Anderson Parra de Paula

Uma Solução de Reconfiguração Leve para Paxos

Sorocaba, SP

Maio de 2015

Anderson Parra de Paula

Uma Solução de Reconfiguração Leve para Paxos

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCCS) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Arquiteturas Distribuídas de Software.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCCS

Orientador: Gustavo Maciel Dias Vieira

Sorocaba, SP

Maio de 2015

Anderson Parra de Paula
Uma Solução de Reconfiguração Leve para Paxos/ Anderson Parra de Paula. – Sorocaba, SP, Maio de 2015-
93 p. : il. (algumas color.) ; 30 cm.

Orientador: Gustavo Maciel Dias Vieira

Dissertação (Mestrado) – Universidade Federal de São Carlos – UFSCar
Centro de Ciências em Gestão e Tecnologia – CCGT
Programa de Pós-Graduação em Ciência da Computação – PPGCCS, Maio de 2015.

1. Palavra-chave1. 2. Palavra-chave2. I. Orientador. II. Universidade xxx. III. Centro de xxx. IV. Título

CDU 02:141:005.7

Anderson Parra de Paula

Uma Solução de Reconfiguração Leve para Paxos

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCCS) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Arquiteturas Distribuídas de Software.

Gustavo Maciel Dias Vieira
Orientador

Professor
Dr. Lásaro Jonas Camargos
UFU / FACOM

Professora
Dra. Yeda Regina Venturini
UFSCar / DComp

Sorocaba, SP
Maio de 2015

Para toda minha família, especialmente para minha esposa Ligia.

Agradecimentos

Agradeço,

ao Laboratório de Sistemas Distribuídos do Instituto de Computação da UNICAMP pela cessão das máquinas utilizadas nos experimentos aqui apresentados;

ao meu orientador, Dr. Gustavo Maciel Dias Vieira pelos ensinamentos e orientação durante esses anos;

a Menina Izildinha, que me acompanhou durante toda essa jornada.

"Quem pensa pouco, erra muito."
Leonardo da Vinci

Resumo

Paxos é um mecanismo de replicação ativa que consegue manter um mesmo estado compartilhado entre servidores que atendem a requisições de uma aplicação. É incomum encontrar aplicações onde a parte principal do processamento acontece através de um algoritmo de replicação como Paxos devido ao seu custo em termos do número de mensagens trocadas, o que limita a escalabilidade do sistema para algumas poucas réplicas. Para aumentar a aplicabilidade de replicação ativa, gostaríamos de ser ser capazes de, não só tornar a capacidade de processamento proporcional ao número de servidores empregados, mas também de variar essa capacidade dinamicamente em resposta às mudanças da demanda gerada. Nessa dissertação exploramos a questão da reconfiguração em sistemas de replicação ativa. Em particular, cobiçamos transformar a biblioteca de replicação Treplica em um sistema reconfigurável. Propomos dois novos mecanismos: (1) protocolo eficiente para transferência de estado; e (2) adição de novas réplicas sem aumentar de forma significativa o custo de manutenção da consistência do sistema como um todo. Nossa estratégia utiliza os dois mecanismos para criação de réplicas leitoras, que são capazes de atender todas as requisições da aplicação sem no entanto participarem ativamente das operações custosas do algoritmo Paxos.

Palavras-chaves: Replicação ativa. Paxos. Reconfiguração. Transferência de estado.

Abstract

Paxos is an active replication algorithm that keeps the same shared state consistently among servers that handle requests from an application. It is unusual to find applications where the main processing happens through a replication algorithm such as Paxos, mostly due to the high number of exchanged messages required to keep the state consistent. This restricts the system scalability to a handful of replicas. To increase the applicability of active replication, we would like be able to not only make the capacity of processing proportional to the number of servers employed, but also change dynamically the number of server according to demand. In this dissertation we explored reconfiguration on systems that use active replication. We proposed two mechanisms: (1) efficient protocolo for state transfer; and (2) incorporation of new replicas in the system with no significant increase in the cost to keep the whole system consistent. Our approach uses both mechanisms to create reader replicas, capable of answering all application requests without taking an active part in the costly operations of the Paxos algorithm.

Key-words: Active replication. Paxos. Reconfiguration. State transfer.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
IP	Internet Protocol
P2P	Peer-to-Peer
SGBD	Sistema de Gerenciamento de Banco de Dados
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Lista de símbolos

Ω	Letra grega Ômega

Sumário

Introdução

Atualmente, muitos dos programas que estamos acostumados a utilizar no dia-a-dia são distribuídos. Simples rotinas diárias como ler correio eletrônico ou navegar na Internet envolvem algum tipo de computação distribuída (CACHIN; GUERRAOUI; RODRIGUES, 2011). Podemos definir sistemas distribuídos como um conjunto de servidores (físicos ou virtuais) independentes que apresentam-se a seus usuários como um sistema único e coerente (TANENBAUM; STEEN, 2007). Também é verdade que as falhas nos servidores podem ocorrer de maneira independente. Essa noção de falhas parciais é bem ilustrada pela definição bem humorada apresentada por Lamport (1996) em que sistemas distribuídos são sistemas onde a falha de uma máquina que você nem sabia que existia pode tornar sua própria máquina inutilizável.

Quando existe uma falha em um servidor, o desafio para os que ainda estão operacionais é manter consistência e progresso na sincronização de suas atividades. Ou seja, a colaboração entre os servidores deve ser suficientemente robusta para suportar falhas parciais (CACHIN; GUERRAOUI; RODRIGUES, 2011). Sendo assim, o objetivo de sistemas tolerantes a falhas é continuar a prover serviços mesmo na ocorrência de defeitos em alguns de seus componentes, podendo até levar a reconfiguração do sistema para exclusão do componente defeituoso (TANENBAUM; STEEN, 2007).

Uma estratégia amplamente empregada em sistemas distribuídos para prover tolerância a falhas e aumento na capacidade de processamento é a replicação de dados. A *replicação ativa* (SCHNEIDER, 1990) é uma estratégia de replicação voltada para manutenção de um mesmo estado compartilhado entre servidores que atendem requisições de uma mesma aplicação, sendo cada um desses servidores chamados de *réplica*. A replicação ativa é baseada na re-execução em cada uma das réplicas das operações que alteram o estado compartilhado, devidamente ordenadas por um algoritmo apropriado (SCHNEIDER, 1990). Dentre os vários algoritmos de replicação, um dos mais amplamente usados e estudados atualmente é o algoritmo Paxos (LAMPORT, 1998).

Algoritmos de replicação são parte fundamental de várias arquiteturas distribuídas de software (CHANDRA; GRIESEMER; REDSTONE, 2007; HUPFELD et al., 2008; MACCORMICK et al., 2004), sendo particularmente usados como soluções para coordenação entre processos que implementam aplicações com garantias de consistência relaxadas (BURROWS, 2006) ou fazendo parte de algum esquema hierárquico de bloqueios (LAMPSON, 1996). Um exemplo de sistema que permite construir uma aplicação usando replicação ativa é a biblioteca Treplica (VIEIRA; BUZATO, 2008; VIEIRA; BUZATO, 2010), que provê uma forma simples e orientada a objetos para a construção de aplicações

altamente confiáveis. Utilizando essa biblioteca, a aplicação resultante preserva a consistência de uma aplicação centralizada e adiciona a tolerância a falhas de uma aplicação distribuída. Por outro lado, é incomum encontrar aplicações onde a parte principal do processamento acontece através de replicação ativa devido ao fato que essa estratégia possui um custo considerável em termos do número de mensagens trocadas, o que limita a escalabilidade do sistema além de algumas poucas réplicas (LAMPSON, 1996).

Apesar da maior confiabilidade, construir uma aplicação somente com replicação ativa potencialmente limita o quanto que essa aplicação pode tirar proveito dos ganhos de escala advindos de ser uma aplicação distribuída. Resultados experimentais mostram o impacto negativo do aumento da escala no desempenho da implementação de Paxos encontrada na biblioteca de replicação Treplica (VIEIRA; BUZATO, 2009). Gostaríamos de ser ser capazes de, não só tornar a capacidade de processamento proporcional ao número de servidores empregados, mas também de variar essa capacidade dinamicamente em resposta às mudanças da demanda gerada. Dessa forma, teríamos aplicações com a simplicidade de programação de aplicações centralizadas e características de aplicações distribuídas.

Nesse trabalho queremos explorar a questão da reconfiguração em sistemas de replicação ativa. Em particular, cobiçamos transformar Treplica em uma biblioteca reconfigurável, aproveitando o fato que essa biblioteca possuí o código fonte disponível tendo sido desenvolvida pelo orientador desse trabalho. No entanto, o problema de reconfiguração é complexo, principalmente na presença de falhas e assincronia. Sua resolução pode ser obtida através de duas estratégias: (1) transições de visões do conjunto de réplicas operacionais (BIRMAN; JOSEPH, 1987a; BIRMAN; JOSEPH, 1987b); (2) definição, via consenso, de uma nova configuração a partir da construção de uma barreira que, quando alcançada pelas réplicas, faz com que elas abandonem a configuração vigente e ingressem na nova configuração definida (caso elas façam parte dela) (LAMPORT; MALKHI; ZHOU, 2010). Em geral, sistemas de replicação ativa não suportam mecanismo de reconfiguração, sendo definidos como grupos de processos estáticos (CHANDRA; TOUEG, 1996; LAMPORT, 1998). Do ponto de vista teórico podemos encontrar um tratamento formal do problema em (LAMPORT; MALKHI; ZHOU, 2010), mas sistemas práticos tendem a evitar esta questão de forma a simplificar a sua construção (CHANDRA; GRIESEMER; REDSTONE, 2007).

Nessa dissertação propomos uma forma diferente de simplificar a reconfiguração considerando apenas um sub-conjunto desse problema: a adição de réplicas sem memória persistente. A nossa proposta está fundamentada na utilização de *réplicas leitoras*, que são capazes de atender a todas as requisições da aplicação sem, no entanto, requerer acesso à memória persistente e sem, na prática, participarem ativamente das operações custosas do algoritmo Paxos. Identificamos a necessidade da criação de um mecanismo

eficiente para *transferência de estado* entre réplicas. Estamos preocupados com o impacto inerente para implantar uma nova réplica em um aglomerado em tempo de execução. Tal impacto dificulta a viabilidade das técnicas de autogestão, porque dependendo do cenário, adicionar uma nova réplica pode comprometer o desempenho de um sistema sobrecarregado (VILAÇA; PEREIRA; OLIVEIRA, 2009).

Esse trabalho apresenta duas novas funcionalidades para expansão da biblioteca Treplica:

1. Mecanismo de transferência de estado: criação de um protocolo eficiente para transferência de estado entre réplicas.

2. Réplicas leitoras: possibilidade da utilização de réplicas que não participam do processo de decisão de instâncias de consenso.

Implementamos e validamos experimentalmente a nossa proposta obtendo resultados de pesquisa interessantes. Observamos que o mecanismo de transferência de estado minimizou o impacto para adição de uma nova réplica no aglomerado. No presente momento, o mecanismo de transferência de estado proposto foi implementado somente no contexto de réplicas leitoras, mas ele é genérico o suficiente para suportar qualquer tipo de réplica. Fomos capazes de provisionar uma réplica leitora em um tempo mais curto do que o mecanismo anterior implementado em Treplica, utilizando de forma mais eficiente os recursos disponíveis. Pudemos também constatar que o uso de réplicas leitoras permitiu a expansão do aglomerado experimental de forma simples conseguindo evitar a difícil tarefa de reconfiguração total, porém sem um grande ganho de desempenho. Mesmo assim, acreditamos termos construído mecanismos úteis para sistemas de replicação ativa.

O restante desta dissertação está organizada da seguinte forma: O Capítulo 1 introduz a estratégia de replicação ativa para tolerância a falhas, o modelo computacional utilizado e descreve o algoritmo Paxos e a biblioteca Treplica. Já o Capítulo 2 expõe em detalhes a ideia de reconfiguração em Treplica e os componentes necessários para execução de tal tarefa. O Capítulo 3 apresenta e discute os resultados experimentais. Por fim, a Conclusão explicita conclusões e trabalhos futuros.

1 Replicação Ativa e Paxos

Normalmente, uma aplicação distribuída é estruturada em termos de clientes e serviços. Cada serviço suporta uma ou mais operações invocadas pelos clientes através de requisições remotas. Embora o uso de um único servidor (computação centralizada) seja a maneira mais simples para implementar um serviço, a confiabilidade desse serviço resultante está diretamente ligada a tolerância a falhas do servidor. Se esse nível de tolerância é inaceitável, então a mesma versão desse serviço deve ser replicada em servidores diferentes visando mitigar os erros. O isolamento físico das unidades de processamento em um sistema distribuído garante que as falhas serão independentes (SCHNEIDER, 1990).

Redundância é a técnica chave para tolerância a falhas e alta disponibilidade. O princípio básico dessa técnica é que se um dos componentes redundantes falha, os demais podem continuar o trabalho em seu lugar, gerando o mínimo de interrupção do serviço. Criar um *grupo de réplicas* para suportar a implementação de um serviço minimiza a percepção de falhas do cliente. Entretanto, para que o grupo de réplicas seja transparente ao cliente, é preciso adotar uma estratégia de replicação capaz de coordenar as réplicas no momento que falham (JALOTE, 1994).

A coordenação do componente replicado é realizado por software, usando estratégias como *replicação ativa* capaz de tolerar falhas de maneira transparente visando o progresso da aplicação. O princípio por trás da replicação ativa é a suposição de que cada *réplica* opera como uma máquina de estados determinista, de forma que elas mantenham-se idênticas ao executarem as mesmas transições de estado, na mesma ordem (SCHNEIDER, 1990). A consistência de atualização de dados é mantida por algum protocolo, executado pelas réplicas, que permita difundir e ordenar totalmente essas transições.

Neste capítulo apresentamos os aspectos conceituais de aplicações que utilizam o modelo de replicação ativa através do algoritmo Paxos necessários para compreensão do restante do trabalho. Na Seção 1.1, começamos definindo conceitos fundamentais sobre o modelo computacional adotado e a Seção 1.2 define o modelo operacional utilizado na construção da aplicação. A Seção 1.3 examina detalhadamente o modelo de replicação ativa e na Seção 1.4 descrevemos o funcionamento do algoritmo Paxos e seus principais componentes. Em seguida, na Seção 1.5 apresentamos a biblioteca Treplica que é uma implementação do modelo de replicação ativa utilizando Paxos. Na Seção 1.6, falamos dos principais aspectos de reconfiguração em um sistema distribuído executando Paxos. Finalmente, a Seção 1.7 encerra o capítulo mostrando os trabalhos relacionados a esta dissertação.

1.1 Modelo Computacional

Um sistema distribuído pode ser definido como um conjunto de réplicas autônomas, interconectadas via rede, que se comunicam através de troca de mensagens. Para este trabalho, adotamos o modelo modelo computacional *falha-e-recuperação assíncrono*. As premissas em relação às réplicas são: 1) réplicas passam por períodos de instabilidade e falham, mas conseguem se recuperar, porém perdendo a sua memória volátil; 2) uma réplica é considerada defeituosa se não consegue se recuperar mais ou se fica em um ciclo constante de falha e recuperação (CACHIN; GUERRAOUI; RODRIGUES, 2011).

Essas réplicas trocam mensagens por um enlace de *perda-justa*, que possui as seguintes propriedades (CACHIN; GUERRAOUI; RODRIGUES, 2011):

- Perda-justa: se um processo correto p envia a mensagem m infinitas vezes para um processo correto q, então q entrega m um número infinito de vezes;
- Duplicação finita: se um processo correto p envia um número finito de vezes a mensagem m para um processo correto q, então m não pode ser entregue infinitas vezes por q; e
- Sem criação: se algum processo q entrega a mensagem m, então m foi anteriormente enviada por algum processo p para q.

Enlaces de perda-justa podem ser implementados diretamente sobre uma rede de comunicação, ou seja, a implementação da abstração não garante as propriedades, apenas percebe que elas existem no substrato escolhido. O protocolo UDP é um exemplo de protocolo que implementa o modelo perda-justa, na ausência de falhas graves. É importante ressaltar que a propriedade que garante entrega confiável de mensagens entre réplicas não é garantida pelo modelo de comunicação perda-justa. Essa propriedade é inerente ao algoritmo Paxos.

Relacionado ao modelo computacional assíncrono, as seguintes premissas são relevantes: 1) não existe limite para a diferença de velocidade de processamento entre duas réplicas; 2) não existe limite para a latência de transferência de uma mensagem. Então, podemos afirmar que não existe nenhuma suposição de sincronia entre réplicas baseado nessas propriedades.

1.1.1 Tolerância a Falhas

Um componente é considerado defeituoso quando seu comportamento não está de acordo com sua especificação (SCHNEIDER, 1990). Para tolerância a falhas, a perspectiva do usuário é a mais importante. Por isso, gostaríamos que as aplicações distribuídas continuassem progredindo apesar das falhas (JALOTE, 1994).

Os principais componentes de um sistema distribuído são: processadores, redes, relógios, armazenamento não volátil e software. Normalmente esses componentes não são construídos para suportar tolerância a falhas e são suas estruturas que utilizamos para apoiar os mecanismos de tolerância a falhas. Exceto software, todos os outros componentes são físicos e sua falha pode ter causas físicas subjacentes. Suportar por software falhas em mecanismos físicos é o principal foco da maioria dos mecanismos de tolerância a falhas, especialmente em servidores e redes de comunicação (JALOTE, 1994).

Tais falhas podem ser classificadas de acordo com o comportamento defeituoso apresentado no momento da falha (JALOTE, 1994). Essa classificação pelo comportamento da falha especifica quais pressupostos podem ser feitos sobre o comportamento do componente quando ele falha. Uma dessas classificações é dada por Cristian, Aghili e Strong (1986), onde cada falha pertence a uma de quatro categorias: Colapso (*Crash*), Omissão (*Omission*), Tempo (*Timing*) ou Bizantina (*Byzantine*).

- Falha por Colapso: esta falha faz com que a réplica paralise e perca seu estado interno. Esse tipo de falha nunca leva o componente a sofrer uma transição de estado incorreta quando ele falha.

- Falha por Omissão: ocorre quando a réplica deixa de responder a algumas requisições enviadas a ela. Quando a réplica falha por omissão e deixa de responder indefinidamente, considera-se que a semântica da falha é silenciosa (CRISTIAN, 1991).

- Falha por Tempo: quando uma réplica responde uma requisição demasiadamente tarde é classificada como falha por tempo.

- Falha Bizantina: ocorre quando não é feita nenhuma restrição quanto ao comportamento falho da réplica. Nessa semântica, a réplica pode deixar de responder, atrasar envio de respostas, enviar respostas erradas, enviar respostas diferentes para diferentes destinos, recusar o recebimento de requisições ou ainda, maliciosamente, fazer se passar por outra réplica (JALOTE, 1994).

Esse grupo de falhas forma uma hierarquia, sendo a falha por colapso a mais simples e restritiva (ou bem definida) e a falha Bizantina a menos restritiva (JALOTE, 1994). A relação de inclusão é ilustrada na Figura 1 (CRISTIAN; AGHILI; STRONG, 1986):

Projetamos sistemas distribuídos tolerantes a falhas para fornecer um serviço confiável e contínuo, apesar das falhas de alguns de seus componentes. Um mecanismo básico utilizado na construção é o *detector de falhas*, que a grosso modo, fornece algumas informações sobre as réplicas defeituosas. Em ambientes assíncronos, a informação de falha

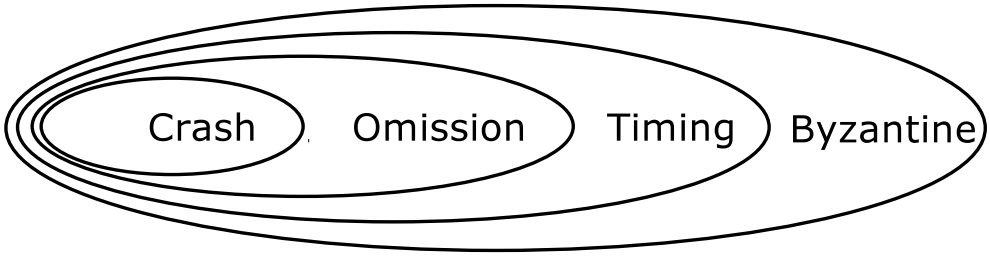


Figura 1: Classificação de falhas

normalmente é dada através de uma lista de suspeitas, que nem sempre estão atualizadas ou corretas. Um detector de falhas pode levar um longo tempo para suspeitar de uma réplica que deixou de funcionar e pode erroneamente suspeitar de uma réplica correta, na prática isto pode ser devido à perda de mensagens e/ou atrasos (CHANDRA; TOUEG, 1996; CHEN; TOUEG; AGUILERA, 2002).

1.2 Modelo Operacional

Para facilitar a construção de uma aplicação que utiliza o modelo de replicação ativa, esse trabalho emprega uma arquitetura separada em camadas. Essa divisão visa facilitar a resolução do problema, mitigar os erros e facilitar a compreensão dos experimentos.

