Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Diversidade em Sistemas de Quóruns Bizantinos

Cayke Gabriel dos Santos Prudente

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2018

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Diversidade em Sistemas de Quóruns Bizantinos

Cayke Gabriel dos Santos Prudente

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. Edison Ishikawa
CIC/UnB

MsC. Lucas Maciel Vieira
CIC/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 05 de julho de 2018

Dedicatória

Dedico este trabalho e a conclusão do meu curso aos meus pais, André e Fabiane, à minha namorada Luísa, às minhas irmãs, Gabriele e Ana Júlia, e à minha avó Ione. Agradeço todo incentivo desde o início da caminhada até os dias atuais. Sem eles, esse momento não seria possível.

Agradecimentos

Agradeço ao meu orientador Dr. Eduardo Alchieri pela oportunidade e ensinamentos ao longo deste projeto e aos meus colegas de curso que tornaram a jornada mais tranquila e divertida.

Resumo

Vulnerabilidades podem comprometer todo um sistema quando adequadamente exploradas por um atacante, fazendo com que haja um prejuízo financeiro e até social a depender da aplicação. Uma forma de mitigar riscos de indisponibilidade do sistema em caso de falhas ou ataques são os sistemas de quóruns. Esses sistemas geralmente compreendem um conjunto estático de servidores que fornecem um registro tolerante a falhas de leitura/escrita acessado por um conjunto de clientes. Porém como os nós dos sistemas de quóruns são réplicas uns dos outros, possíveis falhas e vulnerabilidades presentes em uma instância do sistema estarão presentes em todo sistema. Para evitar esse problema há a possibilidade da implementação de diversidade, trazendo para o sistema alta disponibilidade e tolerância a falhas devido ao uso de sistema de quóruns e diversidade conjuntamente. O presente projeto visa aumentar a segurança de replicação através de sistemas de quóruns Bizantinos, por meio do emprego de diversidade na implementação das réplicas que suportam o sistema.

Palavras-chave: Sistemas de quóruns, falhas bizantinas, diversidade

Abstract

Vulnerabilities can compromise a system when correctly explored by an attacker. It can lead to a financial loss and even a social loss depending on the application. One way to defend your system against unavailability caused by crashes or atacks are quoruns systems. These system compreends a group of servers that have one fault tolerant register for write/read operations and can be accessed by a group of clients. Because systems are replicas of each other, possible failures and vulnerabilities that exists on an instance of the system should be present on the whole system. To avoid that problem, there is the possibility to implement diversity on the system, making it highly available and fault tolerant due to the use of quoruns system and diversity together. This project aims to increase the security of replication through Bizantine Quoruns Systems using diversity when implementing the replicas of the system.

Keywords: Quorum system, Byzantine failure, diversity

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API Application Programming Interface.

CFT Crash Fault-Tolerance.

COTS Commercial O-The-Shelf.

DQS Dissemination Quorum Systems.

IDL Interface Definition Language.

MOM Message-Oriented Middleware.

MQS Masking Quorum Systems.

Q-RPC Quorum Remote Procedure Call.

RPC Remote Procedure Call.

Capítulo 1

Introdução

Este capítulo tem como objetivo apresentar a motivação para o desenvolvimento desse trabalho, abordar os objetivos do mesmo e descrever a organização de todo o texto.

1.1 Motivação

Vulnerabilidades, sejam de hardware ou software, sempre estiveram presentes na computação. Mesmo com a evolução das técnicas de programação, testes de sistemas e avanço dos protocolos de segurança, ainda não foi possível criar um sistema totalmente livre de vulnerabilidades e falhas. Com a presença cada vez maior da tecnologia no dia a dia das pessoas, vulnerabilidades podem comprometer todo um sistema, causando prejuízos financeiros e até sociais.

Com isso em mente, os programas computacionais devem ser construídos de forma a tolerarem falhas inesperadas na execução. Sistemas computacionais confiáveis são aqueles que garantem seu funcionamento correto mesmo com a falha de um ou mais atributos, como hardware, software e rede. Funcionar corretamente significa que o sistema mantém suas propriedades como disponibilidade, integridade e confidencialidade [1].

Para tolerar falhas os sistemas devem ser desenvolvidos utilizando uma ou mais soluções *Crash Fault-Tolerance (CFT)*. Porém, tal domínio de soluções resiste apenas a falhas em que um componente fica indisponível e deixa de se comunicar com o restante do sistema, não tolerando o caso em que o componente passa a agir de forma maliciosa. Para tolerar tal ação deve-se utilizar uma abordagem tolerante a falhas bizantinas [2].

Uma abordagem muito utilizada para tolerar falhas, tanto crash quanto bizantinas, são os sistemas de quóruns bizantinos [3]. Esse consiste de um sistema de réplicas que possuem um registrador que permite a leitura e escrita de dados. Os sistemas de quórum bizantinos garantem consistência e disponibilidade mesmo na ocorrência de falhas bizantinas em alguma das réplicas.

Embora existam muitas propostas de sistemas e protocolos sobre sistemas de quórum bizantinos, esse sistema possui uma limitação visível, a réplica de falhas e vulnerabilidades. Ao replicar um sistema como um servidor, todos as características são as mesmas em todos os nós. Um atacante pode facilmente comprometer todas as instâncias ao descobrir alguma vulnerabilidade em apenas uma delas, ou um *bug* de linguagem ou de programação pode facilmente afetar todo o sistema.

