator more than the minimum necessary. Our method works by splitting validation in two concurrent activities: activation and recovery proper. It turns out, only activation is strictly required for a coordinator to be able to start Phase 1 of all uninitiated consensus instances. Our improved procedure provides seamless coordinator validations, with reduced coordinator blocking and less disruptive performance oscillations. A more complete description of this optimization can be found in [90].

Fail Fast Rounds Another optimization found in Treplica implementation of Paxos reduces the number of rounds required for a coordinator to successfully validate. Whenever a validating coordinator fails to receive a quorum of responses to its validation request before a timer expires, it assumes the validation has failed and creates a new round with a larger round number. However, if this failure was motivated by a previous coordinator that has created a much larger round number before being

demoted, it may take a long time before the new coordinator can produce a round large enough if it naively increases the round number. A simple solution to this problem is for the acceptors to send the coordinator a special message indicating they are unable to reply to the round number just received because they have replied to a larger round number. This way, the validating coordinator knows how large its round number must be and, if no two contending coordinators are active at the same time, uses this number to successfully validate. This approach has the added benefit that the acceptor can use this message to inform the coordinator of any situation where a round number cannot be acted upon because of another larger round number. This way, a coordinator can actually discover if its status as single coordinator is being contended before a timeout.

Instance Management Once activated, a coordinator can begin processing messages from the proposers asking it to pass a proposal. Even in Fast Paxos, where the proposers act independently, proposal requests are still sent to the coordinator if they fail to be decided in a fast round. When a proposal arrives, the coordinator first checks to see if the selected instance is decided or is in progress. If it is in progress the coordinator does nothing. If the instance is decided the coordinator informs the proposer of the instance outcome using a special message. If the instance is neither decided or in progress, the coordinator starts Phase 2 of the Paxos algorithm or Phase 1 of the Fast Paxos algorithm. That is, for Paxos the coordinator continues the round started during the validation. For Fast Paxos, the coordinator assumes the proposal has failed and immediately starts a new round.

After deciding how to process a new proposal, the coordinator receives and processes the remaining Paxos messages exactly as described in Section 2.5.3. As described above, the coordinator won't restart a consensus instance in progress if it receives any proposal request for the same instance. This happens because this instance is now under the coordinator control, and it will monitor and keep retrying it until it decides any value. The coordinator maintains a timer for each instance, and initiates a new round, with a greater round number, each time this timer expires. The proposers should not be allowed to influence this timing. They will maintain local timers for their proposals, but the coordinator refusal in restarting an instance in progress allows both timers to be integrated.

Coordinator Self-Stabilization During normal operation of the Paxos algorithm there should be only a single coordinator. However, the leader election procedure can make mistakes and another process can briefly believe itself to be a coordinator. This process j will then start a coordinator that will compete with the correct

coordinator in process i , generating increasing round numbers. This is a momentary situation that is caused by election procedure mistakes, not by any real process or network failure. As such, it is important for the system to self-stabilize after these instability periods. This is specially relevant if we consider that such unstable periods are quite frequent under heavy load [90].

If the leader election module in process i effectively detects it has momentarily lost the leadership and deactivates the coordinator, this recovery is automatic. This is equivalent to an actual failure. Once the system stabilizes, i will re-validate its coordinator agent and a large enough round number will be promptly discovered. However, if the leader election module in i doesn't recognize any contending coordinator, the coordinator in j could have finished validation with a larger round number. After the election module stabilization, the coordinator in i continues processing unaware of the fact it has effectively lost the coordination position and that no acceptor will reply to its current round number. If this happens, Paxos consistency isn't violated, but all proposals with the round number from the original validation of the coordinator in i will fail. As a consequence, after a costly timeout a new round will have to be enacted in full. This has a severe impact on the throughput of the system. To avoid this problem, and ensure proper stabilization after failure detector mistakes, we adopt a simple solution. Whenever an acceptor receives a Phase 2a message and ignores it because of a larger round number, it sends the coordinator a special message indicating this fact. When this message is received, the coordinator restarts, acquiring per the rules of validation a new large enough round number.

In Fast Paxos the problem is more subtle. A possible result of a contended coordination is a set of proposers that assume they can send proposals directly to the acceptors using a round number from the coordinator in i . Because of the competition from the coordinator in j , proposals coming directly from the proposers with this round number will timeout. This happens because there won't be enough acceptors that agree to vote on these proposals. Only after a timeout the coordinator intervention is requested, and it will eventually decide the instance. However, fast rounds are impossible and the coordinator is oblivious to this fact. To fix this, the coordinator re-sends at regular intervals the round number it has associated with its last validation. This effectively renews the authorization it has made for this round number to be used directly by the proposers. If an acceptor notices one of these notifications with a smaller round number it informs the coordinator. As in Paxos, when this message is received, the coordinator restarts, acquiring a new large enough round number.

Acceptor

The acceptor is an agent responsible for voting in consensus instances according to the Paxos algorithm. This agent reflects very closely the behavior of Paxos described in Section 2.5.3. It concerns itself mostly with the safety of the algorithm. The acceptor waits for Phase 1a messages starting a new consensus round and answers them, if appropriate, with an account of the vote it has cast last. This enables a round to proceed and the acceptor casts a vote in this round when it receives a suitable Phase 2a message. In Treplica this behavior has only two small changes that increase the performance of the system: the acceptor reduces a vote to small constant size message and it actively warns the coordinator of decided consensus instances.

Constant Size Votes Usually, a Phase 2b message carries the round number and the proposal being voted. This way, any learner listening to broadcast votes can compute, independently, the outcome of any round. However, this is wasteful of network resources as the same proposal is sent once in the Phase 2a message and n additional times, once for each of the n acceptors in a quorum. A solution to this problem implemented in Treplica is to configure the acceptor to send only a unique identifier of the proposal in its Phase 2b messages. This identifier is generated by the learner agent that created and submitted the proposal. This way, proposals are uniquely identified in the votes and it is possible to discover the outcome of the round as long as it is possible to map this identifier to the actual proposal. To this end, learners are also changed to monitor Phase 2a messages and keep a local cache of proposals presented for voting, indexed by proposal id. The end result is that the Phase 2b messages, that account for the larger number of messages send in a Paxos round, are reduced to two integers: the round number and the proposal identifier.

Succeed Fast Rounds Another improvement is the reduction of the number of messages required for a coordinator to finish an election for an already decided consensus instance. A coordinator can always enact a complete Paxos round with all its phases, regardless of the state of a consensus instance. If this consensus instance is already decided, the coordinator will always compute a majority of votes selecting the already decided proposal. This happens frequently in Fast Paxos whenever a process hosting the coordinator loses many messages and its local learner has to catch up on the decided consensus instances. To short circuit this process, an acceptor agent warns the coordinator of a decided consensus instance by broadcasting the decided value. Thus, the coordinator and any learner that may have missed the voting can now discover the decision for this consensus instance without requiring a complete election.

2.6 Applications

Virtually all distributed applications require replication. The amount and importance of replicated state to the application varies, depending on consistency requirements and overall cost. Keeping all information replicated consistently is potentially very costly and may limit the system efficiency or availability [43]. Thus, it is extremely important to be able to replicate data consistently and integrate this data with the rest of the components of the system. Due to its modularity, Treplica can be used as a tool to assist in the construction of practically all types of distributed applications.

As described in Section 2.2, Treplica can help in creating complete applications, programmed as if they were centralized. For example, good candidates for replication using Treplica are web applications that use a database for data storage and sharing. In general, these applications do not use all properties of a relational database and only use it as a convenient way of persisting data. As a consequence, these applications end up needlessly tied to a centralized point of failure. Treplica can offer a more direct programming abstraction to persistence while providing replication and fault tolerance. We expect Treplica to be a viable option to build an enterprise wide application or a small scale Internet shop. We analyze the performance of Treplica in one of these applications in Section 2.7.

Another way Treplica can be used is in the construction of a central coordination point for more loosely coupled components. A simple way to coordinate the access of a shared resource by several independent agents is through the use of locks and leases [56, 59]. The replicated state machine is the perfect abstraction to build a cluster of reliable lock servers that can be accessed through a RPC interface. For example, distributed file systems maintain large amounts of data stored on stable storage, replicated for fault tolerance and reliable access. Due to the amount of data and to the performance requirements, this involves only two or three replicas with a primary-backup scheme. Nonetheless, the state of these replicas can be controlled by a replicated state machine, such as the identity and status of the replicas are always consistently updated and made available to both file system replicas and clients. The Google File System [41] and Chubby [26] are systems built with this software architecture. Apart from a small prototype, we have not extensively tested Treplica in a similar environment. Nonetheless, we believe Treplica would fit nicely in a similar architecture because of its simple design and performance. More data about Treplica performance can be found in [88, 90]. While these works do not put Treplica performance in the perspective of a standard benchmark, they give an idea of the expected Treplica performance.

2.7 Performance

We have made an extensive study of Treplica performance, in the context of a small scale Internet shop. This application, called RobustStore, is an implementation of the TPC-W benchmark [81] using Treplica. We use TPC-W as a benchmark for a complete application running on Treplica, in such way that the real systems that TPC-W is expected to assess are faithfully represented by our experimental setup. The complete performance analysis, including details of the experimental setting, can be found in [20]. We reproduce in this section a brief description of TPC-W metrics and key Treplica performance charts.

The TPC-W benchmark specifies all the functionality of an on-line bookstore, defining the access pages, their layout, and image thumbnails, the database structure. The bookstore application is based on a standard three-tier software architecture. TPC-W defines three workloads that are differentiated from each other by varying the ratio of browsing (read access) to ordering (write access) in the web interactions. Performance is measured in *web interactions per second* (WIPS), with *web interactions response time* (WIRT) as a complementary metric. An example of a read-only interaction is the search for books by a given author, while an example of an update interaction is the placement of an order. The shopping profile specifies that 80% of the accesses are read-only and that 20% generate updates. The browsing profile specifies that 95% of the accesses are read-only and that only 5% generate updates. Finally, the ordering profile fixes a distribution where 50% of the accesses are read-only and 50% generate updates.

In the remaining of this section we show two of the experiments we designed using this benchmark. The first experiment (Speedup) characterizes the scalability of the application, increasing the number of replicas in the system and the load handled by them. The second experiment (One Failure) shows the resiliency to failures of the application, injecting a failure that disables a replica. The experiments were carried out in a cluster with 18 nodes interconnected through the same 1Gbps Ethernet switch. Each node has a single Xeon 2.4GHz processor, 1GB of RAM, and a 40GB disk (7200 rpm). The software platform used is organized with Fedora Linux 9, OpenJDK Java 1.6.0 virtual machine, Apache Tomcat 5.5.27 and HAProxy 1.3.15.6.

Speedup The speedup experiment evaluates the maximum possible increase in performance obtained when RobustStore's scale goes from 4 to 12 replicas. Figure 2.6 shows the speedup values obtained for the three workloads and an initial state size of 500MB. It is possible to observe that RobustStore scales well, specially with a large proportion of reads. Write performance is still good and it indicates the application

can handle gracefully varying load profiles.

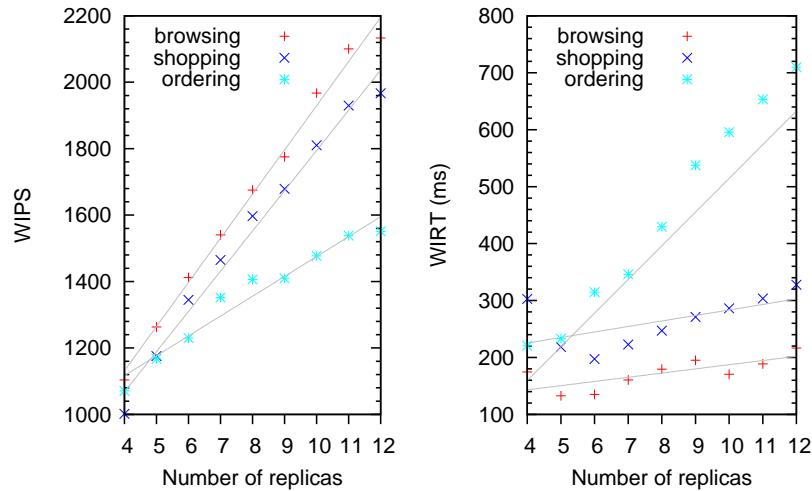


Figure 2.6: Speedup

One Failure In the one failure experiment one process is crashed 270 seconds in a 600 seconds run. The crash is immediately followed by the automatic triggering of recovery by a local replica watchdog. Figure 2.7 shows the behavior of a five-replicas RobustStore for the three workload profiles. The application throughput can be characterized as very resilient and stable in the presence of the crashes, failover, and recoveries used in the experiments. In this and other dependability experiments we have performed RobustStore loses less than 13% of its average performance during recovery in the worst case [20].

2.8 Related Work

The idea of main-memory storage, with a persistent operations log used as a fault tolerance mechanism, is described by Birrell et al. [15]. The current API of Treplica was influenced by the Prevayler [95] persistence layer, specifically in its use of features of modern dynamic languages like Java and C# to simplify implementation and provide a more straightforward API. Compared to these centralized systems, Treplica goes a step further as it uses this operation-based persistence approach as a basis for replication.

Data replication with strong consistency has been frequently used as basis for control mechanisms in large scale distributed systems [19, 48, 49]. These systems

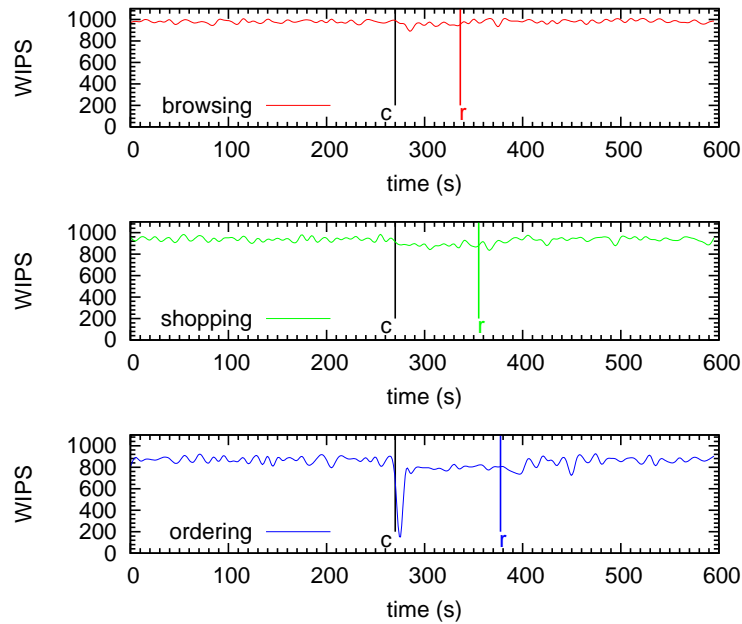


Figura 2.7: One Failure: 5 Replicas

require some mechanism to control the operations of a large number of processors and services as autonomously as possible. These mechanisms present a lock based programming abstraction coupled with a configuration data repository. Similarly to Treplica, many of these systems use the Paxos algorithm to implement replication.

The Chubby locking service is used to power a myriad of distributed applications at Google [19]. Although a locking system is a different type of abstraction, Chubby shares many architectural features with Treplica, including a “persistent log”, very similar to a persistent queue. The designers of Chubby state the intention of using this queue abstraction for the construction of other distributed applications [26]. Chubby is a special purpose application used to provide lock services and doesn’t export its internal replicated state service. In comparison Treplica exports only the replicated state service, a base where locking primitives can be build upon. Chubby uses the Classic Paxos algorithm to implement replication, while Treplica uses both the Classic and Fast Paxos variants.

Other distributed locking and distributed control systems are FaTLease [48] (part of the XtremFS object-oriented file system [47]), Zookeeper³ and Autopilot [49]. All these systems are used as control centers for distributed applications operating in a cluster. As such, they are replicated to ensure uninterrupted operation of these applications in the presence of failures. These systems provide a more specialized

³<http://hadoop.apache.org/zookeeper/>

service than Treplica. Nonetheless, it is interesting to observe that the designers of these systems decided to employ a consensus-based mechanism to replicate vital data. Moreover, this data is kept in main memory, using stable memory as an accessory for fault tolerance. These are design decisions similar to the ones we have made in Treplica. We expect Treplica to be an excellent tool for the constructions of such systems.

Boxwood is a framework for the construction of distributed storage applications [62]. Boxwood creators advocate the use of generic data structures as a foundation where to build more complex distributed systems. One of the proposed abstractions is a generic consensus service based on Paxos. This module is used by several other Boxwood components, including its distributed locks manager. Boxwood is focused in one domain of application (file systems and databases) and provides a more low level interface to its services, while Treplica offers a higher level programming API.

It is common in the literature the acknowledgment that Paxos, despite its simplicity, is full of subtleties that increase the complexity of an actual implementation [16, 26, 59]. A work that deals exclusively with a detailed description of a Paxos implementation can be found in [7]. This paper describes all aspects of a complete state machine replication using Paxos. Also, it presents a fairly complete study of the performance of the described implementation. The authors describe mechanisms for flow and congestion control, leader election and other implementation aspects not usually detailed. They observe that these mechanisms are generally considered to be implementation details, but that in fact they are central to the maintenance of many algorithm properties, specifically its liveness. With Treplica, we have proposed and implemented a modular abstraction for active replication including requirements for persistence. This way, Paxos is central for the development of Treplica but it encompasses a broader theme: object-oriented replication.

Group communication toolkits provide a service of message diffusion to a group of processes according to diverse ordering guarantees. Many of these systems exist, from the original Isis [12], to JGroups [9], Spread [5] and Appia [67], to list a few. The central idea behind these toolkits is the virtual synchrony [13, 14] application programming model. Treplica shares similar goals with these systems but does not implement the virtual synchrony model, nor does it support many message ordering guarantees, only a totally ordered message sequence. Treplica is designed to offer a simpler programming abstraction with built in support for persistence, thus the application programmer is free from the difficult task of guaranteeing state consistency. In a way, Treplica can be seen as a higher-level abstraction than group communication, and these toolkits could be used to create an implementation of the Treplica API.

The asynchronous persistent queues abstraction is very similar to the publish /

subscribe pattern of communication for process groups implemented in message oriented middleware (MOM) [10]. The message exchange in MOM is asynchronous and even a failed or inoperative processes can expect to be delivered all messages sent, in the same order seen by all the other processes. Besides message diffusion, MOM allows the construction of elaborate message flow graphs and may perform message format conversion as messages are transported through this graph. Examples of such systems are the IBM WebSphere MQ⁴ and Apache ActiveMQ⁵ products. These systems are heavy-weight compared to Treplica and are usually implemented on top of a centralized relational database, inheriting the failure behavior of these systems. Also, Treplica is designed for more tightly coupled processes and transports application objects. As a consequence, it does not provide explicit message flow and message format conversions.

2.9 Conclusion

Correct, efficient and resilient replication of applications is a hard problem faced by many programmers of distributed applications. Unfortunately, there is limited support for completely handling replication in face of process failures and recoveries in the tools currently used for the construction of such applications. To address this problem we have created an object-oriented specification for replication and implemented it in the Treplica library. This paper described the object-oriented specification proposed and the software architecture of its implementation.

The advantages of using the proposed object-oriented specification for replication are twofold. First, it makes transparent to the programmer much of the complexity of dealing with a highly-available application. Second, it allows the middleware effectively implementing the replication to optimize many factors now outside of the programmer reach. In Treplica we use extensively this property to employ consensus-based active replication to effectively get application durability for free, after paying the cost of replication. The observed performance of the final system is a good indication of the success of this approach.

⁴<http://www-306.ibm.com/software/integration/wmq/>

⁵<http://activemq.apache.org/>

Capítulo 3

Dynamic Content Web Applications: Crash, Failover, and Recovery Analysis

This work assesses how crashes and recoveries affect the performance of a replicated dynamic content web application. RobustStore is the result of retrofitting TPC-W's on-line bookstore with Treplica, a middleware for building dependable applications. Implementations of Paxos and Fast Paxos are at the core of Treplica's efficient and programmer-friendly support for replication and recovery. The TPC-W benchmark, augmented with faultloads and dependability measures, is used to evaluate the behaviour of RobustStore. Experiments apply faultloads that cause sequential and concurrent replica crashes. RobustStore's performance drops by less than 13% during the recovery from two simultaneous replica crashes. When subject to an identical faultload and a shopping workload, a five-replicas RobustStore maintains an accuracy of 99.999%. Our results display not only good performance, total autonomy and uninterrupted availability, they also show that it is simple to develop efficient recovery-oriented applications using Treplica.

3.1 Introduction

In this work, we evaluate how *crashes, failovers, and recoveries* affect the performance and availability of RobustStore, a highly available dynamic content web application. RobustStore has been implemented by retrofitting the stand-alone on-line bookstore specified by TPC-W [81] with Treplica, a middleware for building dependable applications [87]. Thus, the assessment of RobustStore is, in fact, the assessment of the fitness of Treplica as a high-availability support for dynamic content web applications. The TPC-W benchmark, augmented with faultloads and dependability measures, is used to evaluate the behaviour of RobustStore.

The process of recovering failed replicas is a main concern for highly available applications because it has a negative impact on their availability and reliability. Recovery time is primarily a function of application state size, so a larger application state should have a larger negative impact on the application, leading to performance loss. One could expect even more pronounced performance oscillations in scenarios with multiple overlapping crashes followed by multiple recoveries. We show that this is not the case for RobustStore. In fact, even in the worst case failure scenarios performance stays close to the levels delivered before the failures occurred.

Experiments apply faultloads that cause sequential and concurrent replica crashes. For example, RobustStore's performance drops by less than 13% during the recovery from two simultaneous replica crashes. When subject to an identical faultload and a shopping workload, a five-replicas RobustStore maintains an accuracy of 99.999%. The good performance, total autonomy and uninterrupted availability displayed by RobustStore in the experiments indicate that Treplica offers an efficient support for the construction of highly available distributed applications.