Em inglês, duas palavras são comumente utilizadas para se falar de divisão em camadas: *layers* e *tiers*. Ao falar em *layers*, normalmente pensa-se em separações lógicas, em como organizar a aplicação de maneira a diminuir o acoplamento entre diferentes partes do código e evitar que mudanças em um lugar afetem outros. Já a divisão em *tiers* visa as separações físicas entre partes do sistema (SILVEIRA et al., 2011).

1.2.0.1 Layers

Para concepção do presente trabalho, supomos uma aplicação Web, conforme ilustrada na Figura 2, utilizando um *middleware* que implementa o modelo de replicação ativa. A função principal desempenhada por esse *middleware* de replicação é gerenciar o estado da aplicação, executando as requisições dos clientes através de uma abstração de uma *máquina de estados replicada* (SCHNEIDER, 1990). Essa abstração é implementada utilizando o algoritmo Paxos (LAMPORT, 1998).

Supomos que uma aplicação que use replicação ativa o faz na forma de um servidor que atende *requisições* de *clientes*. Esses clientes não têm conhecimento de como as requisições são executadas. As réplicas recebem essa requisições e as processam de forma a manter a consistência do estado replicado (compartilhado). Cada uma dessas requisições executa o equivalente a uma chamada de função no contexto do estado da aplicação. Se

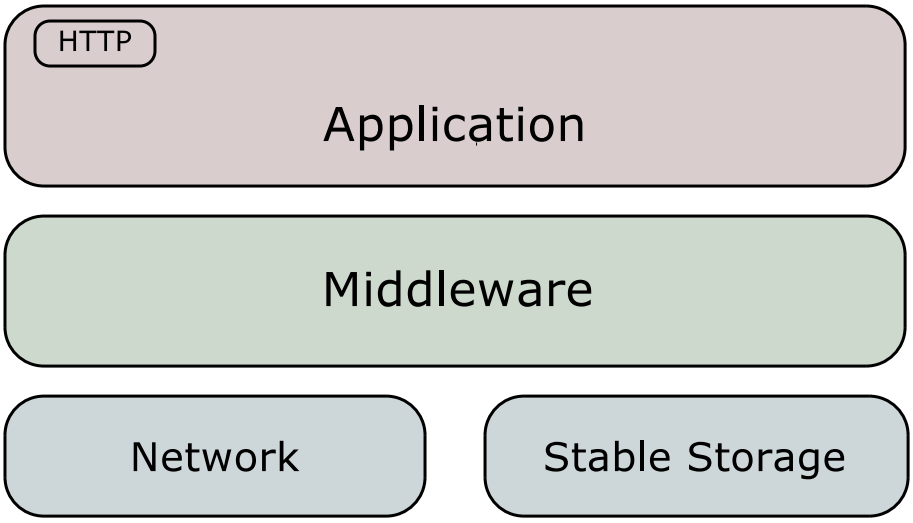


Figura 2: Separação em *tiers*

a função subjacente altera o estado, esta é uma *requisição de escrita* e deve ser difundida e ordenada entre todas as réplicas antes de ser executada. Caso a função subjacente não altere estado, esta é uma *requisição de leitura* e pode ser executada localmente sem coordenação entre as réplicas.

1.2.0.2 Tiers

Do ponto de vista da estruturação física para suportar a aplicação, supomos um conjunto de réplicas que é gerenciada por um servidor balanceador de carga, usado para melhorar o desempenho de serviços Web distribuindo requisições entre vários servidores. Todas as requisições dos clientes passam pelo balanceador de carga, configurado para receber e rotear as requisições para as réplicas ativas, usando um algoritmo de revezamento circular (*round-robin*), como ilustrado na Figura 3.

O balanceador de carga monitora as réplicas para tentar detectar quando a aplicação está disponível. Assim que uma réplica está disponível ela passa a receber requisições roteadas pelo balanceador de carga. O monitoramento é realizado através de pequenos pacotes *keepalive* que são enviados para cada réplica do aglomerado. Todos os componentes estão conectados através de um *switch* que se comporta como uma canal de perda-justa.

1.3 Replicação Ativa

Em geral, é uma boa ideia replicar serviços ou dados (estado) de uma aplicação. A replicação é um mecanismo fundamental em sistemas distribuídos, proporciona maior disponibilidade e ajuda a equilibrar a carga entre componentes, o que potencialmente resulta em melhor desempenho (TANENBAUM; STEEN, 2007). O acesso a serviços ou dados replicados deve ser transparente para o usuário, ele deve ser realizado da mesma

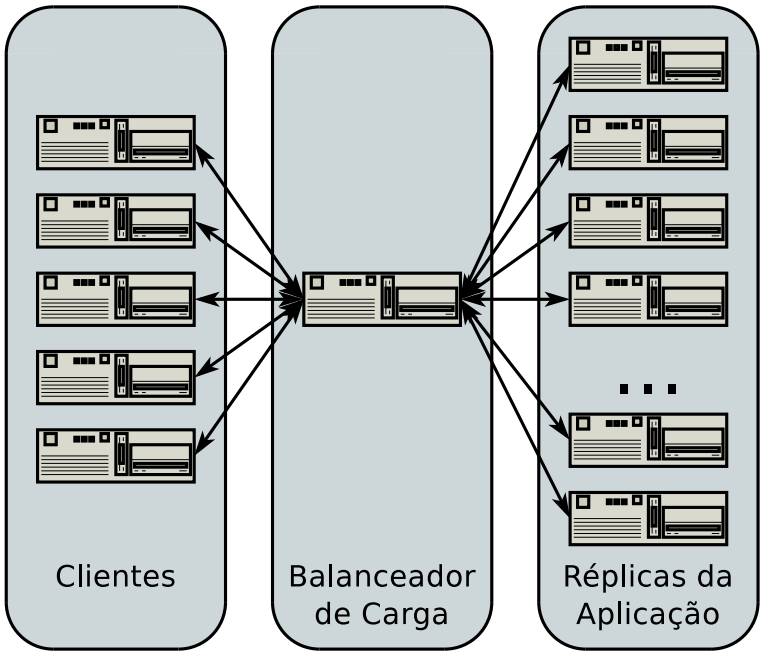


Figura 3: Separação em *layers*

forma que o faria como se não houvesse replicação. A consistência dos dados replicados deve ser garantido automaticamente pelas réplicas e mesmo que mais de uma réplica responda a uma requisição, apenas uma resposta deve ser entregue ao usuário final.

A criação de um grupo de réplicas minimiza a percepção de falhas de um cliente que acessa um estado compartilhado. Entretanto, para que o grupo de réplicas seja transparente para aplicação cliente, é preciso adotar uma estratégia de replicação a fim de coordenar as réplicas preparando-as para que, no caso de ocorrer uma falha, uma réplica possa dar continuidade ao processamento do sistema (JALOTE, 1994). Existem duas técnicas seminais de replicação de processos: a *replicação ativa* e a *replicação passiva* (JALOTE, 1994). Esses modelos de replicação são capazes de tolerar falhas de maneira transparente para o cliente.

Os processos na replicação ativa (COULOURIS; DOLLIMORE; KINDBERG, 2011; GUERRAOUI; SCHIPER, 1997) ou replicação por máquina de estados (SCHNEIDER, 1990), funcionam como máquinas de estado deterministas, ou seja, o estado atual é determinado única e exclusivamente por um estado inicial e uma sequência de transições. Desta forma, se os processos replicados recebem a mesma sequência de transições eles atingirão estados idênticos.

Cada operação que altera o estado de uma réplica (requisição de escrita) deve ser executada de maneira determinista no conjunto de réplicas, ou seja, todas as réplicas recebem e processam a mesma sequência de requisições, conforme ilustrado na Figura 4.

A propagação das requisições de mudança no conjunto de réplicas garante a sincronização consistente global do estado, pois é replicando a atualização de estado sofrida por

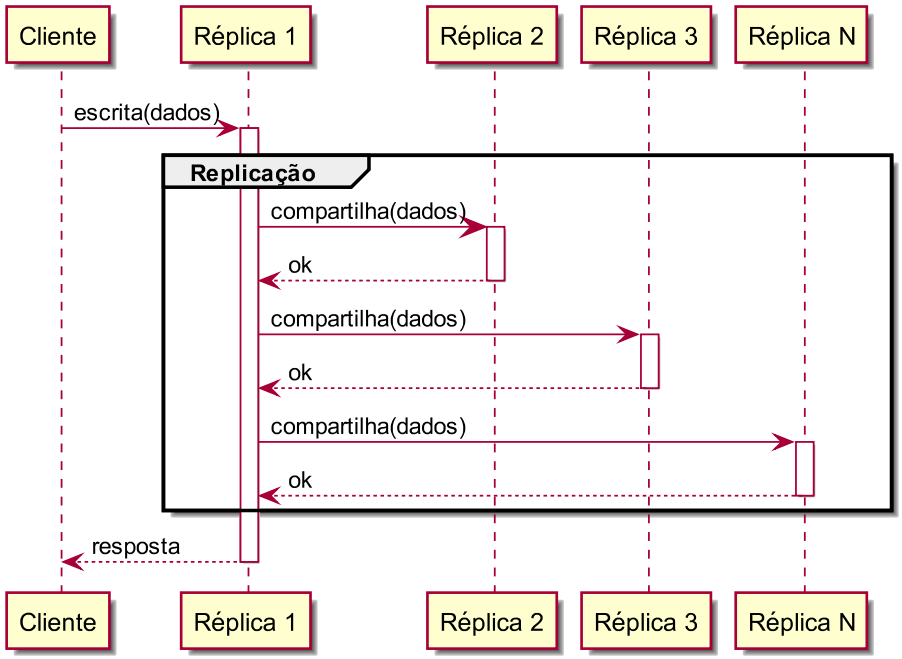


Figura 4: Requisição de escrita

uma réplica em todas as outras réplicas que uma requisição de leitura pode ser executada localmente em qualquer réplica com a certeza do mesmo resultado. A Figura 5 ilustra uma requisição de leitura, repare que não existe propagação da requisição para outras réplicas, a requisição é processada localmente.

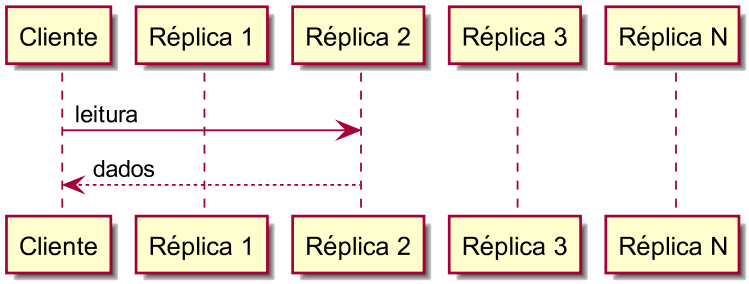


Figura 5: Requisição de leitura

Não é necessário modelar o funcionamento de uma aplicação que use replicação ativa como uma máquina de estados explícita, basta considerar as funções da aplicação que mudam o estado como transições. As funções que não mudam o estado não têm correspondência na abstração de máquina de estados, sendo simples consultas ao estado armazenado e não exigem ordenação.

Uma desvantagem da replicação ativa é que por requerer réplicas deterministas ela restringe o uso de múltiplos fluxos de execução (*multithreading*) e outros mecanismos que podem causar indeterminismos, mas que podem melhorar o desempenho. O requisito de ordem total pode ser relaxado se as requisições dos clientes forem comutativas ou idempotentes. Para um sistema $f-$tolerante a falhas, são necessários $f+1$ réplicas ati-

vas para garantir progresso da aplicação (JALOTE, 1994; LAMPORT; MALKHI; ZHOU, 2010). Requisições de escrita encarecem a aplicabilidade desse modelo, pois precisamos escrever em um número grande o suficiente de réplicas de tal forma que a escrita não seja esquecida.

Replicação ativa consistente é uma tarefa complicada e cara porque no modelo computacional adotado não há uma ideia de relógio global compartilhado. Ou seja, processos em máquinas diferentes têm sua própria ideia do que é o tempo e como ele passa, logo não podem determinar com certeza se alguma das outras réplicas estão defeituosas. Para garantir a consistência, as réplicas devem executar a mesma sequência de eventos, dessa forma é possível manter o estado análogo em qualquer réplica, mesmo que o mesmo evento em máquinas diferentes seja processado em momentos distintos (TANENBAUM; STEEN, 2007). Ordenar as requisições de forma única, na presença de falhas, é equivalente a resolver o problema de consenso (CHANDRA; TOUEG, 1996).

1.4 Paxos

O algoritmo Paxos (LAMPORT, 1998) é um protocolo de difusão atômica utilizado para replicação ativa através consenso (BARBORAK; DAHBURA; MALEK, 1993). Paxos é destinado ao modelo assíncrono de computação com falha-e-recuperação que utiliza detectores de falha não confiáveis (*unreliable failure detectors* (CHANDRA; TOUEG, 1996)). Em particular, ele garante que o estado de nenhuma réplica irá divergir do das demais, mesmo na presença de falhas e de comportamento assíncrono de processamento e canais de comunicação.

Paxos emprega um protocolo baseado em quórum para atualizar os dados replicados em um grupo. O objetivo do algoritmo é coordenar as réplicas para que todas tenham o mesmo estado (CACHIN; GUERRAOUI; RODRIGUES, 2011). As garantias de progresso (*liveness*) e a correção (*safety*) do algoritmo são dadas pelas seguintes propriedades:

- Validade uniforme: se um processo decide v então algum processo anteriormente propôs v.
- Acordo: não existem dois processos corretos que decidem valores diferentes.
- Encerramento: se todos os processos corretos propõem um valor, então mais cedo ou mais tarde, eles serão decididos.

Por utilizar um mecanismo baseado em quórum, o algoritmo não depende de que todas as réplicas estejam ativas para garantir que a decisão seja durável. Entenda por

durabilidade[1] que uma vez que a proposta foi confirmada, ela permanecerá assim, mesmo em caso de falhas ou erros. Ou seja, como estamos no modelo falha-e-recuperação, os dados foram gravados em memória não-volátil.

1.4.1 Algoritmo Básico

No Paxos os processos no sistema são agentes reativos que podem assumir vários papéis: *proponente* (*proposer*) que pode propor valores, *receptor* (*acceptor*) que escolhe um único valor ou *aprendiz* (*learner*) que aprende o valor escolhido. Para resolver o consenso, agentes do Paxos executam várias *rodadas* (*rounds*), onde cada rodada possui um *coordenador* (*coordinator*) e é unicamente identificada por um inteiro positivo, o *número de rodada*. Proponentes enviam a sua *proposta* para o coordenador que tenta alcançar consenso sobre a proposta em uma rodada, sendo que cada proposta corresponde a uma ou mais requisições de escrita da aplicação sendo replicada.

O coordenador é responsável por essa rodada e executa um número definido de passos de comunicação para garantir que a decisão tomada nessa rodada seja aceita pelos demais processos, ou seja, um consenso. Esse agente de Paxos é capaz de decidir, após aplicar uma regra local, se outras rodadas tiverem 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 na aplicação (LAMPORT, 1998).

Cada rodada acontece em duas fases, com dois passos cada, como ilustrado na Figura 6 (todos os processos podem assumir todos os papéis):

- 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 informando a última proposta votada por esse receptor e a rodada em que esse 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 que poderia ter sido decidida em rodadas com número menor que r. Ele então pede a esses receptores para votar nessa proposta, ou caso ela seja *null*, para votar em uma das propostas feitas pelos proponentes.

[1] Semelhante a propriedade ACID definida em transações de banco de dados (HAERDER; REUTER, 1983)

- 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 receber mensagens da Fase 2b de um quórum de receptores, onde todos votaram em p na mesma rodada r.

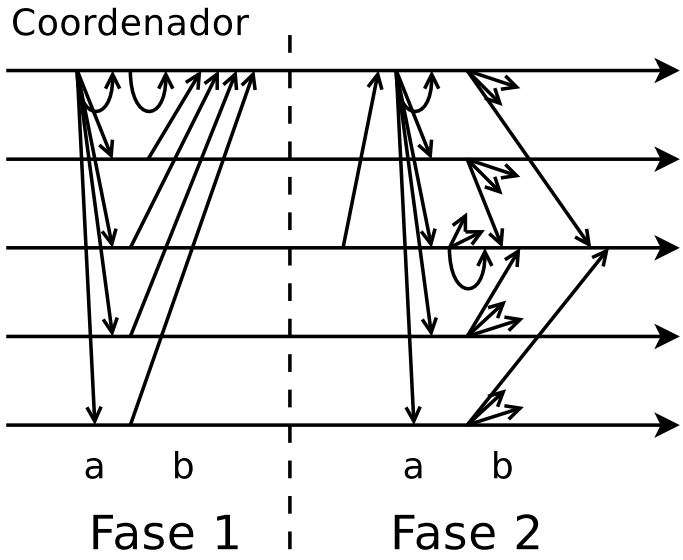


Figura 6: Paxos

Cada rodada corresponde à decisão de uma proposta apenas. Porém Paxos define também uma forma de entregar um conjunto de requisições de escrita totalmente ordenadas. A ordem de entrega dessas requisições é determinada pela sequência dos inteiros positivos, tal que cada inteiro corresponde a uma *instância* de consenso. Cada instância i terá um valor decidido, que corresponde a i-ésima requisição (ou conjunto ordenado de requisições) a ser executada na sequência de requisições. Cada instância de consenso é independente das demais e várias instâncias podem estar em curso ao mesmo tempo.

Como Paxos é definido no modelo falha-e-recuperação assíncrono, esse algoritmo exige que os agentes armazenem estado em memória não volátil (LAMPORT, 2006). Esse estado é composto por um registro das instâncias iniciadas, os números de rodadas usados e as propostas feitas e votadas, entre outros dados. De forma resumida, o coordenador escreve na memória persistente na Fase 1a e os receptores o fazem nas Fases 1b e 2a. Não há a necessidade de que a proposta decidida seja registrada para garantir a correção do algoritmo, mas isto é normalmente feito para acelerar a recuperação. A escrita de dados em memória persistente faz parte do caminho crítico de desempenho da execução das duas fases do algoritmo Paxos. Logo, dependendo do custo de comunicação de rede, esse é o principal gargalo de desempenho para a execução de Paxos.

Em Paxos, qualquer processo pode agir como o coordenador de uma rodada enquanto ele seguir a regra para escolher uma proposta coerente como o resultado das rodadas anteriores na Fase 2a. A escolha do 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 progresso. Especialmente, 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 esses múltiplos coordenadores competem pela atenção dos receptores com números de rodada que crescem rapidamente. Por esta razão, o progresso 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.

1.4.2 Recuperação

Uma consideração em relação ao funcionamento típico de Paxos é o que acontece quando uma réplica inicia depois que o sistema já está operando ou quando reinicia após uma falha prolongada. Esse caso não é explicitamente definido na descrição clássica de Paxos, porém o algoritmo permite que um número arbitrário de rodadas aconteçam *antes* ou *depois* que o consenso seja alcançado (LAMPORT, 1998). Dessa forma, podemos supor um mecanismo de recuperação simples onde uma réplica que ficou fora de operação por algum tempo atualiza o seu estado pela decisão sucessiva das instâncias de consenso que ela não tem conhecimento. Esse mecanismo é apropriado para pequenas interrupções, como a perda de algumas mensagens ou uma falha transiente. Porém, caso uma réplica tenha um grande estado para atualizar esse procedimento pode ter impacto adverso no desempenho do sistema.

Para exemplificar o possível impacto de uma recuperação, vamos supor as seguintes propriedades relacionadas a atualização de uma réplica desatualizada:

- Consistência: Uma réplica desatualizada r volta a computação após n rodadas de consenso, onde n é um número arbitrariamente grande. Se r aplicar as n decisões de consenso sucessivamente atingirá o mesmo estado das demais réplicas.

- Volatilidade do estado: Se o número de rodadas de consenso n muda em uma velocidade maior que a réplica desatualizada r consegue atualizar seu estado, r estará sempre desatualizada (VILAÇA; PEREIRA; OLIVEIRA, 2009).

Baseado nessas propriedades de consistência e volatilidade não podemos afirmar

nada com relação ao tempo de recuperação de estado de uma réplica. O mecanismo de recuperação por aplicação de decisões de consenso é ineficiente quando uma réplica possui um estado muito antigo ou quando a velocidade de recuperação é inferior a vazão do sistema. Para atendermos eficientemente este cenário, precisamos de uma nova política de recuperação. Uma alternativa que propomos nessa dissertação é transferir para a réplica desatualizada o estado de uma réplica atualizada, como descreveremos na Seção 2.2.1. Dessa forma, supriremos a lacuna de desatualização substituindo por completo o estado desatualizado por um atualizado, semelhante à ideia de um transplante.

1.4.3 Estado persistente

Para satisfazer as propriedades de consenso no modelo falha-e-recuperação assíncrono é preciso utilizar um mecanismo capaz de gravar dados de forma persistente (não-volátil). Dessa forma, após um período de instabilidade uma réplica é capaz de recuperar suas decisões de consenso tomadas anteriormente para continuar participando corretamente do protocolo. A cada fase do algoritmo acontecem dois acessos a disco, conforme ilustrado na Figura 7.