Para evitar esse cenário, seria necessário desenvolver um sistema de quóruns utilizando diferentes linguagens. A técnica de desenvolver sistemas com diferentes técnicas para minimizar os riscos de falhas e vulnerabilidades é conhecida como diversidade. Apesar de existirem muitas propostas e protocolos sobre sistemas de quórum bizantinos, nenhum deles fornece suporte para aplicação de diversidade nas réplicas [4, 5, 6, 7, 8]. Ou seja, implementar as réplicas em diferentes linguagens de programação, aumentando o grau de independência do sistema. Dessa forma, uma falha não seria replicada para todo o sistema, já que duas ou mais réplicas em diferentes linguagens não falharão pelo mesmo motivo.[7]

1.2 Objetivos

1.2.1 Geral

O principal objetivo deste trabalho é aumentar a confiabilidade dos sistemas de quóruns bizantinos por meio do emprego de diversidade na implementação das réplicas. Propõe-se através do presente trabalho mitigar riscos na utilização de réplicas. Sendo assim, uma vulnerabilidade presente em uma instância não será uma vulnerabilidade que poderá ser explorada em todo o sistema e um bug em uma instância não comprometerá todo o sistema.

1.2.2 Específicos

Os objetivos específicos deste trabalho são listados a seguir:

- Estudar os conceitos de sistemas de quórum, falhas bizantinas e diversidade.
- Propor uma arquitetura capaz de suportar diversidade na implementação das réplicas de um sistema de quóruns bizantinos, bem como na implementação dos clientes. A arquitetura deve possibilitar a comunicação entre diferentes linguagens de programação e apresentar baixa perda de desempenho devido a comunicação entre diferentes ambientes de execução.
- Implementação dos servidores em diversas linguagens.

- Implementação de clientes em diversas linguagens.
- Execução de experimentos nos mais variados cenários com o objetivo de verificar o impacto no desempenho causado pelas soluções propostas.

1.3 Organização do texto

O trabalho foi divido em 5 capítulos. Neste foram introduzidas as motivações e objetivos para realização desse trabalho. No segundo capítulo é apresentado o referencial teórico para obter-se o entendimento necessário para a análise do projeto. No capítulo 3 é detalhada a arquitetura do sistema desenvolvido para apresentar-se os experimentos e resultados no quarto capítulo. Por fim, no último capítulo são feitas as conclusões do trabalho.

Capítulo 2

Sistemas Distribuídos

Esse capítulo apresenta uma visão sobre Sistemas Distribuídos apresentando seu conceito e suas principais características. Também serão abordados os desafios presentes na área.

Sistemas de computação estão sempre em evolução. De 1945 a 1985 os computadores eram enormes, ocupavam salas inteiras e eram muito caros. Mesmo com a criação do microcomputador, o custo ainda era de milhares de dólares. Como resultado disso, as empresas possuíam apenas um punhado de computadores e por falta de uma forma de conectá-los, eles operavam independentemente uns dos outros.

A partir dos anos 80, houve grandes avanços e foi possível o desenvolvimento de duas tecnologias que mudaram a situação descrita acima. A primeira foi a criação do microprocessador. Com ele foi possível oferecer o poder de computação de um mainframe por uma fração do preço. A segunda foi a invenção das redes de computadores de alta velocidade. As redes locais, ou LANs, permitiram que centenas de máquinas no mesmo espaço físico pudessem ser interligadas e transmitissem dados entre si. Redes de área ampla, ou WANs, possibilitaram que máquinas ao redor do globo terrestre fossem conectadas e trocassem dados.

O resultado da utilização dessas tecnologias é a viabilidade e facilidade da criação de sistemas computacionais compostos por uma grande número de máquinas conectadas por uma rede de alta velocidade, os chamados sistemas distribuídos.

Tanenbaum define os sistemas distribuídos como: "Um sistema distribuído é um conjunto de computadores independentes entre si que se apresenta a seus usuários como um sistema único e coerente"[9]. Uma outra definição, por Coulouris é: "Um sistema distribuído é aquele no qual os componentes de hardware ou software, localizados em computadores interligados em rede, comunicam-se e coordenam suas ações apenas passando mensagens entre si"[10].

A partir das definições percebe-se dois aspectos, os computadores participantes do sistema devem ser autônomos e os usuários creem estar lidando com um sistema único,

há uma transparência para o usuário. Esses pontos chaves moldaram as principais características dos sistemas distribuídos. Independente de existir grandes diferenças entre computadores, seja de hardware, software ou a maneira que eles se comunicam-se entre si, esses detalhes são em grande parte escondidas do usuário. Usuários e aplicações podem interagir com um sistema distribuído de maneira consciente e uniforme, independentemente de quando e onde são feitas as interações.

Por princípio, sistemas distribuídos devem ser fáceis de expandir e escalar. Também deve estar disponível continuadamente mesmo que alguma parte esteja temporariamente indisponível.

2.1 Desafios

A partir das definições e características abordadas anteriormente, percebe-se que existem diversos desafios e possíveis problemas na implementação de um sistema distribuído. Nessa seção, abordam-se os quatro principais desafios para a construção de um sistema distribuído [9, 10]. Um sistema distribuído deve disponibilizar recursos de maneira fácil; deve esconder que os recursos estão distribuídos através da rede; deve ser aberto; e deve ser escalável.

2.1.1 Heterogeneidade

O principal objetivo dos sistemas distribuídos é facilitar o acesso dos usuários a recursos remotos e o compartilhamento dos mesmos de maneira eficiente e controlada. Recursos podem ser impressoras, computadores, dados, arquivos, entre outros. Porém, ambas as definições de sistemas distribuídos analisadas nesse trabalho não possuíam premissa alguma sobre os tipos de computadores que compõem o sistema. Ou seja, um sistema distribuído pode possuir computadores com hardwares totalmente distintos. A variedade também pode ocorrer em outros aspectos como rede, sistema operacional, linguagens de programação, implementações de diferentes desenvolvedores, etc.