The remainder of the paper is structured as follows. Section 3.2 describes Treplica, and its use of Paxos [56] and Fast Paxos [57]. Treplica has been designed with performance, modularity and ease-of-use as primary objectives. The toolkit offers two very simple programming abstractions for programmers: state machine and asynchronous persistent queue. Section 3.3 summarizes the features of TPC-W, a web application benchmark widely accepted by industry and academia. In Section 3.4 we show how we have dealt with non-determinism, randomness, and database substitution during the development of RobustStore. Section 3.5 measures how the performance and availability of RobustStore is affected by crashes, failovers, and recoveries. Section 3.6 brings a summary of research that is related to our work. Section 3.7 summarizes our results and contributions.

3.2 Treplica

This section describes the features of Treplica that are relevant to this work; additional information can be found in [87]. Treplica supports the construction of highly available applications through either the *asynchronous persistent queue* or the *state machine* programming interfaces. The main programming abstraction is the persistent queue, a totally ordered collection of objects with the usual `enqueue(Object)` and `Object dequeue()` methods. `enqueue(Object)` is, for efficiency reasons, implemented as an asynchronous primitive. `Object dequeue()` has a synchronous (blocking) semantics, as usually provided by queue implementations available in programming libraries. Persistence means that a replica bound to a queue can crash, recover and bind again

to its queue, certain that the queue has preserved its state and that it has not missed any additional enqueues made by any other active replicas. Thus, by relying on the total order guaranteed by the queue and the fact that queues are persistent, individual processes can become active replicas while remaining stateless; the persistence of their state has been delegated to the queue.

The asynchronous persistent queue is implemented using the Paxos [56] and Fast Paxos [57] algorithms. These algorithms were chosen because they were designed to provide continuous operation of the application under the occurrence of partial failures, without requiring the programmer to use reconfiguration protocols. As a consequence of our choice, *Treplica* transparently transfers to the application the resiliency qualities of these algorithms. In particular, for N processes the configuration of *Treplica* used in this work uses Fast Paxos as long as $\lceil 3N/4 \rceil$ processes are working. If fewer processes than $\lceil 3N/4 \rceil$ but at least $\lfloor N/2 \rfloor + 1$ are available, *Treplica* falls back on Paxos. If fewer than $\lfloor N/2 \rfloor + 1$ processes are operational, the algorithm blocks until enough failed processes have recovered.

To ease the task of creating replicated applications out of the objects (operations) held by the asynchronous persistent queue, *Treplica* provides a higher level abstraction that supports the construction of replicated state machines. The state machine programming interface does not contain explicit support for the definition of states, events (transitions), conditions, and actions. Instead, it considers an application a black-box component whose public methods (interface) implement the set of events, conditions, and actions of a deterministic state machine. The application programmer uses the state machine programming interface of *Treplica* to treat all events, conditions, and actions as generic *actions*—Java objects—that can be managed by the asynchronous persistent queue and delivered to the application for execution.

A newly (re-)activated state machine sets its state to a consistent state. After that, the only way to change the state of the replica is through the execution of *actions* triggered at the replica by the `execute()` method of *Treplica*'s state machine. At any moment it is possible to obtain a snapshot of the most recent consistent state of a state machine by invoking its `getState()` method.

Actions invoked at one replica are guaranteed to be performed by it only after they have been converted into a message and enqueued into the asynchronous persistent queue for delivery to the other replicas. The original invoker of the action sees its execution as a call to a (synchronous) blocking method. A successful return of the call guarantees that the action has been performed by the invoker's replica and that the effects of the execution are now visible in the local state.

Recovery: Suppose a replica crashes and some time later recovers. Initially, a stateless instance of the application is created and its constructor, in turn, instantiates a state machine and invokes its `getState()` method. The method `getState()` interacts with the replica's asynchronous persistent queue. It is the responsibility of the asynchronous persistent queue to provide the recovering replica with the state to which it must be reset, in the form of a locally obtained checkpoint and an associated suffix of the queue's history. After resetting its state to that of the checkpoint, the recovered replica rejoins the remaining replicas. The queue's suffix necessary to complete the re-synchronization of the recovered replica is learned from the active replicas using Paxos. As soon as the queue re-synchronization ends, the recovered replica is ready to proceed as if it had not crashed. From the point of view of the programmer, all that needs to be done is to call `getState()`, the rest is transparently handled by Treplica.

3.3 The TPC-W Benchmark

The TPC-W benchmark specifies all the functionality of an on-line bookstore, defining the layout of access web pages, application semantics and the database structure. The bookstore application is based on a standard three-tier software architecture. Enterprises [81] and Universities [36, 37, 64] have extensively used implementations of TPC-W to assess the performance of machines, operating systems, and databases as supports for web services. The TPC-W implementation created at University of Wisconsin-Madison [21] has been used as the basis for our experiments. Performance is measured in *web interactions per second* (WIPS), with *web interactions response time* (WIRT) as a complementary metric. TPC-W defines three workload profiles that differ from each other by varying the ratio of book browsing interactions (read access) to book ordering interactions (write access). The shopping workload profile specifies that 80% of the accesses are read-only and that 20% generate updates. The browsing profile specifies that 95% of the accesses are read-only and that only 5% generate updates. Finally, the ordering profile defines a distribution where 50% of the accesses are read-only and 50% updates. TPC-W names each of these workload profiles differently to make clear from the metric name which workload has been used in every experiment. The unit name WIPS is assigned to the shopping workload profile, WIPSB is used for the browsing profile and WIPSO for the ordering profile.

During an experiment, workloads are generated by remote browser emulators (RBE). To emulate the behaviour of human interactions, the RBE specification includes a *think time*, defined by TPC-W as 7 seconds. Thus, the number of web interactions per second (WIPS) generated by a set of emulated RBEs is given by $\#RBEs / \text{think time}$. TPC-W also has a very strict definition of database model (conceptual and physical)

and of the type and amount of data generated to populate the database.

3.4 RobustStore

In this Section, we summarize the changes we made to the implementation of the TPC-W online bookstore [21] to implement RobustStore. The method described here is general enough to guide the retrofitting of any application with Treplica. The steps are the following: (I) determination of the application state to be replicated; (II) review of the application methods that change the state and their transformation into deterministic actions. In the case of RobustStore, we had to deal with the non-determinism generated by calls to date and time system functions, and random number generation. The retrofitted application is structured as shown in Figure 3.1.

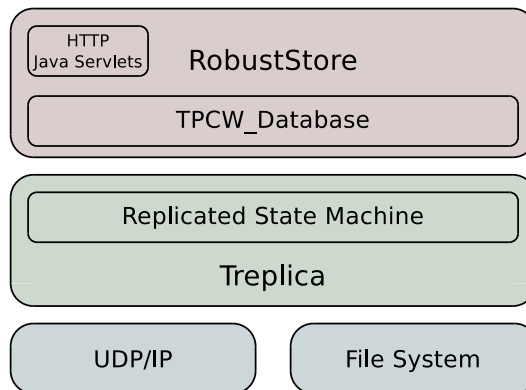


Figura 3.1: RobustStore components

Task (I) requires the design of an object model to represent the application objects that are going to be replicated. In the case of the online bookstore, we devised an object model composed by 9 classes that represent the entities and relations of TPC-W's online bookstore conceptual model. These classes and their instances represent the critical state of the bookstore and as such have to be programmed using the state machine abstraction provided by Treplica. The methods of these classes represent all the database functionality required by the bookstore. The original bookstore was structured as a set of web components (*servlets*) that accessed the database through a facade class (TPCW_Database) that served as a higher-level abstraction for the actual database. RobustStore has kept this structure intact, but the facade class now uses Treplica's state machine to execute operations equivalent to the original SQL transactions. The conversion of the facade class demanded 0.5 man-month. In total, about 2300 lines of code were changed. The final program had 3145 lines of code, 147 less

than the original implementation. We did not have to change the code of the servlets, remote browser emulator or any other support program.

Task (II) has to do with non-determinism removal. The use of random numbers, dates and time is not a problem for a centralized system, but it is a problem for a replicated system. For example, whenever a new book order is created the order creation time is set to the current time. If each replica read its local clock inside the create order method to obtain the timestamp of the order, then each of the replicas would very likely stamp its order with a different timestamp. To avoid this, the code in the facade responsible for the creation of actions in the state machine reads its local clock *before* the action is created, and passes the resulting timestamp as an argument to the action's constructor. This simple procedure guarantees that every replica receives an order with exactly the same timestamp. Calls to random number generators are handled in the same way. For example, to generate the value of the discount applied to orders of a new customer, the random number generator is called before the action that creates a customer is instantiated and the value is passed as a parameter to the action.

It is important to note that the retrofit of TPC-W's bookstore with Treplica—execution of tasks (I) and (II)—did not require the programmer to think about replication, persistence, or the replica recovery process.

3.5 Evaluation

In this Section, we seek answers to four questions. First, how long can RobustStore be expected to run without interruption? Second, how much service can RobustStore be expected to deliver during failure-free and failure-prone operation periods? Third, what accuracy can be expected of RobustStore in the presence of crashes, failovers, and recoveries? Fourth, what level of human intervention is necessary to maintain RobustStore operational? We devised four sets of experiments to gather results associated with these questions. The first set contains speedup and scaleup experiments that show how RobustStore behaves in deployments of different scales. The other sets assess the dependability of RobustStore using the three TPC-W workloads and three different faultloads.

3.5.1 Method

The experiments were carried out in a cluster with 18 nodes interconnected through the same 1Gbps Ethernet switch. Each node has a single Xeon 2.4GHz processor, 1GB of RAM, and a 40GB disk (7200 rpm). The software platform used is organized

with Fedora Linux 9, OpenJDK Java 1.6.0 virtual machine, Apache Tomcat 5.5.27 and HAProxy 1.3.15.6.

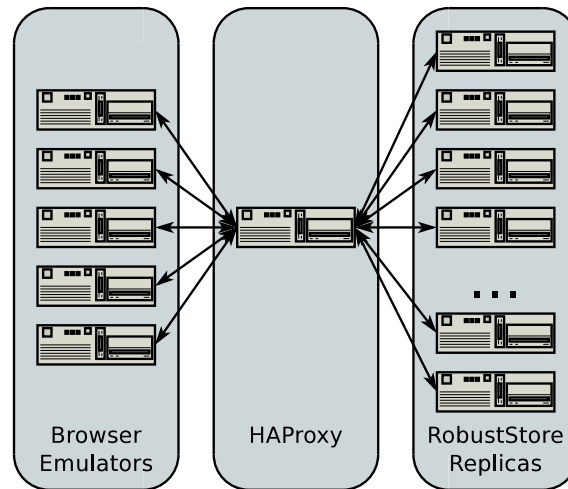


Figura 3.2: Experimental setup

The cluster has been divided into three disjoint sets of nodes as shown in Figure 3.2. The first set is composed by 5 client nodes that run the RBEs. Each client node holds the same number of RBEs. Instantiation and finalization of RBEs is done by a user initiated script, that computes and starts the exact number of RBEs necessary to generate the desired workload. Performance metrics are written by the RBEs into log files stored in the local disk. The second set contains from 4 to 12 server replicas that run the bookstore application. Each node of this set runs a copy of Tomcat that serves both static and dynamic web content. The application itself uses Treplica, as described in Section 3.4 and is configured to write only to the local disk. The final set contains only one node and runs the reverse proxy HAProxy, that has a load balancing module. The HAProxy is responsible for the failover mechanism. First, it actively queries the state of all of the server replicas using an HTTP probe. If it senses a replica is down (after 4 unsuccessful tries), it removes it from its servers list until it is probed active again. Second, requests are balanced among the server replicas using a hash mechanism based on unique client identifiers that are included in all interactions. If a server fails during the execution of a client request, HAProxy will close the connection and the client will observe an error.

RobustStore does not rely on a database, but the changes we have made to the application do not affect the data stored or the transactional semantics of the original application interactions. As a consequence, our experiments maintain the value of all experimental parameters as recommended by TPC-W, with one minor exception. To reduce the number of RBEs effectively required to provide a given load we changed

the default 7s think time of the TPC-W specification to 1s. With a 7s think time the workloads generated by the RBEs of the 5 client nodes were not sufficient to saturate RobustStore. It is important to note that shorter think times do not change either the read to write ratios nor the probabilistic characteristic of the workloads. Even with the reduced think time, we still had to set aside 5 nodes only to generate load. This left a maximum of 12 nodes to hold replicas, but this number is sufficient to emulate most commercial deployments of replicated application servers. Thus, the real systems that TPC-W is expected to assess are faithfully represented by our experimental setup.

The replicas were populated using the standard TPC-W population procedure, with 10,000 items and 30, 50 and 70 emulated browsers, even though we instantiated a larger number of RBEs. The parameter *number of browsers* was chosen to generate initial application state sizes of 300MB, 500MB, and 700MB, respectively. For the most write intensive profile (ordering) the average state size at the end of the measurement interval was approximately 550MB, 750MB, and 950MB, respectively. This respects the experimental requirement that all state must fit into main-memory. This is important to guarantee as much as possible that the performance variations observed are solely related with Treplica's activity on the network and on the disk. For all experiments the ramp-up, measurement interval and ramp-down periods follow TPC-W's specification; they were set to 30 seconds, 9 minutes and 30 seconds, respectively.

The TPC-W benchmark consists of a system specification, a workload and a metric. A dependability benchmark consists of a system specification, a faultload, a workload and a metric. Thus, to turn TPC-W into a dependability benchmark we added to it a faultload and metric specifications [35]. The faultload consists of environment or operator generated faults injected at precise times; all machines had their clock synchronized using NTP with clock skew smaller than 100ms. The time of failure was chosen to guarantee that full recovery of all failed replicas was observed within the experiment measurement interval. The abrupt server shutdown (crash) has been emulated by killing the application server at the operating system level. The abrupt server reboot (initiates a recovery) has been emulated by re-instantiating the application server. Re-instantiation of application servers is carried out automatically by a simple watchdog process that monitors the application server and re-instantiates it as soon as it detects the crash.

The dependability measures used in the experiments are availability, performability, accuracy, and autonomy [35]. The system under test is available when it is able to provide the service requested by the workload. **Availability** is defined as the ratio between the time the application is operational and the total duration of the run. **Performability** gives an idea of the impact of failures on the performance of the application. It is defined as the ratio between the average performance (AWIPS) during

the failure free period of the measurement interval and the average performance during the period of recovery. **Accuracy** is defined as the ratio between the number of requests with error and the total number of requests of the experiment. **Autonomy** is defined as the ratio between the number of human interventions required to restart a failed replica and the number of faults injected.

3.5.2 Speedup

Speedup experiments evaluate the maximum possible increase in performance obtained when RobustStore’s scale goes from 4 (baseline system) to 12 replicas. The relative speedup for a k -replicated RobustStore is defined by $S_k = \pi_k / \pi_4$, where π_k is the performance of a k -replicated application. Figure 3.3 shows the speedup values obtained for the three workloads and an initial state size of 500MB. For example, for the browsing workload, $S_8 \approx 1.59$, $S_{10} \approx 1.81$, and $S_{12} \approx 1.97$; the addition of four replicas to the baseline system increases its performance by nearly 60%. Treplica’s sub-linear speedups are a function of the costs associated with Paxos and Fast Paxos: the message complexity, latency complexity and the latency derived from writing data to stable storage. Thus, the different read/write ratios defined by the workloads pose increasing demands on Treplica’s efficiency in terms of network and stable storage. Web interactions that only read values can be executed without resorting to the total order broadcast. This is the case of browsing workload that has only 5% of updates, so 95% of requests (reads) can be fulfilled locally. Also, the small proportion of updates reduces access to disk. So, in this case the good speedup observed (Figure 3.3 browsing) can be explained by (i) the read-bound workload; (ii) the main-memory residence of the state; and (iii) the light use of the asynchronous persistent queue (total order).

The shopping workload generates 20% of updates, meaning that total order is going to be invoked for at least 20% of operations. In this scenario, the speedup is practically identical to the speedup obtained with the browsing workload. The maintenance of the good speedup for shopping can be explained by the same factors used to explain it for the browsing workload, despite the fact that the shopping load has *four* times the number of updates of the browsing workload. Here, the replicas can no longer be considered independent of each other due to their heavier use of the asynchronous persistent queue (Paxos). Each replica added produces a performance gain of $\approx 11.3\%$, with an associated increase in response time of $\approx 4.29\%$. The shopping workload is TPC-*W*’s *reference workload*. So, Treplica continues to speed up well when subject to TPC-*W*’s reference workload, but there must be a workload threshold after which the cost of uniform total-ordering impedes the maintenance of the good spee-

dups observed so far. Figure 3.3 shows that the ordering workload has by far crossed the threshold. In this case, RobustStore's S_8 has dropped to ≈ 1.29 . The change can be explained by the growth in the costs related to Treplica that now has to totally order half of the requests. Each replica added yields a performance gain of $\approx 5.35\%$, at the expense of a $\approx 37\%$ increase in the average response time.

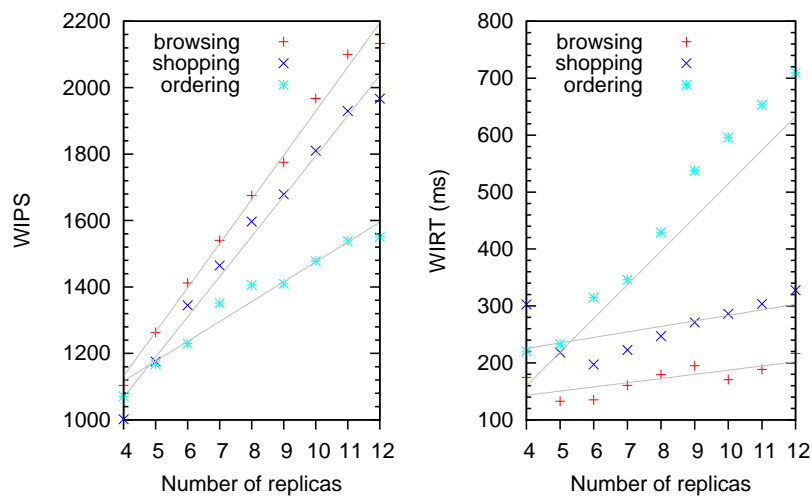


Figura 3.3: Speedup

3.5.3 Scaleup

Figure 3.4 shows how the system scales for a fixed workload of 1000 WIPS and increasing number of replicas. This measurements serve as a baseline to later assess the behaviour of Treplica in the presence of partial failures. An initial replica size of 300MB is used; this size has been chosen to minimize as much as possible interferences caused by swapping. A perfectly scalable system should show an horizontal scaleup line. The determination of the scaleup curves shown by RobustStore for each workload is important as it characterizes its behaviour when the scale is changed. To determine the curves we used regression analysis. The best fit for every set of points is given by a straight line, plotted in gray (confidence coefficients omitted) along the scaleup values (Figure 3.4). Additionally, we can ask ourselves how throughput (WIPS) is related to response time (WIRT). Correlation analysis of the two variables for each workload reveals that they are linearly correlated, with correlation coefficients: $r^2 = 0.8788$ for browsing, $r^2 = 0.9976$ for shopping, and $r^2 = 0.9958$ for ordering. The case $r^2 = 1.0$ corresponds to the maximum possible linear association between WIPS and WIRT, meaning that all data points will lie exactly on a straight

line. Thus, we have a system that has performance linearly correlated to response time and that scales up linearly. In Section 3.5.4 we use these observations to explain the behaviour of RobustStore after a crash.

RobustStore shows an ideal scaleup for the browsing workload, for the same reasons RobustStore shows a good speedup for browsing. For the shopping profile, RobustStore's scaleup is sublinear but with a gradual *linear* decrease in performance, approximately 0.85% per replica added, with a correspondent average increase of WIRT of $\approx 27.3\%$ (Figure 3.4). This is a good characteristic, showing that the expected impact of Treplica on the performance is *constant* as the system scales up. In fact, the actual cost of Treplica is smaller than 0.85% for this workload, because the costs inherent to RobustStore and its execution environment (JVM and Tomcat) were not subtracted from the 0.85%. For the ordering profile, each replica added to the configuration causes a constant performance drop of $\approx 2.1\%$, with an expected increase in WIRT of $\approx 25.9\%$ per replica added (Figure 3.4).

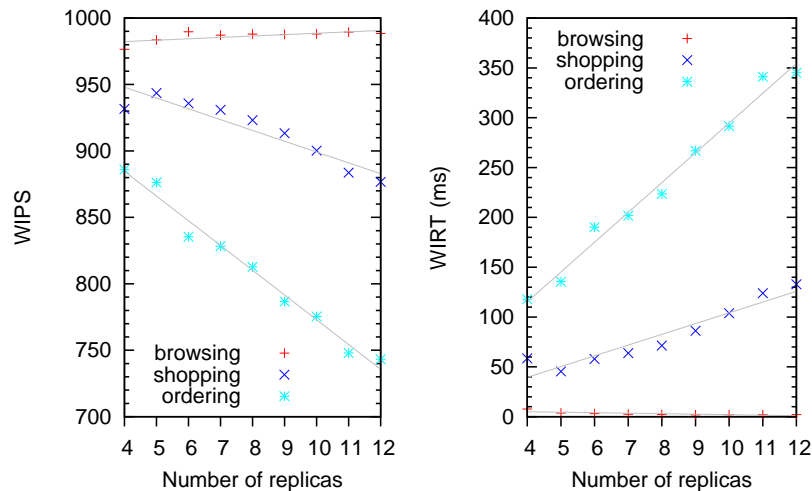


Figure 3.4: Scaleup for 1000 WIPS

The speedup and scaleup results characterize the behaviour of RobustStore in the absence of failures, but our main focus is not on raw performance but on what happens to performance and other important dependability indicators when RobustStore is subject to crashes.