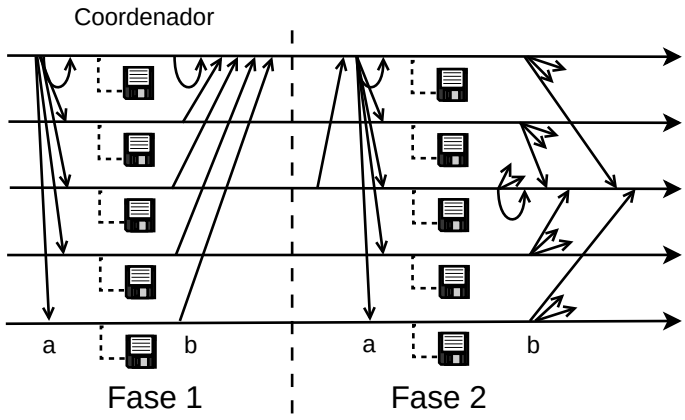


Figura 7: Acesso a disco em Paxos

- Na Fase 1, após receber um convite do coordenador para participar da rodada r, todo receptor que aceitou o convite, escreve r em disco.
- Na Fase 2, antes de enviar seu voto para o coordenador os receptores escrevem no disco o par (rodada, voto).

Operações de escrita em memória persistente são custosas e têm grande potencial para reduzir o desempenho do algoritmo. Por exemplo, considere que cada operação de escrita em disco leva cerca de $1ms$ para ser concluída e que uma rodada de Paxos, pelo menos, requer duas escritas em memória estável, acabamos de adicionar uma latência de $2ms$ em todas as rodadas de consenso. Aguilera, Chen e Toueg (2000) propõem a

combinação de réplicas com diferentes graus de uso de memória persistente para diminuir o custo do armazenamento dos dados. Esse trabalho apresenta a resolução de consenso nesse modelo com um número variável de réplicas com memória persistente, desde que seja sempre possível entrar em contato com uma réplica que use memória persistente ou com uma réplica sem memória persistente que nunca falhe. A proposta de réplicas leitoras proposta nessa dissertação garante que os quóruns responsáveis pelo consenso sejam sempre compostos por réplicas que usam memória persistente, como descreveremos na Seção 2.2.2

1.5 Treplica

Treplica (VIEIRA; BUZATO, 2008; VIEIRA; BUZATO, 2010) se situa a meio caminho entre flexibilidade de baixo nível de um sistema de comunicação em grupo (BIRMAN, 1993) baseado em sincronia virtual (FRIEDMAN; RENESSE, 1996; BIRMAN, 2005) e os vastos recursos de processamento de dados de um SGBD. A principal característica de Treplica é a ideia de se apresentar ao programador como uma abstração de programação unificada para replicação e persistência, propondo o uso de consenso como a fundação para a construção dessa ferramenta.

Treplica foi projetada para prover uma forma simples e orientada a objetos de se construir aplicações altamente confiáveis. Essas aplicações podem se estender ao sistema inteiro ou se restringir à subsistemas onde consistência e confiabilidade são cruciais. Para alcançar esse objetivo, Treplica decompõe o problema de se implementar replicação em componentes com interfaces simples e bem definidas.

Dessa forma, um desenvolvedor que deseje implementar uma aplicação distribuída não precisa pensar em termos de processos, mensagens e falhas. Ao invés disso, ele pensa sobre a execução das operações da aplicação, transições de uma máquina de estados replicada, que são disparadas por eventos disponibilizados através de uma fila persistente assíncrona (VIEIRA; BUZATO, 2010). A Figura 8 mostra a interface destes componentes e sua relação com a aplicação e entre si.

A principal decisão de projeto por trás de Treplica é que o programador pode considerar a sua aplicação como não tendo estado persistente, ficando a durabilidade da mesma garantida pela biblioteca. Essa decisão é suportada pela observação que os mesmos requisitos de replicação ativa podem ser usados para prover um mecanismo simples e poderoso de persistência.

A replicação ativa exige que a aplicação execute ações que modificam o seu estado de forma determinista. Essas ações são então propagadas, na mesma ordem, para todas as réplicas de um serviço que as reexecutam localmente. Dentro dessa organização, as ações não são apenas enviadas para as outras réplicas, mas arquivadas em memória estável

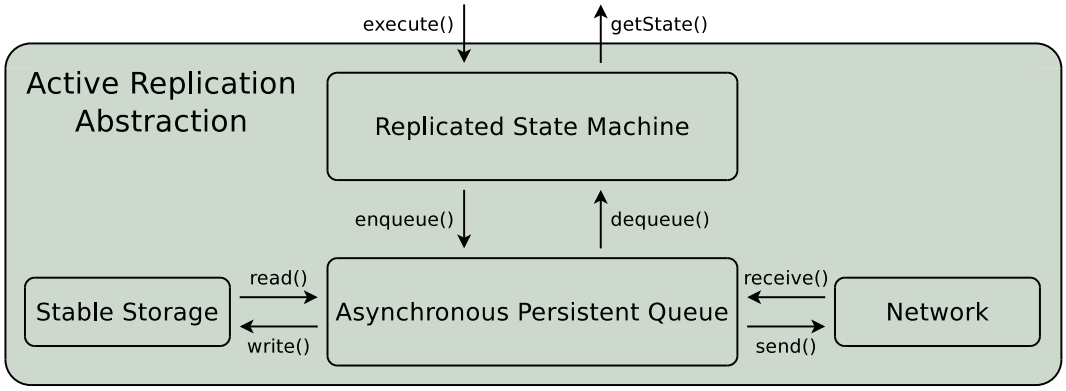


Figura 8: Componentes para replicação

(BIRRELL; JONES; WOBBER, 1987). Dessa forma, é possível se recuperar de uma falha reexecutando o arquivo de ações. O determinismo garante que após cada recuperação a aplicação reiniciará com o mesmo estado que possuía antes da falha. Por razões de eficiência, a aplicação deve caber inteiramente em memória principal, pois a biblioteca não armazena a aplicação em si, apenas mudanças de estado.

A máquina de estados replicada provê uma abstração da operação de qualquer aplicação determinista. Ela permite a manutenção do estado da aplicação estipulando uma interface simples para consultar e modificar esse estado. Esse componente é acessado diretamente pela aplicação que usa os seus serviços para armazenar, replicar e persistir o seu estado. Todas essas operações são realizadas de forma transparente e não exigem intervenção por parte do usuário desse componente.

A fila persistente assíncrona é uma abstração de uma fila de objetos persistentes e tolerante a falhas. Ela representa um registro ordenado de objetos enviados a um grupo de processos, que é garantidamente disponível mesmo se todos os processo deste grupo falhem. Ela pode ser usada como um registro persistente dos eventos que disparam transições na máquina de estados distribuída e é implementada usando algum mecanismo de difusão ordenada de mensagens, como Paxos descrito na Seção 1.4.

1.6 Reconfiguração

O processo de modificar o conjunto de réplicas que compõem o sistema chama-se *reconfiguração*. Alterar o número de réplicas participantes de uma aplicação que utiliza o modelo de replicação ativa não é uma tarefa trivial pois a informação de *cardinalidade* do conjunto de réplicas é relevante para o mecanismo de replicação gerenciar a consistência do estado compartilhado. Segundo Lamport, Malkhi e Zhou (2010) reconfigurar é custoso, pois o algoritmo Paxos trabalha com consenso e para que uma rodada de consenso ocorra é preciso que o número de réplicas participantes seja claramente definido. Isso obriga que

cada expansão ou redução do grupo de réplicas seja precedido de reconfiguração.

Paxos é um protocolo baseado em quóruns, então a retirada de uma réplica da computação sem a devida reconfiguração afeta diretamente o funcionamento correto do algoritmo que foi projetado para trabalhar com um grupo estático de réplicas (CHANDRA; TOUEG, 1996; LAMPORT, 1998), onde não é permitida a entrada e saída de servidores durante a execução. Essa abordagem não é adequada para sistemas que permanecerão em execução por um longo tempo, pois limitam a atuação de um administrador, que em tempo de execução não pode adicionar máquinas ao sistema (para suportar um aumento na carga de processamento) ou trocar máquinas antigas (para efetuar um reparo no hardware) (ALCHIERI et al., 2014). Oferecer suporte a esses requisitos caracteriza, em sua forma mais simples, um comportamento elástico que a aplicação deve suportar para refletir mudanças do ambiente operacional durante o período de execução.

Para formar uma maioria que votou em uma mesma proposta é preciso que o mecanismo de decisão de consenso conheça o número exato de réplicas em uma determinada rodada. Por exemplo, caso existam apenas $\lfloor n/2 \rfloor + 1$ potenciais réplicas e uma réplica r for removido de forma ingênua, a formação legítima de quóruns não é mais possível. Nesse caso, réplicas corretas serão levadas a 1) suspeitar indevidamente que r falhou, mas na realidade r não pertence mais ao grupo de réplicas; 2) esperar indefinidamente por uma resposta de r para determinar a maioria. Essa espera indevida pode impedir a progresso do algoritmo.

Por outro lado, uma aplicação configurada para executar com n réplicas não pode adicionar uma nova réplica de forma arbitrária sem ser precedida de reconfiguração. Caso existam $n+1$ réplicas ativas, a formação de um quórum que contenha todas as decisões de consenso pode não ser mais verdade, pois não podemos mais garantir interseção entre os diferentes quóruns do conjunto de réplicas. Dessa forma, afetamos diretamente a correção do algoritmo.

Para oferecer o dinamismo necessário na *ampliação* e *redução* de grupos de réplicas em sistemas distribuídos modernos, Lamport, Malkhi e Zhou (2010) propõem a criação de um mecanismo de visões. Nessa proposta uma nova visão v do sistema é criada sempre que uma réplica r for adicionada ou retirada da computação. Nessa visão v, r pode fazer parte ou não do processamento computacional, dependendo da operação executada. Durante a execução de uma aplicação, ela pode passar por diferentes visões (configurações) sem afetar seu progresso e correção, mas sistemas práticos tendem a evitar mudanças de configuração de forma a simplificar a construção de aplicações (CHANDRA; GRIESEMER; REDSTONE, 2007).

Projetar mecanismos autônomos capazes de realizar reconfiguração, sintonizados para reagir rapidamente às mudanças do sistema e gerar novas réplicas com base no uso de recursos e medidas de desempenho, é um assunto atual e relevante para pesquisa. Por

outro lado, estamos preocupados com o impacto inerente de implantar uma nova réplica em um aglomerado em tempo de execução. Se perceptível tal impacto dificulta a viabilidade das técnicas de autogestão, porque adicionar uma nova réplica pode comprometer o desempenho de um sistema sobrecarregado (VILAÇA; PEREIRA; OLIVEIRA, 2009).

Segundo o estudo de Vilaça, Pereira e Oliveira (2009), a política de reconfiguração deve levar em consideração a velocidade das mudanças de estado da aplicação no momento que se deseja reconfigurá-la. Isso ocorre pois quando a eficiência da transferência é menor que a velocidade de mudanças no estado, a nova réplica nunca receberá o estado mais atual da aplicação podendo não participar das rodadas atuais de consenso.

Então temos dois problemas em mãos: (1) alterar o numero de réplicas sem violar o progresso do algoritmo quando remover réplicas e a correção do algoritmo quando adicionar réplicas. Dessa forma, prover uma capacidade elástica preservando a consistência de Paxos; (2) provisionar réplicas de forma que não comprometa o desempenho, aumentando o suporte aos picos de acesso, denominados *flash crowds*, existentes na Internet (TANENBAUM; STEEN, 2007).

1.7 Trabalhos Relacionados

A ideia de se combinar réplicas com diferentes graus de uso de memória persistente no modelo falha-e-recuperação assíncrono foi formalizada por Aguilera, Chen e Toueg (2000). Para esse trabalho supomos a resolução de consenso nesse modelo, utilizando o algoritmo Paxos com um número fixo de réplicas com memória persistente e um número variável de réplicas que não utilizam memória persistente. Estabelecemos a forte premissa que os quóruns responsáveis pelo consenso sejam sempre compostos por réplicas que usam memória persistente, como descreveremos na Seção 2.2.2.

Do ponto de vista de engenharia de sistemas, a nossa proposta se assemelha às arquiteturas de cache distribuídas. Um exemplo notável é o Memcached[2] que é um repositório de chave/valor. Esse sistema é usado para armazenar, de forma distribuída pelo aglomerado, dados que podem evitar que a aplicação faça consultas custosas a um banco de dados centralizado. Usando Memcached o projetista de aplicação deve modificar o seu programa para registrar as informações úteis para atender requisições de leitura no cache distribuído. De forma similar, a nossa proposta procura evitar que seja feito acesso a um recurso nobre, nesse caso as réplicas que possuem memória persistente. Porém, o uso de réplicas sem memória persistente é transparente ao programador de aplicação que não precisa se preocupar em particionar os seus dados entre aqueles que são armazenáveis no cache e aqueles que não são.

[2] <http://memcached.org/>

Um outro sistema que faz uso extensivo de cache é o gerenciador de bloqueios distribuídos Chubby (BURROWS, 2006) usado no Google. Nesse sistema o número de réplicas que fornecem o serviço é fixo e relativamente pequeno (5 réplicas), e os clientes acessam o serviço apenas através de uma réplica mestre. A chave para o desempenho desse sistema é o fato de que os clientes acessam o serviço usando uma biblioteca especial que constrói um cache local, usando *lease* regidos por tempo (LAMPSON, 1996) para garantir a consistência dos mesmos. Em comparação, a nossa proposta não exige um cliente especial, o que a torna mais indicada para aplicações em geral, especialmente no ambiente Web. Dessa forma, temos uma implementação de cache que também é transparente ao cliente que acessa a aplicação.

Quando pensamos em reconfiguração na biblioteca Treplica é natural pensar no trabalho de Lamport, Malkhi e Zhou (2010), uma vez que Treplica utiliza Paxos como algoritmo de consenso. No entanto existem outras abordagens de reconfiguração mais completas, mas com complexidade comparável. Dentre elas vale ressaltar o protocolo Raft (ONGARO; OUSTERHOUT, 2014), FreeStore (ALCHIERI et al., 2014) e o trabalho de (BIRMAN K.; RENESSE, 2010). Esse último explora o modelo de reconfiguração dinâmica unificando duas abordagens: sincronia virtual, definido como um conjunto de protocolos para comunicação confiável em grupo e replicação de máquina de estados utilizando Paxos.

O protocolo Raft promete uma abordagem mais fácil de entender do que Paxos. Esse mecanismo já foi concebido com a ideia de reconfiguração utilizando dois passos: (1) o aglomerado muda para uma configuração de transição, chamada de *joint consensus*; e (2) uma vez que o *joint consensus* é alcançado, o sistema faz a transição para a nova configuração. Por outro lado, o protocolo FreeStore estabelece uma sequência de visões no sistema que são causadas por entradas e saídas de membros, semelhante a ideia proposta por Lamport, Malkhi e Zhou (2010).

2 Treplica Reconfigurável

Cobiçamos para Treplica um mecanismo de autogestão capaz de realizar autointegração de réplicas sem intervenção humana, inteligente o suficiente para detectar oscilação na demanda e reagir a elas, proporcionando assim maior eficiência na utilização de recursos físicos. Projetar mecanismos autônomos sintonizados para reagir rapidamente às mudanças no sistema é um assunto atual e relevante para pesquisa.

O primeiro passo para suportar tal anseio é dado nesse trabalho, que tem como objetivo transformar Treplica em uma biblioteca reconfigurável. O problema de reconfiguração é complexo, principalmente na presença de falhas e assincronia. Sua resolução é obtida basicamente a partir de duas maneiras: (1) abordagem baseada em transições de visões do conjunto de réplicas participantes (e corretas) (BIRMAN; JOSEPH, 1987a; BIRMAN; JOSEPH, 1987b); (2) definição, via consenso, de uma nova configuração a partir da construção de uma barreira que, quando alcançada pelas réplicas, faz com que elas abandonem a configuração vigente e ingressem na nova configuração definida (caso elas façam parte dela) (LAMPORT; MALKHI; ZHOU, 2010).

O processo de reconfiguração descrito por Lamport, Malkhi e Zhou (2010) pode ser bem complexo, então nessa dissertação estaremos interessados em um sub-conjunto desse problema onde apenas réplicas que não possuem estado inicial serão adicionadas. Isso é mais simples porque o estado inicial dessas réplicas é sempre vazio, consequentemente não precisamos de uma política para tratar réplicas com estado. Em outras palavras, a política para tratar recuperação de falhas em Treplica não foi alterada.

Para implementar esse mecanismo, identificamos a necessidade de um mecanismo eficiente capaz de transferir estado entre réplicas. Treplica ainda não possui tal mecanismo capaz de adicionar novas réplicas e recuperação de falhas. Estamos preocupados com o impacto inerente para implantar uma nova réplica em um aglomerado em tempo de execução. Tal impacto dificulta a adoção das técnicas de autogestão, porque dependendo do cenário, adicionar uma nova réplica pode comprometer o desempenho de um sistema sobrecarregado (VILAÇA; PEREIRA; OLIVEIRA, 2009).

Neste capítulo apresentaremos as alterações propostas em Treplica, começando pelos principais componentes da biblioteca que serão de fundamental importância para compreensão das alterações propostas na biblioteca. Iniciaremos com a Seção 2.1 apresentando os componentes de suporte fundamentais para arquitetura de Trepica e examinamos detalhadamente os componentes utilizados para construção do algoritmo Paxos. Em seguida, na Seção 2.2 apresentamos as alterações propostos para Treplica dividas em: (1) Protocolo para transferência de estado; e (2) Paxos com réplicas leitoras.

2.1 Visão arquitetural de Treplica

A arquitetura de Treplica é baseada em uma *fila persistente assíncrona* (Figura 9), sua implementação segue muito de perto a decomposição modular de Paxos apresentada por Lamport (2006). A fila é composta por classes internas da biblioteca que representam a funcionalidade dos quatro agentes de Paxos:

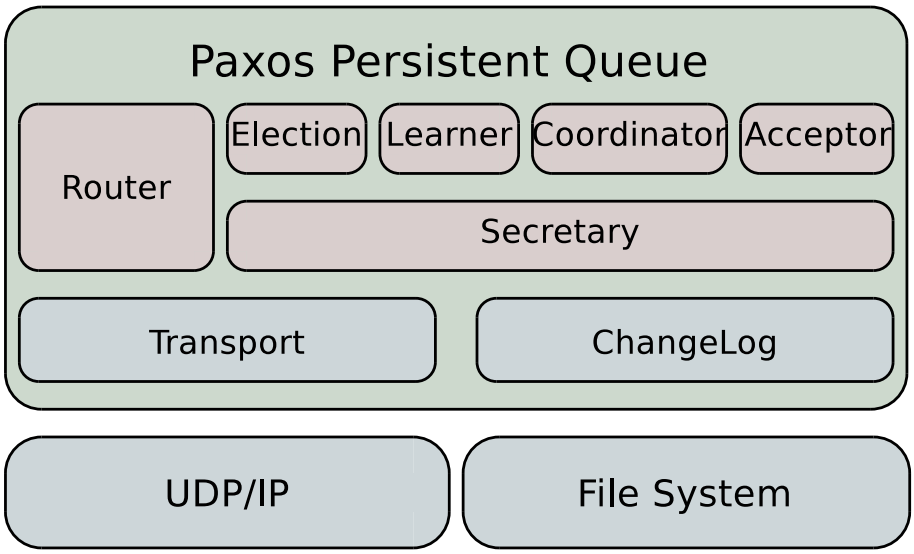


Figura 9: Paxos Persistent Queue

- `Learner`: combina os agentes proponentes e aprendiz em uma única classe responsável pelo monitoramento do fluxo de instâncias de Paxos, convertendo mensagens em objetos para serem entregues para a fila;
- `Acceptor`: atua como um receptor;
- `Coordinator`: atua como um coordenador;
- `Election`: algoritmo para eleição do líder, usado para selecionar um único coordenador.

As classes que implementam esses agentes foram concebidas para serem independentes, tornando possível a criação de réplicas que executam Paxos com diferentes subconjuntos de agentes. Especificamente, uma réplica contendo apenas o agente `Learner` poderá propor e aprender valores, efetivamente executando uma fila completa, sem participar do processo de consenso (votação). Usamos um réplica configurada dessa forma como base da nossa proposta de réplicas sem memória persistente.

As principais classes de suporte possuem estruturas semelhantes aquelas dos agentes de Paxos: são módulos encapsuladores de comportamento estritamente reativos. Elas operam através da transformação de mensagens endereçadas a elas pela classe `Router`

e também podem enviar novas mensagens à rede, armazenar informação na memória não-volátil ou entregar objetos para a aplicação. Essas tarefas são realizadas pela classe `Secretary`, que oferece uma interface uniforme a todas as tarefas de E/S requeridas pelos agentes. A abstração fornecida pelo `Secretary` oferece aos agentes uma forma de enviar mensagens e acessar a memória estável. Essa classe baseia seus serviços nas classes `Transport` e `ChangeLog` para acessar, respectivamente, a rede e o armazenamento estável. Por sua vez, essas duas classes fornecem abstrações que protegem os demais módulos dos detalhes de implementação. O componente de transporte é implementado com base em *multicast* sobre redes UDP/IP e o `ChangeLog` é baseado em sistema de arquivos simples.