Todas as variações citadas acima podem ocorrer dentro de um sistema distribuído e o mesmo deve ser implementado de maneira que a comunicação ocorra entre todos os nós do sistema sem perdas. Os protocolos de redes são utilizados para tornar a comunicação entre todos os entes transparente. No entanto, algumas vezes é necessário o uso de uma camada entre os clientes/aplicações com os servidores para assegurar a coerência na comunicação.

Para suportar computadores e redes heterogêneas através de uma interface única, os sistemas são organizados com uma camada logica de software que irá fazer a comunicação com o mundo externo. Essa camada também é conhecida como middleware. O middleware é um conjunto de funções e padrões que atua oferecendo uma abstração para

a comunicação e representação dos dados, permitindo que aplicações executando em diferentes plataformas se comuniquem de maneira transparente. Na Figura 2.1 pode-se observar a estrutura de um sistema distribuído que utiliza middleware.

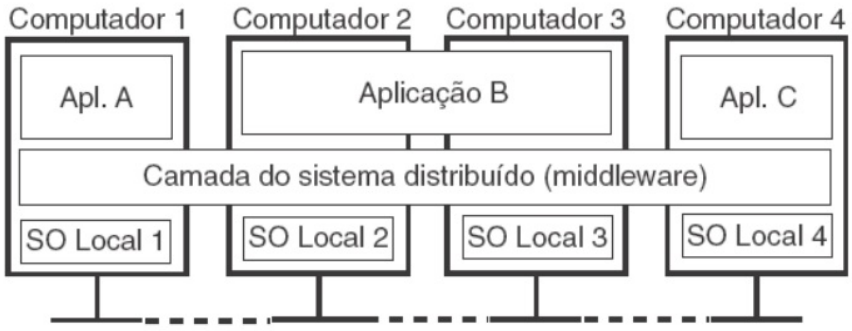


Figura 2.1: Sistema distribuído organizado como middleware. A camada de middleware se estende por várias máquinas e oferece a mesma interface para cada aplicação (Fonte: [9]).

2.1.2 Transparência

Como dito na seção anterior, o sistema distribuído deve ser transparente para os usuários e aplicações. Para Tanembaum, um sistema distribuído é dito transparente quando o mesmo é capaz de se apresentar como um único sistema computacional para os usuários e aplicações [9].

Um sistema distribuído transparente deve ocultar o fato de que seus processos e recursos estão separados fisicamente uns dos outros e prover uma ideia de que é um único e simples sistema. Existem diversos tipos de transparência dentro da área de sistemas distribuídos, alguns conceitos foram mapeados e detalhados na Tabela 2.1.

Transparência de acesso diz respeito ao tratamento de dados e em como um recurso é acessado por um usuário. É desejável esconder diferenças que podem existir devido a diferentes arquiteturas de hardware e sistemas operacionais. Diferentes nós do sistema distribuído podem ter diferentes sistemas de arquivos, que não necessariamente utilizaram a mesma convenção para os caminhos dos arquivos. Diferenças de nomes bem como de formas de manipular os arquivos devem ser transparentes para os usuários e aplicações.

Outro importante grupo é a transparência de localização. Essa transparência refere-se a impossibilidade dos usuários dizerem onde um recurso está localizado fisicamente no sistema. A nomeação tem um papel importante na utilização da transparência de localização. A transparência citada só pode ser alcançada com a atribuição de no-

Tabela 2.1: Diferentes formas de transparência em um sistema distribuído (ISO,1995)

Transparência	**Descrição**
Acesso	Ocultar diferenças em representações de dados e em como um recurso é acessado
Localização	Ocultar onde um recurso está alocado
Migração	Ocultar que um recurso pode ser movido para outro local
Relocalização	Ocultar que um recurso pode ser movido para outro local enquanto em uso
Replicação	Ocultar que um recurso é uma réplica
Concorrência	Ocultar que um recurso pode ser compartilhado por diversos usuários
Falha	Ocultar a falha e recuperação de um recurso

mes lógicos a recursos. Um exemplo de nomes lógicos são as URLs, ao acessar o site https://www.instabuy.com.br/index.html não é possível saber a localização do servidor da Instabuy. Também não é possível saber se o arquivo index.html sempre esteve no diretório atual ou foi movido recentemente. Sistemas distribuídos que permitem mover recursos sem afetar o acesso de terceiros possuem transparência de migração. Um caso ainda mais especifico é mover um recurso enquanto o mesmo está em uso por um usuário ou aplicação e não há percepção de alteração pelo mesmo, nesse caso o sistema suporta transparência de relocalização. Outro exemplo de relocalização acontece quando uma pessoa está utilizando a rede wireless da universidade enquanto se move pelo campus e nunca é derrubado da rede apesar de estar constantemente trocando de ponto de acesso.

Transparência de replicação é muito importante em sistemas distribuídos. Alguns recursos podem ter de ser replicados para aumentar a disponibilidade ou para aumentar o desempenho da aplicação, como quando um sistema copia um arquivo para um servidor mais próximo ao cliente para otimizar o tempo de acesso. Como estamos replicando recursos, é necessário que também haja a transparência de localização ao se utilizar transparência de replicação, ou seria impossível referenciar as réplicas em diferentes locais.