3.5.4 One crash, one autonomous recovery

For the first experiment, one crash was injected at $t = 270s$, followed by the automatic triggering of recovery by the local replica watchdog. Figure 3.5 shows the behaviour

of a five-replicas RobustStore for the three workload profiles. As expected, all curves show a performance drop. Let us start with the curve for the ordering workload. There is a short (≈ 14 s) and sharp (≈ 700 WIPS) drop in performance. This load surge is caused by the HTTP proxy redistribution of the excess load among the active replicas. What is interesting to note is that after this short period, the recovery is still going to last for another 113s, but the average performance is already close to the performance before the failure. RobustStore's linear correlation between WIPS and WIRT (Section 3.5.3) can be used to analyse what happens in this scenario. Due to the correlation, a good estimate of the **worst case WIRT** can be obtained by simply considering WIPS as inversely proportional to WIRT. For example, in Figure 3.5, to estimate the latency at $t=275$ s (the bottom of the deepest valley for the ordering workload) we can subtract ≈ 140 WIPS from 841.4 average WIPS (Table 3.1, line 5/o, column failure-free AWIPS) to obtain the magnitude of the performance drop: ≈ 700 WIPS. Thus, in the worst case, the latency at $t=275$ s is estimated as ≈ 700 ms. Before the crash it was ≈ 50 ms, as estimated by the regression line in the scaleup WIRT (Figure 3.4) for 5 replicas. The value sampled by the RBE for the interval of 5s that includes the valley shows a latency of ≈ 613 ms. In Figure 3.5 it is possible to observe that the browsing and shopping workloads have much lower variability, so do, in the same proportion, the response times associated with them.

Table 3.1 contains the performability measurements for this experiment. Column R/P shows the replication degree and workload profile. For example, 5/b means five replicas, browsing workload. The variability of the load is characterized by the coefficient of variation (CV): the ratio of the standard deviation of the workload to its mean. The column PV shows the Performance Variation as a percentage of the failure-free AWIPS. Line 5/b shows that RobustStore delivers an average 977.4 WIPS_b with a CV of 0.01, almost no variation, during a failure-free run. It also shows that during the recovery period the performance drops to 898.28 WIPS_b (-8.1%); a small drop. For the shopping profile PV is smaller than 4% during recovery; performance remains practically stable during recovery. The CV values show that the browsing and shopping workloads have low variability, meaning that the PVs can be trusted to have been caused by the recovery. This is not the case for the ordering workload, with a CV of ≈ 0.20 for 5/o, and ≈ 0.33 for 8/o, they render the average WIPS useless as indicators of performance variation. The only resource available in this case is the WIPS histogram (Figure 3.5). There, it is possible to confirm that there was a performance drop during recovery, and that performance went back to its pre-crash level after the end of the recovery, but the estimated magnitude of performance drop during recovery, $\approx 13\%$, cannot be trusted due to the high CVs (Table 3.1, line 5/o, column PV).

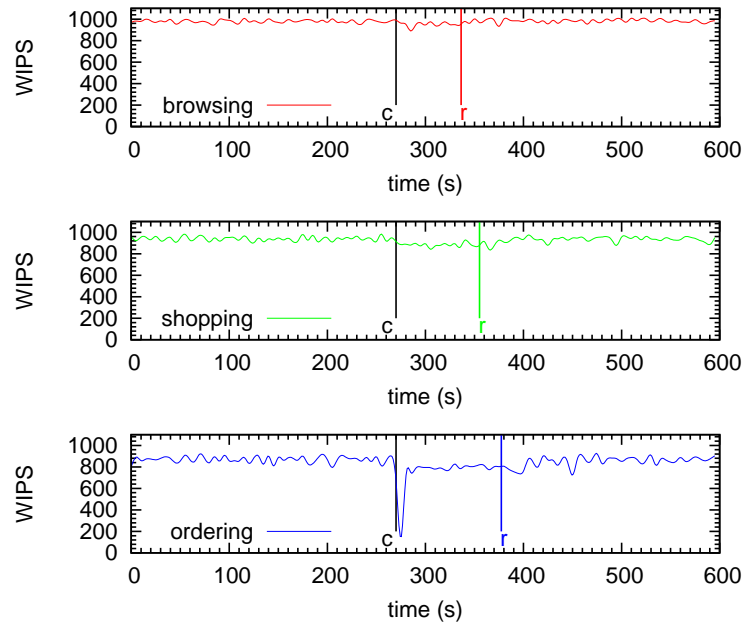


Figura 3.5: One failure: 5 replicas

R/P	failure free		recovery		PV (%)
	AWIPS	CV	AWIPS	CV	
5/b	977.4	0.01	898.28	0.01	-8.1
5/s	928.1	0.06	884.46	0.07	-4.7
5/o	841.4	0.20	732.33	0.24	-12.9
8/b	985.3	0.01	980.4	0.01	-0.5
8/s	916.8	0.01	903.88	0.09	-1.4
8/o	790.8	0.33	761.74	0.34	-3.7

Tabela 3.1: One failure: performability

As expected, the recovery times grow as the replica size grows. Figure 3.6 shows the recovery times for all one-failure experiments for three initial sizes of replica state (300MB, 500MB, and 700MB). For any replication degree, it is clear that recovery times grow faster for the browsing and shopping profiles, than they do for the ordering profile. This can be explained by the way recovery is handled by Treplica. Once a replica is rebooted, the application rebinds to its asynchronous persistent queue and requests the loading of the most recent checkpoint from stable memory. In parallel, the asynchronous persistent queue starts the recovery of the operations that have been enqueued by the remaining replicas since its failure, its backlog. For the browsing and shopping profiles the cost of queue resynchronization is relatively smaller than the cost of loading the most recent checkpoint from disk, so parallelization helps but still the time to recover is dominated by the loading of the checkpoint from disk. For the ordering profile, both state transfers become larger. In this case, the parallelization of the tasks contributes to a noticeable reduction of the total time of recovery, leveling the recovery times as we move across different state sizes, and reducing the impact of Treplica on RobustStore’s performance during recovery. For the next experiments we have omitted the recovery times to save space, but the same recovery pattern was observed.

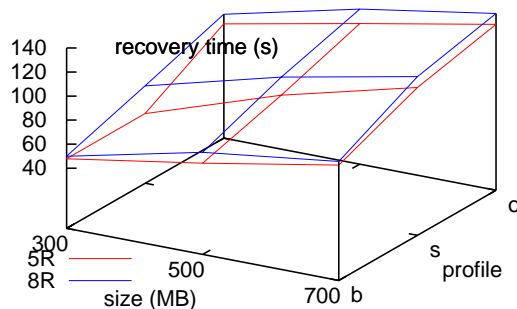


Figura 3.6: One failure: recovery times

Table 3.2 shows the accuracy of RobustStore in the presence of one crash. Clearly, RobustStore produces very few erroneous outputs when subject to one crash-recover failure.

replicas	browsing	shopping	ordering
5	99.999	99.999	99.985
8	99.999	99.999	99.986

Tabela 3.2: One failure: accuracy

3.5.5 Two crashes, autonomous recoveries

In this set of experiments RobustStore is subject to two concurrent crashes, followed by autonomous recoveries of the crashed replicas. The replicas to be crashed were chosen at random and crashed at $t=240s$ and $t=270s$. The WIPS histogram (Figure 3.7) shows small performance losses during recovery for all three workloads. For the browsing profile, the first replica crashed becomes operational at $t = 303s$, approximately 63s after the crash. The second replica re-joins RobustStore at $t=336.8s$, 66.8s after it crashed. In a little more than a minute the two replicas, with state sizes greater than 500MB, had already rejoined RobustStore. The shopping and ordering profiles also show that RobustStore recovers gracefully from the concurrent crashes even when exposed to increasingly write-intensive workloads. Table 3.3 shows that the largest PV is inferior to 5%, a drop that can be considered small given the adverse crash scenario generated by the faultload. The CVs for the ordering profile are high and similar to the ones observed before for one crash. Table 3.4 shows that RobustStore has maintained a high accuracy when submitted to concurrent crashes. From the point of view of maintainability and autonomy, RobustStore has so far shown that it can recover fully automatically, to a good extent due to its reliance on the simple recovery mechanism offered by Treplica (Section 3.2).

R/P	failure free		recovery		PV (%)
	AWIPS	CV	AWIPS	CV	
5/b	971.5	0.02	942.24	0.02	-3.0
5/s	910.4	0.09	876.58	0.09	-3.7
5/o	841.5	0.21	801.96	0.22	-4.7
8/b	982.8	0.01	962.6	0.01	-2.0
8/s	907.9	0.01	891.32	0.01	-1.8
8/o	787.1	0.33	763.96	0.34	-2.9

Tabela 3.3: Two overl. crashes: performability

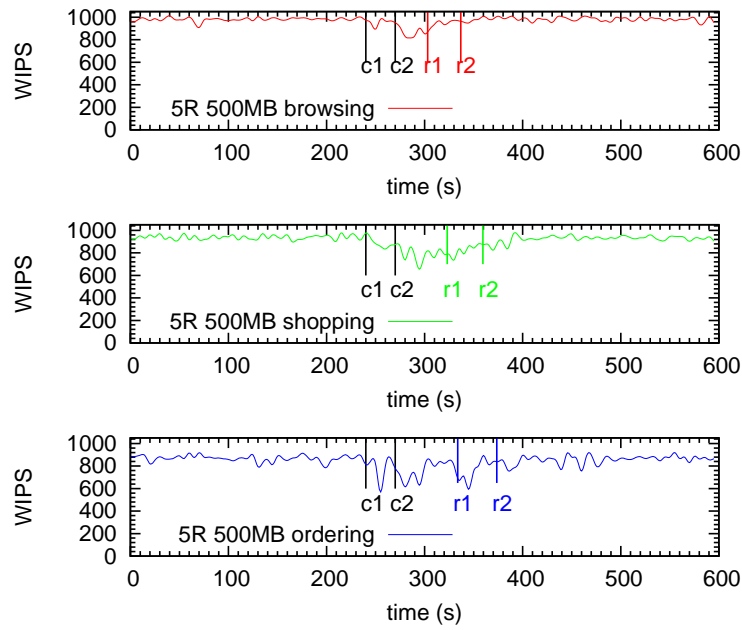


Figura 3.7: Two overlapped crashes

replicas	browsing	shopping	ordering
5	99.998	99.993	99.978
8	99.999	99.998	99.978

Tabela 3.4: Two overlapped crashes: accuracy

3.5.6 Two crashes, one autonomous, one delayed recovery

The last experiment has been designed to show how Treplica influences the performance of RobustStore in a scenario where a replica recovers long after it crashed. This is an important issue for Treplica because of how Paxos and Fast Paxos work. During the downtime of the crashed replica, the active replicas have delivered a large number of operations to the application. This means that the recovering replica is going to have to load the checkpoint from stable memory and spend a larger period learning (state transfer) from the other replicas, before it re-synchronizes itself and can resume normal operation. In this scenario (Figure 3.8), both replicas are crashed at $t=240s$. The recovery of one of the crashed replicas is triggered automatically. The recovery of the second replica is triggered manually at $t=390s$. Consider the shopping profile. At this moment, the first failed replica has already ended its recovery process, that took $\approx 70s$. The throughput curve shows that the recovery process implemented by Treplica has a small impact on performance of RobustStore for all workloads. Table 3.5 does not contain the CV values because they are very similar to the CV values obtained for the other two faultloads. Consider, for example, the shopping workload and five replicas. The impact on performance for the first failure is similar to the one verified in the case of two concurrent crashes. During a period of time RobustStore operates with 3 replicas, then the first failed replica recovers, taking RobustStore to 4 replicas. In the scaleup experiments using failure-free runs, we have observed that the addition of a replica causes an average performance drop of $\approx 8\%$. So a four-replicas RobustStore should perform an average 8% better. Recall that this reasoning is only valid because of the very low CVs shown by the shopping workload. The AWIPS during the period from r_1 to r_2 is 902.78 WIPS. The four-replicas RobustStore does not perform better because it is still processing the backlog of operations created by the two simultaneous failures, but it has recovered to a performance level that is only 1.4% below the performance before the crashes; the shopping workload has a CV = 0.09. The second recovery affects even less the performance of RobustStore, because the extra broadcasts demanded by the recovering replica to re-synchronize itself with the active replicas are processed concurrently by Treplica (Paxos). The consequence of this characteristic of Treplica is a reduced impact on performance stability, at the expense of a longer recovery time. (Figure 3.8). The same reasoning is valid for the other workloads, but, as stated before, the values of PV for the ordering profile are not valid because of the high variability of this workload. During these experiments, RobustStore's accuracy (Table 3.6) remained high and consistent with the accuracies found in the previous experiments.

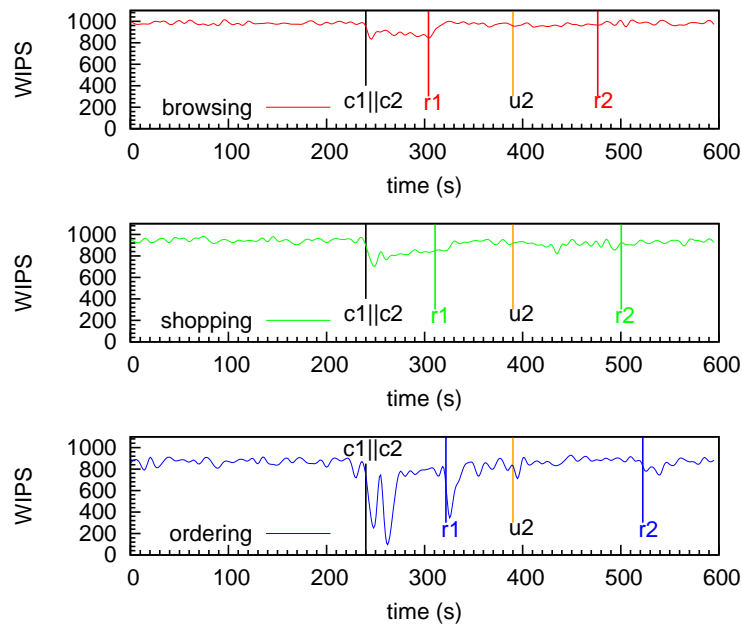


Figura 3.8: Delayed recovery

	no failures	recovery R1		recovery R2	
R/P	AWIPS	AWIPS	CV (%)	AWIPS	CV (%)
5/b	966.6	858.49	-11.1	919.58	-4.8
5/s	915.3	813.09	-11.2	905.89	-1.0
5/o	821.2	603.31	-26.5	852.12	+3.8
8/b	985.1	949.3	-3.63	948.65	-3.7
8/s	915.0	864.94	-5.5	906.01	-1.0
8/o	785.6	686.67	-12.6	802.08	+2.1

Tabela 3.5: Delayed recovery: performability

replicas	browsing	shopping	ordering
5	99.990	99.988	99.957
8	99.998	99.995	99.974

Tabela 3.6: Delayed recovery: accuracy

3.5.7 Discussion

Four questions were posed at the beginning of this Section. **1.** How long can RobustStore be expected to run without interruption? In the presence of only benign crashes, as assumed, RobustStore will remain operational forever. **2.** How much service can RobustStore be expected to deliver during failure-free and failure-prone operation periods? RobustStore's throughput can be characterized as very resilient, and stable in the presence of the crashes, failover, and recoveries used in the experiments. We have carried out 18 dependability experiments, 6 for each faultload specified. For each replication factor (5 or 8) three initial sizes of RobustStore replicas were instantiated (300, 500, and 700MB). All these experiments have shown that RobustStore loses less than 13% of its average performance during recovery in the worst case, which occurs with the faultload that injects two concurrent crashes, later followed by autonomous recoveries. The longest recovery occurred in the experiment with two crashes and delayed recovery of one replica. It took the second recovering replica about 180s to become operational in a setting with 8 replicas, ordering profile, and a 700MB state size. During the 180s recovery the average throughput practically remained at the same level displayed during the failure-free period. For the shopping profile, the profile considered by TPC-W as the one that best approximates the behaviour of a dynamic content web service, RobustStore worst average performance loss is inferior to $\approx 4.0\%$. **3.** What accuracy can be expected of RobustStore in failure-prone executions? Very high, three 9s, in the worst case. **4.** What level of human intervention is necessary to maintain RobustStore operational? None, when subject to the faultloads presented here, RobustStore has shown total autonomy. The combined effect of high accuracy, throughput resilience, and full autonomy allows the conclusion that RobustStore is indeed a highly available dynamic content web application.

3.6 Related Work

Paxos and recovery. Here we comment on work whose applications were built upon middleware that uses uniform repeated consensus (total order broadcast) [33]. Specifically, we are interested in toolkits that implement Paxos [56]. Examples of applications that satisfy this criteria include a lock service [19], data center management [49], data storage systems [62, 77], database replication [37], a distributed hash table system [51], and dynamic content web services [32, 69]. The projects listed in Table 3.7 have successfully employed the state machine approach [55] and *uniform* total order broadcast based on Paxos to replicate *critical* application state, with systems often combining different replication mechanisms to obtain the required degree of relia-

bility and performance. A key aspect of all papers listed in Table 3.7 is that their experiments were primarily designed to assess performance, not dependability, with the exception of FAB that shows fault-tolerance results for disk arrays. Most of the systems opted for the traditional message passing interface to expose Paxos, with the exception of Chubby. By contrast, we have opted to present uniform total order using a queue abstraction; queues are simple and widely-used objects.

There is much research on mechanisms to make dynamic content web applications highly available with emphasis on their performance improvement. Various reliable data management solutions have already been used, from file-based implementations (e.g., [29]) to database-based implementations (e.g. [22, 37, 8]). Tashkent’s experiments (Table 3.7) were carried out using a dynamic web content application. Sprint, FAB, and Chubby (Table 3.7) can be used as supports to build highly available dynamic content web applications.

Institution	Project Name	Paxos		1st Publ. Date
		Classic	Fast	
HP	FAB [77]	•		2004
Microsoft	Boxwood [62]	•		2004
EPFL/USI	Tashkent [36]	•		2006
Microsoft	Autopilot [49]	•		2007
Google	Chubby [19]	•		2007
USI	Sprint [22]	•		2007
UNICAMP	Treplica [87]	•	•	2008

Tabela 3.7: Paxos and Application Availability.

Replicated databases and recovery. Liang and Kemme [61] compare two recovery strategies: (i) total versus (ii) partial copy of the database. They assess the trade-offs of (i) and (ii) in runs where a single failure occurs. Manassiev [64] reports, using TPC-W and a faultload with a single crash, on the availability of a multiversion master-slave in-memory database that tolerates a single failure. They show that it is possible to reduce the impact of recoveries on the availability of the replicated database. Treplica offers a simpler recovery and failover solution that does not require the maintenance of hot backups for fast failover. Wu and Kemme [94] consider different recovery strategies depending on the failure scenario: (i) a single failed replica must be recovered or (ii) all replicas have to be recovered.

3.7 Conclusion

We have presented a dependability analysis of RobustStore, a highly available dynamic content web application built upon Treplica. Treplica's programming interface, based on only 8 methods, simplifies the programming tasks associated with the construction of highly available applications, relieving the programmer of important concerns related to the recovery. We like to consider Treplica as Paxos made simple *in practice*, a great benefit for developers of highly available applications. The experimental results show that RobustStore/Treplica performs well in the presence of crashes and recoveries, showing very good performance stability, continuous availability and high accuracy. They also contribute to a better understanding of the impact of Paxos and Fast Paxos when used as building blocks of a replication middleware.

From the point of view of dependability benchmarking, we have shown that not all workloads of TPC-W can be used as off-the-shelf indicators in dependability experiments. The coefficient of variation of the browsing and shopping workloads warrant them as good workloads for dependability assessment. Unfortunately, the same cannot be said about the ordering workload because of its high variability. This shortcoming of TPC-W can motivate further research on the development of dependability benchmarks for dynamic content web applications.

Capítulo 4

The Performance of Paxos and Fast Paxos

Paxos and Fast Paxos are optimal consensus algorithms that are simple and elegant, while suitable for efficient implementation. In this paper, we compare the performance of both algorithms in failure-free and failure-prone runs using Treplica, a general replication toolkit that implements these algorithms in a modular and efficient manner. We have found that Paxos outperforms Fast Paxos for small number of replicas and that collisions are not the cause of this performance difference.

4.1 Introduction

The construction of highly available asynchronous systems is intrinsically linked to solutions to the problem of consensus, because this problem is equivalent to a very powerful communication primitive: total order broadcast [28]. Among the consensus algorithms available, Paxos [56] and Fast Paxos [57] have recently been used to implement important systems [26] for at least the following reasons: (i) they implement uniform consensus; (ii) they are simple and elegant; and (iii) they are efficient. In theory, the number of communication rounds and the message complexity required by Paxos and Fast Paxos to reach consensus should be the determinant factors of their expected performance [75]. Fast Paxos, with smaller theoretical latency, should be faster and Paxos should be more resilient, by tolerating a larger number of failures. Fast Paxos reduces latency by being optimistic, that is, if the messages exchanged to reach consensus happen to be in a favorable order, then it is fast. This is the picture painted by theory. Practice can paint different pictures. Junqueira et al. [52] have pinpointed a scenario where Paxos shows a smaller overall consensus latency, if one of the communication steps is always much slower than the others. Their results serve

well to illustrate that determining the practical performance of Fast Paxos and Paxos can be a challenging task whose answers depend on careful experimentation.

In this work, we address the challenge of assessing Paxos and Fast Paxos efficiencies in practice in a LAN setting. We decided to start our study in the LAN environment because it houses most of the applications requiring the use of Paxos [26]. The evaluation presented here was only possible because we have programmed and tested both algorithms while building Treplica [87], a general replication toolkit that can be instrumented to generate the indicators necessary to assess the performance of these consensus algorithms. Our assessment method is based on looking at what the theory prescribes for the behaviour of the algorithms to design experiments that are intended to observe whether or not the prescribed behaviour occurs in practice. Examples of aspects assessed include number of messages ordered, latency of messages, quorum sizes and collisions. We have experimentally found, among other results, that Paxos outperforms Fast Paxos for small number of processes. Surprisingly, this isn't caused by unjustified optimism in Fast Paxos, but by the network and the extra load generated by the uncoordinated activities of its processes.

The remaining of the paper is organized as follows. Section 4.2 details the theoretical aspects of Paxos and Fast Paxos and the key differences between them. The prescriptions listed here were used as a guide for the design of the experiments. Section 4.3 describes our experimental setup, the experiments, and the results obtained. It also contains our assessment of the results and what they mean when contrasted with the theoretical predictions. We conclude the paper with a section on related work and a few concluding remarks.

4.2 Theory

Informally, the *consensus* problem consists in all processes in a distributed system proposing an initial value and all processes eventually deciding on the same value from the ones proposed. In this section we describe how Paxos and Fast Paxos solve consensus and we argue that there are many factors found in real systems that can affect their performance expectations.