Nas seções a seguir descreveremos esses módulos com mais detalhes. Eles são apresentados de baixo para cima, começando com os módulos de suporte e, em seguida, descrevendo os agentes de Paxos. Para cada módulo vamos mostrar a sua principal função, como ele interage com outros módulos e as implicações de sua estrutura para a execução Paxos. O principal foco é apresentar os componentes alterados na biblioteca que propusemos nessa dissertação. A descrição completa desses componentes encontra-se em Vieira e Buzato (2010).

2.1.1 Módulos de suporte

Em Treplica, os módulos de suporte são uma abstração dos mecanismos subjacentes a Paxos, claramente definidos por interfaces que oferecem a flexibilidade necessária para substituir o comportamento de um componente. A principal motivação para essa segregação foi simplificar a API utilizada pelos agentes de Paxos. Dessa forma esses componentes encapsulam os detalhes inerentes sobre endereçamento de réplicas, envio de mensagens *multicast* e *unicast*, gerenciamento de memória não volátil (disco) e detecção de falhas.

2.1.1.1 Transport

A abstração do transporte de dados é definida pela interface `Transport`. Ela oferece a seus clientes um mecanismo de envio e recebimento de mensagens *multicast* e *unicast*. Suas propriedades estão alinhadas com as premissas da rede para uma aplicação construída no modelo computacional assíncrono, ou seja, são enlaces de perda-justa (Seção 1.1):

Canais perda-justa podem perder, duplicar ou trocar a ordem de mensagens. Mesmo assim, essa abstração é adequada para o `Transport` por duas razões: (1) correspondência com as exigências da rede imposta por muitos algoritmos de consenso no modelo assíncrono, incluindo Paxos; e (2) reflete de perto as garantias efetivamente prestadas pelo modelo de transporte de rede utilizado pelo componente: UDP/IP.

Paxos não exige um modelo de transporte com propriedades de entrega de men-

sagens confiável, pois essa propriedade está intrinsecamente implementada pelo próprio algoritmo, incluindo mecanismos de buferização de mensagens e retransmissão. Se utilizarmos um mecanismo de transporte confiável, duplicaremos as propriedades para garantia de confiabilidade. Além disso, a entrega confiável fornecida por um transporte como TCP/IP só funciona para o modelo de falhas falha-e-para[1] (ABDELLATIF; LACHAIZE, 2004). No modelo falha-e-recuperação, o algoritmo de consenso ainda precisa verificar se as mensagens foram entregues mesmo quando se usa TCP/IP.

2.1.1.2 ChangeLog

A abstração criada na classe `ChangeLog` protege os agentes de Paxos dos detalhes pertinentes ao armazenamento não-volátil (disco). Basicamente, o serviço fornecido é um *log* persistente de todas as mudanças ocorridas em um objeto, com suporte para *checkpointing*. Na verdade, a interface desse componente é muito simples oferecendo métodos para escrita de alterações (acréscimo no *log*), com o apoio explícito para a recuperação. As alterações nos objetos podem ser acrescentadas ao final do arquivo de *log* de forma persistente e um objeto pode ser mais tarde reconstruído a partir da repetição dessas mudanças armazenadas. Com relação ao mecanismo de *checkpointing*, vale ressaltar que ele é utilizado para melhorar o desempenho da reconstrução de objetos a partir do *log*. Ele armazena, de forma intercalada, cópias completas do objetos, possibilitando a reconstrução de objetos em um único passo, dispensando a necessidade da aplicação de todas as alterações para chegar no último estado do objeto.

2.1.1.3 Ledger

A classe `Ledger` é a abstração do estado persistente para implementação de Paxos. É uma estrutura de dados comum, compartilhada por todos os agentes de Paxos implementados por Treplica, que conseguem através de uma interface acessar os dados de qualquer instância de consenso. A implementação dessa interface suporta persistência dos dados de forma não-volátil.

A classe `LoggingLedger` é utilizada para persistir em *log* (disco) as alterações, quando utilizada em conjunto com `ChangeLog`. Para simplificar o uso de *log* de alterações, essa implementação tem suporte para detectar e isolar as alterações feitas em seu estado interno. A gravação de mudanças de estado e o acesso aos dados persistidos são funcionalidades da classe `ChangeLog`, enquanto `LoggingLedger` registra e replica as mudanças. O `Ledger` armazena o estado completo de cada instância do consenso por réplica, mantendo todos os dados exigidos por todos os tipos de agentes de Paxos implementados por

[1] No modelo falha-e-para, um processo não tem capacidade de se recuperar de uma falha, ou seja, a partir do momento que o processo falha ele permanecerá falho até o infinito (CACHIN; GUERRAOUI; RODRIGUES, 2011).

Treplica (Seção 1.4.3). Dessa forma, é possível que qualquer agente recupere seu estado, inclusive o coordenador.

2.1.1.4 Secretary

A classe `Secretary` apresenta uma abstração unificada de E/S para os agentes de Paxos. Esse componente utiliza memória persistente usando os componentes `ChangeLog` e `Ledger`, lida com a passagem de mensagens usando o módulo de `Transport` e lida também com a fila de objetos utilizada para entregar objetos para a aplicação. A principal razão para a criação dessa abstração em Treplica foi sintetizar as operações de E/S em *threads* diferentes das que executam as operações de Paxos. Operações de E/S em disco tem grande potencial para reduzir o desempenho do algoritmo Paxos por duas razões: (1) todas as requisições de escrita que estabeleceram consenso devem ser persistidas de forma não-volátil antes do progresso do algoritmo; (2) alguns passos do algoritmo Paxos podem demandar vários acessos à memória persistente.

Vieira e Buzato (2010) mostram que a E/S é tratada apenas pelo `Secretary` de forma assíncrona, possibilitando a ocorrência de paralelismo entre instâncias diferentes, mesmo com apenas acessos sequenciais ao disco. Isso é feito através de uma fila que agrupa gravações lógicas distintas e retém os dados para realizar uma única gravação física quando o disco estiver livre. Essa abordagem é vantajosa porque o tamanho dos dados de escrita no disco afeta muito pouco a latência da chamada de sistema *sync()* usada para tornar a escrita estável. Dessa forma, a implementação de `Secretary` absorve a latência da *system call* mantendo uma *thread* separada para persistência dos dados.

2.1.1.5 PersistentQueue

Filas persistentes assíncronas são uma maneira para que um grupo de réplicas compartilhem informações na forma de objetos. Esses objetos são enviados por qualquer réplica ligado à fila e são transmitidos para as outras réplicas de forma totalmente ordenada e com entrega garantida, independentemente de falhas. Esse comportamento pode ser mais precisamente descrito pelas seguintes propriedades:

- Objetos são entregues na mesma ordem para todas as réplicas.
- Objetos são entregues para todas as réplicas, mesmo que uma réplica falhe e mais tarde se recupere.
- Objetos são persistentes e sobrevivem a falhas em todas as réplicas.

Em Treplica, esse componente é definido como uma interface genérica que pode ser implementada utilizando outras abordagens diferentes de consenso, desde que respeite

as propriedades definidas acima. Apesar da viabilidade para suportar diversas implementações, Treplica utiliza somente uma implementação baseada em consenso para o componente de fila persistente, definida pela classe `PaxosPersistentQueue`.

2.1.1.6 Router

A classe `Router` é um componente simples, mas vital para o funcionamento da classe `PaxosPersistentQueue`, porque inicia todo o conjunto de agentes necessário para o funcionamento de Paxos. Sua função principal é prover o *main loop* da implementação de Paxos, que recebe mensagens do componente de transporte e, de acordo com o tipo da mensagem, encaminha para processamento no agente apropriado. Dessa forma, a execução desse agente é sequencial e não precisa de controle de concorrência.

Esse é o único componente (*thread*) que monitora o temporizador central e gera eventos de *timer*[2]. O código de processamento dos agentes não possui operações demoradas ou bloqueantes, eles são programados como simples manipuladores de eventos caracterizando uma arquitetura de processamento assíncrono baseada em eventos (*event-driven*). É responsabilidade do `Router` instanciar agentes e componentes de apoio e, também, inicializar a `PaxosPersistentQueue`.

2.1.2 Agentes de Paxos

Em Treplica, os agentes de Paxos efetivamente implementam o algoritmo Paxos. Esses agentes são baseados na especificação do algoritmo e são responsáveis pelo seu correto funcionamento. Os agentes descritos aqui utilizam os componentes de suporte descritos na seção anterior.

2.1.2.1 Election

Esse agente é responsável pela eleição do líder requerido pelo protocolo Paxos para garantir progresso no algoritmo. Ele expõe a seus clientes a interface de uma eleição de líder Ω. Resumidamente, Ω garante que todo agente de eleição confie em uma réplica do sistema como correta e que existe um tempo futuro em que todos os agentes de eleição confiarão na mesma réplica (CHANDRA; TOUEG, 1996). Se a réplica que todos os agentes acreditam estar correta executar o agente coordenador, a propriedade de progresso é garantida.

Para garantia de progresso, Paxos deve ter um único agente coordenador em execução. O agente de eleição não exige que os clientes consultem seu serviço para perceber mudanças de liderança. Especificamente, ele é capaz de detectar quando a réplica é eleita como líder e inicia a execução do agente coordenador em resposta a esse evento. Por outro lado, quando ele detecta que a réplica deixou a liderança o agente coordenador é parado.

[2] Os eventos de *timer* simbolizam a passagem do tempo para a aplicação. Esse evento atinge todos os componentes que necessitam de um relógio para seu correto funcionamento.

2.1.2.2 Learner e Proposer

Em Treplica, a classe `Learner` implementa as funcionalidades de aprendiz e proponente de Paxos. Ela é responsável por: (1) tratamento das requisições oriundas dos clientes da fila persistente; (2) criação de propostas que encapsulam essas requisições; e (3) acompanhamento das propostas até que elas sejam ordenadas e entregues.

Para entender por que há uma combinação entre a funcionalidade desses dois agentes no mesmo módulo, basta observar as atividades realizadas por esse agente. É possível classificar as duas primeiras tarefas como pertencentes ao agente proponente e só a última tarefa como atividades do aprendiz. No entanto, a terceira tarefa é fundamental para o funcionamento correto da implementação de Treplica utilizando a fila persistente e exige conhecimento detido por ambos proponente e aprendiz. Isso acontece porque Treplica suporta Paxos e *Fast Paxos* (LAMPORT, 2006) na mesma implementação e *Fast Paxos* retira do agente coordenador a responsabilidade exclusiva de propor valores para consenso.

Em *Fast Paxos* qualquer proponente pode propor valores e essas propostas ocorrem de forma descentralizada. Desta forma, na Fase 2a agentes proponentes diferentes podem propor diferentes propostas para uma mesma instância de consenso, provocando uma colisão. Quando essa situação é detectada, o agente aprendiz deve estar ciente do ocorrido para propor novamente a proposta em outra instância até conseguir decidi-la.

2.1.2.3 Coordinator

Coordenador é o agente responsável por conduzir a rodada de consenso, como descrito na Seção 1.4.1. Ele é o agente que envia a mensagem iniciando uma nova rodada r na Fase 1a. Após a formação de um quórum em r, o coordenador atua novamente na Fase 2a ordenando a votação de uma determinada proposta. Na Fase 2b, o coordenador computa os votos e estabelece o consenso de r.

É permitido a existência de apenas um coordenador por instância. Em caso de falha na réplica que executa o agente coordenador uma eleição de líder deve ser convocada para estabelecer que uma réplica correta execute o agente coordenador. Como o algoritmo é executado no modelo computacional falha-e-recuperação, uma réplica coordenadora defeituosa pode voltar à computação acreditando que ainda é coordenadora. Nesse caso, uma nova eleição de líder deve ser convocada para restabelecer a unicidade de coordenador por instância.

2.1.2.4 Acceptor

O receptor é um agente responsável pela votação das propostas de consenso do algoritmo Paxos. Esse agente reflete muito de perto o comportamento de Paxos descrito

no Seção 1.4.1. O agente receptor aguarda pela mensagem da Fase 1a que inicia uma nova rodada de consenso e responde, se for o caso, com o valor que ele votou em alguma rodada anterior. Isso permite que a rodada progrida e habilita o receptor a votar na rodada corrente, assim que receber a mensagem adequada na Fase 2a. Em Treplica esse comportamento tem apenas duas pequenas modificações que aumentam o desempenho do sistema (VIEIRA; BUZATO, 2010): (1) o receptor reduz o voto para uma pequena mensagem de tamanho constante e (2) ele avisa ativamente o coordenador sobre instâncias de consenso decididas.

2.1.2.5 PaxosPersistentQueue

`PaxosPersistentQueue` é responsável por agrupar todos os agentes de Paxos. Essa classe possui o *main loop* responsável pelo recebimento das mensagens trocadas pela biblioteca, sejam elas mensagens de Paxos ou mensagens de configuração. A classe `PaxosPersistentQueue` foi criada para fornecer uma implementação para fila persistente, base da arquitetura de Treplica, utilizada como canal ordenado para troca de informações.

2.2 Alterações propostas

Apresentamos os conceitos utilizados para criação da biblioteca Treplica, os detalhes de seus componentes e como eles estão relacionados. A partir de agora, iremos focar nossa discussão nas alterações propostas para expansão da biblioteca. O objetivo dessa seção é a apresentar e detalhar as seguintes funcionalidades:

- Protocolo para transferência de estado: mecanismo eficiente para transferência de estado entre réplicas.
- Réplicas leitoras: a ideia principal dessa abordagem é reconfigurar o sistema de forma mais simples, utilizando réplicas réplicas que não participam de processo de decisão de instâncias de consenso.

2.2.1 Protocolo para transferência de estado

A ideia principal dessa abordagem é a criação de um mecanismo que possibilite transferir, de forma eficiente, o estado entre réplicas. Para isso, criamos um protocolo que orquestra as interações entre réplicas e os bloqueios de estado necessários para garantia de consistência. Visando maior clareza de exposição, quando necessário, chamaremos as réplicas que recebem o estado de *réplicas receptoras* e réplicas que transferem seu estado de *réplicas doadoras*.

A necessidade de um mecanismo para transferência de estado surgiu a partir da suposição da equalização de estados divergentes entre réplicas da mesma aplicação. A

disparidade de estados em uma ambiente que emprega replicação ativa utilizando Paxos pode suceder a partir de: (1) falha-e-recuperação de uma réplica ou falha transiente do enlace de comunicação. Durante o período defeituoso, uma réplica pode perder n decisões de consenso criando uma grande lacuna entre seu estado local e o estado corrente da aplicação; ou (2) a divergência de estados pode ter um motivo mais nobre: expansão do aglomerado.

Independente da motivação para aplicação de uma transferência de estado, essa operação não deve gerar grande impacto para o processamento de requisições e deve preservar a correção do algoritmo. Tendo em vista a garantia de consistência, implementamos essa operação como uma tarefa *síncrona*[3]. As seguintes premissas foram supostas para construção do protocolo:

- Quando uma réplica receptora inicia o processo de transferência, todas as mensagens recebidas não pertencentes ao protocolo de transferência devem ser ignoradas.
- A réplica doadora não deve processar nenhuma operação de escrita enquanto realiza a transferência de estado.
- Novas réplicas estarão aptas para processar requisições (leitura ou escrita) somente após a configuração de um estado inicial.

Baseado nessas premissas, podemos afirmar que a operação de transferência de estado é custosa para o desempenho de Paxos, pois estamos bloqueando temporariamente a participação de uma réplica no processo de decisão de instâncias de consenso.

2.2.1.1 Funcionamento do protocolo

Estabelecemos a premissa de que é responsabilidade da réplica receptora encontrar uma réplica doadora (réplica disposta a transferir o seu estado). A simplicidade do mecanismo de seleção de doador é herdada de Treplica: as réplicas conhecem somente seu próprio identificador de rede e podem alcançar todas as outras réplicas por uma primitiva simples de difusão. Criamos o protocolo com a preocupação de minimizar a degradação de desempenho causada pela operação de transferência de estado. Ele foi dividido em três fases, conforme ilustra a Figura 10.

- Na Fase 1 (*Fase de Negociação*) a réplica receptora faz a solicitação de transferência de estado, obedecendo a uma *política contratual*.

[3] No modelo requisição/resposta, o emissor dos dados fica bloqueado até receber uma resposta do receptor (COULOURIS; DOLLIMORE; KINDBERG, 2011)

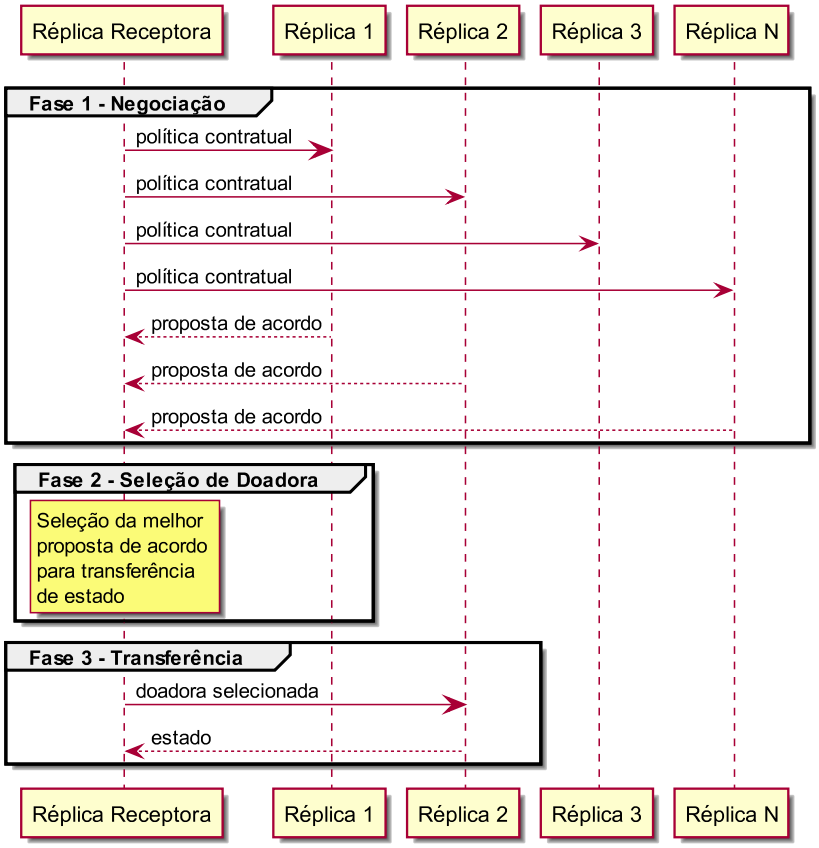


Figura 10: Fases do protocolo de transferência de estado

- Na Fase 2 (*Seleção de Doadora*) ocorre a apuração do melhor acordo proposto pelas réplicas que atendem às exigências estabelecidas pela réplica receptora. Somente uma réplica doadora é selecionada.

- Finalmente, na Fase 3 (*Transferência*) o estado da réplica doadora eleita na Fase 2 é transferido para a réplica receptora.

Na Fase de Negociação a réplica receptora estabelece as regras da transferência de estado através da mensagem `PolicyMessage`. Por exemplo, supomos que uma réplica r deseje receber um estado a partir da instância de consenso i. Então, r inicia a negociação de estado difundindo uma mensagem com essa política contratual, o predicado: a maior instância de consenso decidida é maior que i. As réplicas que recebem essa mensagem de solicitação de negociação de estado são solidárias e tentam atender essa requisição. Elas avaliam as exigências contidas na mensagem e, caso estejam de acordo, enviam somente para a réplica receptora (remetente do contrato) a mensagem `DealMessage`. Essa proposta de acordo, contém informações referentes ao estado que a réplica doadora está oferecendo. Caso a réplica receptora não receba nenhuma proposta de acordo após um tempo pré-estabelecido, ela reinicia a Fase de Negociação até encontrar uma réplica doadora. Podemos nos beneficiar dessa etapa inicial do protocolo para criarmos diferentes

políticas contratuais para transferência de estado. Nessa versão proposta de Treplica, supomos configurações de duas politicas: uma mais agressiva e outra mais ingênua. Lembrando que todas as políticas devem incluir qual é a instância de consenso pretendida.

- Somente réplicas leitoras: essa política é mais restritiva, busca um acordo com uma réplica leitora (Seção 2.2.2). Acordos com réplica leitoras são preferíveis visando minimizar possíveis impactos na computação de Paxos.
- Qualquer réplica: essa política é a menos restritiva possível, busca um acordo independente da configuração da réplica.

Devemos ter cuidado ao eleger qualquer réplica como réplica doadora, pois podemos gerar impacto direto no desempenho da aplicação. A réplica doadora não participará da decisão de um nova requisição durante o período que está transmitindo seu estado para garantir a consistência. Lembrando que, no algoritmo Paxos, a partir do momento em que a maioria dos receptores concordam com a alteração do estado, mais cedo ou mais tarde todas as réplicas chegarão ao mesmo estado. A partir do momento que retiramos temporariamente da computação uma réplica votante, a probabilidade de atingir consenso pela maioria diminui, podendo até impossibilitar o progresso do algoritmo.

O protocolo progride quando a réplica receptora possui propostas de acordo. Quando essa condição é alcançada, ela inicia a Fase de Seleção de Doadora, que executa um algoritmo simples capaz de eleger a réplica que propôs o melhor acordo. Para fins experimentais nós adotamos a política simples de selecionar a primeira réplica doadora que responder. Porém, a seleção pode ser mais sofisticada de acordo com as políticas contratuais estabelecidas. Somente a partir do momento que o algoritmo estabelece uma réplica doadora a Fase de Transferência inicia.