Já a transparência de concorrência é quando usuários compartilham o mesmo recurso. Por exemplo quando dois usuários estão acessando a mesma tabela de um banco de dados. É importante que a transparência mantenha a consistência do recurso. Para evitar a

criação de inconsistências pode-se utilizar mecanismos para garantir o acesso exclusivo ao recurso, como locks e semáforos.

Por fim, um sistema distribuído transparente a falha significa que o usuário não percebe que um recurso falhou na sua execução e que o sistema subsequentemente se recuperou da falha. O usuário nem mesmo desconfia que houve algum problema durante algum passo da execução.

2.1.3 Abertura

Um sistema distribuído é considerado aberto quando ele oferece seus recursos seguindo regras definidas sobre a semântica e sintática para um terceiro consumir seus recursos[9]. O sistema deve prover as regras para que a comunicação seja feita, especificando formato e conteúdo das mensagens trocadas. Essas regras para comunicação são conhecidas como protocolo.

Um protocolo é um conjunto de regras e convenções estritamente definidas para que a comunicação entre duas ou mais partes seja feita de maneira correta, a fim de permitir a troca de informações entre os mesmos. Em sistemas distribuídos, os protocolos são definidos e após isso cria-se normalmente uma interface para comunicação.

As interfaces para comunicação são conhecidas como Application Programming Interface (API). Uma API é um conjunto particular de regras e especificações que os sistemas computacionais devem seguir para se comunicar. Ele serve como uma interface entre diferentes programas de software e facilita sua interação. Permite que programas em diferentes linguagens possam trocar informações.

Essas interfaces devem ser documentadas e os sistemas devem ser implementados seguindo as regras das mesma. A documentação costuma ser feita utilizando Interface Definition Language (IDL). A IDL descreve com precisão os nomes das funções, tipos de parâmetros, métodos que devem ser utilizados, valores de retorno, possíveis exceções que podem ser lançadas, entre outros. A descrição é feita de maneira a ser independente de linguagens de programação. Pode-se usar linguagem natural ou pseudocódigo, conforme observado na Figura 2.2. A partir disso cada sistema deve implementar o protocolo utilizando a sua linguagem de programação própria. Dessa forma consegue-se integrar por exemplo um servidor escrito em C++ com um cliente Java.

Outra característica de um sistema distribuído aberto é que o sistema deve ser fácil de configurar utilizando diferentes componentes [9]. O sistema deve permitir a adição ou remoção de componentes sem que isso afete seu comportamento. Um sistema aberto deve ser flexível, deve funcionar ao alterar-se o sistema operacional, sistema de arquivos, hardware, etc.

Criando uma transação

Para fazer uma cobrança, você deve usar a rota `/transactions` para criar sua transação, que pode ser feita por cartão de crédito ou por boleto bancário.

Parâmetro	Descrição
api_key obrigatório	Chave da API (disponível no seu dashboard)
amount obrigatório	Valor a ser cobrado. Deve ser passado em centavos. Ex: R$ 10.00 = `1000`
card_hash obrigatório*	Informações do cartão do cliente criptografadas no navegador. **OBS**: Apenas para transações de **cartão de crédito** você deve passar **ou** o `card_hash` **ou** o `card_id`
card_id obrigatório*	Ao realizar uma transação, retornamos o `card_id` do cartão para que nas próximas transações desse cartão possa ser utilizado esse identificador ao invés do `card_hash`
payment_method default: `credit_card`	Aceita dois tipos de pagamentos/valores: `credit_card` e `boleto`
postback_url	Endpoint do seu sistema que receberá informações a cada atualização da transação. Caso você defina este parâmetro, o processamento da transação se tornará assíncrono.
async default: `false` ou `true` caso utilize `postback_url`	Utilize `false` caso queira utilizar POSTbacks e manter o processamento síncrono de uma transação.
installments mínimo: 1, máximo: 12	Se o pagamento for boleto, o padrão é 1
boleto_expiration_date default: data atual + 7 dias	Prazo limite para pagamento do boleto
boleto_instructions default: `null`	Campo instruções do boleto. Máximo de 255 caracteres
soft_descriptor	Descrição que aparecerá na fatura depois do nome da loja. Máximo de 13 caracteres
capture default: `true`	Após a autorização de uma transação, você pode escolher se irá capturar ou adiar a captura do valor. Caso opte por postergar a captura, atribuir o valor `false`

Figura 2.2: Documentação da API de um gateway de pagamentos eletrônicos.

2.1.4 Escalabilidade

Escalabilidade é um dos mais importantes objetivos a serem alcançados na construção de um sistema distribuído. A escalabilidade pode ser mensurada em três aspectos: tamanho, geografia e administração [11].

Um sistema é escalável em tamanho quando pode-se aumentar o número de usuários e recursos do sistema. Um exemplo atual de sistema escalável de fácil uso são os servidos da Amazon AWS. Na AWS existe a possibilidade de se fazer auto escala, em que o serviço automaticamente aloca mais capacidade de acordo com a demanda dos seus usuários. O aumento da capacidade pode ser desde alocar mais recursos na máquina atual até a disponibilidade de mais máquinas executando seu sistema.

Usuários e recursos podem estar distantes uns dos outros, com a Internet é possível acessar sistemas em diferentes continentes. Porém ao acessar um recurso distante pode haver um maior tempo de resposta. Para melhorar esse cenário os sistemas devem ser escaláveis geograficamente, ou seja, deve ser possível distribuir os recursos em diferentes pontos geográficos. Grandes empresas como o Facebook possuem servidores espalhados por todo o globo terrestre. Isso melhora a experiência dos usuários, tornando o acesso muito mais rápido. Estando em Brasília, é muito mais rápido obter a resposta de um servidor em São Paulo do que um servidor na América do Norte.