4.2.1 Paxos and Fast Paxos

We give here a brief description of Paxos and Fast Paxos, to create a guide for the experiments. Full descriptions of both algorithms can be found in [57], including the computational and failure models assumed by them. Processes in the system are reactive agents that can perform multiple roles: a *proposer* that proposes values,

an *acceptor* that chooses a single value, or a *learner* that learns what value has been chosen.

To solve consensus, Paxos agents execute multiple rounds, each round has a *coordinator* and is uniquely identified by a positive integer. Proposers send their proposed value to the coordinator that tries to reach consensus on it in a new round. The coordinator is responsible for that round and is able to decide, by applying a local rule, if previous rounds were successful or not. The local rule of the coordinator is based on quorums of acceptors and requires that at least $\lfloor N/2 \rfloor + 1$ acceptors take part in a round, where N is the total number of processes in the system [57, 86]. Each round progresses through two phases with two steps each:

- In Phase 1a the coordinator sends a message requesting every acceptor to participate in round i . An acceptor accepts the invitation if it has not already accepted to participate in round $j \geq i$, otherwise it declines the invitation by simply ignoring it.
- In Phase 1b every acceptor that has accepted the invitation answers to the coordinator with a reply that contains the round number and the value of the last vote it has cast for a value, or *null* if it has not voted.
- In Phase 2a, if the coordinator of round i has received answers from a quorum of acceptors then it executes its local rule on the set of values suggested by acceptors in Phase 1b and picks a single value v . It then asks the acceptors to cast a vote for v in round i , if v is not *null*, otherwise the coordinator is free to pick any value and picks the value proposed by the proposer.
- In Phase 2b, after receiving a request to cast a vote from the coordinator, acceptors can either cast a vote for v in round i , if they have not voted in any round $j \geq i$, otherwise, they ignore the vote request. Votes are cast by sending them together with the round identifier to the learners.
- Finally, a learner learns that a value v has been chosen if, for some round i , it receives Phase 2b messages from a quorum of acceptors announcing that they have all voted for v in round i .

Fast Paxos changes Paxos by allowing the proposers to send proposed values directly to the acceptors. To achieve this, rounds are separated in *fast* rounds and *classic* rounds. Fast and classic rounds have different quorums with properties such that the local rule of the coordinator is still able to detect if a previous round was successful. These quorums are larger than the ones used by Paxos and can assume many values

that satisfy the requirements of the local rule. In particular, it is possible to minimize the number of processes in a fast quorum ensuring that both a fast and classic quorums contain $\lfloor 2N/3 \rfloor + 1$ processes. Another option is to minimize the number of processes in classic quorums requiring the same number of processes as in Paxos ($\lfloor N/2 \rfloor + 1$) but requiring $\lceil 3N/4 \rceil$ processes in the fast quorums [57, 86]. A Fast Paxos round progresses similarly to a Paxos round, except that Phase 2 is changed:

- In Phase 2a, if the coordinator of round i has received answers from a quorum of acceptors then it executes its local rule on the set of values suggested by acceptors in Phase 1b and picks a single value v . It then asks the acceptors to cast a vote for v in round i , if v is not *null*, otherwise, if i is a fast round the coordinator sends a *any* message to the proposers indicating that any value can be chosen in round i . In this case, the proposers can ask the acceptors directly to cast a vote for a value v of their choice in round i .
- In Phase 2b, after receiving a request to cast a vote from the coordinator (if the round is classic) or from one of the proposers (if the round is fast), acceptors can either cast a vote for v in round i , if they have not voted in any round $j \geq i$, otherwise, they ignore the vote request.

The above description of both algorithms considers only a single instance of consensus. However, these algorithms are more commonly used to deliver a set of totally ordered messages, where a sequence of repeated instance of consensus maps to a predefined position in the message ordering. In this case, it is possible to run Phase 1 and Phase 2a only once for all still unused instances. This factorization of phases is carried out immediately after the election of a coordinator. At this point, most of the consensus instances have not been started yet, allowing the coordinator in Paxos to “save” these instances for future use or, in Fast Paxos, allowing it to send Phase 2a *any* messages.

The improvement brought about by this factorization allows Paxos to achieve consensus in three communication rounds and Fast Paxos in only two communication rounds. Moreover, in Fast Paxos once the coordinator sends the *any* messages, consensus can be reached without the need of further coordinator intervention. Unfortunately, Fast Paxos cannot always be fast. Proposers can propose two different values concurrently, in this case their proposals may collide. Also, process and communication failures may block a round from succeeding. Different recovery mechanisms can be implemented to deal with collisions and failures, but eventually the coordinator intervention may be necessary to start a new classic round [57]. In both algorithms, any process can act as the coordinator as long as it follows the rule for choosing a value, if any, that is proposed in Phase 2a. The choice of coordinator and the decision

to start a new round of consensus are made relying in some timeout mechanism, as both Paxos and Fast Paxos assume a partially synchronous computational model to ensure liveness.

4.2.2 Performance Expectations

Before discussing the performance characteristics of Paxos and Fast Paxos experimentally, it is useful to map the theoretical notion of broadcast onto the actual primitive available in the experimental setup: high speed wired local area networks (LAN). The technology most often used to implement these LANs is Ethernet, in one of its several variations. Because of this heritage, it is commonly assumed that LANs use some sort of shared medium that must be collectively managed by the stations connected to the network. As a consequence, LANs messages can be broadcast to all stations with the same latency of sending a single message and, due to the shared nature of the medium, only one of such broadcasts can happen at the same time. This characteristic is very desirable, specially for optimistic algorithms such as Fast Paxos. However, not all variants of Ethernet work through a shared medium. In particular, 100Mbps and 1Gbps Ethernet are usually implemented with a full-duplex dedicated twisted-pair link connecting each station to a central switch in a star topology. In these networks communication is centrally arbitrated by the switch and there is no need for stations to manage access to the medium. This setup has many advantages to point to point communication, including full-duplex communication at full speed and a maximum aggregated bandwidth larger than the individual bandwidth of any link. Broadcast is still available, but it is not as straightforward as it was in the shared medium case. In these networks broadcast is just a single message multiplied by the switch and put in the dedicated medium of each station. As such, every one of these messages traverses a different queue and can potentially be ordered differently from other concurrent broadcasts and unicasts. Moreover, it is not uncommon for IP stacks to deliver locally a broadcast message even before it reaches the network interface.

Within this environment, what are the main differences between Paxos and Fast Paxos concerning the *expected* performance of both algorithms? Paxos requires 3 communication rounds for each instance of consensus while Fast Paxos needs only 2 communication rounds. Moreover, Fast Paxos doesn't require the active participation of a single process, the coordinator, in all instances of consensus. However, Fast Paxos requires the participation of a larger number of active processes than Paxos and the performance advantage of Fast Paxos is only realized in the optimistic case where there is no conflict. Considering these properties, it might be tempting to conclude that as long as the optimistic ordering of messages expected by Fast Paxos holds this

algorithm has the performance advantage. For each of the potential advantages of each algorithm we list now some reasons why this isn't necessary true:

Communication rounds: The main claim for the theoretical performance of Fast Paxos is that two communication rounds are better than three. However, both Paxos and Fast Paxos contain a communication step where all processes in a quorum broadcast a message at the same time. No matter how efficient the switch is, all these broadcasts will have to be serialized as they are transferred to all destination ports and they will be received as k individual messages. In this case, we can conceivably fold in a communication round all processing latency, but the propagation and transmission latency must be counted individually. That is, communication *complexity* is important.

Single coordinator: All Paxos messages must be relayed through a single coordinator. Although this process isn't a single point of failure, it is a potential performance choke point. Fast Paxos might perform better if load on the coordinator is high, but the centralizing nature of the coordinator can act as more robust way to decide on an order for the messages than relying on chance.

Larger quorums: Fast Paxos requires larger quorums and this has the direct consequence that the algorithm tolerates less process failures. Depending on the selection of quorums Fast Paxos can revert to Paxos quorums ($\lfloor N/2 \rfloor + 1$) if consensus is not optimistically reached, but this requires even larger quorums for the optimistic case. This fact has performance implications. Larger quorums require more messages to be successfully and timely delivered for consensus to be reached, making Fast Paxos vulnerable to network overload and timing violations.

Collisions: Fast Paxos is optimistic. It succeeds in two communication rounds as long as messages are naturally ordered. But, in switched LANs broadcasts are implemented as many messages send to each station, not necessarily ordered. If only a majority of these messages are ordered, consensus will be reached but will require more messages to be timely received. If not even a majority of messages is ordered, a collision occurred and consensus is not possible. There is nothing in the network that orders messages. If they arrive ordered it is more likely that they were not sent concurrently in the first place, thus collisions increase as the message rate increases [72].

Observing the uncertainties related to each supposed advantage of Fast Paxos, it is possible to reach the conclusion that these two algorithms are basically incomparable

without a clear characterization of the network properties. In the next section we present a set of experiments designed to extract data on this characterization for our target high speed local networks.

4.3 Practice

This section presents the basic organization of Treplica and where Paxos and Fast Paxos were used in the toolkit. Here, we also present the experiments we have carried out to assess Fast Paxos and Paxos, their results, and what they indicate in relation to the expected behaviour indicated by theory.

4.3.1 Treplica

Treplica is a replication toolkit that simplifies the development of high-available applications by making transparent the complexities of dealing with replication and persistence. We present here the basic organization of Treplica and where Paxos and Fast Paxos fit in the toolkit. Additional information on Treplica can be found in [87]. Treplica supports the construction of highly available applications through either the asynchronous persistent queue or the state machine programming interfaces. A queue is a *totally ordered* collection of objects with the usual *enqueue* and *dequeue* operations. Persistence guarantees that a process can crash, recover and bind again to its queue, certain that the queue has preserved its state and that it has not missed any additional enqueues made by any active replicas. An asynchronous persistent queue maintains a history of the objects it has ever held since its creation. Thus, by relying on the total order guaranteed by the queue and in the fact that queues are persistent, individual processes can become active replicas while remaining stateless; the persistence of their state has been delegated to the queue. The state machine programming interface leverages the persistent queues to provide a simple abstraction of an object that only changes state through deterministic command objects. To use this abstraction, applications must adhere to the state machine approach [55, 79].

To provide these two programming abstractions and still be able to provide reasonable performance, Treplica uses a *uniform* total order delivery mechanism built on top of Paxos. The uniformity of the consensus component is fundamental to the efficiency of Treplica. Usually, uniform consensus algorithms are more expensive than non-uniform consensus algorithms [33], however the higher price paid by such algorithms simplify tremendously the task of synchronizing persistent data local to the replica, specially in the case of failure. It also allows for a natural way to aggregate the local stable storage of each replica in a global persistent store, without requiring

any single replica to assume special duties. In Treplica the *ledger* abstraction of Paxos is the central data structure of the whole toolkit. As a consequence, there is just a thin software layer between the application and the Paxos implementation. Thus, Treplica doesn't add much overhead to the algorithm and our performance data is very close to a "pure" Paxos implementation.

However, there are two factors that characterize and separates the data obtained with Treplica from other Paxos implementations: state machine execution and operation parallelism. First, we made our experiments using the state machine abstraction of Treplica, so our response times are not equivalent to the consensus latency, but operation execution latency. This means that, on top of the consensus latency, we have to add the processing time required to apply the command object to the local replica. As described in the next section, we selected an application such as to minimize this cost, but nevertheless this latency is present. Second, as the state machine abstraction requires sequential execution of command objects, we must employ parallelism internally in Treplica to avoid the critical path comprised by the Paxos ordering and command execution to become a bottleneck. Thus, we try as much as possible to pack many command objects in the same Paxos message, without adding to the overall latency. This way, a multithreaded application can obtain a higher throughput but the final response time deviates further from the basic consensus latency. As our objective is to relatively compare Paxos and Fast Paxos, these effects can be factored out as they affect both implementations equally. Moreover, both Paxos and Fast Paxos are implemented by the same code inside Treplica, actually a Fast Paxos implementation that can be configured to generate only classic rounds, behaving exactly like Paxos. Thus, all implementation details are shared by the two algorithms and the comparison obtained is as fair as possible.

4.3.2 Experimental Setup

The experiments were carried out in a cluster with 18 nodes interconnected through the same 1Gbps Ethernet switch. Each node has two Intel Xeon 2.4GHz processors, 1GB of RAM, and a 40GB disk (7200 rpm). System software in each node include Fedora Linux 9 and OpenJDK Java 1.6.0 virtual machine. We used 4 to 16 nodes in our experiments and each node operated as a server replica and as a load generator.

The server replicas run a simple replicated hash table. The application is a wrapper over the standard Java hash table implementation, with the same API, but adding replication and persistence support through Treplica. As such, only operations that change the internal state of the hash table employ Treplica, the read only operations are executed directly. Treplica is configured in the server replicas to use local disk as

its persistent data store, and no network activity is expected of each node beyond the one generated by Treplica.

The load generation consists in a sequence of put operations, where each operation associates a sequential integer with a random 5 character string. It would be possible to interleave read with writes in our load, creating distinct usage profiles. However, due to the simplicity of the application, this probably would only increase the observed performance by the proportion of reads used as they are orders of magnitude cheaper than writes. Thus, we decided to concentrate on a load composed only of hash table writes to analyze the data as if Treplica were the only possible bottleneck. The generated load is measured in operations per second (op/s) and is generated with a fixed rate divided equally among all the load generators of the system. Server replicas and load generators share the same hosts, but care was taken to ensure that the load generation wasn't competing with the application processing and that the specified load rate was being generated.

4.3.3 Experiments

Based on the performance expectations of Paxos and Fast Paxos listed in Section 4.2.2 we devised five experiments and four metrics to compare both algorithms:

Scale up: For a fixed generated load of 2000 op/s we increase the scale of the system from 4 to 16 replicas. For each point we count the load served in op/s and the average response time for each operation.

Speed up: For a fixed number of 4 and 8 replicas we increase the generated load from 100 op/s to 4000 op/s. For each point we count the load served in op/s and the average response time for each operation.

Quorum size: We perform the scale up and the 8 replicas speed up experiments with a modified version of Paxos that uses a larger quorum than necessary ($\lfloor 2N/3 \rfloor + 1$). We count the load served in op/s.

Retries and collisions: We extract the number of failed consensus instances and collisions from the scale up and the 8 replicas speed up experiments. We count the number of failed consensus rounds and the number of collisions per total consensus rounds.

Failures: For a fixed number of 8 replicas and a fixed load of 2000 op/s we simulate the failure of a non-coordinator replica or a coordinator replica. We count the load served in op/s.

We run all experiments with Paxos, Fast Paxos with large fast quorums ($\lceil 3N/4 \rceil$) and Fast Paxos with small fast quorums ($\lfloor 2N/3 \rfloor + 1$). The scale up and speed up experiments were intended to give a general performance evaluation, and can be used to assess if the smaller number of communication rounds required by Fast Paxos and the fact that this algorithm doesn't have a single performance bottleneck make it more efficient. The quorum size experiment allows us to measure the cost of waiting and processing a larger number of messages to achieve consensus, indicating if the larger quorums required by Fast Paxos are acceptable. The retries and collisions experiment will show the number of retried consensus instances, an indication of the number of lost messages and timing failures that can be used to quantify the cost of larger quorums and of a single coordinator. This experiment also shows the proportion of collisions found in the other experiments to make explicit the cost that Fast Paxos pays for being optimistic. The failures experiment shows how both algorithms handle failures and if a single coordinator can negatively affect the performance of Paxos in case of failure.

4.3.4 Scale Up

Figure 4.1 shows the data for the scale up experiment with a constant load of 2000 op/s. The chart on the left shows the served operations per second as a function of the number of replicas in the system. The chart on the right shows the response time for the same points.

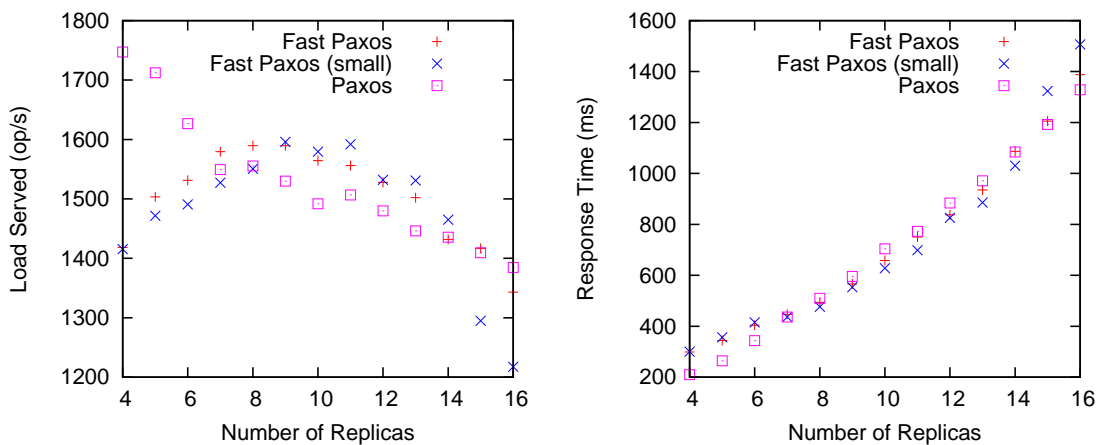


Figura 4.1: Scale up (2000 op/s)

The most striking observation from this experiment is that Paxos outperforms Fast Paxos for small replica numbers. Up until 7 replicas Paxos is better, and with more

than 7 replicas both are roughly the same. Many factors can justify this behavior, as pointed in Section 4.2.2, but we believe it is caused by the stabilizing effect the single coordinator creates in the system, reducing timing violations. To fully justify this supposition we need to analyze the data from the quorum size and retries experiments. Another interesting behavior is the fact that Fast Paxos increases its performance up to a maximum at about 9 replicas. Again, we believe this effect is related to timing violations and we justify it using the data for the retries experiments. Both variants of Fast Paxos fare similarly in all replica configurations, with a slight advantage for the large quorums version. This indicates that the quorum size has a role in the performance of the algorithms but it isn't a very important one. Once more, this explanation will be verified by the quorum size experiment data. Average response time grows with the number of replicas and all algorithms tested have roughly similar numbers. This is mostly a consequence of the fact that many operations are being ordered in the same Paxos instance and that the load generated is dependent on the load served.

4.3.5 Speed Up

Figure 4.2 and Figure 4.3 show data for the speed up experiment for 4 and 8 replicas, respectively. In both figures, the chart on the left shows the served operations per second as a function of the rate of generated operations per second. The chart on the right shows the response time for the same points.

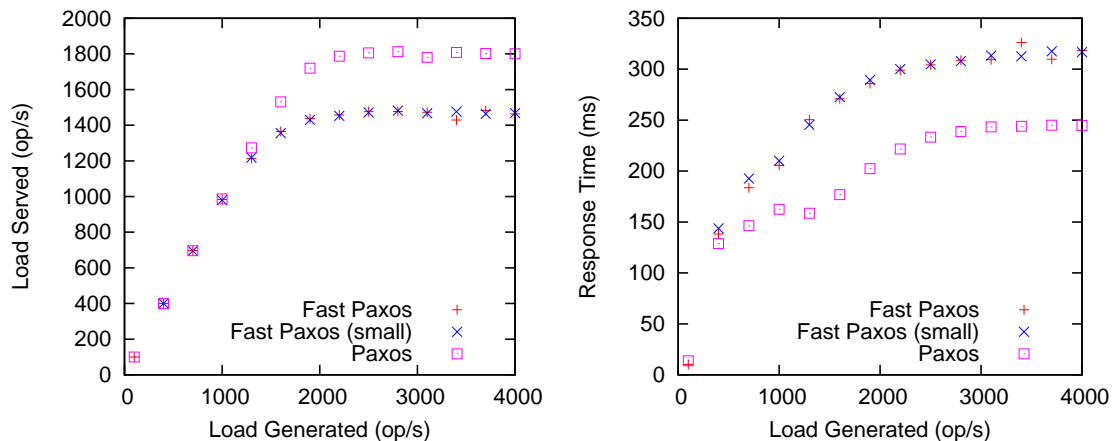


Figure 4.2: Speedup (4 Replicas)

For both 4 and 8 servers the increasing tendency of served operations is similar. The served load rises linearly, following the generated load, up until a peak point where it stabilizes. This was expected and shows that the performance difference

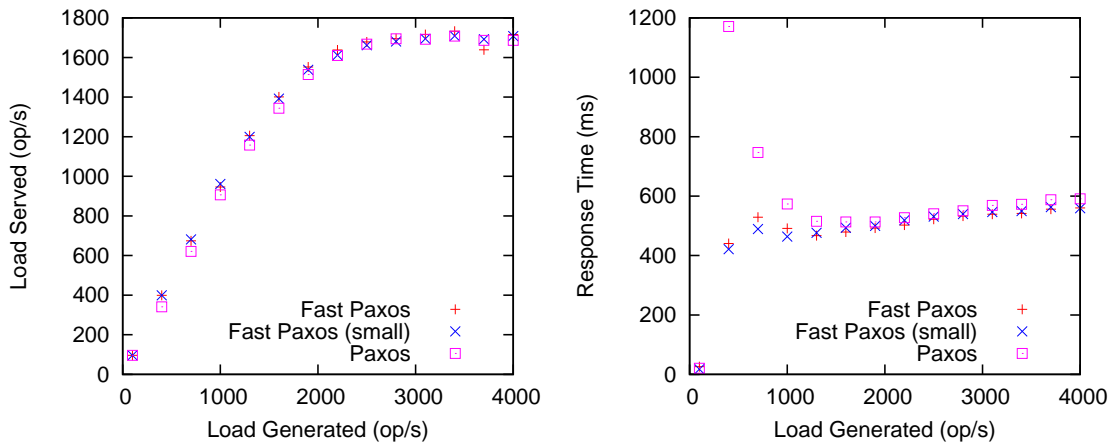


Figura 4.3: Speedup (8 Replicas)

among the algorithms, when present, only shows after the peak load is reached. Before that point Paxos and Fast Paxos should behave the same way, only trading places as the number of replica increases as shown in the scale up experiment. The latency charts are more interesting. Latency also rises to reach a plateau, but much faster in the case of 4 replicas and even surpassing it in the 8 replicas case. This is explained by the fact that many operations are bundled in the same consensus instance, and such instances are fairly costly. In our data sets a little more than 150 consensus instances are completed per second in the best case. Thus, when the load is light a less aggressive bundling takes place and latency suffers. This is a property of our implementation and not necessarily will be found in other environments.