Para a execução da Fase de Transferência, os seguintes aspectos de Treplica foram considerados: (1) o estado de uma aplicação pode ser tão grande quanto a capacidade de memória de uma réplica; (2) todas as mensagens são trocadas utilizando o protocolo UDP. Optamos então pela utilização do protocolo TCP para transferência de estado cobiçando maior vazão dos dados nessa operação (ABDELLATIF; LACHAIZE, 2004). Somente o estado é enviado via TCP, todas as outras mensagens pertencentes ao protocolo utilizam comunicação UDP, nativa de Treplica. Sendo assim, a réplica receptora abre um *socket* TCP e envia uma mensagem UDP `GETMessage` para a réplica doadora com o endereço do *socket* TCP recentemente aberto. Por sua vez, a réplica doadora bloqueia suas atividades e estabelece a conexão TCP com a réplica leitora. Finalmente a transferência de estado é executada.

Assim que a operação é concluída, a conexão TCP entre as réplicas é finalizada. A réplica doadora volta para computação e a réplica receptora começa a processar as requi-

sições encaminhadas pelos seus clientes. Estabelecemos um *timeout* para evitar bloqueios indevidos por falha em alguma das réplicas envolvidas na transação. Caso todas as etapas do protocolo não sejam concluídas em um tempo máximo pré-estabelecido, a negociação de estado é reiniciada até que se obtenha êxito.

2.2.1.2 Implementação

A Figura 11 ilustra o funcionamento do protocolo e suas respectivas mensagens: `PolicyMessage`, `DealMessage`, `GetMessage` e `StateMessage`. Todas essas mensagens são representadas por classes Java e implementam a interface `StateTransferMessage`, possibilitando assim distinguir as mensagens do protocolo de transferência de estado das mensagens de Paxos. Todas as mensagens que implementam `StateTransferMessage` são roteadas no *main loop* de Treplica para a nova classe `Diplomat`, independente da configuração existente na réplica. No restante dessa seção descreveremos a implementação do protocolo de transferência de estado pelo detalhamento desses componentes.

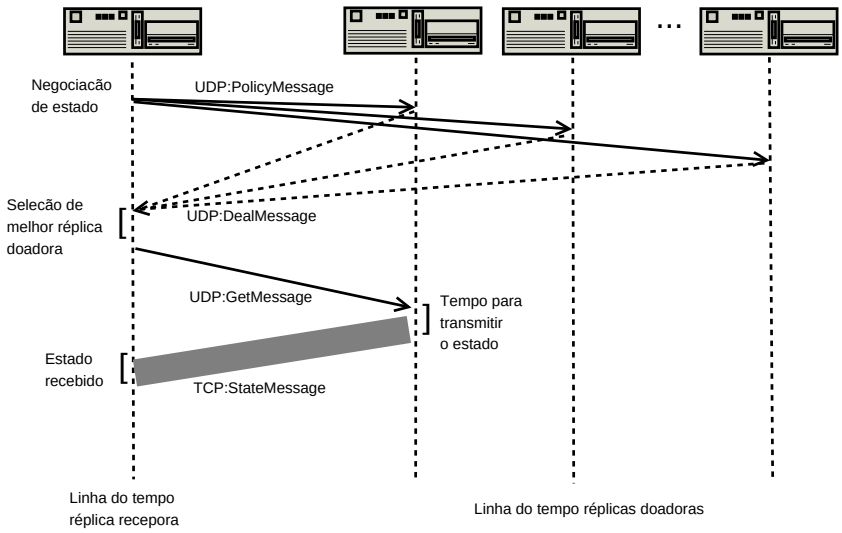


Figura 11: Protocolo de transferência de estado

2.2.1.2.1 PolicyMessage

A classe `PolicyMessage` é a primeira mensagem do protocolo para transferência de estado. Essa classe além de sinalizar para as outras réplicas do grupo que existe uma réplica interessada em receber um estado, contém todas as informações sobre o estado pretendido pela réplica receptora.

Os dados contratuais são abstraídos pela classe `Policy`. Essa classe por sua vez contém o número da instância de consenso desejado (informação mínima para proposta de um acordo) e uma lista de regras qualificadoras, que uma réplica deve atender para ser uma doadora. O número da instância de consenso serve como identificador único de

um estado por representar a sequência de operações que foram efetivamente aplicadas, ou seja, mudaram o estado.

Uma regra qualificadora deve implementar a interface `Rule`. A implementação pode ser customizada de acordo com as necessidades da aplicação. Ela deve deve avaliar se o estado da réplica candidata a doadora está de acordo com os requisitos estabelecidos pela réplica receptora. Uma réplica pode se tornar doadora se atender todos os requisitos do contrato.

2.2.1.2.2 DealMessage

Essa classe abstrai uma proposta de acordo. Ela é uma sinalização para a réplica receptora da existência de uma réplica disposta a doar seu estado, ou seja, a réplica doadora está em conformidade com a política estabelecida pela réplica receptora. Quanto menos restritivas são as políticas que definem a seleção de um estado mais mensagens de acordo serão enviadas para a réplica receptora, potencializando a seleção de uma doadora.

2.2.1.2.3 GetMessage

A classe `GetMessage` é a abstração da mensagem de pedido de estado enviada somente para a réplica doadora. As informações do endereço para envio do estado (através de *socket* TCP) são extraídas dessa mensagem.

2.2.1.2.4 StateMessage

A classe `StateMessage` é a abstração do estado enviado da réplica doadora para a réplica receptora. Essa classe possui como atributos o número da instância de consenso, indicando a última operação que causou alteração no estado, e um estado que implementa `Serializable`. Definimos essa classe como uma cópia instantânea, tirada durante o período que a réplica estava bloqueada para alterações.

A partir do conteúdo armazenado por `StateMessage`, qualquer réplica que possui um estado identificado por um número de instância menor[4] pode se beneficiar da substituição de seu estado local corrente (desatualizado) por um mais atual. Em outras palavras, a réplica receptora executa uma operação de *avanço de estado* para o estado i.

O avanço de estado implica na execução instantânea de uma série de operações que, do ponto de vista da aplicação, não causam problemas de acordo com as regras de Treplica. Do ponto de vista de Paxos, o resultado das instâncias menores que i ficam potencialmente indefinidas, mas a réplica receptora não procura ativamente decidi-las. Apenas participa delas se for requisitada, de forma a sempre garantir o quórum.

[4] Podemos supor comparações de estados através da última instância de consenso, já que o mesmo é um limiar crescente (VIEIRA; BUZATO, 2010).

2.2.1.2.5 Diplomat

A classe `Diplomat` é responsável por implementar o tratamento das mensagens do protocolo de transferência. Toda réplica possui uma instância da classe `Diplomat` para representar seus interesses frente outras réplicas. A abstração desse componente foi inspirada nas funções de diplomacia (ciência e arte referentes às relações entre Estados (FERREIRA, 1999)) exercidas por um Diplomata no modelo atual de relação entre nações[5].

O diplomata adéqua suas funções de trabalho conforme o estado corrente da réplica. Em uma réplica receptora, estão habilitadas as funções responsáveis por obter um novo estado: criação de política, seleção de réplica doadora e abertura de *socket* TCP. Por outro lado, em uma réplica doadora as funções para ceder o estado é que estão habilitadas: análise de aderência a política de transferência, bloqueio de alterações de estado e o envio de estado.

Para seleção da melhor proposta de transferência de estado o diplomata armazena em memória todas as propostas recebidas e após um tempo pré-estabelecido a seleção é realizada. Consideramos como melhor proposta de estado aquela que possui maior instância de consenso. Em caso de eventuais erros e/ou *timeouts* as propostas são removidas da memória para evitar conflitos com as futuras propostas de acordo, que serão recebidas devido a atuação do mecanismo de reinicialização.

2.2.1.3 Política de Reconfiguração

Identificamos dois potenciais pontos que se beneficiariam da utilização do protocolo de transferência de estado: (1) expansão do aglomerado e (2) recuperação de erros. Para a construção desse trabalho, focamos na aplicação do protocolo na expansão do aglomerado, conforme ilustra a Figura 12. Supomos alguns aspectos para serem considerados:

- O estado da nova réplica que deseje se juntar ao aglomerado está defasado com relação ao estado das réplicas que já participam do grupo. É imprescindível que essa defasagem seja suprimida.
- A operação para igualar o estado da nova réplica com o estado compartilhado pelo grupo deve ser consistente.
- O impacto gerado para o processamento do aglomerado deve ser mínimo.

Investimos nosso esforço na construção de um mecanismo de reconfiguração leve, consequentemente evitamos reconfigurações de réplicas completas e focamos na criação e

[5] Segundo Ferreira (1999), diplomata é um funcionário que representa um governo junto de outro governo.

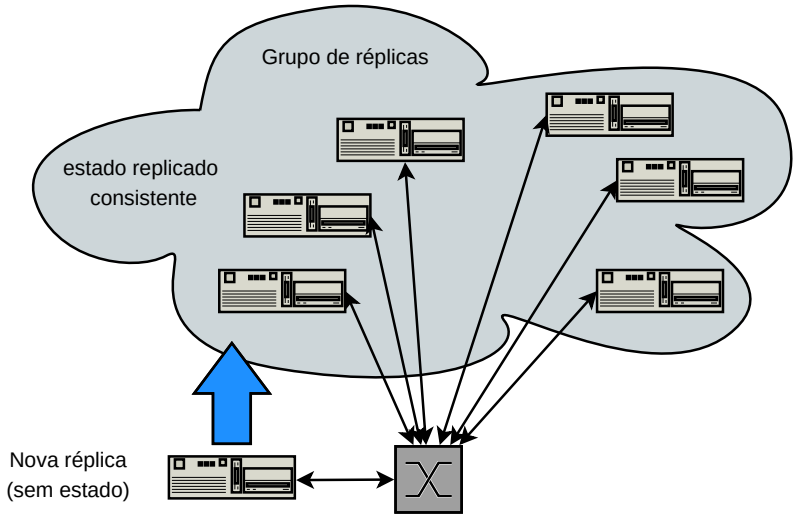


Figura 12: Inclusão de réplica

remoção de réplicas parciais, que chamamos de réplicas leitoras e que serão descritas na Seção 2.2.2.

A partir da suposição de um mecanismo geral para "preenchimento de lacunas" no estado utilizando o protocolo de transferência, as réplicas leitoras são um caso particular que pode ser atendido por tal mecanismo, de forma que elas apenas usam transferência de estado caso a lacuna no seu estado seja muito grande. O custo da política de reconfiguração pode ser derivado do custo da aplicação do protocolo de transferência que está diretamente ligado ao tamanho da lacuna que se deseja preencher.

O trabalho de especificação de parâmetros para essa política ainda está em seu estágio inicial, dependendo de estudos mais aprofundados para caracterizar os custos envolvidos, como descrevemos brevemente na conclusão desse trabalho.

2.2.2 Paxos com Réplicas Leitoras

A ideia principal da abordagem proposta é utilizar o protocolo de transferência de estado para instanciar réplicas que não participem do processo de decisão de instâncias de consenso. A motivação por trás da criação de *réplicas leitoras* é permitir que o sistema reaja de forma autônoma a picos de carga sem comprometer o desempenho do mesmo. No entanto, sem uma política cuidadosa de reconfiguração corre-se o risco de gastar muitos dos recursos do sistema no próprio processo de reconfiguração, anulando quaisquer ganhos advindos do acréscimo de novas réplicas leitoras.

A adoção de réplicas leitoras é uma subclasse do problema de reconfiguração de Paxos, no entanto é uma forma mais simples de ser implementada pois não altera o número estático de réplicas com o poder de alterar o estado. Definimos réplicas leitoras como réplicas onde apenas parte dos agentes do algoritmo Paxos estão executando. Para

maior clareza de exposição, quando necessário, chamaremos as réplicas contendo todos os agentes ativos de *réplicas votantes*

2.2.2.1 Réplicas Leitoras

Réplicas leitoras são réplicas onde apenas os agentes proponente e aprendiz estão executando. Dessa forma, do ponto de vista do conjunto de processos que implementam o algoritmo Paxos, uma réplica leitora é capaz apenas de propor operações a serem aplicadas no estado replicado e de aprender operações decididas pelo conjunto de receptores. Do ponto de vista do cliente da aplicação replicada um réplica leitora se comporta como uma réplica votante: ela atende requisições de qualquer tipo garantindo a execução atômica das mesmas.

As réplicas leitoras não assumem um papel fundamental na execução do algoritmo Paxos, no entanto elas se integram de forma consistente com a operação das réplicas votantes por meio de suas funções fundamentais: propor e aprender requisições de escrita. As réplicas leitoras propõem novas requisições a serem executadas em nome de seus clientes através de seu agente proponente. O proponente encaminha a operação ao coordenador que por sua vez decide, em conjunto com os receptores, a decisão da alteração do estado através de uma instância de Paxos, como descrito na Seção 1.4. Uma vez que a decisão é alcançada, a mesma é difundida para o resto do sistema. Nesse momento o agente aprendiz da réplica leitora toma conhecimento da decisão e atualiza o seu estado interno, sem a participação ativa do coordenador ou de qualquer receptor.

Tanto o processo de proposta quanto o de aprendizado executado por uma réplica leitora devem usar as mesmas estratégias de implementação das réplicas votantes. Na verdade, em nossa implementação usando Treplica, as réplicas leitoras foram construídas a partir da separação modular dos agentes que implementam Paxos. Dessa forma, reutilizamos os mesmos componentes e por consequência essas réplicas são capazes de detectar e reenviar propostas perdidas, detectar e corrigir lacunas na sequência de instâncias de consenso, fazer controle de fluxo e de congestionamento, entre outras operações fundamentais para uma operação eficiente de Paxos (VIEIRA; BUZATO, 2010).

Uma consequência importante do uso de réplicas leitoras é que essas réplicas, consistentemente com as funções que elas assumem no algoritmo Paxos, não precisam de memória persistente para sua operação. Isso se deve ao fato de que elas não executam as Fases 1 e 2 do algoritmo, descritas na Seção 1.4. Porém, pode ser interessante que essas réplicas registrem em memória persistente a proposta decidida de forma a não precisar realizar uma recuperação completa em caso de falha. Na nossa proposta de réplicas leitoras decidimos não fazer esse registro de forma a remover completamente a escrita em memória persistente do caminho crítico de execução. É interessante observar que a escrita eliminada ocorre somente quando a réplica leitora atualiza o seu estado de acordo com

as propostas decididas pelos receptores das réplicas votantes. Dessa forma, as réplicas leitoras conseguem manter seu estado atualizado com as réplicas votantes com um custo mínimo. Elas também são capazes de processar requisições de escrita com um custo similar àquele gerado pelas réplicas votantes ao executar as mesmas requisições. Podemos argumentar que esse custo é menor, na medida que as réplicas leitoras aliviam as réplicas votantes do custo de manter as conexões abertas com os clientes.

Utilizando réplicas leitoras, podemos formar grupos de réplicas com diferentes graus de uso de memória persistente (AGUILERA; CHEN; TOUEG, 2000). Uma configuração simples seria mesclar réplicas votantes e leitoras formando um conjunto híbrido de réplicas transparente para o cliente, conforme ilustra a Figura 13 (a). É concebível ainda uma configuração onde as réplicas votantes não entram em contato com os clientes, sendo essa operação completamente delegada às replicas leitoras, configuração ilustrada pela Figura 13 (b).

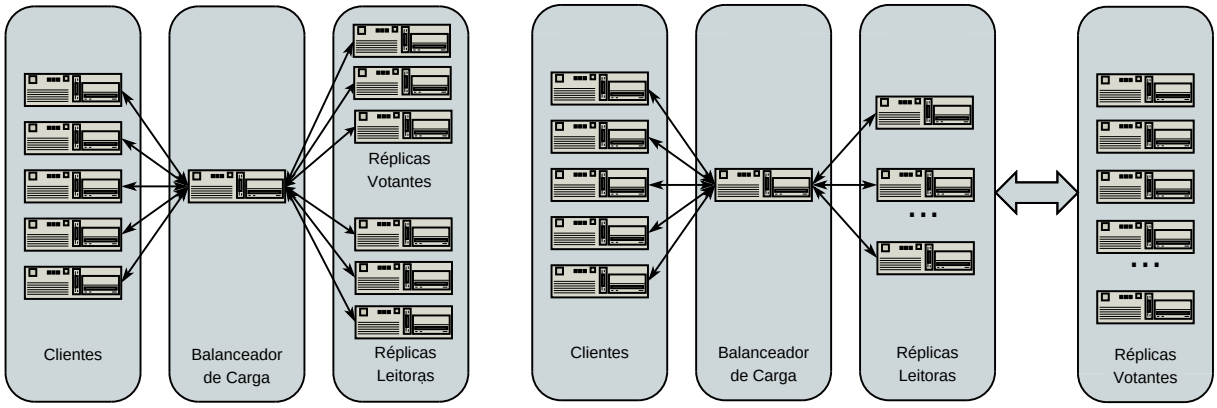


Figura 13: Configuração de Paxos com réplicas votantes e leitoras

As réplicas leitoras funcionam então como uma espécie de cache *write-through* distribuído. O estado replicado na memória destas réplicas permite atender diretamente as requisições de leitura dos clientes, enquanto as requisições de escrita são repassadas ao receptores. Podemos supor que a vazão de operações de leitura tem o potencial de crescer linearmente com o número de réplicas leitoras disponíveis.

2.2.2.2 Implementação

Originalmente, Treplica apresenta uma única configuração de réplicas que emprega todos os agentes de Paxos utilizando memória persistente, conforme descrito na Seção 2.1.2. Para implementar réplicas leitoras foi necessário alterar a forma como é feita o agrupamento de agentes.

A classe denominada `PaxosPersistentQueue` é responsável por implementar o agrupamento de todos os agentes de Paxos, caracterizando assim uma réplica votante. Uma réplica votante possui como uma das suas *threads* ativas o *main loop* da classe `PaxosPersistentQueue`, como descrito na Seção 2.1.2. Em contra partida, uma réplica leitora executa o *main loop* da classe `PaxosReadonlyQueue`. Essa classe, implementa somente a agregação dos agentes proponente e aprendiz, sem a utilização de memória persistente. Detalharemos nas próximas seções a implementação proposta para esse componente.

2.2.2.2.1 PaxosReadonlyQueue

A classe `PaxosReadonlyQueue` foi criada para fornecer o mesmo comportamento da classe `PaxosPersistentQueue`: disponibilizar uma fila assíncrona usando o algoritmo Paxos. Dessa forma, as operações suportadas por `PaxosReadonlyQueue` são as mesmas. Do ponto de vista do processamento de mensagens postadas na fila, as seguintes propriedades caracterizam uma réplica leitora:

- Abdicar liderança: todas as mensagens relacionadas à eleição de líder não são processadas, logo é eliminada qualquer possibilidade de uma réplica leitora se tornar coordenadora de uma rodada de Paxos.
- Inelegível ao voto: mensagens relacionadas a votação de uma proposta são ignoradas. Dessa forma, réplicas leitoras não participam do processo de decisão de consenso e não são essenciais para o progresso do algoritmo Paxos.
- Aprendizado: todas as mensagens endereçadas ao componente `Learner` são processadas pela fila. Consequentemente, os mecanismos para descobrir qual foi o consenso de uma determinada instância são habilitados.

O principal objetivo dessa classe é participar das operações que não exigem dados persistentes para garantir a correção do algoritmo, oferecendo instâncias capazes de atuar parcialmente nas fases de Paxos. Do ponto de vista do cliente da aplicação um réplica leitora se comporta como uma réplica votante: ela atende requisições de qualquer tipo garantindo a execução atômica das mesmas. Sendo assim, as seguintes premissas não podem ser violadas:

- Mensagens que alteram o estado (escrita): são resolvidas pelo aglomerado de réplicas orquestrado pelo protocolo Paxos.
- Mensagens que não alteram estado (leitura): são resolvidas localmente independente do tipo da réplica.

Do ponto de vista de uma réplica votante, não é possível distinguir se a proposta é oriunda de uma réplica votante ou leitora. Como Treplica utiliza mecanismo de difusão para enviar as mensagens de Paxos, uma nova réplica conectada na rede passa a receber todas as mensagens do protocolo. A réplica leitora está programada para ignorar as mensagens de formação de quórum e processar somente as mensagens roteadas para seus agentes ativos: proponente e aprendiz. Dessa forma, garantimos que as instâncias de consenso são lidas por todas as máquinas, garantindo a transparência da adoção de réplicas leitoras e podemos afirmar que o progresso e a correção de Paxos não são violados.

A principal razão para a criação dessa abstração em Treplica foi eliminar a operação de persistência em disco, gerando um componente volátil, capaz de oferecer as operações essenciais para o progresso do algoritmo sem o ônus da escrita em disco. Desse forma, foi necessário substituir a classe `Secretary` por um equivalente não persistente, a classe `WeakSecretary`.