Por último, um sistema pode ser administrativamente escalável. Para isso, deve ser fácil de gerenciar mesmo que haja muitas instâncias. Infelizmente, é difícil escalar um sistema distribuído nos 3 aspectos. Em geral, ao aumentar o tamanho do sistema ou replicá-lo em diferentes locais, torna-se mais complexa a administração do mesmo.

Um problema de se escalar sistemas distribuídos são os serviços centralizados. Muitas vezes os serviços são implementados de forma a serem executados em uma máquina especifica do sistema distribuído. Porém ao executar em uma máquina específica, ao aumentar-se o número de usuários cria-se um gargalo na aplicação. Há um limite para o número de acessos a uma máquina, mesmo que possua o melhor hardware disponível no mercado.

Para evitar maiores problemas ao escalar os sistemas, deve-se utilizar algoritmos descentralizados na implementação. Segundo Tanenbaum [9], os algoritmos descentralizados possuem quatro características que os diferem dos algoritmos centralizados:

- Nenhuma máquina possui informações completas sobre o estado do sistema.
- As máquinas tomam decisões com base apenas em informações locais.
- Falha em uma máquina não arruína o algoritmo.
- Não há suposição da existência de um relógio global.

2.2 Concorrência

Sistemas distribuídos podem ser acessados simultaneamente por diversos clientes, devido a isso é necessária uma atenção para evitar possíveis problemas. Por permitir o acesso simultâneo, alguns recursos do sistema podem ter de ser compartilhados entre os usuários. Os recursos podem variar desde partes físicas (como impressoras) até recursos lógicos(como arquivos, dados em memória). O consumo de recursos deve ser controlado para evitar que os mesmos tenham comportamentos inesperados.

Uma aplicação distribuída deve ser desenvolvida de maneira que vários clientes possam se conectar ao mesmo tempo a ela e o resultado das operações não deve apresentar

inconsistências. Deve ter o mesmo comportamento quando acessado por um cliente ou por vários clientes simultaneamente. Para isso torna-se necessária a utilização de threads e mecanismos de controle de concorrência.

Por padrão, um servidor deve criar uma nova thread sempre que um cliente novo conecta-se a ele. Com a criação de diversas threads os clientes não ficam travados esperando o término da operação de outro, neste caso a execução é concorrente. Cada nova thread é responsável pelo cliente para qual foi criada, porém as threads compartilham os recursos do processo pai.

Como são executadas concorrentemente, pode-se haver inconsistência em operações caso duas ou mais threads estejam fazendo operações de escrita/leitura no mesmo dado. Por exemplo, um cliente A está lendo um arquivo e antes de terminar a leitura, um cliente B faz uma alteração no mesmo, e por fim, o cliente A finaliza a leitura. Percebe-se no caso citado que o cliente A terá uma inconsistência. O dado lido do arquivo será diferente do dado ao iniciar-se a leitura. Existem diversos erros que podem ser causados pelo acesso mutuo a recursos computacionais.

Para evitar as inconsistências, é necessário controlar o acesso as regiões críticas garantindo a exclusão mútua, evitando possíveis condições de corrida [12]. O controle das regiões deve ser feito na implementação do sistema, para tal utilizam-se diversos mecanismos de sincronização, sendo os mais populares locks e semáforos.

Contudo, a má utilização dos mecanismos de sincronização frequentemente ocasiona outros problemas, podendo até mesmo congelar o progresso de um programa. Esse problema é conhecido como deadlock [13]. Ocorre quando um membro A tenta alocar um recurso já reservado pelo membro B, e o membro B tenta alocar um recurso já reservado por A. Ambos ficam então travados numa espera eterna.

Um sistema concorrente deve garantir a consistência dos seus dados ao mesmo tempo que garante o progresso da aplicação. A consistência é garantida quando todos os usuários do sistema possuem um visão consistente dos dados, ou seja, operações realizadas por outros usuários devem produzir resultados que satisfazem os critérios de correção dos dados.

2.3 Comunicação

Um ponto chave dos sistemas distribuídos é a comunicação. Processos executando em diferentes máquinas devem conseguir se comunicar e trocar informações. A comunicação em sistemas distribuídos baseia-se na transmissão de mensagens de baixo nível através da rede subjacente [9]. Devido a ausência de memória compartilhada entre os nós, toda a comunicação é baseada em enviar e receber mensagens.

Para um processo A enviar uma mensagem para um processo B, o primeiro deve montar a mensagem e então fazer uma chamada de sistema para que o sistema operacional envie a mensagem para B através da rede. Para que a comunicação seja possível, é necessário que os processos utilizem as mesmas estruturas de comunicação, ou seja, utilizem o mesmo protocolo para que a comunicação faça sentido para ambas as partes.

A comunicação pode ser ou não, orientada a conexão. No caso de uma comunicação orientada a conexão, antes de enviar a mensagem, os processos envolvidos devem estabelecer uma conexão explícita entre eles. Com a conexão estabelecida, os dados podem ser enviados. Ao finalizar a troca de mensagens, deve-se encerrar a conexão.

Existem diversos modelos para comunicação via troca de mensagens, sendo os mais conhecidos Remote Procedure Call (RPC) e Message-Oriented Middleware (MOM).

Remote Procedure Call (RPC)

Muitos sistemas distribuídos são baseados na troca explícita de mensagens. Os procedimentos de leitura e escrita não escondem a comunicação entre as partes, que é importante para alcançar a transparência de acesso dos sistemas distribuídos. Para alcançar tal transparência, pode-se utilizar o Remote Procedure Call (RPC)[9].