4.3.6 Quorum Sizes

To test the effect of quorum sizes we run the scale up and 8 replicas speed up experiments using a modified version of Paxos that uses quorums of $\lfloor 2N/3 \rfloor + 1$ replicas and compare it with regular Paxos. Figure 4.4 shows the data obtained.

Data from this experiment confirms that quorum sizes aren't a relevant factor for performance when the number of replicas is moderate (less than 15). This is also true for the scale up experiment and the two variants of Fast Paxos. Two factors justify this finding. First, with the total number of replicas in the 4 to 15 range, the absolute difference in the cardinality of quorums is very small, two replicas at most. Second, timing violations are more probable if a learner has to receive a message from more processes. This second hypothesis is confirmed by the data collected on consensus rounds retries presented next.

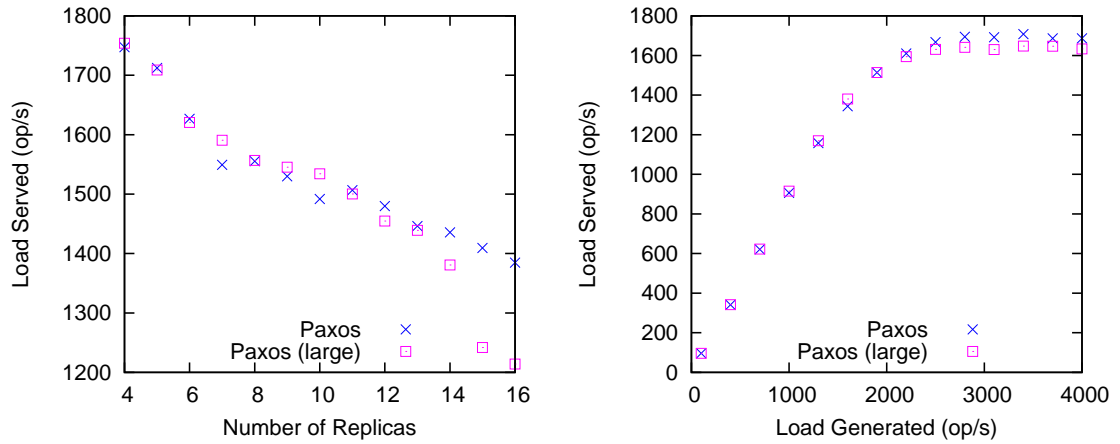


Figura 4.4: Paxos with Large Quorums

4.3.7 Retries and Collisions

Figure 4.5 shows the number of retried consensus instances for Paxos and Fast Paxos and the number of collisions for Fast Paxos observed in the scale up and 8 replicas speed up experiments. Both numbers are presented as relative values to the total number of consensus instances executed.

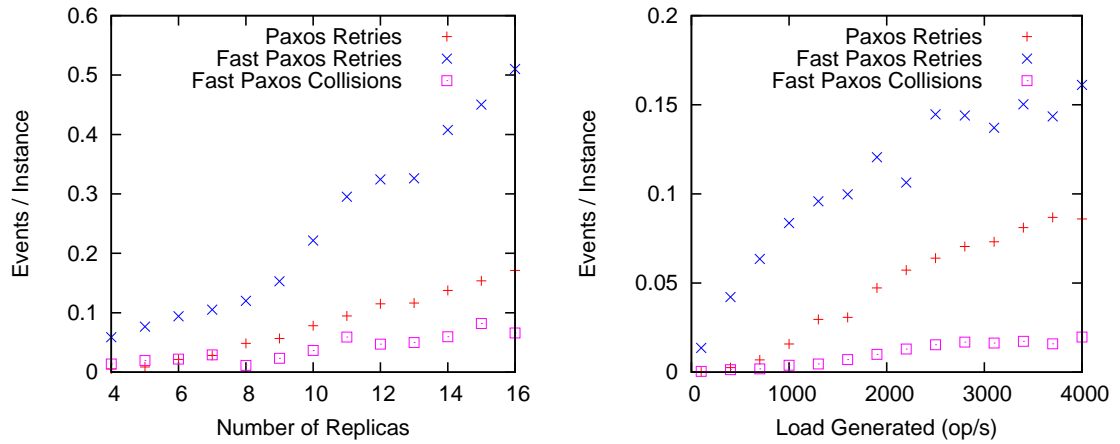


Figura 4.5: Retries and Collisions

This experiment produced vital information about the performance of Paxos and Fast Paxos. The optimism of Fast Paxos could be considered its weak spot and could justify its inferior performance with fewer processes. However, our data shows that collisions do occur but they are responsible for only a small percentage of the retried consensus instances of Fast Paxos. Lost messages or, more likely, timing violations are

responsible for the most part of consensus failures. Each consensus failure triggers a regular *Paxos* consensus round, even for Fast Paxos, and this round is costly as it must execute all 2 phases of the algorithm. The number of failed consensus attempts in Fast Paxos is sometimes 3 times larger than in Paxos and can account for the decreased performance. The cause of these timing violations is probably the fact that timeouts in Fast Paxos are managed by all replicas at the same time. Any replica that believes a consensus round should have been finished alerts the coordinator that in turn starts a full Paxos round, thus we multiply the possibility of a timing violation by the number of replicas in the system. In Paxos, only the coordinator decides when a round must be retried. It may not be more accurate, but the possibility of timing failure is smaller. Moreover, even when a conflict does not arise in a Fast Paxos round, it may be possible for the processes in the system to observe a “partial conflict” where some, but less than a majority, of replicas vote for a different operation. In this case, more messages must be timely received for the consensus to be reached, increasing the chance for timing violations. While this accounts for Fast Paxos limitations, it is still necessary to explain why Paxos loses its advantage at about 8 replicas. The first cause is that the single coordinator only acts as a stabilizing factor as long as it is not overloaded. As soon as the coordinator gets overloaded it starts dropping messages and prematurely restarting consensus rounds.

4.3.8 Failures

Figure 4.6 shows one execution with 8 replicas and load of 2000 op/s that suffers the failure of a single replica. The failure is simulated by killing the replica at the operating system level and by immediately re-instantiating it back in operation. The charts in the left show the failure of a regular replica and the charts in the right show the failure of the coordinator replica. In all charts the first vertical bar shows the moment when the replica is forcibly shutdown and the second bar shows the moment when the replica finishes its *local* recovery and starts to coordinate with the other replicas.

In both cases it is possible to notice that failure itself doesn't impact the throughput of the system. This is reasonable considering the data from the scale up experiment; less replicas can potentially give more performance. The interesting observations is that it is the replica reintegration that negatively affects the throughput of the system. When a replica finishes its local recovery it has only learnt the operations up to the moment of its failure, and must catch up with the others replicas. This process puts demand on the network and on the coordinator as all missed decisions are relayed to the recovering replica. Another intriguing aspect is the large difference in local reco-

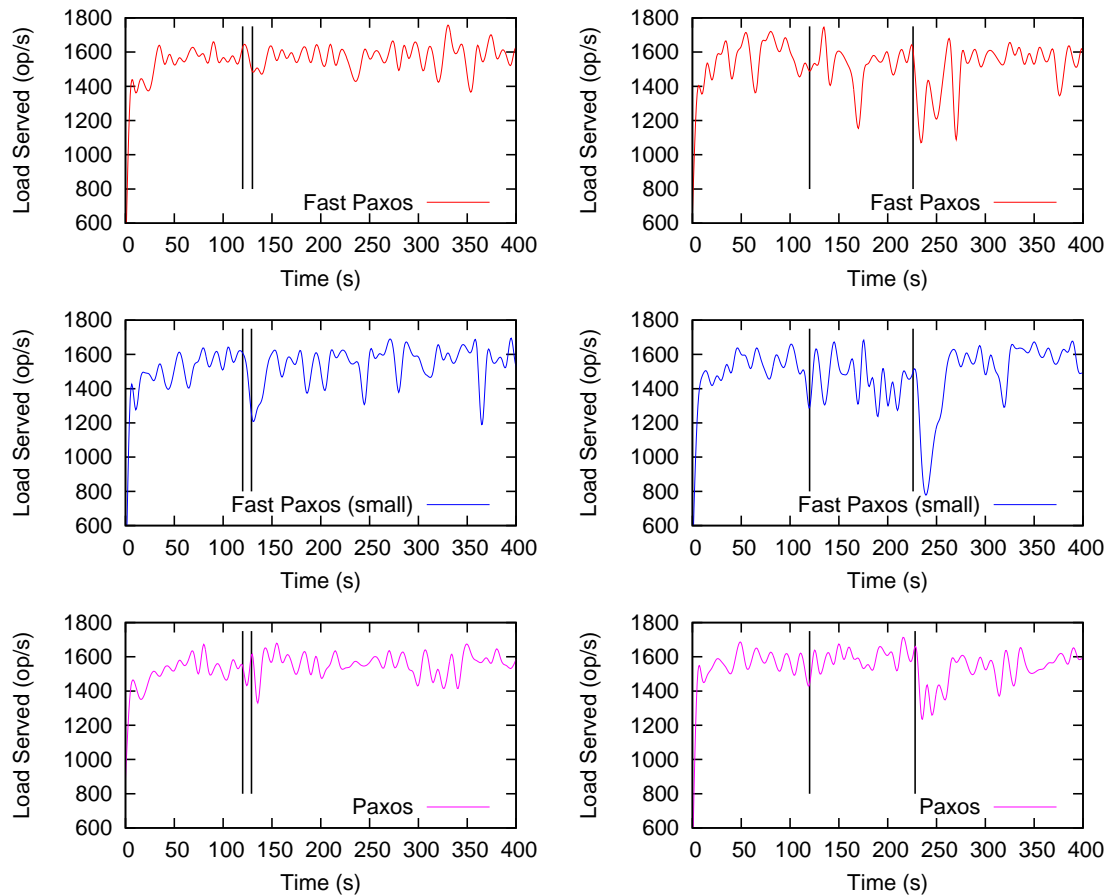


Figure 4.6: Single Failure (8 replicas, 2000 op/s)

very times between a normal replica and a coordinator replica. Due to the observed fast local recovery, a normal replica easily reintegrates in the system and only creates minimal disruption. The failure of the coordinator replica isn't felt any differently by the system, as a new coordinator is promptly elected, but the local recovery of the coordinator replica takes a longer time. This happens due to the larger state held in memory by the coordinator, that requires more information to be brought back from disk on recovery. As a very damaging side effect, the longer a replica stays out of the computation for any reason, the longer its reintegration will take and larger the disruption caused by it will be. Finally, all tested algorithms displayed a very similar behavior under failures, even when the coordinator has failed. This indicates that the coordinator only affects the performance of Paxos as a bottleneck in the steady state. In the presence of failures, coordinator election is performed without interrupting the operations flow.

4.4 Related Work

Paxos and Fast Paxos are well understood algorithms, but until recently, seldom implemented. A very clear and concise description of both algorithms can be found in [57]. The theoretical performance of Paxos is described in detail in [75]. Probably due to the lack of actual implementations, one of the first works to delve in the Paxos performance employed simulation [83], and compared Paxos to Chandra-Toueg rotating coordinator consensus algorithm [28].

Recently, motivated by the need of dependable coordination services for scalable distributed systems, Paxos implementations are becoming more common and works analysing their performance are being published. The Chubby system used at Google is described in [26], with some basic performance figures. A detailed description of a Paxos implementation encompassing all aspects of a complete state machine replication system can be found in [7]. In this work it is presented a fairly complete study of the performance of the described implementation under different state machine replication suppositions. A description of a variant of the Paxos algorithm optimized for the implementation of a distributed lock management system and an analysis of its performance can be found in [48].

All of the above cited works evaluate only Paxos. The only work we have knowledge of that attempts to quantitatively compare Paxos and Fast Paxos is [52]. This work employs simulation to study a particular configuration where Fast Paxos doesn't have a better consensus latency than Paxos. Restricted as the studied configuration might be, this work showed for the first time that increased latencies of individual messages can drastically change the behavior of Paxos and Fast Paxos.

4.5 Conclusion

We have presented a comparative analysis of the performance of Paxos and Fast Paxos in the context of high speed local area networks. We have discovered scenarios where Paxos has lower latency than Fast Paxos and we showed evidence of the cause of such behavior. To the best of our knowledge this is the first such comparison.

Our experimental data indicates that Paxos is faster for a small set of replicas and owns its performance to the stability provided by its single coordinator. The Paxos coordinator makes fewer timeout mistakes, needs to retry consensus rounds less often and is immune to collisions, however it can be overloaded by a large number of replicas. Fast Paxos suffers from timing failures and lost messages, but its lack of reliance on a single coordinator allows it to operate more efficiently with more replicas. We have also discovered that quorum sizes and collisions aren't very determinant in the

relative performance of these algorithms and that the single coordinator of Paxos isn't particularly affected by failures.

As replication is used as a device for fault tolerance, the fact that Fast Paxos is more effective with a larger number of replicas is effectively cancelled by the fact that it requires larger quorums of active replicas to function. For example, a system using Paxos needs 7 replicas to tolerate 3 replica failures while Fast Paxos requires 12 replicas to guarantee the same resilience. Thus, unless Fast Paxos can be made more efficient in its use of the available network, avoiding the timing failures observed, its use is hardly justified.

Capítulo 5

On the Coordinator's Rule for Fast Paxos

Fast Paxos is an algorithm for consensus that works by a succession of rounds, where each round tries to decide a value v that is consistent with all past rounds. Rounds are started by a coordinator process and consistency is guaranteed by the rule used by this process for the selection of v and by the properties of process sets called quorums. We show a simplified version of this rule for the specific case where the quorums are defined by the cardinality of these process sets. This rule is of special interest for implementors of the algorithm.

5.1 Introduction

The problem of deciding a single value out of a set of values proposed by processes is known as the *consensus* problem. This problem is easy to solve in the absence of failures, but it is impossible to solve in an asynchronous distributed system even if a single process fails by permanently stopping [38]. A better approximation of the failures that processes of a real distributed system can suffer is the one where processes stop but may later recover. Unfortunately, the impossibility also holds for these systems. One of the ways to get around the impossibility is to design algorithms that do not violate their safety requirements while the system behaves asynchronously and are certain to make progress if the system behaves partially synchronously for periods long enough to satisfy the progress requirements. Designing consensus algorithms for the asynchronous crash-recovery model is a difficult task of practical interest and probably Paxos [56] and Fast Paxos [57] are the most studied solutions so far.

Fast Paxos solves the consensus problem through a succession of rounds that lead to the choice of the consensus value. In each round a distinguished process, the *coor-*

dinator, is responsible for picking a single value using a rule that is based on quorums of processes. Quorums of successive rounds are used to guarantee that if a single value has been chosen or might ever be chosen in previous rounds then the same value is going to be chosen in the current round. Thus, quorums are fundamental to the correctness of Fast Paxos because they are ultimately responsible for the validity of consensus.

Lamport [57] shows how quorums can be characterized using the cardinality of sets of processes, defining what minimum number of processes represents a quorum. However, he defines the coordinator's rule in terms of quorum sets and general set operations. This complete characterization of the coordinator's rule is perfect for the purposes of his work, but it does not address thoroughly the needs of a programmer who wants to implement it. Therefore, the main contributions of this paper are (i) an interpretation of the Fast Paxos coordinator's rule only in terms of the cardinalities of quorum sets, and (ii) its simplification. The simplified interpretation is efficient, easier to implement and test; it can help developers to create reliable implementations of Fast Paxos. This is important, as the use of Fast Paxos to build fault-tolerant applications is bound to require the execution of a very large number of consensus instances.

5.1.1 Fast Paxos

Before detailing the coordinator's rule, it is useful to give a very brief overview of Fast Paxos; a complete description of it can be found in [57]. The algorithm is easier to explain in terms of reactive agents that represent a role, such that a single process can enact multiple agents, with each one of them playing a different role. An agent can enact one of the following main roles: a *proposer* that can propose values by sending them to acceptors, an *acceptor* that chooses a single value, or a *learner* that learn what value has been chosen.

To solve consensus, Fast Paxos agents execute multiple rounds, each round has a coordinator and may be either a *fast* round or a *classic* round. Positive integers are used to uniquely identify rounds, each identifier determines the coordinator and indicates the round type: fast or classic. Regardless of its type, each round progresses through two phases with two steps each:

- In Phase 1a the coordinator sends a message requesting every acceptor to participate in round i . An acceptor accepts the invitation if it has not already accepted to participate in round $j \geq i$, otherwise it declines the invitation by simply ignoring it.
- In Phase 1b every acceptor that has accepted the invitation answers to the coordinator with a reply that contains the round number and the value of the last

vote it has cast for a value, or *null* if it has not voted.

- In Phase 2a, if the coordinator of round i has received answers from a quorum of acceptors then it executes its rule on the set of values suggested by acceptors in Phase 1b and picks a single value v . It then asks the acceptors to cast a vote for v in round i , if v is not *null*, otherwise, if the round is fast the coordinator sends a *any* message to the proposers indicating that any value can be chosen in round i . In this case, the proposers ask the acceptors to cast a vote for a value v of their choice in round i .
- In Phase 2b, after receiving a request to cast a vote from the coordinator or from one of the proposers, acceptors can either cast a vote for v in round i , if they have not voted in any round $j \geq i$, otherwise, they ignore the vote request. Votes are cast by sending them together with the round identifier to the learners.
- Finally, a learner learns that a value v has been chosen if, for some round i , it receives Phase 2b messages from a quorum of acceptors announcing that they have all voted for v in round i .

As Fast Paxos agents may crash and recover, they must save their state in stable memory so that agents, once recovered, can remember the votes they have cast earlier. The sequence of steps described above imply that a learner can only learn the value of consensus after a period of at least four message delays. If numerous executions of Fast Paxos are required, then it is possible to run Phase 1 and Phase 2a only once for all these instances. This factorization of phases is carried out immediately after the election of a coordinator. At this point, most of the consensus instances have not been started yet, allowing the coordinator to send Phase 2a *any* messages. The improvement brought about by this factorization allows consensus in two message delays, making Fast Paxos an optimal consensus algorithm [58]. Unfortunately, Fast Paxos cannot always be fast. Proposers can propose two different values concurrently, in this case, their proposals may collide. Also, process and communication failures may block a round from succeeding. Different recovery mechanisms can be implemented to deal with collisions and failures, but eventually the coordinator intervention may be necessary to start a new round [57]. Any process can act as the coordinator as long as it follows the rule for choosing a value, if any, that is proposed in Phase 2a.

As already mentioned, quorums are fundamental for Fast Paxos. Quorums are set of processes and each round has a set of quorums associated with it, classic quorums for a classic round and fast quorums for a fast round. For the proper operation of the algorithm, quorums have to satisfy properties on the sets of processes that form them. Specifically, any two quorum sets must have a non-empty intersection and any

quorum and any two fast quorums from the same round must also have a non-empty intersection [57]. There are many ways to define quorums, but a very interesting one from the point of view of process fault tolerance is the definition based only on the number of processes contained in each quorum. The definition of quorum using this parameter is straightforward and is described in Section 5.2.

The coordinator's rule determines how a coordinator can consistently start a new round, after collecting information about previous rounds from the acceptors. That is, for each round i the coordinator is about to start, it must know if a value v had been decided or might have been decided in previous rounds $j < i$. The coordinator's rule of Fast Paxos must take into account that in a fast round, more than one value might have been proposed and voted concurrently. Quorums are defined to guarantee that only one, if any, of the conflicting proposed values is selected through the application of the coordinator's rule. Section 5.3 presents the original coordinator's rule as defined in [57] and then shows how this rule can be effectively implemented using a cardinality-based definition of quorums. Section 5.4 brings our derivation of a simplified cardinality-based coordinator rule, it is stricter than the one presented in Section 5.3, but it is easier to understand and to implement. Section 5.5 closes the work by commenting on the practical value of our main result: a simplified coordinator's rule for Fast Paxos.

5.2 Choosing Quorums

The quorum requirements for Fast Paxos assert that: (a) any two quorums must have non-empty intersection, (b) any two fast quorums and any classic or fast quorum from the same round have a non-empty intersection [57]. We can satisfy these conditions by considering only the number of process in each quorum, where N is the number of acceptors, and F and E are the maximum number of failed acceptors in classic and fast rounds, respectively [57]. A classic quorum is formed by $N - F$ acceptors and $N - E$ acceptors form a fast quorum. As the requirements for fast quorums are always stricter than those for classic quorums, we can always assume that $E \leq F$. The quorum conditions [57] are then stated as:

$$N > 2F \tag{5.1}$$

$$N > 2E + F \tag{5.2}$$

For a fixed N , F and E can be chosen in various different ways and a natural way of choosing them is by maximizing one or the other [57]. As we have $E \leq F$, maximizing

E leads to $E = F$. Thus, we can satisfy the system only with $N > 3F$ and:

$$N > 3F \Leftrightarrow F < N/3 \Leftrightarrow F \leq \lceil N/3 \rceil - 1$$

For this case, the cardinality of any classic quorum ($|Q_c|$) or fast quorum ($|Q_f|$), expressed only as a function of N , is:

$$|Q_c| = |Q_f| \geq N - \lceil N/3 \rceil + 1 \geq \lfloor 2N/3 \rfloor + 1$$

If instead we maximize F , the limit for its value is given by the Equation 5.1, thus:

$$N > 2F \Leftrightarrow F < N/2 \Leftrightarrow F \leq \lceil N/2 \rceil - 1$$

In this case E must be chosen to satisfy Equation 5.2, considering the value of F we have just chosen:

$$N > 2E + F \Leftrightarrow N > 2E + \lceil N/2 \rceil - 1 \Leftrightarrow 2E \leq N - \lceil N/2 \rceil \Leftrightarrow E \leq \lfloor N/4 \rfloor$$

For this case, the cardinality of any classic quorum ($|Q_c|$) and fast quorum ($|Q_f|$), expressed only as a function of N , is:

$$\begin{aligned} |Q_c| &\geq N - \lceil N/2 \rceil + 1 \geq \lfloor N/2 \rfloor + 1 \\ |Q_f| &\geq N - \lfloor N/4 \rfloor \geq \lceil 3N/4 \rceil \end{aligned}$$

5.3 Coordinator's Rule

The original coordinator's rule for Fast Paxos [57] is:

LET Q be any i -quorum of acceptors that have reported their last votes to the coordinator.