2.2.2.2.2 WeakSecretary

A classe `WeakSecretary` apresenta uma abstração de E/S sem persistência de dados em disco. Esse componente é uma versão leve da classe `Secretary`, que é utilizada pelos agentes de Paxos para enviar mensagens pela rede, através do intermédio do componente `Transport`. Essa classe foi projetada para trabalhar com dados somente em memória, dessa forma todos os dados normalmente encaminhados a memória persistente são perdidos na presença de defeitos na réplica. No entanto, na ausência de falhas nos beneficiamos da eliminação de uma operação custosa relacionada com E/S em disco.

`WeakSecretary` também é responsável por lidar com o componente `Ledger` e com a fila de objetos utilizada para entregar mensagens para a camada da aplicação. Essa classe utiliza a implementação `TransientLedger` para fornecer informações sobre os decretos conhecidos. Essa implementação fornece uma abstração capaz de trabalhar com um estado transplantado e responde `null` para todas as solicitações que utilizam um número de decreto menor que o decreto ligado ao estado transplantado. Em outras palavras, do ponto de vista de uma réplica leitora, é tolerável não saber o que aconteceu em instâncias de consenso anteriores ao seu estado inicial.

O comportamento mais importante dessa classe é que eliminamos a fila de escalonamento de entrada e saída, sendo todas as operações concluídas sincronamente.

2.2.2.3 Provisionamento de Réplicas Leitoras

É possível utilizar os mecanismos tradicionais de Treplica para provisionar uma nova réplica leitora. Em resumo, uma réplica que se integra ao sistema pela primeira vez ou após uma falha demorada deve recuperar o seu estado. Esse processo acontece através

de um mecanismo de preenchimento de lacunas, que observa que não pode executar novas requisições de escrita sem antes executar as requisições anteriores (VIEIRA; BUZATO, 2010).

Esse procedimento de recuperação é voltado para reparar pequenas interrupções e não a recuperação do estado completo de uma réplica. Em particular, no caso de uma réplica leitora sem estado persistente, o tamanho dessa recuperação pode ser muito grande em termos do número de *requisições* a serem reexecutadas, pois ela sempre parte do estado inicial vazio.

Decidimos então usar sempre o protocolo de transferência de estado para provisionar uma réplica leitora, pois essas réplicas não possuem estado persistente e, por definição, sempre devem recuperar tudo o que aconteceu desde o início da computação. Nessa versão, o mecanismo de transferência de estado é aplicado somente para réplicas leitoras. A configuração desse tipo de réplica se torna mais fácil pelo fato que não existe estado anterior para ser conciliado com o novo estado sendo recebido. Em particular, a adição ou remoção de uma réplica leitora não altera o número de receptores executando o algoritmo, não havendo necessidade de se realizar um reconfiguração custosa (LAMPORT; MALKHI; ZHOU, 2010).

2.2.3 Transferência de estado para recuperação de falhas

Treplica possui outro potencial candidato para empregar o protocolo de transferência de estado: o componente detector de lacunas na sequência de instâncias de consenso. Lacunas podem surgir por diferentes motivos: falha-e-recuperação na réplica, perda de mensagens ou ainda réplicas com grandes diferenças de capacidade de processamento. Para preencher lacunas detectadas o componente solicita retransmissão da instância de consenso em uma determinada rodada. É perceptível que na presença de grandes lacunas essa abordagem exigirá uma longa sequência de retransmissões.

Para exemplificar o potencial problema do detector de lacunas, vamos supor que uma aplicação está passando por um grande pico de processamento (*flash crowds*) e que a réplica r possui uma grande lacuna. O mecanismo detector de lacunas solicitará retransmissão das instâncias de consenso não conhecidas por r, aumentando a concorrência no meio compartilhado pelas réplicas: a rede. Como solução alternativa, podemos utilizar o mecanismo de transferência para concentrar em uma única mensagem o estado resultante da execução das operações codificadas pelas instâncias de consenso. Dessa forma, um mecanismo de detecção de lacunas deveria ter a capacidade de:

- Retransmissão de instância de consenso: utilizado para preencher pequenas lacunas na sequência de consenso.

- Protocolo de transferência de estado: utilizado para preencher grandes lacunas, onde retransmissão tem potencial perda de desempenho.

3 Avaliação e Resultados

O Capítulo 2 apresentou três funcionalidades para expansão da biblioteca Treplica:

1. Protocolo para transferência de estado: criação de um mecanismo eficiente para transferência de estado entre réplicas.

2. Réplicas leitoras: possibilidade da utilização de réplicas que não participam do processo de decisão de instâncias de consenso.

3. Equalização de estado: proposta para novo componente de preenchimento de lacunas originadas por possíveis períodos de instabilidade da réplica e/ou falhas.

Supomos duas hipóteses baseado-se nas alterações propostas 1 e 2. A alteração proposta pelo item 3 não será validada por esse trabalho. Mantivemos a descrição dessa alteração para enriquecimento do conteúdo e possível utilização em trabalhos futuros.

1. Se possuirmos um mecanismo de transferência eficiente, podemos recuperar o estado de uma réplica de forma mais rápida do que se fôssemos reexecutar todo o histórico de operações perdidas por essa réplica.

2. Se adicionarmos uma réplica de forma a evitar uma reconfiguração total do sistema, ganharemos a flexibilidade para expansão do aglomerado de acordo com a demanda de clientes.

Para validar que as hipóteses 1 e 2 podem aumentar o desempenho de um sistema que utiliza a biblioteca Treplica, criamos dois experimentos onde supomos uma carga de trabalho na qual uma parcela significativa das requisições solicitadas para aplicação seja de leitura. Essa é uma suposição razoável para a maioria das aplicações de Internet (TPC, 2002) e proporciona o cenário, que acreditamos ser o mais adequado, para execução eficiente utilizando réplicas leitoras. Esses experimentos são:

1. Experimento para transferência de estado: criamos um cenário onde uma nova réplica leitora é adicionada no aglomerado. Dessa forma, o protocolo de transferência de estado é acionado. Para criar uma base de comparação, repetimos o experimento adicionando uma réplica votante exigindo a atuação do mecanismo de retransmissão de instância de consenso.

2. Experimento de réplicas leitoras: fundamentamos esse experimento na combinação de réplicas com diferentes graus de uso de memória persistente. Utilizamos duas formações de aglomerados submetidos a mesma carga: (1) três réplicas votantes e duas réplicas leitoras; e (2) 5 réplicas votantes para criação de uma base de comparação. O objetivo nesse cenário é testar o desempenho das réplicas leitoras em comparação com uso apenas de réplicas votantes.

Começamos esse capítulo com a Seção 3.1, apresentando os detalhes da aplicação e as bibliotecas utilizadas para sua concepção. Em seguida, na Seção 3.2 descrevemos o ambiente experimental usado na execução dos experimentos. A Seção 3.3 e Seção 3.4 apresentam, respectivamente, os experimentos de transferência de estado e de réplicas leitoras com seus respectivos resultados e análise. Encerramos o capítulo com uma breve descrição de um problema de conflitos de instâncias encontrado durante a análise experimental na Seção 3.5.

3.1 Aplicação

Visando a validação experimental das hipóteses apresentadas na seção anterior, desenvolvemos uma aplicação Web simples que mapeia uma cadeia de caracteres para um valor numérico de 32 bits. Em outras palavras, criamos uma aplicação caracterizada como um mapa que disponibiliza dois serviços aos clientes remotos através de uma interface HTTP:

1. Operação GET `/replicated-map/map/key/<chave>`
2. Operação PUT `/replicated-map/map/key/<chave>/value/<valor>`

A operação GET é uma operação de leitura que permite ao cliente buscar o valor armazenado em uma determinada chave. Definimos que uma operação bem-sucedida no método GET retorna o código de status HTTP 200 e um JSON com o valor da chave requisitada `{"value":"<valor>"}`. A operação PUT define operações de escrita, ela é responsável por armazenar um valor em uma determinada chave. Definimos que uma operação de escrita sem falhas não possui corpo de retorno, o método retorna apenas o código de status HTTP 201.

Utilizamos a linguagem Scala[1] para criação da aplicação, sendo que o estado da aplicação é gerenciado pela biblioteca Treplica que atua como um *middleware* de replicação ativa, conforme ilustra a Figura 14. Treplica é uma biblioteca Java e mostrou boa interoperabilidade com a aplicação. Não relatamos nenhum problema oriundo da utilização de bibliotecas Java com a linguagem Scala.

[1] <http://www.scala-lang.org>

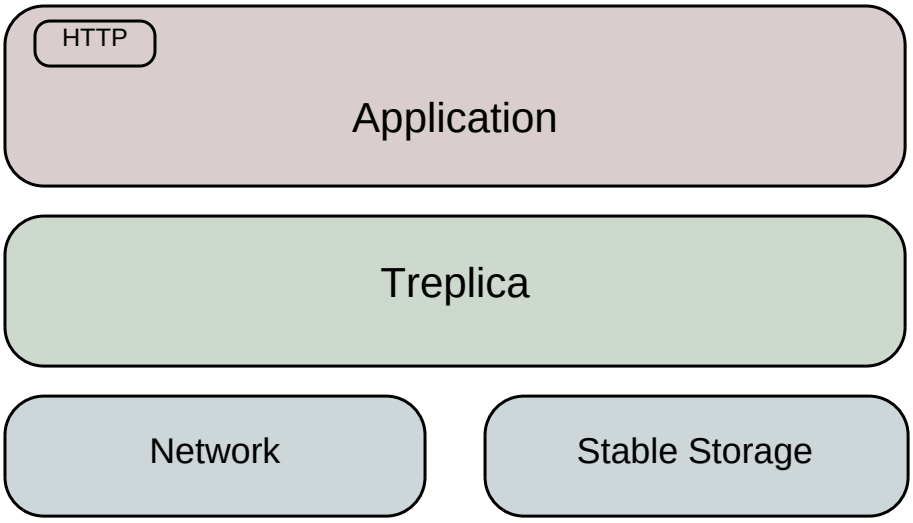


Figura 14: Treplica como *middleware* de replicação ativa

Sabemos que um mapa pode ser replicado usando outras estratégias mais eficientes que replicação ativa devido a independência das chaves, mas selecionamos essa aplicação por sua simplicidade de implementação e geração de carga. Dessa forma a aplicação de teste é executada em um aglomerado de máquinas e oferece pra seus clientes, a garantia de que toda operação de escrita será replicada para outras instâncias de forma ativa, usando Treplica. Com base nessa propriedade, garantimos a elegibilidade dessa aplicação para execução de experimentos no modelo computacional utilizando replicação ativa.

Cada uma das réplicas instanciadas possui seu próprio estado e pode assumir diferentes graus de persistência para os dados em memória, definido-as como réplicas votantes ou réplicas leitoras, de acordo com Seção 2.2.2:

- Réplica votante: utiliza memória persistente, premissa do algoritmo Paxos para garantia de correção no modelo computacional falha-e-recuperação.

- Réplica leitora: utiliza memória volátil.

A utilização de réplicas votantes e leitoras irá depender do experimento que está sendo executado.

3.1.1 Dependências

A Tabela 1 lista as bibliotecas das quais o projeto depende, com a respectiva versão utilizada para compilação e execução da aplicação. A tabela lista também a responsabilidade que a biblioteca exerce no projeto.

Tabela 1: Tabela de dependência de bibliotecas

Biblioteca	Versão	Responsabilidade
treplica	0.3.2	*midleware* de replicação ativa
vraptor	3.5.1	controlador MVC

Nota: Vraptor: <http://vraptor3.vraptor.org>

Nota: Model-View-Controller, é um padrão de projeto de software que separa a representação da informação da iteração do usuário

3.2 Descrição do Ambiente Experimental

Os experimentos foram realizados em um aglomerado com 16 nós, interligados através de um *switch* Ethernet de 1 Gbps. Cada nó possui a seguinte capacidade de processamento:

- 1 processador Xeon E5620 (2.4 GHz);
- 12 GB de memória RAM;
- 2 placas de rede Ethernet de 1 Gbps
- 500 GB de armazenamento local (disco 7200 rpm);
- Plataforma de 64 bits;

Os nós utilizam o sistema operacional GNU/Linux Debian 6.0. A Tabela 2 lista os aplicativos que foram utilizados para execução dos experimentos e suas respectivas versões.

Tabela 2: Tabela de aplicativos utilizados nos experimentos

Aplicativo	Versão	Descrição
JVM	HotSpot 64-Bit 1.7.0_45	Máquina Virtual Java
Apache Tomcat	7.0.32	*Servlet container*
HAProxy	1.5-dev19	*HTTP Load balancer*

Nota: Apache Tomcat: <http://tomcat.apache.org>

Nota: HAProxy: <http://www.haproxy.org>

O conjunto de réplicas é gerenciado por um servidor balanceador de carga configurado com HAProxy, usado para melhorar o desempenho de serviços Web distribuindo requisições entre vários servidores. As requisições executadas em um experimento são

iniciadas por geradores de carga, descritos na Seção 3.2.1, e passam pelo HAProxy que está configurado para receber e rotear as requisições para as réplicas ativas, usando um algoritmo de revezamento circular (*round-robin*), como ilustrado na Figura 15. Nessa configuração os geradores de carga não compartilham recursos com os processos da aplicação, ou seja, ou uma máquina executa processos clientes ou executa processos da aplicação.

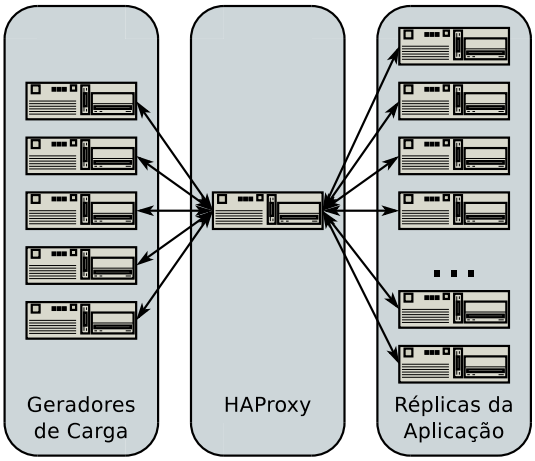


Figura 15: Configuração experimental

3.2.1 Carga

O gerador de carga tem como objetivo produzir a carga de trabalho requirida pelos experimentos e coletar medidas de vazão do ponto de vista do cliente. Sendo assim, cada instância do gerador de carga registra em *log* o *timestamp* de início e fim de cada atividade, para que seja possível o computar a vazão alcançada pela aplicação.

Os geradores de carga são processos Java que executam requisições de leitura (requisições para o método HTTP GET) e escrita (requisições para o método HTTP PUT) respeitando um percentual configurável para realização de cada operação. Para todos os experimentos dedicamos 5 máquinas para atuarem como clientes, sendo que cada uma executa 1000 requisições/segundo, variando entre requisições de leitura e escrita. Esse valor é maior do que Treplica consegue atender para essa aplicação, em todos os cenários testados. Assim garantimos que o sistema está sempre em sua capacidade máxima. O gerador de carga acompanha o número de requisições em aberto e limita esse número a 160 requisições em cada instância, de forma a regular o fluxo de criação de requisições com o fluxo de processamento das mesmas.

Do ponto de vista da aplicação, requisições de leitura serão atendidas localmente por uma das réplicas, enquanto uma requisição de escrita será transformada em uma proposta e submetida a uma instância de consenso. Avaliando-se o custo de cada tipo de requisição, podemos afirmar que requisições de escrita são mais caras porque envolvem mais trocas de mensagens que requisições de leitura.

Para execução de todos os experimentos foi gerado uma carga utilizando todas as 5 máquinas disponíveis, totalizando uma carga máxima para a aplicação de 5000 requisições/segundo, mantida durante 300 segundos para cada medida experimental. Para a análise resolvemos considerar intervalos diferentes de acordo com o cenário de teste. Tal intervalo, ao qual chamaremos de *período de análise*, será identificado em cada análise. Os experimentos foram divididas em duas configurações, conforme mostra os dados na Tabela 3.

Tabela 3: Configuração da carga experimental

Experimento	Requisições por segundo	Total de escrita	Total de leitura
1	5000	2500 (50%)	2500 (50%)
2	5000	1000 (20%)	4000 (80%)

Identificaremos os experimentos como Experimento 1 e Experimento 2. O objetivo da configuração do Experimento 1 é criar um cenário que exigirá bastante esforço do protocolo Paxos devido o equilíbrio da carga de requisições de leitura e escrita. Em contra partida a configuração do Experimento 2 proporciona um cenário com menos trocas de mensagens de Paxos, já que a maioria das requisições (80%) são de leitura e são atendidas localmente em cada réplica.

3.3 Experimento Transferência de Estado

A análise de desempenho para esse experimento visa responder duas questões sobre o mecanismo de transferência de estado em Treplica:

1. Qual é o desempenho do mecanismo de transferência de estado?
2. Qual é a melhor estratégia em Treplica para recuperação de estado: transferência de estado ou aplicação uma a uma das instâncias de consenso?

Para construção dos cenários experimentais supomos um aglomerado com cinco réplicas, todas previamente configuradas com a aplicação de teste. Nosso objetivo é a geração da divergência do estado replicado em uma das réplicas. Para alcançar tal objetivo iniciamos quatro réplicas simultaneamente e postergamos a inicialização de uma das réplicas. Definimos então dois instantes no tempo:

- $t1$ início da execução das requisições dos clientes;
- $t2$ inicialização da aplicação na quinta réplica, $t2$ acontece exatamente 180 segundos após $t1$.

Nessa configuração, o momento $t1$ possui quatro réplicas prontas para atender as requisições dos clientes e no momento $t2$, a quinta réplica é adicionada ao aglomerado. Essa nova réplica, poderá ser votante ou leitora dependendo da configuração do experimento, conforme ilustra a Figura 16. Nessa versão de Treplica o mecanismo de transferência de estado é exclusivo para replicas leitoras e vamos usar esse fator para facilitar a comparação com o mecanismo tradicional de Treplica. Dessa forma, ao adicionar uma réplica leitora acionamos o mecanismo de transferência de estado e ao adicionar uma réplica votante acionamos o mecanismo de retransmissão de instância de consenso (preenchimento de lacunas).

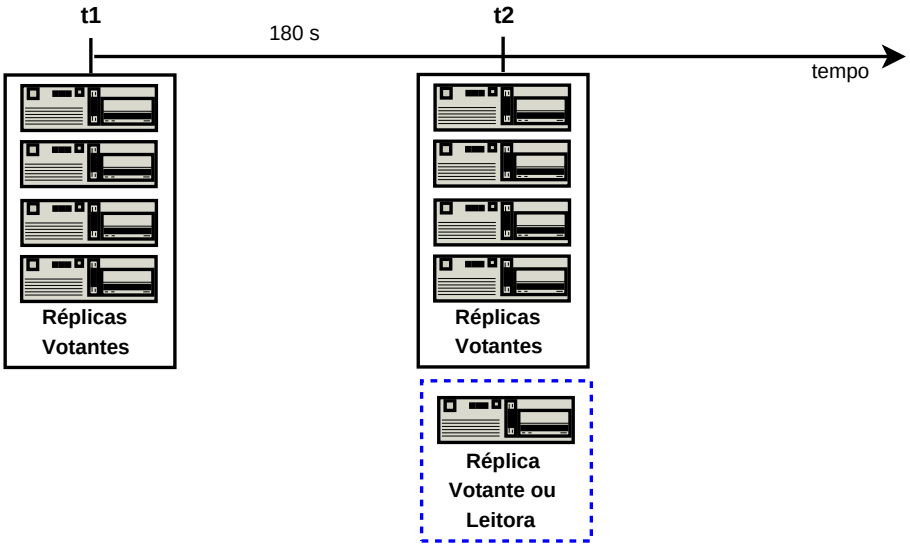


Figura 16: Linha do tempo experimento transferência de estado

Nesse experimento focamos nossa análise no instante $t2$. Vamos medir e comparar o impacto no desempenho do aglomerado causado pela adição de uma nova réplica. O foco nesse caso não é comparação de réplicas votantes ou leitoras, mas sim os diferentes mecanismos utilizadas por elas para receber o estado compartilhado pelo aglomerado.

Para garantir a confiabilidade dos resultados decidimos variar o tamanho do estado para medir a capacidade do mecanismo de transferência. Para criação de tais cenários, combinamos as configurações do Experimento 1 e do Experimento 2, detalhados na Seção 3.2.1, com a adição de uma réplica leitora, conforme descrito anteriormente. A configuração do Experimento 2 produz um estado menor devido a quantidade de requisições de leitura. Teoricamente esse cenário é favorável para o mecanismo de transferência de estado uma vez que a quantidade de dados trafegados pela rede é menor.

Para esse experimento foram geradas requisições com uma chave aleatória e um valor de 500 bytes como na Figura 17. Assim, o estado crescerá proporcionalmente a quantidade de requisições, em uma velocidade suficiente para gerar um estado da ordem de centenas de *megabytes* no tempo que o experimento executa.