O método RPC é definido pelo cenário observado na Figura 2.3. Um processo A chama um procedimento de um processo B, o processo A é suspenso, e a execução do procedimento ocorre em B. A informação é transportada de A para B através de parâmetros definidos no protocolo, e B retorna o resultado para A. Nenhuma troca de mensagem é visível para o usuário.

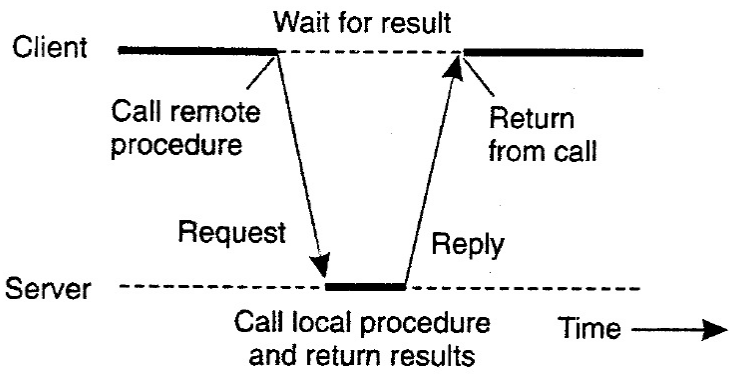


Figura 2.3: Principio RPC aplicado a um cliente e servidor. (Fonte: [9]).

Message-Oriented Middleware (MOM)

RPCs contribuem em esconder a comunicação em sistemas distribuídos, trazendo uma transparência de acesso ao mesmo. Porém, esse mecanismo nem sempre é o mais apropriado. Principalmente quando não há garantia que o servidor executa as requisições imediatamente após o recebimento, é necessário utilizar outro método de comunicação. Nesses casos, a natureza síncrona do RPC, onde um cliente fica bloqueado até a resposta do procedimento, precisa ser substituída por algo assíncrono como as trocas de mensagem.

Muitos sistemas distribuídos e aplicações são construídas em cima do modelo de troca de mensagens oferecida pela camada de transporte. Para isso, utilizam-se sockets para a comunicação entre processos. Socket é um end-point de comunicação onde um processo pode escrever e ler dados utilizando a rede.

Na comunicação orientada a mensagem utilizando sockets, tanto clientes quanto servidores necessitam ter seus próprios sockets, porem cada um irá implementar suas particularidades, conforme a Figura 2.4.

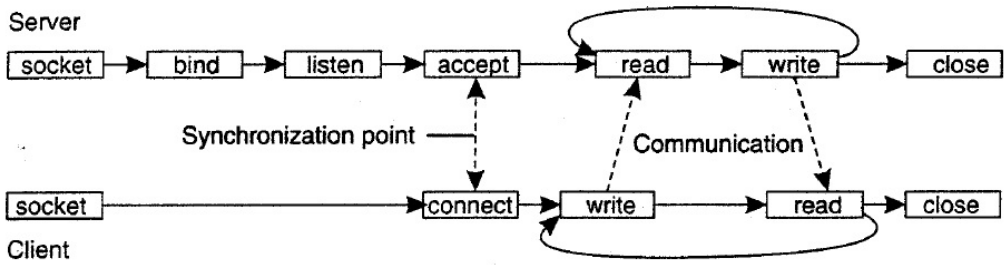


Figura 2.4: Comunicação orientada a conexão(TCP) utilizando sockets. (Fonte: [9]).

Antes da troca de mensagens, o servidor deve executar quatro passos. No primeiro, deve criar o socket e determinar qual protocolo será utilizado, UDP ou TCP. Após isso, o servidor deve dar um bind no socket, ou seja, deve registrar um endereço para o socket, que será formado pelo endereço IP da máquina atual mais uma porta escolhida. Em seguida o servidor fica em estado de escuta, na espera de um cliente. Quando um cliente se conecta ao socket, o servidor deve aceitar o estabelecimento de uma conexão entre ambos, caso utilize o TCP.

Após o accept, costuma-se criar uma nova thread no servidor para tratar a comunicação que virá a seguir. Dessa forma o servidor continua na fase de listening, podendo receber novos clientes, ao mesmo tempo em que comunica-se com um cliente já conectado.

Com a conexão realizada, cliente e servidor conseguem escrever no socket do outro e ler seu próprio socket. Dessa forma, as mensagens são trocadas entre processos através

dos sockets. Como já dito anteriormente, essas mensagens precisam seguir um padrão para que ambos consigam entender o significado de cada dado enviado e possam fazer as operações necessárias para tal. Após o termino da comunicação, os sockets são fechados e a conexão é interrompida. Possíveis threads criadas são eliminadas nesse momento.

Na parte do cliente, tem-se menos passos para a comunicação. O cliente cria o socket com o endereço IP e porta do servidor, após isso ele conecta-se com o servidor. Após a conexão pode-se criar uma thread para fazer a comunicação, mantendo dessa forma o cliente não bloqueado.

2.4 Falhas

Uma característica dos sistemas distribuídos é a possibilidade de falhas parciais. Uma falha parcial ocorre quando um componente do sistema distribuído apresenta comportamento falho. Apesar da ocorrência da falha, o sistema pode continuar funcionando, visto que a falha ocorreu em uma parte do sistema. Já em sistemas não distribuídos, uma falha provavelmente ocasiona a perda de todo serviço.