$vr(a)$ and $vv(a)$ be the round and the value voted by acceptor a .

k be the largest value of $vr(a)$ for all $a \in Q$.

V be the set of values $vv(a)$ for all $a \in Q$ with $vr(a) = k$.

$O4(v)$ be true iff there is a k -quorum R such that $vr(a) = k$ and $vv(a) = v$ for all $a \in (Q \cap R)$.

IF $k = 0$ THEN let v be any proposed value.

ELSE IF V contains a single element

THEN let v equal that element.

ELSE IF there is some $w \in V$ satisfying $O4(w)$

THEN let v equal that w (unique).

ELSE let v be any proposed value.

We now show how this rule can be interpreted in terms of the cardinality-based quorum definitions presented in the previous section. When $k = 0$ or V contains a single element the rule is trivial to evaluate no matter the quorum implementation used. However, the evaluation of $O4(w)$ is more complex because it requires the evaluation of all possible intersections $Q \cap R$ for all k -quorums R . Considering only the cardinality of the quorums involved we have that $O4(w)$ is true if at least $|Q \cap R|$ acceptors voted for w for some R . As we don't know, and don't want to know, all possible quorums R , we must consider the smallest possible $|Q \cap R|$, assuming as implied by $O4(w)$ that all acceptors *outside* of Q also voted for w in ballot k . Considering that V can only contain more than one element if k was a fast round, we have two situations: i is a classic quorum or i is a fast quorum. Let T be the number of votes for the value w in V . If we want T to be at least as large as the smallest $|Q \cap R|$ then we have:

$$T \geq \begin{cases} N - E - F & \text{if } i \text{ is classic} \\ N - 2E & \text{if } i \text{ is fast} \end{cases}$$

$O4(w)$ can now be evaluated by simply counting the number of votes for w in V . So, any value w that satisfies the condition above satisfies $O4(w)$ and is by definition unique. Considering that $E < (N - F)/2$ from Equation 5.1, when i is a classic round, we have:

$$T \geq N - E - F \Rightarrow T > N - (N - F)/2 - F \Leftrightarrow T > (N - F)/2 \Leftrightarrow T \geq \lfloor |Q_c|/2 \rfloor + 1$$

Similarly, for the case where i is a fast round we have:

$$\begin{aligned} T \geq N - 2E &\Rightarrow T > N - E - (N - F)/2 \Leftrightarrow 2T > N - E + (F - E) \\ &\Rightarrow T > (N - E)/2 \Leftrightarrow T \geq \lfloor |Q_f|/2 \rfloor + 1 \end{aligned}$$

In all cases T is at least as large as $\lfloor |Q|/2 \rfloor + 1$, so if any value w satisfies $O4(w)$, then it has been voted in round k by a majority of processes *inside* the quorum Q .

5.4 Simplified Coordinator's Rule

We have shown that a value w satisfies $O4(w)$ if this value has been voted in round k by a majority of acceptors in Q . We can use this observation to derive a simplified coordinator's rule for Fast Paxos. First, the fact that w has been voted by a majority in Q implies that it is the value most often voted in V . Thus, we can check this condition before testing if w satisfies $O4(w)$, obtaining an equivalent coordinator's rule.

IF $k = 0$ THEN let v be any proposed value.

```

ELSE IF  $V$  contains a single element
  THEN let  $v$  equal that element.
ELSE IF there is a single  $w \in V$  voted most often
  THEN IF  $w$  satisfies  $O4(w)$ 
    THEN let  $v$  equal that  $w$ .
    ELSE let  $v$  be any proposed value.
  ELSE let  $v$  be any proposed value.

```

If w does not satisfy $O4(w)$, we are free to choose any value as v . We use this freedom to always select the most often voted w . We have now removed some freedom from the coordinator, but all values w that satisfy $O4(w)$ are correctly selected. We can remove the $O4(w)$ test, obtaining the following rule:

```

IF  $k = 0$  THEN let  $v$  be any proposed value.
  ELSE IF  $V$  contains a single element
    THEN let  $v$  equal that element.
  ELSE IF there is a single  $w \in V$  voted most often
    THEN let  $v$  equal that  $w$ .
    ELSE let  $v$  be any proposed value.

```

If V contains a single element, then this element surely has been voted most often than any other element in V . Thus, we can remove the single element test, giving our final simplified rule:

```

IF  $k = 0$  THEN let  $v$  be any proposed value.
  ELSE IF there is a single  $w \in V$  voted most often
    THEN let  $v$  equal that  $w$ .
    ELSE let  $v$  be any proposed value.

```

5.5 Conclusion

We have showed how the coordinator's rule of Fast Paxos [57] can be simplified by resorting exclusively to counting the number of votes for each of the proposed values. Our rule is more restrictive than the original rule, as for some consensus instances it forbids the coordinator of freely choosing any value when he would be allowed otherwise by the original rule. However, the restriction imposed by the simplification does not lead to any disadvantage because if some value received votes in a previous round and the consensus value isn't decided yet, it is reasonable to consider that the coordinator will try to decide on that value first. Our simplified rule is easier to

implement, has the advantage that it is independent of the type of a round (fast or classic), and it has to consider only the cardinality of the quorum Q .

Capítulo 6

A Recovery Efficient Solution for the Replacement of Paxos Coordinators

In Paxos, failures can cause the replacement of its coordinator. The replacement of the coordinator, in its turn, leads to a temporary unavailability of the application implemented atop Paxos. Solutions to the unavailability problem have been sought because of the widely recognized utility of Paxos as a building block of fault-tolerant distributed applications. So far, the problem has been addressed by reducing the coordinator replacement rate through the use of better failure detection derived from stable leader election algorithms. We have observed that the recovery process of the newly elected coordinator's state is at the core of the unavailability problem. Thus, in this paper we present a new solution to the problem that allows the recovery to occur concurrently with new consensus rounds. We show that our solution has a very small impact on the communication and message complexity of Paxos. Experimental results show that our solution effectively solves the temporary unavailability problem. The main benefit of our solution for the application is its uninterrupted execution with better performance.

6.1 Introduction

Paxos [56] is a consensus algorithm for asynchronous distributed systems; it relies on a key agent, the coordinator, to ensure its safety. The algorithm also guarantees liveness as long as there is one, and only one, coordinator. When Paxos is used to decide multiple instances of consensus, as in the case of the delivery of totally ordered messages, the requirement of a single coordinator can hinder its progress. The reasons for this are the higher workload processed by the coordinator [24] and the inherent cost of replacing the failed coordinator [26]. The coordinator faces higher

CPU and I/O loads than the other Paxos agents because of its main task. It acts as a sequencer and processes all application messages that need to be ordered. It does so by initiating many consensus instances and keeping track of their outcome.

Even when the coordinator can handle the higher workload without compromising the overall performance of the fault-tolerant application, it is still subject to failures that eventually will cause its replacement. Coordinator *replacement* is carried out in two steps: a new coordinator is elected, and then it is validated [56]. Coordinator *election* is handled by any unreliable leader election mechanism that is equivalent to an Ω failure detector [27]. The unreliability of this mechanism means that it can erroneously change coordinators many times, but it will select a single coordinator eventually. Coordinator *validation* requires the new coordinator to have its role ratified by a majority of Paxos agents. To achieve this, the new coordinator has to receive a potentially large prefix of the current state of each member of this majority. Validation ensures that the new coordinator is up to date with the state of all active consensus instances. A newly elected coordinator can resume its activities only after the completion of validation. Thus, the replacement of a coordinator triggers a costly operation that is inevitably going to happen many times in the presence of partial failures and incomplete or inaccurate failure detection. The *temporary unavailability problem* occurs because normal Paxos operation can only be resumed after a successful validation.

The periods of unavailability are a real concern for fault-tolerant systems based on Paxos. Burrows [19] provides a concrete example of the troubles caused by the replacement of Paxos coordinators in a production system. Finding suitable solutions for the temporary unavailability problem is an interesting research challenge with practical implications. The most common way to mitigate the unavailability problem is to devise a mechanism that makes it harder to replace the coordinator. Malkhi et al [63] have proposed a failure-detector based on an election procedure with built-in leader stability. Using this procedure a coordinator is only replaced if it isn't able to effectively perform its actions. Another approach is to grant an implicit lease to the current coordinator [26]. This ensures it won't be demoted needlessly, but increases the time it takes to detect an actual failure. However, these approaches only mitigate the problem of coordinator replacements caused by inaccurate failure detection. They cannot really help in the event of a real coordinator failure and the ensuing coordinator replacement.

In this paper we show an alternative approach to solving the temporary unavailability problem that stems from breaking coordinator validation in two concurrent activities: activation and recovery. Coordinator *activation* corresponds to the actual ratification of a coordinator by a majority. We show that it is possible to reduce the information necessary to activate the new coordinator to a single integer. In fact, we

show that the coordinator doesn't need to rebuild the complete state of a majority of processes before it can resume its work, it just needs to discover the highest consensus instance that a majority of processes agrees is free to use. This can be done using only a single exchange of fixed size messages, allowing the new coordinator to resume operation in a very short time. Coordinator *recovery* then becomes a secondary task that can take much longer to finish, but that does not block the application during the validation. The result is a much briefer coordinator validation whose time is limited primarily by the activation time. The coordinator's state recovery, the longer step, occurs while the coordinator is already managing new consensus instances. From the point of view of the application our new procedure guarantees coordinator replacements with less disruptive performance oscillations, namely, *seamless coordinator validations*.

Experimental results show that our concurrent validation procedure guarantees progress with increased throughput in the presence of coordinator replacements caused by process failures. While these coordinator replacements happened, we have observed the uninterrupted operation of the application, a clear indication that our validation procedure solves the temporary unavailability problem. Additionally, we have observed that failure detector mistakes can trigger the replacement of a coordinator, even if there are no process failures. The results of this set of experiments show that the new coordinator validation procedure increases the throughput of the application even when there are no process failures. In short, the results show that our coordinator validation mechanism makes coordinator replacement seamless to the application and increases its performance.

The paper is structured as follows. In Section 6.2 we give an overview of the Paxos algorithm and introduce the terms used throughout the paper. Section 6.3 discusses the original coordinator validation procedure of Paxos. Section 6.4 describes our seamless coordinator validation procedure and prove its correctness. Section 6.5 discusses the results of the experiments carried out to compare the original with the seamless validation procedure. Section 6.6 analyzes the applicability of our results to the Fast Paxos algorithm. Section 6.7 describes related work. Section 6.8 provides concluding remarks.

6.2 Paxos

Informally, the *consensus* problem consists in each process of a distributed system proposing an initial value and all processes eventually reaching a unanimous decision on one of the proposed values. The Paxos algorithm is both a solution to the consensus problem and a mechanism for the delivery of totally ordered messages that can be

used to support active replication [79]. In this section we give a summarized description of Paxos and make explicit the key role performed by the coordinator. Full descriptions of the algorithm can be found in [56, 57].

6.2.1 Core Algorithm

Paxos is specified in terms of roles and agents; an agent performs a role. Different implementations of Paxos may choose different mappings between agents and the actual processes that execute them. Agents communicate exclusively via message exchanges. The usual asynchronous crash-recovery computation model is assumed. The roles agents can play are: a *proposer* that can propose values, an *acceptor* that chooses a single value, or a *learner* that learn what value has been chosen. To solve consensus, Paxos agents execute multiple *rounds*, each round has a *coordinator* and is uniquely identified by a positive integer. Proposers send their proposed value to the coordinator that tries to reach consensus on it in a round. The coordinator is responsible for that round, and is able to decide, by applying a local rule, if any other rounds were successful or not. The local rule of the coordinator is based on quorums of acceptors and requires that at least $\lfloor N/2 \rfloor + 1$ acceptors take part in a round, where N is the total number of acceptors in the system [57]. Each round progresses through two phases with two steps each:

- In Phase 1a the coordinator sends a message requesting every acceptor to participate in round r . An acceptor accepts the invitation if it has not already accepted to participate in round $s \geq r$, otherwise it declines the invitation by simply ignoring it.
- In Phase 1b, every acceptor that has accepted the invitation answers to the coordinator with a reply that contains the round number and the value of the last vote it has cast for a proposed value, or *null* if it has never voted.
- In Phase 2a, if the coordinator of round r has received answers from a quorum of acceptors, it analyzes the set of values received and picks a single value v . It then asks the acceptors to cast a vote for v in round r , if v is not *null*, otherwise the coordinator is free to pick any value and picks the value proposed by the proposer.
- In Phase 2b, after receiving a request from the coordinator to cast a vote, acceptors can either cast a vote for v in round r , if they have not voted in any round $s \geq r$, otherwise, they ignore the vote request. Votes are cast by sending them and their respective round identifiers to the learners.

- Finally, a learner learns that a value v has been chosen if, for some round r , it receives Phase 2b messages from a quorum of acceptors announcing that they have all voted for v in round r .

This description of the algorithm considers only a single instance of consensus. However, Paxos also defines a way to deliver a set of totally ordered messages. The order of delivery of these messages is determined by a sequence of positive integers, such that each integer maps to a consensus instance. Each instance i eventually decides a value v and this value is the message (or ordered set of messages) to be delivered as the i th message of the sequence. The value v is input by the proposers, and they can either select a suitable i from their local view of the instance sequence or ask the coordinator to select i from its view. Each consensus instance is independent from the others and many instances can be in progress at the same time. In fact, for any agent its local view of the set of all instances can be divided in three proper subsets: the decided instances, the undecided instances that were initiated (Phase 1a) and the infinite set of uninitiated instances. Figure 6.1 shows an example of the status of the consensus instances as seen by an agent. In this example the set of decided instances is $\{1,2,4\}$, the set of undecided instances is $\{3,5,7\}$ and the set of uninitiated instances is $\mathbb{N} \setminus \{1,2,3,4,5,7\}$.

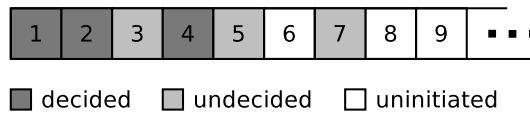


Figura 6.1: Local View of an Agent

6.2.2 Stable Memory Requirements

Paxos assumes a process failure model where agents crash and later recover. When a process crashes, it loses all state it has stored in its local volatile memory. Unfortunately, key information must be restored exactly as it was before the crash to guarantee the correctness of the algorithm. Thus, parts of the local state are recorded into stable memory that can be recovered after a crash. Access to stable storage is usually slow, so its use must be minimized. The coordinator must store the value of the last round it has started, say $crnd_c$, to ensure it won't start the same round twice [57]. Similarly, each acceptor must store in stable memory:

- rnd_a : the last round they have taken part (Phase 1a);
- $vrnd_a$: the last round where they have cast a vote;

- $vval_a$: the value of the vote cast in $vrnd_a$ (Phase 2a).

The stable memory requirements for the set of consensus instances in Paxos are the same for a single instance, but multiplied by the number of instances. Thus, each agent must store an array of instances, where for each instance i it records $rnd_a[i]$, $vrnd_a[i]$, $vval_a[i]$ and $crnd_c[i]$. Additionally, the learner agent may store $dval_l[i]$, the value decided in instance i , but this isn't strictly necessary as a new successful round will yield the same value. Usually, all agents are implemented in each process and agents may use the information stored by other agents to implement optimizations. For instance, a coordinator can inform proposers that their selected instance number i is already decided, or similarly, acceptors can inform a coordinator that an instance i it is about to start is already decided.

6.2.3 Liveness

In Paxos, any process can act as the coordinator as long as it correctly chooses a value, if any, that has been proposed in Phase 2a. There can be only one active coordinator at any given time for the algorithm to ensure progress. If two or more processes start coordinator agents, the algorithm can stall while the multiple coordinator candidates cancel each other rounds with fast increasing round numbers. For this reason, liveness of the algorithm resides on a coordinator selection procedure. This procedure doesn't need to be perfect. Safety is never compromised if zero or more than one coordinator are active at any time. However, the coordinator selection needs to be robust enough to guarantee that only a single coordinator is active most of the time.

It is clear that the coordinator in Paxos has a very important role, as all successful consensus instances must be started (Phase 1) and lead to completion (Phase 2) by a coordinator. After receiving all Phase 1b messages the coordinator discovers that no value was previously voted for most of the consensus instances. This is expected as only rounds that happen after failed rounds carry a potentially decided value. It is possible to use this observation to reduce the latency to reach consensus through a validation procedure. During validation the coordinator tries to start a new round for all uninitiated consensus instances concurrently. If successful, the coordinator is then able to use this round to continue any instance directly from Phase 2. This way, it is possible to reduce from five to three message delays the time required to reach consensus [57].

6.3 Original Coordinator Validation

We now describe in more detail how validation is performed in the original Paxos specification [56]. During validation a coordinator selects a round number r and starts *all* consensus instances at the same time with a single message, as the Phase 1a message carries only the round number. If r is large enough, acceptors will respond to this message with a finite number of Phase 1b messages with the actual votes and an infinite number of Phase 1b messages with no votes. Lamport [56] notes that only the finite set of messages containing an actual vote need to be sent back to the coordinator, framed in a single physical message. No message has to be sent to the coordinator for each of the infinite instances that have had no vote yet. The coordinator processes all Phase 1b messages received and it *assumes* that the infinitely many omitted messages correspond to Phase 1b messages with no vote. All messages received or presumed voteless are processed as usual and a suitable value will be selected to be voted for each instance, or the instance will be marked free (no previous value) and will be used when necessary. This way a coordinator can start the Phase 2 of any free instance as soon as it receives a proposal, and consensus for this instance can be reached in three message delays [57].

This validation procedure requires the coordinator to learn the status of all decided and undecided consensus instances of a quorum of acceptors to determine the exact identities of the infinite uninitiated consensus instances. So, the combined state of a quorum of acceptors represents the state footprint a new coordinator must recover to be able to start passing new consensus instances. To reduce the footprint of the recovery state, it is possible to determine a point d_c in the instance sequence as seen by the coordinator such that all instances i , with $i \leq d_c$, belong to the decided set. The point d_c doesn't necessarily determine all instances in the decided set, but this isn't necessary. The coordinator can then indicate the prefix d_c of the instances it already knows are decided and the acceptors need only send information about larger instances [56]. This combined message is finite, but even with the footprint reduction it can be very large and must be fully recovered so the coordinator can (1) discover all instances this acceptor has voted and (2) use this information to infer the set of instances the acceptor *has not* voted. Moreover, the coordinator must expect complete responses from a quorum of acceptors before it can complete Phase 1 for all instances. While this happens, the coordinator remains blocked and no progress is possible.

6.4 Seamless Coordinator Validation

Our proposal for a seamless coordinator validation is based on the observation that validation can be broken in two concurrent activities: activation and recovery. Activation is the procedure where acceptors inform the newly elected coordinator about the instances they have not voted. Recovery is the procedure where the coordinator's view of the consensus instances is updated, it learns the outcome of decided instances, and initiates rounds for the undecided ones. This compound view of the validation procedure is interesting because only activation is required to be finished before a coordinator can resume its activities. Recovery, while necessary, does not pose any restriction on the coordinator's use of uninitiated consensus instances. This happens because a coordinator doesn't need to immediately start the consensus instances that belong to the undecided set. For these instances, the coordinator doesn't know whether it can instantly input a value or not, as a consequence, it can learn their status later, during recovery. In order to use this fact to create a more efficient validation we have to devise an activation procedure that avoids the transfer of the finite, but possibly very large, set of decided and undecided consensus instances that make up the recovery state. Before we describe how the coordinator can achieve this economy in the state transferred from the acceptors, it is important to understand why the coordinator doesn't need to have knowledge of the status of the consensus instances currently in progress to function.

If we look at the sequence of round numbers effectively used in a consensus instance, it is possible to notice that these numbers are distinct and increasing but that they need not to be sequential. In fact, if they are partitioned among the processes in a way that gives each process equal chance of having a larger number; so they are never sequential. Thus, a coordinator picks a round number, say i , to be any round number larger than $crnd_c[i]$, but not necessarily $crnd_c[i] + 1$. From this simple observation it is easy to see that if the coordinator only records the largest round number initiated for *all* instances it is guaranteed to be able to always choose a larger round number for any individual instance when necessary. In this case, the stable memory footprint of the coordinator can be reduced to the memory necessary to store a single integer $crnd_c$, no matter how many instances of consensus were ever initiated by it. Clearly, the coordinator still must keep track of the progress of the rounds it initiates, including the round numbers of the rounds in progress, but this information may be stored in volatile memory.

This simple modification makes clear the fact that the coordinator doesn't concern itself with the decided or proposed values of consensus, but only with the proper initiation and progress of rounds. It still computes the Phase 2a rule, but can do

so only in volatile memory. Furthermore, the coordinator can still keep this very compact view of the sequence of consensus instances even when it shares a process with other agents. For example, to avoid starting a round for an already decided instance of consensus, the coordinator can query the stable memory of its co-located learner. In this setup, the coordinator depends on the functionality of other agents to implement optional functionality, but still requires just a single integer in stable storage to guarantee safety.

In general, the maintenance of only a very small state in stable memory will require the coordinator to query the acceptors on all extra information it needs to complete a consensus round. More importantly, it depends on the information held by the acceptors to execute the activation procedure and be able to start rounds immediately. Then, it is crucial to understand the view of the consensus instances held by each acceptor. Recall that in Paxos, each acceptor keeps a persistent history of its execution with the following variables for each consensus instance i : $rnd_a[i]$, $vrnd_a[i]$ and $vval_a[i]$. Each instance i is initially inactive and belongs to the uninitiated set, their corresponding variables have been initialized as *null*. As Paxos progresses, values computed by the agents are stored in the fields of the consensus instances and they pass to the set of active but undecided instances. Eventually, a consensus is reached for an instance and it is promoted to the decided set. As instance identifiers are picked by the proposers from the set of positive integers in strict incremental order, it is possible to establish a point f_a in the instance sequence, as viewed by acceptor a , such that every instance i , $i \geq f_a$, has not received a vote yet. It is also possible to find the identifiers of instances smaller than f_a that have also not received a vote yet, but we know for sure that all instances larger than or equal to f_a have never received a vote. For example, in the local state depicted in Figure 6.1 we have $f_a = 8$.