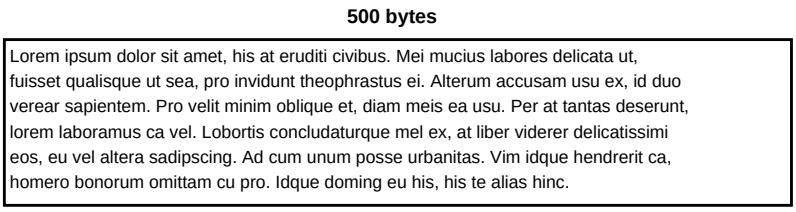


Figura 17: Carga associada a cada requisição de escrita

3.3.1 Procedimento de teste

Os passos para execução do experimento de transferência de estado são listados a seguir:

1. Iniciar manualmente o serviço do HAProxy em uma instância dedicada para o serviço de *load balancer*;
2. Configurar, através de *script*, cinco instâncias que atuarão como servidoras, respeitando os parâmetros que definem se a réplica atuará como votante ou leitora. Nesse passo é executado a instalação da JVM e Tomcat, a aplicação de teste foi previamente instalada no Tomcat;
3. Configurar, através de *script*, cinco instâncias dedicadas para o serviço de cliente. Nesse passo também é instalado a JVM nas instâncias clientes e o programa de teste;
4. Iniciar através de *script* o serviço do Tomcat em quatro réplicas;
5. Iniciar através de *script* o programa de teste nas cinco máquinas dedicadas para atuarem como cliente;
6. Aguardar 180 segundos e iniciar através de *script* o serviço do Tomcat na quinta máquina (retardatária);
7. Aguardar mais 180 segundos e, através de *script*, encerrar os processos Java iniciados (cliente e servidor) e recolher os arquivos de *log* dos clientes e servidores.

3.3.2 Resultados e Análise

Nesta subseção responderemos as duas perguntas feitas no início dessa seção: (1) Qual é o desempenho do mecanismo de transferência de estado? (2) Qual é a melhor estratégia em Treplica para recuperação de estado: transferência de estado ou aplicação uma a uma das instâncias de consenso?

Os valores das métricas foram coletados ao longo de 10 execuções e os valores apresentados correspondem à média dessas execuções.

3.3.2.1 Desempenho absoluto - Pergunta 1

Para medir o desempenho do mecanismo de transferência de estado inciamos no instante $t2$ uma réplica leitora e estabelecemos os seguintes parâmetros para coleta e medição de desempenho:

- Tempo total para envio do estado da réplica receptora para réplica doadora, incluindo tempo para estabelecer a conexão TCP entre as réplicas;
- Tamanho do estado enviado;
- Tempo total para ativação de uma réplica leitora. Essa medida leva em consideração o tempo que a réplica demorou para estar apta a processar mensagens da aplicação.

A Figura 18 (a) exibe o tamanho do estado transferido pelo Experimento 1 (50% de requisições de leitura) e pelo Experimento 2 (80% de requisições de leitura). Comparando os resultados de tempo de transferência, ilustrados na Figura 18 (b), podemos afirmar que o desempenho do mecanismo de transferência de estado está ligado diretamente com o tamanho do estado a ser transferido. Em outras palavras, quanto maior o estado maior o tempo gasto para transferi-lo.

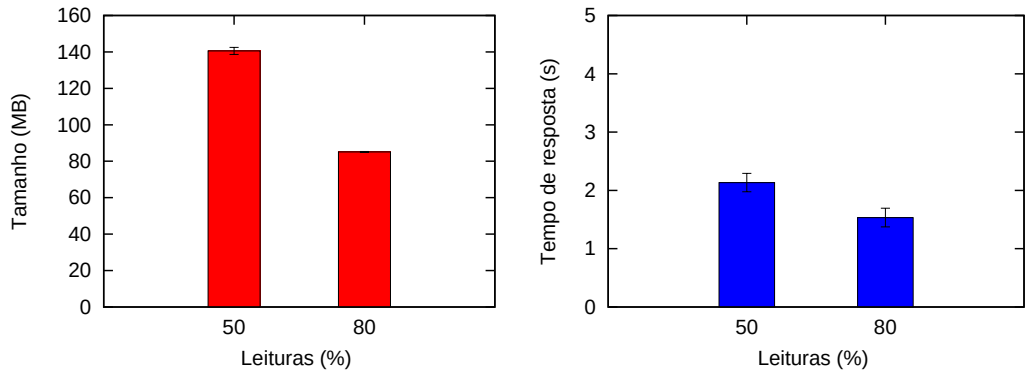


Figura 18: Resultado transferência de estado

Os resultados ilustrado pela Figura 18 também são exibidos nas Tabela 4 (Experimento 1) e Tabela 5 (Experimento 2).

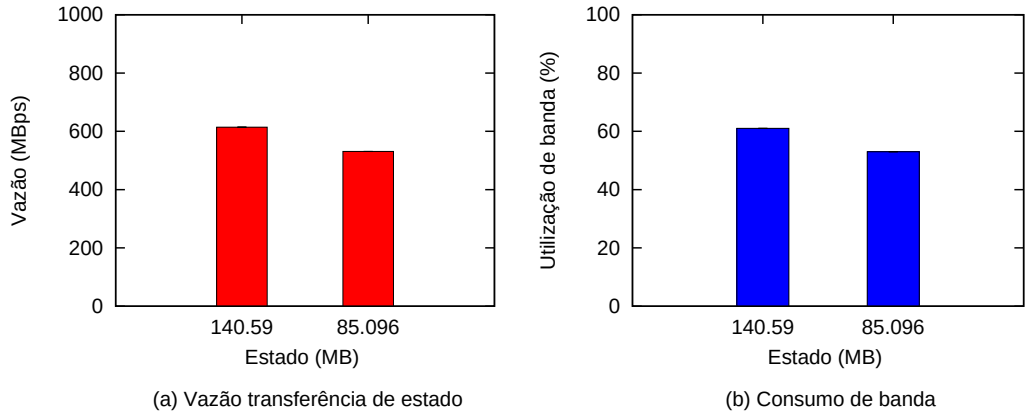
Tabela 4: Desempenho da transferência de estado no Experimento 1

	Tempo de Transferência	Tamanho do Estado	Tempo de Ativação
Média	1.832s	140.590MB	2.133s
Mediana	1.867s	141.535MB	2.209s
Desvio Padrão	0.140	1.934	0.157

Tabela 5: Desempenho da transferência de estado no Experimento 2

	Tempo de Transferência	Tamanho do Estado	Tempo de Ativação
Média	1.282s	85.096MB	1.533s
Mediana	1.305s	85.150MB	1.513s
Desvio Padrão	0.160	0.296	0.079

Considerando o tempo de transferência e o tamanho do estado, podemos calcular a vazão em *megabytes*/segundo que o mecanismo de transferência de estado atingiu. Também podemos calcular a utilização da banda de cada transferência. A Figura 19 ilustra esses dois cálculos, analisando os resultados podemos verificar que o mecanismo de transferência de estado utiliza a banda disponível de forma eficiente.



(a) Vazão transferência de estado (b) Consumo de banda

Figura 19: Vazão transferência de estado

Os mesmos valores também são listados na Tabela 6.

Tabela 6: Vazão transferência de estado

Estado (MB)	Vazão (Mbps)	Utilização de banda (%)
140.59	614	61
85.096	531	53

Para se ter uma ideia da dimensão dos resultados obtidos com o mecanismo de transferência de estado, executamos um experimento simples utilizando a ferramenta rsync[2]. Essa não é uma comparação justa devido as diferenças entre os mecanismos. A Tabela 7 exibe a média de 10 execuções do comando `rsync -W` para dois arquivos que se equivalem em tamanho com os estados transferidos nos experimentos. Podemos observar que a vazão obtida nos experimentos utilizando o mecanismo de transferência de estado proposto é superior a vazão obtida com a ferramenta rsync.

[2] <https://rsync.samba.org>

Tabela 7: Desempenho da transferência de arquivo com rsync

	Arquivo 140.59MB	Arquivo 85.096MB
Média	2.084s	1.357s
Mediana	2.081s	1.317s
Desvio Padrão	0.012	0.122
Vazão	515Mbps	479Mbps
Utilização de banda	51%	47%

3.3.2.2 Desempenho relativo - Pergunta 2

A segunda pergunta definida no início dessa seção é referente a escolha da melhor estratégia de Treplica para recuperação de estado. Para respondê-la vamos comparar as execuções do experimento da Seção 3.3, onde adicionamos ao aglomerado uma nova réplica usando o mecanismo tradicional de Treplica (réplica votante) ou uma nova réplica usando o mecanismo proposto (réplica leitora). O período de análise está concentrado ao redor do tempo $t2$. Compararemos os dois mecanismos de recuperação de estado analisando a quantidade de operações por segundo atendidas em um espaço de 30 segundos de execução. O período de análise contém registros de 10 segundos antes da $t2$ e 20 segundos após $t2$.

Continuamos com a mesma configuração de carga definidas pelos Experimentos 1 e 2. A Figura 20 ilustra a vazão instantânea do experimento que adiciona réplicas votantes e leitoras com resulução de 5 segundos durante o período de análise tomada como média das 10 execuções. Assim que o balanceador de carga detecta a presença de uma nova réplica, começa a rotear requisições da aplicação para ela. Essa nova réplica também passa a receber mensagens do protocolo Paxos oriundas das outras réplicas do aglomerado.

Analisando especificamente a Figura 20 (a), podemos verificar um vale acentuado no momento da adição da réplica votante rv. Na transferência de estado tradicional utilizado por rv, o estado é replicado instância por instância. Fica claro que a adição de rv causa uma alteração no desempenho do aglomerado que desaparece instantes depois quando a recuperação termina.

Por outro lado, a Figura 20 (b) ilustra o impacto da adição de uma réplica leitora rl. Podemos verificar uma alteração no número de operações suportadas quando rl é adicionada. Porém, fica claro que o distúrbio causado pela adição de uma réplica leitora é menor. Nesse caso, a replicação de estado em rl acontece através da atuação do mecanismo de transferência de estado. Ou seja, o estado é transferido usando um mecanismo de lote mais eficiente (TCP), causando menos impacto no desempenho do aglomerado.

O Experimento 2 está configurado com um maior número de requisições de leitura, a Figura 21 ilustra seu desempenho de forma similar à Figura 20. Mais uma vez podemos

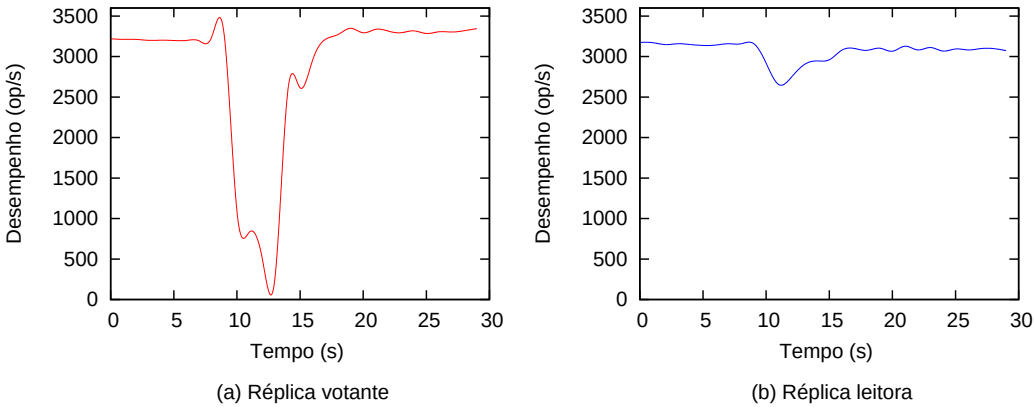


Figura 20: Desempenho da adição de uma nova réplica no Experimento 1

observar menos impacto no número de requisições quando adicionamos uma réplica leitora, caracterizando assim o bom desempenho do mecanismo de transferência de estado nos dois cenários experimentais.

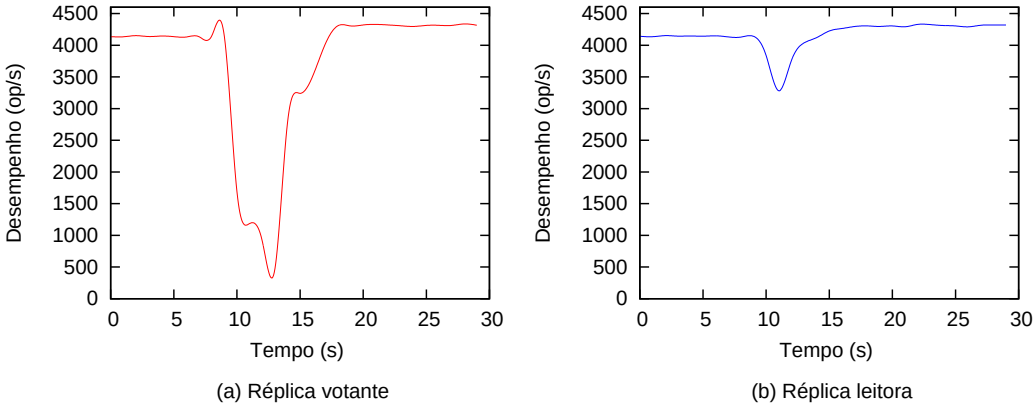
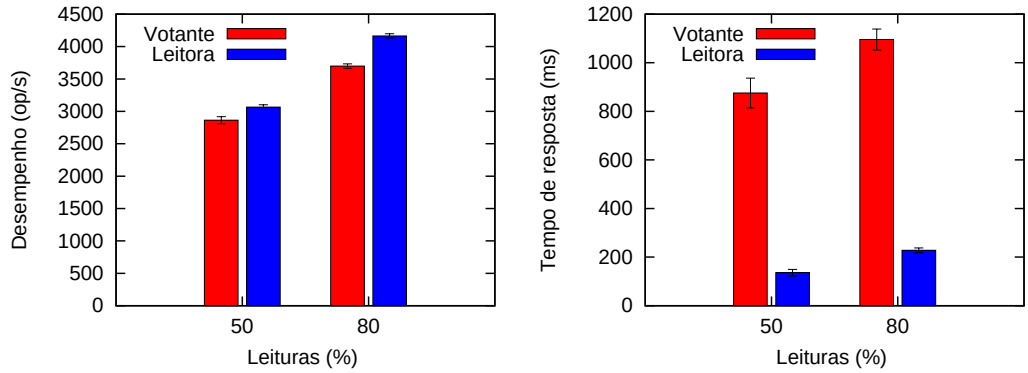


Figura 21: Desempenho da adição de uma nova réplica no Experimento 2

A Figura 22 resume desempenho total do sistema ao longo do período de análise, medido pelo número de operações por segundo atendidas e pelo tempo de resposta dessas requisições. Observando a vazão obtida no experimento (Figura 22 (a)), podemos verificar que para o Experimento 1 a proposta de transferência de estado obteve um desempenho 7.06% superior (202.163 operações por segundo), enquanto que no Experimento 2 o aumento de desempenho foi de 12.60% (466 operações por segundo). Realizamos um teste-t independente para cada uma dessas duas diferenças e o resultado produziu um valor t estatisticamente significativo para o Experimento 1 ($t_{(10)} = 9.74$, $p < 0.0001$) e para o Experimento 2 ($t_{(10)} = 31.15$, $p < 0.0001$).

A maior diferença entre a adição de uma réplica votante ou leitora está no tempo de resposta do período de análise (Figura 22 (b)). No Experimento 1 o mecanismo de transferência de estado obteve tempos de resposta 84.46% menores (739.76 ms) que a ree-

xecução individual de operações, enquanto que no Experimento 2 a melhora foi de 79.20% (867.75 ms). Novamente, realizamos um teste-t independente para cada uma dessas duas diferenças e o resultado produziu um valor t estatisticamente significativo para o Experimento 1 ($t_{(10)} = 37.33$, $p < 0.0001$) e para o Experimento 2 ($t_{(10)} = 61.93$, $p < 0.0001$).



(a) Operações suportadas (b) Tempo de resposta

Figura 22: Comparação do desempenho do mecanismo de transferência de estado

Quando comparamos os dois experimentos, podemos verificar que o Experimento 2 consegue suportar mais operações por segundo (Figura 22 (a)). Isso se deve ao fato da redução de iterações entre as réplicas requiridas por uma requisição de leitura, predominante no Experimento 2. Os resultados ilustrado pela Figura 22 também são exibidos na Tabela 8 (adicionando uma réplica votante) e na Tabela 9 (adicionando uma réplica leitora).

Tabela 8: Desempenho para incorporação de uma réplica votante

Leitura	Operações/segundo	Desvio Padrão	Tempo de Resposta (ms)	Desvio Padrão
50%	2862.933	54.491	875.824	61.181
80%	3698.806	33.976	1095.644	43.157

Tabela 9: Desempenho para incorporação de uma réplica leitora

Leitura	Operações/segundo	Desvio Padrão	Tempo de Resposta (ms)	Desvio Padrão
50%	3065.096	36.611	136.065	13.540
80%	4164.806	32.920	227.890	10.049

3.4 Experimento Réplicas Leitoras

Para validar a hipótese que réplicas leitoras podem aumentar o desempenho de um sistema que utiliza Paxos, precisamos de uma carga de trabalho na qual uma parcela significativa das requisições solicitadas sejam de leitura, pois essas réplicas não participam

da eleição de consenso. Para esse experimento utilizamos a aplicação descrita na Seção 3.1, fazendo uma combinação de réplicas com diferentes graus de uso de memória persistente. Nosso objetivo aqui é medir o desempenho de um aglomerado com réplicas votantes e leitoras.

Supomos duas configurações diferentes de aglomerado para execução dos experimentos, ambas com cinco réplicas cada: (1) cinco réplicas votantes para criação de uma base de comparação, conforme ilustra a Figura 23 (a); e (2) combinação de três réplicas votantes e duas réplicas leitoras, conforme ilustra Figura 23 (b).

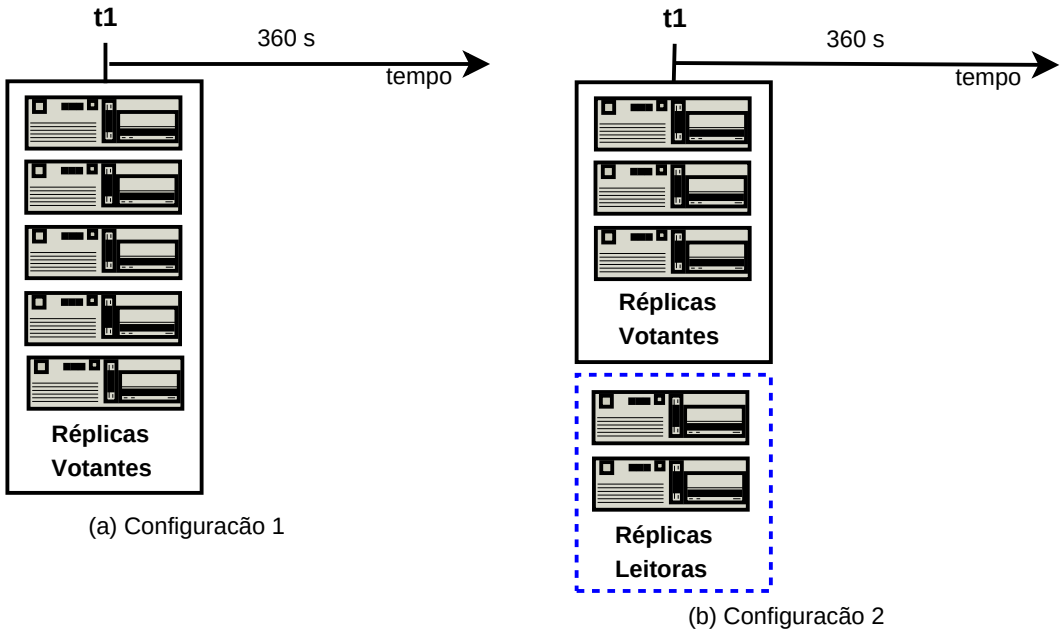


Figura 23: Linha do tempo experimento réplicas leitoras

O tamanho do quórum para eleição do consenso durante as rodadas de Paxos depende da configuração utilizada. Na Configuração 1 não conseguimos consenso quando três réplicas falham enquanto a Configuração 2 já não tolera falha em apenas duas réplicas. Podemos afirmar então que a tolerância a falhas em Treplica está ligada diretamente ao número de réplicas votantes existentes no aglomerado, sendo assim, a Configuração 1 é mais tolerante a falhas.

Nesse experimento iniciamos a aplicação nas réplicas no mesmo instante de tempo para evitar possíveis influências de outros componentes alterados em Treplica. Dessa forma, no instante $t1$ onde se inicia a execução das requisições dos clientes, todas as réplicas participantes do experimento já foram detectadas como ativas pelo servidor HA-Proxy e estão aptas para processar requisições da aplicação.

3.4.1 Procedimento de teste

Os passos para execução do experimento para medir o desempenho de réplicas leitoras são listados a seguir:

1. Iniciar manualmente o serviço do HAProxy em uma instância dedicada para o serviço de *load balancer*;

2. Configurar, através de *script*, as réplicas que atuarão como servidoras, respeitando a configuração de teste desejada. Nesse passo é executada a instalação da JVM e Tomcat, a aplicação de teste foi previamente instalada no Tomcat;

3. Configurar, através de *script*, cinco instâncias dedicadas para o serviço de cliente. Nesse passo também é instalado a JVM nas instâncias clientes e o programa de teste;

4. Iniciar através de *script* o serviço do Tomcat em todas as réplicas servidoras;

5. Iniciar através de *script* o programa de teste nas cinco máquinas dedicadas para atuarem como cliente;

6. Aguardar 360 segundos, e através de *script*, encerrar os processos Javas iniciados (cliente e servidor) e recolher os arquivos de *log* dos clientes e servidores.