É importante desenvolver o sistema distribuído de maneira que o mesmo possa se recuperar de possíveis falhas individuais. A falha parcial não deve prejudicar o sistema como um todo, o mesmo deve continuar operando de maneira satisfatória. Existem diversos tipos de falhas e maneiras de contorná-los. Aborda-se a seguir alguns casos mais comuns.

2.4.1 Conceitos básicos

Para entender a importância da tolerância a falhas em sistemas distribuídos, precisamos definir o que significa um sistema tolerante a falhas. Um sistema tolerante a falhas é também conhecido como um sistema fidedigno. Um sistema fidedigno possui cinco características: confiabilidade, disponibilidade, segurança, integridade e manutenibilidade [1].

Disponibilidade (*Availability*)

É definida como a propriedade em que o sistema sempre está pronto para ser utilizado imediatamente. Refere-se a probabilidade do sistema estar operando corretamente em qualquer instante de tempo e está disponível para executar funções para seus usuários. Um sistema altamente disponível é aquele que sempre está funcionando em um dado instante de tempo.

Confiabilidade (*Reliability*)

É definida como a propriedade em que o sistema pode executar continuamente sem falhar. Ao contrario da disponibilidade, a confiabilidade é medida em um dado intervalo de tempo em vez de um instante no tempo. Um sistema confiável é aquele que produz resultados sem falhas durante um longo período de tempo.

Segurança (*Safety*)

Refere-se a situação em que caso haja uma falha no sistema, nada catastrófico acontece por tabela. Caso uma parte do sistema falhe, o sistema deve conseguir se recuperar do erro.

Manutenibilidade (*Maintainability*)

Está relacionada com a facilidade de reparar erros após uma falha do sistema. Um sistema com alto grau de manutenibilidade em geral apresenta uma alta disponibilidade, especialmente quando falhas são automaticamente encontradas e corrigidas.

Integridade (*Integrity*)

O sistema sempre apresenta as características originais estabelecidas na concepção do mesmo. Não ha existência de estados impróprios e o estado do sistema só é alterado nas execuções de operações corretas.

Como outras características dos sistemas distribuídos já descritas no presente trabalho, percebe-se que nem sempre é possível desenvolver sistemas com todos os atributos desejáveis. Dificilmente um sistema apresentará as cincos características citadas acima conjuntamente, visto que algumas podem ser um pouco conflitantes entre si.

Um sistema falha quando não consegue entregar o esperado. Por exemplo, se um sistema promete um ou mais serviços ao seus clientes e não consegue prover algum deles, é dito que o sistema falhou. Um erro (*error*) é um potencial causador de um defeito (*failure*), sendo que a causa de um erro é definida como falha (*fault*) [1].

2.4.2 Falhas, erros e defeitos

Para entender melhor as diferenças entre falhas, erros e defeitos é necessário fazer uma análise de conceito individual e após isso uma análise da relação entre eles.

- **Falha (*fault*)**: causa real de um erro.
- **Erro (*error*)**: propriedade do estado do sistema que difere do estado esperado do mesmo.

- **Defeito (*failure*)**: ocorre quando o sistema se desvia do comportamento designado em sua especificação, ou seja, o sistema deixa de prover um serviço desejado.

Uma falha é dita ativa quando ele é responsável por produzir um erro, caso contrário é dita dormente. As falhas podem alternar seus estados entre ativas e dormentes, a depender das condições da execução do programa.

Um erro inicial pode ser passado entre diversos componentes e até causar erro em um componente diferente da sua origem. Por exemplo no caso em que um serviço B utiliza um dado fornecido pelo serviço A para realizar uma operação e retornar o resultado da mesma para o usuário. O usuário receberá o dado incorreto e terá a impressão de que o serviço B está com problemas, quando o problema era em A. Esse caso é conhecido como propagação de erros.

Já o defeito é fruto de um erro ou uma propagação de erros. O defeito ocorre quando um serviço retorna um valor que difere do correto e esperado. Diversos erros podem ser gerados antes que um defeito ocorra. Um sistema pode propagar um erro para outro, gerando assim defeitos nos outros. Falhas, erros e defeitos estão ligados em cadeia. A existência de um pode causar o outro, conforme observa-se na Figura 2.5. Uma falha observada em uma parte de um sistema distribuído não necessariamente nasceu no sistema em questão, visto que ela pode estar sendo propagada através do sistema.

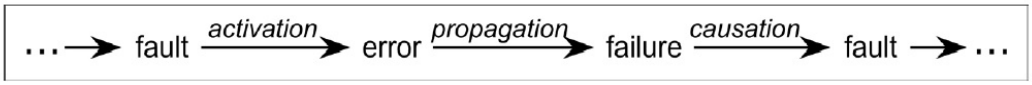


Figura 2.5: A cadeia fundamental das falhas. (Fonte: [1]).

2.4.3 Tipos de falhas

Os sistemas computacionais são suscetíveis a falhas. Mesmo com o avanço da tecnologia e das técnicas de programação, ainda é inviável a construção de um programa integralmente sem falhas. Em sistemas distribuídos essas falhas podem ocorrer em qualquer parte do sistema, e muitas vezes gerando falhas em outros componentes que dependiam dos servidores do sistema falho.

Apesar de uma falha poder gerar erros e até defeitos, nem sempre isso acontece. De acordo com sua duração e ocorrências, as falhas são classificadas em: transientes ou permanentes.

- **Falhas Transientes**: são as falhas de duração limitada, causadas por alguma situação temporária ou um interferência externa. Podem ocorrer por um tempo e depois sumirem.
- **Falhas Permanentes**: são falhas que ocorrem sempre a partir do momento que um componente falha.