6.4.1 Activation Procedure

The seamless coordinator validation is based on an activation procedure that determines a point f_Q of the Paxos consensus history that is consistent with points f_a of the local histories of each acceptor a of a quorum Q . The detailed steps executed by the activation procedure are as follows:

1. The coordinator sends an Activation Phase 1a message, with round number r starting *all instances*.
2. When an acceptor a receives this message it computes its f_a . If r is larger than the last ballot used in another *activation* or there was no previous activation, then

a sends a single Activation Phase 1b message containing its f_a , meaning that it is sending Phase 1b messages for all instances $i \geq f_a$ and *only for these instances*.

- As soon as the coordinator has received Activation Phase 1b messages from a quorum Q of acceptors, it computes f_Q to be the largest of the f_a received, for each $a \in Q$. It then considers that it has received a Phase 1b message with no votes from all acceptors in Q for instances $i \geq f_Q$, and from this point on it proceeds as the original Paxos.

Figure 6.2 shows an example of the activation process for four acceptors a_1 , a_2 , a_3 and a_4 . Assuming all of them are able to take part in the activation, they compute f_a respectively as $f_{a_1} = 5$, $f_{a_2} = 7$, $f_{a_3} = 8$ and $f_{a_4} = 7$. The coordinator computes $f_Q = 8$ and ends its activation.

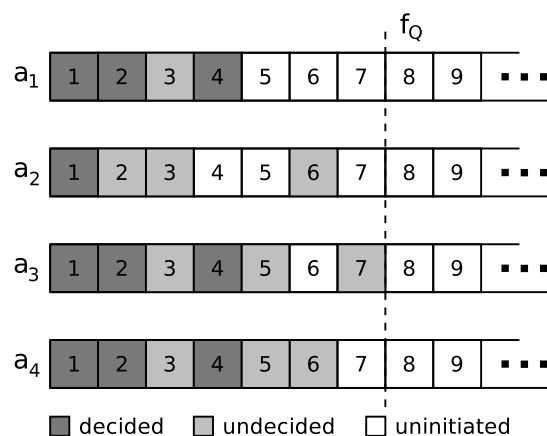


Figure 6.2: Global View as Observed by a Coordinator

The seamless coordinator activation presented here requires considerably less information to be propagated from the acceptors to the coordinator. It takes one broadcast from the coordinator containing the round number and Q unicasts from the acceptors to the coordinator containing a single integer f_a . This contrasts with the original coordinator validation [56] where the activation and recovery are handled sequentially. In the original validation the coordinator broadcast is answered by Q unicasts containing all previous votes for consensus instances $d_c < i < f_a$, as described in Section 6.3. Each vote contains, besides the round number, the contents of the actual application messages (or ordered set of messages) voted in one specific consensus instance. It is not difficult to see that the handling of the transmission, reception and processing of these messages can have a considerable cost for Paxos. Most important, while the sequential validation is underway Paxos stops delivering application messages, generating the unavailability problem.

6.4.2 Correctness

The correctness of the seamless coordinator activation is derived from the correctness of individual Paxos consensus instances. Although in its first step the coordinator initiates many instances at once, each one of them complies strictly with Paxos protocol and with the proofs contained in [56]. So, in this section, we show that the activation procedure we have devised does not perform any operation forbidden by the original Paxos.

The analysis for the first step (Section 6.4.1) is straightforward. Any process that considers itself the coordinator can start a round for any consensus instance, as long as the rules of Paxos for the selection of a new round number are respected. The sending of an Activation message containing the command to start all rounds, if setup with a suitable round number, doesn't violate any of the Paxos invariants.

The correctness of the second step depends on the following facts regarding the behavior of an acceptor: (i) it can always determine the f_a point, (ii) it respects the original Paxos rules when answering Phase 1a messages, and (iii) it doesn't violate the algorithm liveness. The uniqueness of f_a follows from the observation that we can always find an instance larger than the last instance voted because the number of voted instances is finite and there is an infinite number of instances. We can consider the larger instance found as the frontier to the remaining infinite number of instances identified by $i \geq f_a$ that can be treated as a single instance I , with respect to the stable memory storage requirements. This is possible because the only way an instance i can leave the set determined by I is by leaving the uninitiated set, but this leads by construction to $i < f_a$. This means that, as successive coordinator activations lead to evaluations of f_a , initiated instances stop being represented by I and are treated as regular instances. Thus, as we run Paxos for this especial instance I , as part of the activation, we are executing Paxos for all instances $i \geq f_a$. Acceptor a is able to decide whether to answer or not the Activation Phase 1a message of the coordinator by comparing r with the value of $rnd_a[I]$ and it does so by using a single message, and recording the new value of $rnd_a[I]$. What remains to be done now is to ensure that the activation process does not violate liveness by proceeding only with instances $i \geq f_a$. From the Paxos algorithm, an acceptor can refrain from answering a message (for example, if it refers to a smaller round), so once a determines f_a it is free to ignore the requests for instances $i < f_a$. Clearly, not answering to a message indefinitely could cause a liveness violation, but as f_a is always defined, the Activation Phase 1a message is eventually answered. Also, the instances $i < f_a$ are subsequently treated as instances and a sends timely answers for Phase 1a messages not related to activation.

In the third step, we must show that the determination of f_Q allows for the correct evaluation of the coordinator's rule. In any Paxos round, the coordinator is only free

to set an arbitrary value to an instance if it receives only *null* votes from all acceptors in a quorum. For any given acceptor a , the coordinator considers that it has received a *null* vote for all instances $i \geq f_a$. The coordinator receives answers from a quorum Q and establishes the point f_Q to be the largest f_a , for all acceptors $a \in Q$. It is easy to see that only for instances at least as large as f_Q a full quorum of *null* votes is received. All instances smaller than f_Q will miss at least one vote to complete a quorum. The coordinator then can treat all instances $i \geq f_Q$ as started and free to use. This leaves many instances $i < f_Q$, that are not yet decided, from the acceptors where $f_a < f_Q$. These instances will be treated normally later, as they are not required for the coordinator operation.

6.5 Experimental Evaluation

The seamless coordinator validation allows activation and recovery to occur concurrently. It is reasonable to suppose that the added concurrency will reduce considerably the time taken to setup a new coordinator, allowing Paxos to fulfill its function as a support for highly-available applications without interruption. We have designed two sets of experiments to assess our hypothesis. One set compares the performance of the two versions of coordination validation, original and seamless, in the presence of induced coordinator and network failures. The other set investigates the performance of the same coordinator validation versions during executions where processes do not fail, but adverse conditions associated with the environment where Paxos executes make the failure detector misbehave triggering coordinator replacements. The results of both sets of experiments show that the seamless coordinator validation guarantees that Paxos does not stop delivering ordered messages during the replacement of a coordinator, providing a significant performance increase for the application. In the next section we provide an outline of the experiments, with an emphasis on the components used and parameters that are shared by both sets of experiments. The description of experimental conditions and setup that are specific to each experiment set are described in the following two sections.

6.5.1 Method

Our tests were made using Treplica, a modular total-order broadcast toolkit that implements Paxos and Fast Paxos [87]. Treplica has been designed to be easily instrumented to generate the measurable indicators necessary to assess the performance of Paxos. The toolkit provides a programming interface that allows the construction of applications that adhere to the state machine replication approach.

Our experimental method consists in comparing the relative performance of two coordinator validation procedures. So, to minimize any possible effect of the application execution upon the performance measurements we have devised an application that performs very simple operations: a hash table. The object that is replicated using Treplica is a wrapper around the original Java hash table implementation that turns it into a replicated and persistent object. The workload is exclusively composed of a sequence of hash table put operations, where each operation associates an integer, sequentially incremented, with a random five character string. Read operations were ruled out because they do not represent a significant cost for Paxos. This way, we have a workload that is homogeneous in terms of system resource use and that is always guaranteed to make Treplica the only sub-system of the experiment responsible for the performance variations observed. Treplica is configured to use the local disk of the computing system where replica is executed as its persistent data store, so disk accesses do not trigger any network usage. This way we guarantee that the network is used only to carry the messages exchanged by the replicas as a consequence of Paxos activity.

The experiments were carried out in a cluster with 18 nodes interconnected through the same 1Gbps Ethernet switch. Each node has a single Intel Xeon 2.4GHz processor, 1GB of RAM, and a 40GB disk (7200 rpm). The software platform is composed of Fedora Linux 9 and OpenJDK Java 1.6.0 virtual machine (JVM).

6.5.2 Induced Failures

In the first set of experiments we measure the performance of the application while coordinator failures are induced through the controlled injection of faults. The process that runs the coordinator agent is killed, restarted, and elected coordinator again. During the time the coordinator stays down the remaining correct processes advance many consensus instances. Once restarted, the coordinator will have to restore part of its state from the local stable storage and part from remote acceptors.

Workload

Server replicas and workload generators share the same hosts, but care has been taken to ensure that the load generation wasn't competing with the application processing and that the specified workload was being generated. The generated load is measured in operations per second (op/s) and is generated at a fixed rate equally divided among all the load generators of hosts that do not fail. This way, the load is unaffected by failures.

For all experiments we run a system with 5 replicas under a continuous load of 1000 op/s for 10 minutes. The first 90 seconds and final 30 seconds are discarded as ramp up and ramp down time, for a total of 8 minutes of steady-state performance. During the ramp up time the caches of the JVM are being filled up and the just-in-time compiler is generating optimized code. During the ramp down time some operations can be left incomplete as the replicas are brought down. For each faultload (process and network), and for each type of validation procedure (original and seamless) we have performed 10 distinct runs and recorded the average performance in operations per second, continuously throughout the entire execution time.

Faultload

To enable the controlled occurrence of a coordinator crash followed by its recovery, we have changed the coordinator selection procedure implemented by Treplica to always choose as the coordinator the process p_f with the largest identifier. This is easily accomplished by replacing the original leader election algorithm of Treplica by a simple priority-based leader election that assigns the highest priority to p_f . With the new coordinator selection process in place, it is possible to determine which node and process is the host of the coordinator. This way, the fault injector can be setup to inject the desired fault into the right environment component (process or network interface) at the desired moment. All time labels used in the text and figures are relative to the beginning of the steady-state execution time.

Process faultload: The JVM that hosts process p_f is brought down at $t=30s$ and remains down for 30s, until $t=60s$. Once the JVM is restarted it is going to take around 90s for the Paxos agents to perform local recovery.

Network faultload: The network interface of the computer where p_f is executing is brought down at $t=30s$ and after 90s it is brought up again, at $t=120s$.

All faults are injected at the operating system level, using automated scripts that do not require any human intervention during the duration of the experiment.

Results

Figures 6.3 and 6.4 show the data for the process and network failure scenarios, respectively. For both faultloads we observe the same general behavior. As expected, performance is affected by the recovery of the replica brought down by the fault injector. Moreover, for the original validation procedure the throughput of the system is effectively zero during recovery while for the seamless coordinator validation it is

also greatly reduced but maintains itself, on average, at about 20% of the average performance displayed during the failure-free periods. The observation of the results for the process and network faultloads shows that in both cases the seamless coordinator validation has prevented the application from stopping completely.

We proceed by carrying out the analysis of the effects of the process faultload. At $t=30s$ the coordinator is brought down (Figures 6.3 (a) and (b), label d). At this moment, a new coordinator is elected, the one with the highest process identifier among the remaining Paxos agents, validations take place, and normal processing is resumed. The coordinator validation resulting from this replacement does not have a disruptive impact on the performance of the application because the replicas that remained operational had a very similar state. Thus, the recovery phase of the newly elected coordinator represents a small processing overhead. In fact, results shown in [20] allow us to say the performance of the replicated application is inversely proportional to the scale of the system. This explains the performance increase observed right after $t=30s$, when environment is scaled down to 4 replicas. The results confirm that the combined effects of increased performance and relative small recovery state have minimized the effects of the coordinator replacement. At $t=60s$ (Figure 6.3, label u) the JVM of the failed coordinator is restarted but it takes some time, until approximately $t=150s$, for its Paxos agents to be back into activity because of the time it takes to restart the environment and read its state from stable storage. An election happens and is followed either by an original coordinator validation (Figure 6.3 (a)) or by a seamless coordinator validation (Figure 6.3 (b)). The state of the re-activated coordinator is far behind from the state of the replicas that remained functioning correctly. In the case of the original coordinator validation the coordinator is only going to resume its normal activities after it has finished its recovery, the unavailability caused while recovery takes place can be observed in Figure 6.3 (a). By contrast, the concurrency introduced by the seamless coordinator validation guarantees the availability of the application during the replacement of coordinators (Figure 6.3 (b)). Table 6.1 shows the average performance of the original and seamless versions of coordinator validation during these experiments with their respective coefficient of variations (CV), that are within the 5% accuracy.

The injection of faults at the networking component causes no harm to the state stored in the local volatile storage of the agents. This implies that the returning coordinator has the advantage of not having to pay the cost of stable storage access during its recovery. The network faultload isolates the coordinator for a period of 90s, from the moment labeled d up to u in the Figures 6.4 (a) and (b). Different from what occurs with the process faultload, there is almost no delay between the moment the replica is brought back up and the moment it becomes the host of the new coordinator. The

average performance of the application during these runs is approximately 800 op/s (Table 6.1). Thus, on average, as soon as the isolated coordinator is reconnected it is going to receive recovery states from at least a quorum of acceptors, these states will have sizes proportional to the approximately 72000 ($800\text{op/s} \times 90\text{s}$) operations delivered to the application during the isolation period. The amount of work demanded from the coordinator to receive and process such a large amount of data explains the application's heavy performance drop (Figure 6.4). Once again the recovery of the coordinator's state causes the temporary unavailability of the application. In the case of the seamless coordinator validation the application almost stops, but only for a very brief time. Despite this, it is possible to observe the positive effect the seamless coordinator validation has on the performance of the application during recovery (Table 6.1).

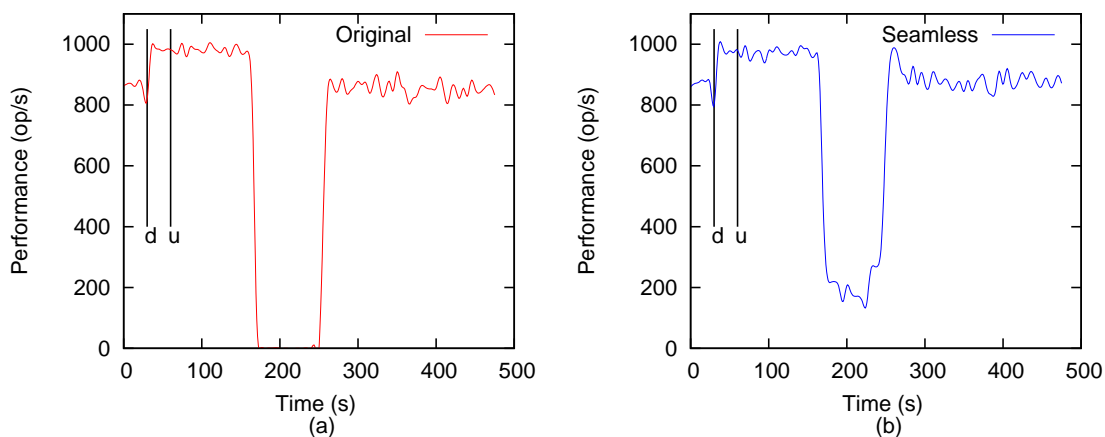


Figura 6.3: Process Faultload

These results allow us to conclude that the seamless coordinator validation introduced here definitely improves the availability of the application supported by Paxos in the presence of a coordinator failure and recover. It is worth observing that the duration of the recovery (width of the valleys) is practically identical for the original and seamless graphs. This is expected as the recovery time is proportional to the size of the state that has to be recovered. The average throughput of the 10 runs for each experiment are listed in Table 6.1, with the corresponding coefficients of variation (CV). The minimum number of runs required per experiment to guarantee an accuracy of 5% for the performance measurements with a confidence level of 99% is 4 [50].

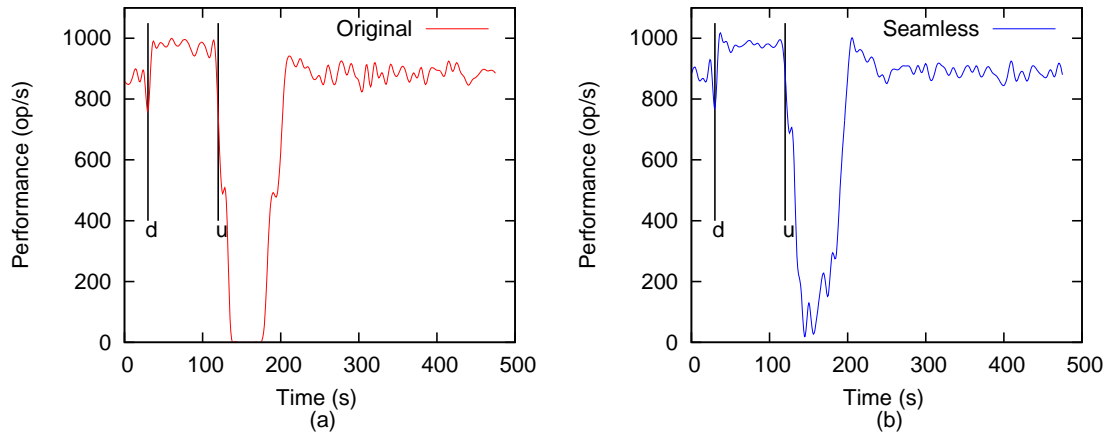


Figura 6.4: Network Faultload

Faultload	Original (op/s)	CV	Seamless (op/s)	CV
Process	732.66	0.0136	795.44	0.0376
Network	783.28	0.0186	814.15	0.0176

Tabela 6.1: Average Performance of the Application under Induced Failures

6.5.3 Intrinsic Failures

The first set of experiments showed that the seamless coordinator validation is better than the original coordinator validation in the presence of coordinator failures. In this set of experiments we would like to verify whether the seamless validation is worthwhile in relation to the original validation when intrinsically occurring transient failures are the adversary of Paxos. Intrinsic failures that occur at the host environment of the replicated application can misguide the failure detectors of each Paxos replica. For example, a failure detector can report a correct coordinator as having crashed due to a transient communication delay or due to a delayed execution of a thread. Irrespective of the underlying causes, failure detector mistakes trigger coordinator changes. As with the first set of experiments, we could have designed a faultload and a workload that would induce the failure detectors to fail. Instead, we have decided to pursue an indirect but more realistic approach: accelerate the rate of occurrence of intrinsic failures by overloading the application's host environment. The increased number of transient failures should increase the likelihood of failure detector mistakes, this, in their turn, should trigger validations and should allow the measurement of any performance differences between the two versions of Paxos, if they exist.

Workload

As for the first set of experiments, the measure used to discriminate between the two versions of coordinator validation is the performance of the replicas. The experiments rely on speedup and scaleup trials to subject the environment to increasingly higher workloads. The configurations used for the speedup and scaleup are as follows:

Speedup: For a fixed number of 9 replicas, the workload varies from 100 op/s to 3000 op/s in steps of 400 op/s.

Scaleup: For a fixed workload of 3000 op/s, the scale of the system goes from 3 to 15 replicas in steps of 2 replicas.

Both configurations were run for 10 minutes for each data point. For the same reasons stated for the first set of experiments, the first 90 seconds and final 30 seconds are discarded as ramp up and ramp down time, yielding a total of 8 minutes of steady-state runtime. For each data point in each workload (speedup and scaleup) and for each type of coordinator validation (original and seamless), we have measured the performance of the replicas in operations per second as the average of five distinct runs. Care has been taken to verify that none of the processes failed during the runs. Thus, any coordinator replacement will have to result from a failure detector mistake, and measurable differences between runs can be attributed to the relative efficiency of the validation strategies being compared.

Results

Figures 6.5(a) and 6.5(b) shows the data for the scaleup and speedup experiments, respectively. In the speedup trials, both the seamless and original coordinator validations have statistically identical performance for all but the 3000 op/s workload. This can be explained by the small state a coordinator will have to obtain during recovery. Contrary to the induced failures experiments, the runs of the intrinsic failures experiments are free of process crashes. Thus, it is reasonable to expect a fairly good synchronization to be maintained by the replicas for most of the workloads. By synchronization we mean that all replicas have a very similar view of the decided, undecided and uninitiated consensus instances. Both versions of coordinator validation show similar performance across all of the speedup runs because the main reason behind the validation unavailability problem is related to the time it takes to recover the state of a replica. As the recovery state is small, the advantage of the seamless validation over the original validation procedure should also be small.

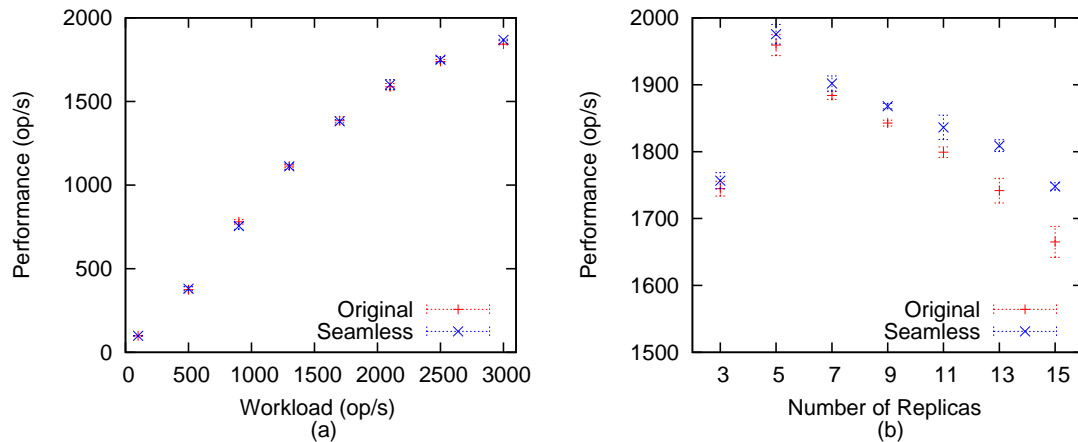


Figure 6.5: Speedup (9 replicas) and Scaleup (3000 op/s)

In the scaleup trials the seamless coordinator validation has performance similar to the performance of the original coordinator validation up to 7 replicas. At these scales, the same analysis used to explain the speedup results is valid. For scales ranging from 9 to 15 replicas the seamless coordinator validation outperforms the original coordinator validation. Interestingly enough the first scale where this happens is the same as the one observed for the speedup trials (9 replicas, 3000 op/s). The behavior of both versions of coordinator validation can be explained by two factors. The first is that the mistakes induced by the failure detectors are now going to affect increasingly larger subsets of the replicas. So, larger sets of acceptors are going to initiate validations concurrently. In the worst case this process can take more replicas out of their relative synchronization, making their states diverge. The second factor is related to Paxos itself. As the scale increases so does the minimum number of replicas required by Paxos to activate the new coordinator. This means that a larger number of recovery states has to be transferred and processed by the new coordinator, at least from a quorum of acceptors. For example, for 13 replicas the coordinator will have to process recovery states from at least 7 replicas, but it will very likely receive 13 responses. As the scale increases, in the absence of process crashes, the combined effect of the two factors cause the growth of the recovery states. In these scenarios, as expected, the advantage represented by the parallelism introduced by the seamless coordinator validation shows its effectiveness and outpaces the original coordinator validation.