Aqui, diferentemente do experimento de transferência de estado, iniciamos a aplicação nas instâncias que atuam como servidoras simultaneamente.

3.4.2 Resultados e Análise

Para análise desse experimento consideraremos as cargas do Experimento 1 e 2, que já foram definidas na Seção 3.2.1. Conforme descrito na Seção 3.4.1, em todas as execuções desse experimento as réplicas foram iniciadas simultaneamente. Sendo assim, eventuais sobrecargas na inicialização de réplicas leitoras foram eliminadas pois não existe nenhum estado para ser transferido. Em outras palavras, no momento que os geradores de carga iniciam todas as réplicas compartilham do mesmo estado inicial.

Nesse experimento redefinimos o período de análise para 260 segundos de execução. Foram descartamos os 20 segundos iniciais e finais visando eliminar efeitos transientes do início e término da aplicação. A análise de desempenho para esse cenário visa responder uma questão: Qual é o desempenho de uma réplica leitora?

Para responder essa pergunta, vamos comparar a execução das Configurações 1 e 2 no Experimento 1 (50% de requisições de leitura) e Experimento 2 (80% de requisições

de leitura), executando dez vezes cada uma das quatro combinações de configuração e experimento. A Figura 24 ilustra a comparação do número médio de requisições atendidas e o tempo de resposta dessas requisições no período de análise para os dois experimentos. Para maior clareza de comparação, os resultados também são exibidos na Tabela 10 (aglomerado configurado somente réplicas votantes) e na Tabela 11 (aglomerado configurado com réplicas votantes e leitoras). Fica evidente a semelhança no desempenho das duas configurações do aglomerado nesses dois critérios de análise.

No Experimento 1 podemos observar uma pequena vantagem na configuração sem réplicas leitoras de 41 operações por segundo, com tempo de resposta 3.09 ms menor. No entanto, realizamos um teste-t independente e observamos que essa diferença não é estatisticamente significativa (com $p < 0.0001$) para vazão ($t_{(10)} = 2.44$, $p = 0.0244$) e para tempo de resposta ($t_{(10)} = 2.47$, $p = 0.0238$). Esse é um resultado que consideramos positivo para as réplicas leitoras, pois conseguem equiparar o desempenho de réplicas votantes e possuem a grande vantagem de não precisarem de reconfiguração total quando são adicionadas ou removidas de um aglomerado. Essa flexibilidade de redimensionamento proporcionada por réplicas leitoras passa a ser um diferencial a ser explorado em Treplica, ao custo de menor tolerância a falhas.

No Experimento 2 a configuração com réplicas leitoras teve um desempenho 52.5 operações por segundo superior e um tempo de resposta 2.6 ms superior. Porém, o grande número de leituras desse experimento tornam o desempenho bem mais previsível com um pequeno desvio padrão, de forma que um teste-t independente mostra que esta pequena diferença é estatisticamente significativa para vazão ($t_{(10)} = 8.46$, $p < 0.0001$) e para tempo de resposta ($t_{(10)} = 27.25$, $p < 0.0001$). No cenário configurado com muitas leituras, réplicas leitoras oferecem um pequeno ganho, que foi constante durante durante as execuções experimentais.

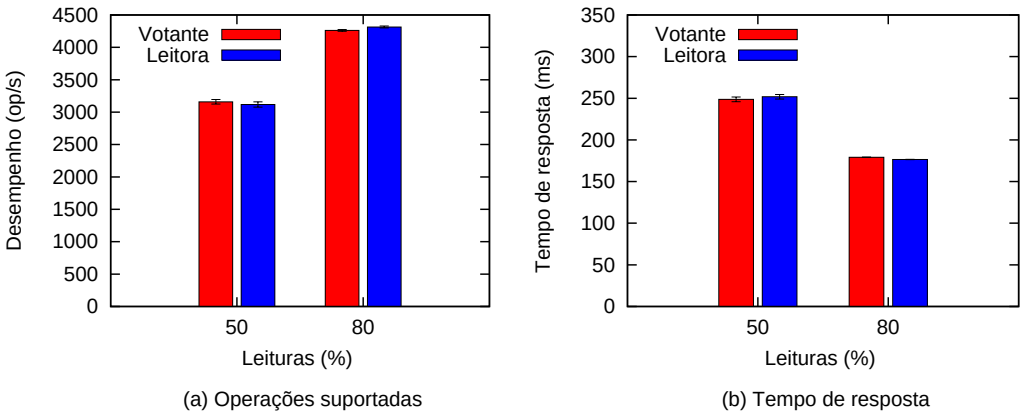


Figura 24: Comparação do desempenho de aglomerados com réplicas leitoras

Tabela 10: Desempenho com réplicas votantes

Leitura	Operações/segundo	Desvio Padrão	Tempo de Resposta (ms)	Desvio Padrão
50%	3118.375	40.235	251.804	2.787
80%	4314.798	14.468	176.524	0.194

Tabela 11: Desempenho com réplicas leitoras

Leitura	Operações/segundo	Desvio Padrão	Tempo de Resposta (ms)	Desvio Padrão
50%	3159.908	35.738	248.718	2.808
80%	4262.307	13.246	179.123	0.231

3.5 Conflitos nas instâncias de Paxos

Uma questão interessante levantada pela nossa análise de desempenho é a redução de desempenho de acordo com o aumento de requisições de escrita. Embora uma requisição de escrita exija mais trocas de mensagens de Paxos entre as réplicas, pudemos notar que essa não era a maior razão para a redução de desempenho. Treplica utiliza um mecanismo descentralizado para geração de número de instância baseado no conhecimento do estado local da réplica. Esse mecanismo não determinista causa problemas de conflito de instâncias quando atua dessincronizado no aglomerado.

Observarmos um grande número de conflitos entre processos tentando propor alterações de estado diferentes na mesma instância. Quando duas réplicas tentam decidir valores diferentes para a mesma instância, apenas uma é decidida e o agente coordenador envia uma mensagem para o proponente que gerou conflito aprender o que já foi decidido na instância conflitante. Sendo assim, o proponente cria uma nova proposta com um número diferente de instância e tenta novamente estabelecer o consenso com a possibilidade de um novo conflito. Notamos que conforme o aumento de propostas (ou agentes proponentes), maior é o número de conflitos.

Infelizmente a exploração desse problema não faz parte do escopo desse trabalho.

Conclusão

Nesse trabalho nós mostramos a implementação e os resultados experimentais de duas novas funcionalidades adicionadas em Treplica: (1) mecanismo de transferência de estado; e (2) réplicas leitoras.

Em relação ao mecanismo de transferência de estado pudemos observar que a adição de uma réplica usando o mecanismo proposto gerou menos impacto no desempenho do aglomerado. Dependendo do tamanho do estado a ser transferido os ganhos são consideráveis. Esse é um resultado interessante, pois mecanismos de reconfiguração completos também podem se beneficiar da utilização do protocolo de transferência de estado no caso da adição de uma nova réplica (LAMPORT; MALKHI; ZHOU, 2010). Os resultados experimentais obtidos com o mecanismo de transferência de estado justificam a extensão desse mecanismo para qualquer tipo de réplica, pois atualmente ele é exclusivo para réplicas leitoras.

No experimento de réplicas leitoras pudemos observar uma equivalência de desempenho entre réplicas leitoras e réplicas votantes no cenário configurado com mais requisições de escrita. No cenário configurado com um maior número de requisições de leitura houve uma vantagem pequena mas bem regular a favor das réplicas leitoras. Essa diferença, apesar de estatisticamente significativa, é muito pequena para justificar o uso de réplicas leitoras apenas por questões de desempenho.

No entanto, alterar o número de réplicas participantes de um sistema que usa replicação ativa não é uma tarefa trivial, como argumentamos na Seção 1.6. Contornar essa dificuldade é a maior vantagem que uma réplica leitora pode oferecer. A equivalência de desempenho com réplicas votantes passa a ser um resultado satisfatório, pois conseguimos não diminuir o desempenho da aplicação ao adicionar uma réplica que remove completamente a necessidade de reconfiguração. Acreditamos que uma maior escalabilidade pode ampliar as oportunidades de aplicação de replicação ativa para novos contextos. Especificamente, aplicações Web com uma parcela considerável de requisições de leitura podem se beneficiar da solução proposta.

Acreditamos que construímos mecanismos úteis para sistemas de replicação ativa, principalmente para Treplica que fica um passo mais próximo da reconfiguração total.

Trabalhos futuros

As propostas apresentadas por esse trabalho são um passo em direção ao mecanismo de autogestão cobiçado para Treplica. Realizar autointegração de réplicas sem inter-

venção humana exige criação de mecanismos para monitorar alterações no comportamento da aplicação e infraestrutura (RENESSE; BIRMAN; VOGELS, 2003; PIERRE; STEEN, 2006). A partir do momento que o sistema está ciente das mudanças de comportamento, pode tomar decisões para iniciar réplicas temporárias quando existe aumento de demanda ou até desligá-las após detectar queda no número de acessos. Essa ideia de dimensionamento elástico leva a uma pergunta: quantas réplicas são necessárias para suportar uma determinada carga de maneira eficiente? A resposta a essa pergunta é o coração da pesquisa para criação de um mecanismo de monitoração em Treplica.

Outro ponto não atacado neste trabalho é a implementação de um mecanismo híbrido de detecção de lacunas no estado. Quando uma réplica tem uma falha transiente de rede ou de operação ela perde o resultado de algumas instâncias de consenso. Essas falhas são relativamente frequentes em sistemas carregados e exigem que a réplica vá atrás de cada instância de consenso perdida. Observamos experimentalmente que quanto maior o estado, melhor a eficiência relativa da transferência de estado proposta. Então existe um certo número de instâncias atrasadas a partir do qual a transferência de estado proposta passa ter melhor desempenho. Pensando nisso, Treplica poderia suportar de maneira mais eficiente falha-e-recuperação nas réplicas selecionando o melhor mecanismo para cada tamanho de lacuna.

Conforme descrito na Seção 2.2.1.1, o protocolo de transferência de estado utiliza uma conexão TCP ponto a ponto entre a réplica doadora e receptora. Essa abordagem pode limitar o desempenho da transferência de estado na medida que estamos limitados a vazão de uma única réplica doadora. Uma outra possível abordagem, não pertencente ao escopo desse trabalho, é a utilização de um mecanismo colaborativo para envio do estado, onde partes do estado seriam enviados por diferentes réplicas doadoras, semelhante ao protocolo P2P. Gostaríamos de saber qual será o impacto que essa abordagem pode causar no desempenho de Paxos.

Gostaríamos também de ressaltar que o mecanismo de transferência de estado trás para Treplica a possibilidade da coleta de lixo. Instâncias de consenso antigas podem ser removidas da memória a partir do momento que nenhuma réplica está mais interessada no resultado desse consenso. Como na presença de falhas não é possível ter certeza que uma instância não será mais necessária para uma réplica, temos que fazer uma coleta de lixo otimista descartando instâncias com fortes indícios de serem desnecessárias. Em caso de erro, será necessário fazer uma transferência de estado para suprir na réplica que depende de uma instância descartada o estado perdido. O resultado dessa abordagem é a utilização mais eficiente da memória.

Esse trabalhamo considerou um sub-conjunto do problema de reconfiguração total descrito por Lamport, Malkhi e Zhou (2010). Adotamos essa abordagem para explorar outros mecanismos como transferência de estado e réplicas leitoras, além de fugir da

complexidade da reconfiguração total. No entanto, pudemos observar nos resultados dos experimentos que a utilização de réplicas leitoras é interessante mas não substituí uma réplica votante. Esse trabalho também ajuda a esclarecer o que precisa ser feito para uma reconfiguração total em Treplica.

É importante ressaltar que o problema de conflitos nas instâncias de Paxos descrito na Seção 3.5 requer uma investigação mais detalhada. Uma proposta de mecanismo para resolução de conflitos beneficiará tanto réplicas tradicionais quanto leitoras e pode garantir ainda mais o aumento de escalabilidade no aglomerado que utiliza Paxos como meio de replicação ativa.

Publicações

Tornando Paxos Mais Escalável com Réplicas Leitoras. Em *Anais do XIII Workshop de Desempenho de Sistemas Computacionais e de Comunicação (WPerformance 2014)*, páginas 2014-2018. Evento organizado pelo Congresso da Sociedade Brasileira de Computação (CSBC) em Julho de 2014 na cidade de Brasília.

Referências

ABDELLATIF, E. C.; LACHAIZE, R. Evaluation of a group communication middleware for clustered j2ee application servers. *Proceedings of the 2004 International Symposium on Distributed Objects and Applications*, p. 1571–1589, 2004. Citado 2 vezes nas páginas 50 e 57.

AGUILERA, M. K.; CHEN, W.; TOUEG, S. Failure detection and consensus in the crash-recovery model. *Distrib. Comput.*, Springer-Verlag, London, UK, v. 13, n. 2, p. 99–125, 2000. ISSN 0178-2770. Citado 3 vezes nas páginas 40, 44 e 63.

ALCHIERI, E. et al. Reconfiguração modular de sistemas de quóruns. In: *SBRC '14: Proc. of the 32th Brazilian Symposium on Computer Networks and Distributed Systems*. Brasília, Brasil: [s.n.], 2014. p. 281–294. Citado 2 vezes nas páginas 43 e 45.

BARBORAK, M.; DAHBURA, A.; MALEK, M. The consensus problem in fault-tolerant computing. *IEEE Computer Society*, p. 140, 1993. Citado na página 36.

BIRMAN K., M. D.; RENESSE, R. van. Virtually synchronous methodology for dynamic service replication. *Microsoft Research*, n. MSR-TR-2010-151, 2010. Citado na página 45.

BIRMAN, K. P. The process group approach to reliable distributed computing. *Communications of the ACM*, v. 36, n. 12, p. 37–53, 1993. Citado na página 41.

BIRMAN, K. P. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. [S.l.]: Springer, 2005. Citado na página 41.

BIRMAN, K. P.; JOSEPH, T. A. Exploiting virtual synchrony in distributed systems. *In SOSP'87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, p. 123–138, 1987. Citado 2 vezes nas páginas 26 e 47.

BIRMAN, K. P.; JOSEPH, T. A. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst*, v. 5, n. 1, p. 47–76, 1987. Citado 2 vezes nas páginas 26 e 47.

BIRRELL, A. D.; JONES, M. B.; WOBBER, E. P. A simple and efficient implementation of a small database. *IEEE Computer Society*, p. 149–154, 1987. Citado na página 42.

BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In: *OSDI '06: 7th USENIX Symposium on Operating Systems Design and Implementation*. [s.n.], 2006. Disponível em: <http://www.usenix.org/events/osdi06/tech/burrows.html>. Citado 2 vezes nas páginas 25 e 45.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. *Introduction to Reliable and Secure Distributed Programming*. Second. [S.l.]: Springer, 2011. Citado 4 vezes nas páginas 25, 30, 36 e 50.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: an engineering perspective. In: *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on*

Principles of distributed computing. New York, NY, USA: ACM Press, 2007. p. 398–407. ISBN 978-1-59593-616-5. Citado 3 vezes nas páginas 25, 26 e 43.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM*, ACM Press, New York, NY, USA, v. 43, n. 2, p. 225–267, 1996. ISSN 0004-5411. Citado 5 vezes nas páginas 26, 32, 36, 43 e 52.

CHEN, W.; TOUEG, S.; AGUILERA, M. K. On the quality of service of failure detectors. *IEEE Transactions on computers*, v. 51, n. 5, p. 561–580, 2002. Citado na página 32.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems: Concepts and Design*. Fifth. [S.l.]: Addison-Wesley, 2011. Citado 2 vezes nas páginas 34 e 55.

CRISTIAN, F. Understanding fault-tolerant distributed system. *16th International Symposium on Fault Tolerant Computing Systems*, v. 34, n. 2, p. 56–78, 1991. Citado na página 31.

CRISTIAN, F.; AGHILI, H.; STRONG, R. Clock syncronization in the presence of omissions and performance faults, and processor joins. *16th International Symposium on Fault Tolerant Computing Systems*, 1986. Citado na página 31.

FERREIRA, A. B. H. *Aurélio século XXI: o dicionário da Língua Portuguesa*. Third. [S.l.]: Nova Fronteira, 1999. Citado na página 60.

FRIEDMAN, R.; RENESSE, R. V. Strong and weak virtual synchrony in horus. *Communications of the ACM*, v. 25, p. 171–220, 1996. Citado na página 41.

GUERRAOUI, R.; SCHIPER, A. Software-based replication for fault tolerance. *Computer*, v. 30, n. 4, p. 68–74, 1997. Citado na página 34.

HAERDER, T.; REUTER, A. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, v. 15, n. 4, p. 287–317, 1983. Citado na página 37.

HUPFELD, F. et al. The XtreemFS architecture—a case for object-based file systems in grids. *Concurr. Comput. : Pract. Exper.*, John Wiley and Sons Ltd., Chichester, UK, v. 20, n. 17, p. 2049–2060, 2008. ISSN 1532-0626. Citado na página 25.

JALOTE, P. *Fault tolerance in distributed system*. [S.l.]: Communications of the ACM, 1994. Citado 5 vezes nas páginas 29, 30, 31, 34 e 36.

LAMPORT, L. *Leslie Lamport research microsoft page*. 1996. Http://research.microsoft.com/en-us/um/people/lamport/pubs. Acesso em: 01 mai. 2015. Citado na página 25.

LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.*, ACM Press, New York, NY, USA, v. 16, n. 2, p. 133–169, 1998. ISSN 0734-2071. Citado 7 vezes nas páginas 25, 26, 32, 36, 37, 39 e 43.

LAMPORT, L. Fast Paxos. *Distrib. Comput.*, Springer-Verlag, London, UK, v. 19, n. 2, p. 79–103, out. 2006. ISSN 0178-2770. Citado 3 vezes nas páginas 38, 48 e 53.

LAMPORT, L.; MALKHI, D.; ZHOU, L. Reconfiguring a state machine. *SIGACT News*, ACM, New York, NY, USA, v. 41, n. 1, p. 63–73, mar. 2010. ISSN 0163-5700. Citado 9 vezes nas páginas 26, 36, 42, 43, 45, 47, 66, 87 e 88.

LAMPSON, B. W. How to build a highly available system using consensus. In: *WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms*. London, UK: Springer-Verlag, 1996. p. 1–17. ISBN 3-540-61769-8. Disponível em: <https://research.microsoft.com/~lampson/58-Consensus/WebPage.html>. Citado 3 vezes nas páginas 25, 26 e 45.

MACCORMICK, J. et al. Boxwood: Abstractions as the foundation for storage infrastructure. In: *OSDI '04: 6th USENIX Symposium on Operating Systems Design and Implementation*. [s.n.], 2004. Disponível em: <https://db.usenix.org/events/osdi04/tech/maccormick.html>. Citado na página 25.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. *In Proceedings of the USENIX Annual Technical Conference*, p. 305–319, 2014. Citado na página 45.

PIERRE, G.; STEEN, M. V. Globule: A collaborative content delivery network. *IEEE Communications Magazine*, v. 44, n. 8, 2006. Citado na página 88.

RENESSE, R. V.; BIRMAN, K.; VOGELS, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *Communications of the ACM*, v. 21, n. 2, p. 164–206, 2003. Citado na página 88.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 22, n. 4, p. 299–319, 1990. ISSN 0360-0300. Citado 5 vezes nas páginas 25, 29, 30, 32 e 34.

SILVEIRA, P. et al. *Introdução a arquitetura e design de software*. [S.l.]: Campus, 2011. Citado na página 32.

TANENBAUM, A. S.; STEEN, M. V. *Distribuited system: principles and paradigms*. Second. [S.l.]: Pearson, 2007. Citado 4 vezes nas páginas 25, 33, 36 e 44.

TPC. *TPC Benchmark W Specification*. [S.l.], 2002. Disponível em: <http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf>. Citado na página 69.

VIEIRA, G. M. D.; BUZATO, L. E. Treplica: Ubiquitous replication. In: *SBRC '08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*. Rio de Janeiro, Brasil: [s.n.], 2008. Disponível em: <http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=7450>. Citado 2 vezes nas páginas 25 e 41.

VIEIRA, G. M. D.; BUZATO, L. E. The performance of Paxos and Fast Paxos. In: *SBRC '09: Proc. of the 27th Brazilian Symposium on Computer Networks and Distributed Systems*. Recife, Brasil: [s.n.], 2009. p. 291–304. Disponível em: <http://www.ic.unicamp.br/~gdvieira/publications/>. Citado na página 26.

VIEIRA, G. M. D.; BUZATO, L. E. *Implementation of an Object-Oriented Specification for Active Replication Using Consensus*. [S.l.], 2010. Disponível em: <http://www.ic.unicamp.br/~reltech/2010/abstracts.html>. Citado 8 vezes nas páginas 25, 41, 49, 51, 54, 59, 62 e 66.

VILAÇA, R.; PEREIRA, J.; OLIVEIRA, R. On the cost of database clusters reconfiguration. *28th IEEE International Symposium on Reliable Distributed Systems*, 2009. Citado 4 vezes nas páginas 27, 39, 44 e 47.