6.6 Fast Paxos

Fast Paxos is a variant of Paxos that reduces the overall latency of the consensus rounds, measured in communication delays, by allowing the proposers to send proposed values directly to the acceptors. To achieve this, rounds are separated in *fast* rounds and *classic* rounds. Fast and classic rounds have different quorums associated with them, with properties such that the coordinator is still able to detect if a previous round was successful, even if the round was directly conducted by the proposers. Fast Paxos quorums are larger than the ones used by Paxos. For example, a possible configuration has both fast and classic quorums containing $\lfloor 2N/3 \rfloor + 1$ acceptors, from a set of N acceptors [57]. A Fast Paxos round is very similar to a Paxos round, except that Phase 2 is changed to:

- In Phase 2a, if the coordinator of round r has received answers from a quorum of acceptors, it analyzes the set of values received and picks a single value v . It then asks the acceptors to cast a vote for v in round r , if v is not *null*. Otherwise, if r is a fast round the coordinator sends a *any* message to the proposers indicating that any value can be chosen in round r . In this case, the proposers can ask the acceptors directly to cast a vote for a value of their choice in round r .
- In Phase 2b, after receiving a request to cast a vote from the coordinator (if the round is classic) or from one of the proposers (if the round is fast), acceptors can either cast a vote for v in round r , if they have not voted in any round $s \geq r$, otherwise, they ignore the vote request.

Coordinator validation is central to the performance of Fast Paxos. When a new coordinator runs validation, it completes Phase 1 and Phase 2a of the algorithm for all undecided consensus instances. It then sends a collective *any* message authorizing the proposers to initiate any of these instances. Proposer initiated instances can reach consensus in only two communication latencies without the need of further coordinator intervention [57]. Unfortunately, Fast Paxos cannot always be fast. Proposers can propose two different values concurrently, in this case, their proposals may collide. Also, process and communication failures may block a round from succeeding. Different recovery mechanisms can be implemented to deal with collisions and failures, but eventually the coordinator intervention may be necessary to start a new classic round [57].

The way Fast Paxos bypasses the coordinator to reduce communication latency removes coordinator validation from the critical processing path required to decide a consensus round. So, one might assume that the reduced role of the coordinator means that there is no need to optimize the coordinator's operation. However, even

in this restricted role, the coordinator still oversees all activity of the algorithm and ensures that instances are decided timely in the presence of collisions or message loss. Moreover, Fast Paxos can only be fast if a suitable coordinator has successfully performed validation, instructing the proposers on how to proceed. Thus, coordinator validation must be quick in the presence of process or network failures, so the system can resume operations in its coordinator-free state.

Seamless coordinator validation is easily adapted to Fast Paxos. During activation the coordinator decides if it will start all unused instances with a classic or fast consensus round. If it decides for a classic round, the procedure is exactly the same as described in Section 6.4. If it decides to start fast rounds, it must wait until it receives Activation Phase 1b messages from a fast quorum Q_F of processes. It then computes f_{Q_F} as described in Section 6.4 and sends a *any* message informing the proposers that instances $i \geq f_{Q_F}$ are prepared and free to use. In essence, the activation step of the seamless coordinator validation doesn't deal with the specific steps required to complete a consensus round. It only divides the consensus instances in sets in a way that it is possible to act on the infinite set of unused instances with a simple message exchange representing Phase 1 of the complete Paxos or Fast Paxos algorithm. This is clearly visible in the simple way the mechanism can be adapted to Fast Paxos.

6.7 Related Work

The importance of the coordinator replacement procedure was observed by Chandra et al. during the design and operation of the Chubby distributed lock system [26]. The designers of this system decided to make it harder for a replica to lose its coordinator status at the cost of slower detection of process failures. This is justified by the low incidence of observed process failures. In general, coordinator stability is considered the best way to deal with the cost of coordinator replacement. For example, the leader election algorithm proposed by Malkhi et al. captures precisely the network connectivity requirements of a working coordinator while guaranteeing stability during failure-free operation [63]. Ensuring stability makes sure a working coordinator will operate for the larger time possible, distributing in time the cost of replacement. However, even the cleverly designed algorithm of Malkhi et al. cannot ensure stability if its weak network connectivity requirements aren't met, even if it only happens for a brief time. In this case, a faster validation procedure is desired.

The fact that Paxos requires a single coordinator is at the root of the unavailability problem. This single process will eventually fail, or be mistakenly taken for failed, requiring a new coordinator to take its place. Another approach was taken by Camargos et al. and consists in not relying in a single one but on a group of coordi-

nators [24]. Their justification is that multiple coordinators make the algorithm more resilient to coordinator failures without requiring the use of Fast Paxos and its larger quorums. The resulting algorithm is considerably complex and increases the number of messages exchanged between the acceptors and the group of coordinators. Our simpler seamless coordinator validation procedure has similar coordinator resilience if we consider the whole set of replicas that can act as a coordinator as a coordinator group where only a master is active at any time and master changes are very cheap.

6.8 Conclusion

In this paper we have shown a novel way to avoid the temporary unavailability problem caused by Paxos coordinator replacements. Our solution is based on the observation that the validation of a new coordinator is composed of two activities: activation and recovery. We have shown that only the completion of the activation is strictly required before the coordinator can resume its operation. This fact has led us to a seamless coordinator validation has two important characteristics. First, it allows activation and recovery to be performed concurrently. Second, it reduces the information required to activate the new coordinator to a single integer exchanged between the acceptors.

We have verified experimentally that the seamless coordinator validation avoids the temporary unavailability problem in the presence of process crashes, providing uninterrupted operation for the application built atop Paxos. We have also observed the seamless coordinator validation performs better than the original validation in scenarios where only intrinsic transient failures make the failure detectors trigger validations. This second set of results showed that the seamless coordinator validation is particularly interesting in environments that change their number of replicas dynamically.

Finally, the seamless coordinator validation have other implications for the research on failure detectors for Paxos. Our enhanced validation procedure removes the restriction that the occurrence of validations must be avoided. In this case, instead of using more complex stable leader elections, it is possible to use very simple leader election mechanisms to choose the new coordinator. A fairly imprecise leader election procedure, but one that responds fast to failures or is simpler to implement, can be used without hindering the performance of Paxos. Actually, one can even consider election procedures that decide which process becomes the new leader not only based on the detection of failures but also on other factors, such as the load experienced by the replicas at the moment of the election. In summary, the seamless validation procedure is not only effective, it encourages research on the combined use of failure

detectors and load balancers to create more adaptive versions of Paxos.

Acknowledgments

The authors thank Prof. W. Zwaenepoel, EPFL, for his support, and Olivier Cramieri, also from EPFL, for his readiness to help with the management of the cluster.

Capítulo 7

Conclusão

7.1 Contribuições

Nesta tese exploramos o problema de como simplificar a construção de aplicações replicadas que sejam capazes de prover alto grau de disponibilidade e desempenho. Neste trabalho desenvolvemos a biblioteca Treplica, implementando uma interface de programação simples baseada em uma especificação orientada a objetos de replicação ativa. Os resultados obtidos com esta abordagem foram muito promissores e acreditamos ter criado um suporte modular e de uso simples para replicação que pode ser usado como primitiva básica para a construção de sistemas distribuídos confiáveis.

Esta tese apresenta as seguintes contribuições:

Especificação orientada a objetos para replicação: Nós propusemos a idéia de apresentar ao programador de aplicação uma abstração orientada a objetos para replicação como forma de simplificar a construção de aplicações confiáveis.

Treplica: Nós descrevemos Treplica, uma implementação da especificação abstrata de replicação.

Consenso como base para replicação: Nós propusemos o uso de consenso como a fundação para a implementação da especificação abstrata de replicação.

Estudo de confiabilidade: Nós realizamos uma análise de desempenho e confiabilidade de uma aplicação completa construída com o Treplica. Nós observamos que a aplicação resultante possui um bom desempenho, mesmo na presença de falhas e recuperações.

Estudo de desempenho: Nós caracterizamos o desempenho dos algoritmos Paxos e Fast Paxos sob várias situações comuns em redes locais de alta velocidade.

Regra do coordenador de Fast Paxos: Nós desenvolvemos uma regra simplificada de consistência a ser usada pelo processo coordenador no algoritmo Fast Paxos. Esta regra simplifica consideravelmente implementações do algoritmo.

Troca de coordenador em Paxos: Nós desenvolvemos um procedimento otimizado para substituição de um coordenador no algoritmo Paxos. Esta regra reduz consideravelmente o custo associado à troca de coordenador, potencialmente alterando o consenso sobre a importância de um único coordenador para este algoritmo.

7.2 Trabalhos Futuros

O trabalho iniciado neste tese levantou algumas questões interessantes a serem abordadas em trabalhos futuros.

Paxos exige apenas uma rede com troca não confiável de mensagens e pode embutir, como parte das suas garantias de *liveness*, mecanismos para retransmissão de mensagens. Aplicado-se o princípio fim a fim, não é aconselhável replicar estas funcionalidades em camadas inferiores da pilha de protocolos de comunicação. Contudo, muitas das implementações de Paxos ignoram esta observação e usam serviços de entrega confiáveis [65], o que gera um impacto de desempenho. Uma razão disto é que a configuração dos parâmetros de entrega de mensagens embutidas no Paxos é algo difícil, como observamos durante a implementação de Treplica.

Uma solução para este problema é a parametrização de Paxos como um protocolo de rede, que garante entrega confiável de mensagens, ordenadas, para um grupo de processos. Esta parametrização deve incluir configurações como *timeouts*, tamanho de *buffers*, protocolos de controle de fluxo e congestionamento, etc. Muitas destas configurações podem ainda apresentar comportamento adaptativo, refletindo condições variáveis da rede subjacente. Este trabalho deve vir acompanhado de modelos de desempenho teóricos e experimentos para comprovar a sua validade.

Um outro problema bem interessante é uma possível mudança da estratégia de construção de ferramentas de comunicação em grupo. A principal abstração de tolerância a falhas da esmagadora maioria destes sistemas é um mecanismo de pertinência a grupo [12]. Na nossa opinião, o principal problema desta abordagem é ligar de forma inseparável a pertinência ao grupo ao mecanismo de detecção de falhas. É razoável se supor que um processo que falhe possa permanecer no grupo até a sua recuperação. Ou então um processo correto pode deixar o grupo mas continuar operando após a sua saída.

O uso de consenso provê uma forma elegante de se resolver este problema, já

que os algoritmos que resolvem este problema exigem um conhecimento muito menos preciso sobre o conjunto de processos corretos no sistema para funcionar [28]. Adicionalmente, um sistema de pertinência ao grupo pode ser construído sobre o mecanismo de consenso, de forma completamente independente em relação ao mecanismo de detecção de falhas. Este mecanismo pode ainda orientar o processo de coleta de lixo do estado dos processos, tornando a recuperação mais eficiente. Pesquisa neste problema deve incluir o estudo de algoritmos e mecanismos que permitem a implementação da semântica tradicional de comunicação de grupo neste ambiente e a efetiva implementação de um protótipo.

Bibliografia

- [1] T. Abdellatif, E. Cecchet, and R. Lachaize. Evaluation of a group communication middleware for clustered J2EE application servers. In *DOA 2004: Proceedings of the 2004 International Symposium on Distributed Objects and Applications*, pages 1571–1589, Agia Napa, Cyprus, Oct. 2004.
- [2] D. Agrawal and A. E. Abbadi. The generalized tree quorum protocol: an efficient approach for managing replicated data. *ACM Trans. Database Syst.*, 17(4):689–717, 1992.
- [3] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*, pages 496–503, Passau, Germany, 1997. Springer-Verlag.
- [4] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distrib. Comput.*, 13(2):99–125, 2000.
- [5] Y. Amir, C. Danilov, and J. R. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society.
- [6] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication subsystem for high availability. In *FTCS-22: Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, MA, USA, July 1992.
- [7] Y. Amir and J. Kirsch. Paxos for system builders. In *LADIS '08: Proceedings of Large-Scale Distributed Systems and Middleware*, New York, Sept. 2008.
- [8] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, 2003.

- [9] B. Ban. Design and implementation of a reliable group communication toolkit for java. Technical report, Cornell University, 1998.
- [10] G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–18, London, UK, 1999. Springer-Verlag.
- [11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Boston, MA, USA, 1987.
- [12] K. P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, 1993.
- [13] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.
- [14] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.
- [15] A. D. Birrell, M. B. Jones, and E. P. Wobber. A simple and efficient implementation of a small database. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 149–154, New York, NY, USA, 1987. ACM Press.
- [16] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *SI-GACT News*, 34(1):47–67, 2003.
- [17] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Reconstructing Paxos. *SIGACT News*, 34(2):42–57, 2003.
- [18] F. V. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 42–50, London, UK, Sept. 2001. Springer-Verlag.
- [19] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.
- [20] L. E. Buzato, G. M. D. Vieira, and W. Zwaenepoel. Dynamic content web applications: Crash, failover, and recovery analysis. In *DSN 2009: 39th International*

- Conference on Dependable Systems and Networks*, pages 229–238, Estoril, Lisbon, Portugal, June 2009.
- [21] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *HPCA '01: Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 229–240, Monterrey, Mexico, 2001.
- [22] L. Camargos, F. Pedone, and M. Wieloch. Sprint: a middleware for high-performance transaction processing. In *EuroSys2007: Proceedings of the 2nd European Conference on Computer Systems*, pages 1–14, Mar. 2007.
- [23] L. Camargos, M. Wieloch, F. Pedone, and E. Madeira. A highly available log service for transaction termination. In *ISPD'08: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing*, pages 335–342, Washington, DC, USA, July 2008. IEEE Computer Society.
- [24] L. J. Camargos, R. M. Schmidt, and F. Pedone. Multicoordinated agreement protocols for higher availability. In *NCA '08: Proceedings of the 2008 Seventh IEEE International Symposium on Network Computing and Applications*, pages 76–84, Washington, DC, USA, 2008. IEEE Computer Society.
- [25] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX 2004 Annual Technical Conference, FREENIX Track*, pages 9–18, 2004.
- [26] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press.
- [27] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.
- [28] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
- [29] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

- [30] B. Charron-Bost, X. Défago, and A. Schiper. Broadcasting messages in fault-tolerant distributed systems: The benefit of handling input-triggered and output-triggered suspicions differently. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 244–249, Washington, DC, USA, 2002. IEEE Computer Society.
- [31] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, 2002.
- [32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. Dynamo: Amazon’s highly available key-value store. In *Proc. of 21st ACM SIGOPS Symp. on Operating Systems Principles*, pages 205–220, 2007.
- [33] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.
- [34] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.
- [35] J. Durães, M. Vieira, and H. Madeira. Dependability benchmarking of web-servers. In *Proc. of 23rd Computer Safety, Reliability, and Security Int. Conf.*, pages 297–310, 2004.
- [36] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys 2006: Proceedings of the 1st European Conference on Computer Systems*, pages 117–130, New York, NY, USA, 2006. ACM Press.
- [37] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *EuroSys 2007: Proceedings of the 2nd European Conference on Computer Systems*, 2007.
- [38] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- [39] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. In *SRDS '96: Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS '96)*, page 140, Washington, DC, USA, 1996. IEEE Computer Society.
- [40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Inc., 1995.

- [41] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.
- [42] D. K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.
- [43] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.
- [44] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM Press.
- [45] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Ada-Europe '96: Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies*, pages 38–57, London, UK, 1996. Springer-Verlag.
- [46] J. Holliday, D. Agrawal, and A. E. Abbadi. The performance of database replication with group multicast. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 158–165, Washington, DC, USA, June 1999. IEEE Computer Society.
- [47] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The XtreamFS architecture—a case for object-based file systems in grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2049–2060, 2008.
- [48] F. Hupfeld, B. Kolbeck, J. Stender, M. Höggqvist, T. Cortes, J. Marti, and J. Malo. FaTLease: scalable fault-tolerant lease negotiation with Paxos. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 1–10, New York, NY, USA, 2008. ACM.
- [49] M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, 2007.
- [50] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.
- [51] Y. Jiang, G. Xue, and J. You. Toward fault-tolerant atomic data access in mutable distributed hash tables. In *Proc. of First Int. Multi-Symp. on Computer and Computational Sciences*, 2006.

- [52] F. Junqueira, Y. Mao, and K. Marzullo. Classic Paxos vs. Fast Paxos: caveat emptor. In *HotDep'07: Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, page 18, Berkeley, CA, USA, 2007. USENIX Association.
- [53] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 134–143, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
- [54] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.
- [55] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [56] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [57] L. Lamport. Fast Paxos. *Distrib. Comput.*, 19(2):79–103, Oct. 2006.
- [58] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, June 2006.
- [59] B. W. Lamson. How to build a highly available system using consensus. In *WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 1–17, London, UK, 1996. Springer-Verlag.
- [60] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, page 52, Washington, DC, USA, 2000. IEEE Computer Society.
- [61] W. Liang and B. Kemme. Online recovery in cluster databases. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 121–132, New York, NY, USA, 2008. ACM.
- [62] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI '04: 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

- [63] D. Malkhi, F. Oprea, and L. Zhou. Ω meets Paxos: Leader election and stability without eventual timely links. In *DISC '05: Proceedings of the 19th International Conference on Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2005.
- [64] K. Manassiev and C. Amza. Scaling and continuous availability in database server clusters through multiversion replication. In *Int. Conf. on Dependable Systems and Networks*, 2007.
- [65] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.
- [66] S. Mena, A. Schiper, and P. Wojciechowski. A step towards a new generation of group communication systems. In *Middleware 2003*, volume 2672 of *Lecture Notes in Computer Science*, pages 414–432, Rio de Janeiro, Brazil, 2003. Springer.
- [67] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 707–710, Washington, DC, USA, Apr. 2001. IEEE Computer Society.
- [68] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Commun. ACM*, 39(4):54–63, 1996.
- [69] J. Ostell. Databases of discovery. *Queue*, 3(3):40–48, 2005.
- [70] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, 2003.
- [71] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distrib. Comput.*, 15(2):97–107, 2002.
- [72] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theor. Comput. Sci.*, 291(1):79–101, 2003.
- [73] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *EDCC-4: Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 44–61, London, UK, Oct. 2002. Springer-Verlag.

- [74] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.
- [75] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243(1-2):35–91, 2000.
- [76] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1206–1217, 2003.
- [77] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. *SIGPLAN Not.*, 39(11):48–58, 2004.
- [78] A. Schiper. Dynamic group communication. *Distributed Computing*, 18(5):359–374, Apr. 2006.
- [79] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.
- [80] A. Supriano, G. M. D. Vieira, and L. E. Buzato. Evaluation of a read-optimized database for dynamic web applications. In *WEBIST 2008: Proceedings of the Fourth International Conference on Web Information Systems and Technologies, Volume 1*, volume 1, pages 73–81, Funchal, Madeira, Portugal, May 2008. INSTICC Press.
- [81] TPC. *TPC Benchmark W Specification*, Feb. 2002.
- [82] P. Urbán, X. Défago, and A. Schiper. Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be? In *SRDS '01: Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 190–193, New Orleans, LA, USA, Oct. 2001. IEEE Computer Society.
- [83] P. Urbán, N. Hayashibara, A. Schiper, and T. Katayama. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 4–17, Washington, DC, USA, 2004. IEEE Computer Society.
- [84] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Softw. Pract. Exper.*, 28(9):963–979, 1998.

- [85] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: a flexible group communication system. *Commun. ACM*, 39(4):76–83, 1996.
- [86] G. M. D. Vieira and L. E. Buzato. On the coordinator’s rule for Fast Paxos. *Information Processing Letters*, 107:183–187, Aug. 2008.
- [87] G. M. D. Vieira and L. E. Buzato. Treplica: Ubiquitous replication. In *SBRC ’08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*, Rio de Janeiro, Brasil, May 2008.
- [88] G. M. D. Vieira and L. E. Buzato. The performance of Paxos and Fast Paxos. In *SBRC ’09: Proc. of the 27th Brazilian Symposium on Computer Networks and Distributed Systems*, pages 291–304, Recife, Brasil, May 2009.
- [89] G. M. D. Vieira and L. E. Buzato. Implementation of an object-oriented specification for active replication using consensus. Technical Report IC-10-26, Institute of Computing, University of Campinas, Aug. 2010.
- [90] G. M. D. Vieira, I. C. Garcia, and L. E. Buzato. Seamless Paxos coordinators. Technical Report IC-10-13, Institute of Computing, University of Campinas, Apr. 2010.
- [91] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *SRDS ’00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS’00)*, pages 206–215, Nürnberg, Germany, Oct. 2000. IEEE Computer Society.
- [92] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *ICDCS ’00: Proceedings of the The 20th International Conference on Distributed Computing Systems*, pages 464–474, Washington, DC, USA, Apr. 2000. IEEE Computer Society.
- [93] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, Apr. 2005.
- [94] S. Wu and B. Kemme. Postgres-R (SI): Combining replica control with concurrency control based on snapshot isolation. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 422–433, 2005.
- [95] K. Wuestefeld. Do you still use a database? In *OOPSLA ’03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 101–101, New York, NY, USA, 2003. ACM Press.