Para obter uma melhor compreensão da gravidade das falhas, foram propostas outras classificações ao longo dos anos, de acordo com o comportamento das mesmas. Uma das mais utilizadas é a feita por Hadzilacos e Toueg (1993) e descrita a seguir:

- **Falhas de Travamento (Crash Faults)**:Um servidor para, mas estava funcionando corretamente até o travamento.
- **Falhas de Omissão (Omission Faults)**: Um servidor falha para responder a uma requisição.
- **Falhas de Timing (Timing Faults)**: A resposta de um servidor está fora do intervalo de tempo esperado.
- **Falhas Bizantinas (Byzantine Faults)**: Um servidor produz uma resposta arbitrária em um tempo arbitrário.

Das falhas citadas acima, as mais graves são as bizantinas. Quando uma falha arbitraria ocorre, os clientes devem estar preparados para o pior caso. O servidor pode produzir uma resposta que nunca deveria, mas que não pode ser detectada como incorreta pelo cliente. Ainda pior, um servidor pode produzir uma resposta maliciosa intencionalmente. As falhas bizantinas agrupam todos os tipos descritos anteriormente a elas, conforme observa-se na Figura 2.6.

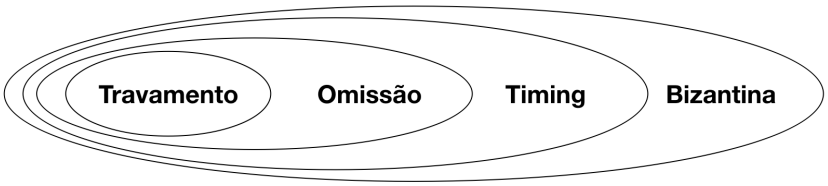


Figura 2.6: Classificação de falhas.

2.5 Segurança na execução

Como dito anteriormente, não há sistema imune a falhas. Devido a isso, é necessário criar maneiras de conviver com a falha e minimizar os possíveis efeitos colaterais. Sistemas distribuídos devem ser capazes de contornar eventuais falhas, a falha de um componente não pode afetar o funcionamento de todo sistema.

Visando manter o bom funcionamento dos sistemas, criou-se diversas práticas para lidar com as falhas, fazendo com que os sistemas se tornem mais seguros. Algumas técnicas são: prevenção de falhas, tolerância a falhas, remoção de falhas e previsão de falhas [1]. A seguir detalha-se cada uma.

Prevenção de falhas

Prevenção de falhas está diretamente relacionada com a parte de desenvolvimento do sistema. Durante a programação do mesmo, deve-se utilizar técnicas e metodologias que visem minimizar a ocorrência de falhas. Também recomenda-se a utilização de testes unitários.

Tolerância a falhas

A tolerância a falhas é alcançada através de duas técnicas: detecção de erros e recuperação do sistema. A detecção de erros é a fase onde são identificadas presenças de erros na aplicação. Após a detecção, a recuperação do sistema é feita. A recuperação consiste em transformar o estado atual (faltoso) do sistema em um estado sem erros detectados e sem falhas que possam ser ativas posteriormente.

A detecção de erro pode ser feita de duas maneiras: preemptiva ou concorrente. Na primeira, o sistema interrompe o fornecimento dos seus serviços e realiza a checagem de erros e falhas dormentes. Já na segunda a checagem é feita ao mesmo tempo que o fornecimento dos serviços. A recuperação do sistema acontece através do tratamento dos erros ou das falhas. No tratamento dos erros, elimina-se os erros do estado do sistema, enquanto no tratamento das falhas é feita a prevenção das falhas para que elas não sejam ativadas novamente.

Remoção de falhas

Pode ser feita tanto no desenvolvimento quanto durante a execução do sistema. A remoção de falhas pode ser caracterizada como preventiva ou corretiva. Na preventiva o objetivo é encontrar e remover falhas que possam causar erros durante a execução do sistema. Por outro lado, a corretiva tem como objetivo a remoção de falhas já reportadas e que causaram erros.

Previsão de falhas

É feita através da avaliação do comportamento do sistema em relação a ocorrência e ativação de falhas. Pode ser quantitativa ou qualitativa. Na quantitativa tem como objetivo identificar e classificar as falhas ou eventos que causam as mesmas. A avaliação qualitativa tem como função estimar probabilisticamente a chance de falhas.

2.6 Sistemas de Quóruns

O significado da palavra Quórum é "o número mínimo de membros de um grupo que devem estar presentes na tomada de decisões do grupo"[14]. Apesar de não existir um padrão para o tamanho dos quóruns, eles em geral são pelo menos a maioria dos membros. Maioria é um exemplo de quórum para votação de grupos. A maioria é necessária no sistema eleitoral por exemplo, e é utilizada para diminuir o particionamento e inconsistência do processo. Já em alguns ritos do legislativo brasileiro, é necessário um quórum de 2/3 dos membros da casa, como no processo de impeachment do presidente da república.

Em sistemas distribuídos, quórum significa a coleção de nós pertencentes a uma rede, que é grande o suficiente para tomar decisões evitando incoerências [15]. Na computação os quóruns são utilizados em grupos, formando os sistemas de quóruns. Segundo Vukolic, um sistema de quóruns é uma coleção de subconjuntos de nós, tipicamente servidores, de tal forma que cada par de quóruns tem uma intersecção não vazia [14].

O principal objetivo do sistema de quórum é garantir consistência. Outros objetivos relevantes são aumentar a disponibilidade, melhorar o balanceamento de carga e tolerar falhas em sistemas distribuídos. A ideia chave é que um cliente ao consumir um serviço não necessita se comunicar com todos os servidores, mas apenas com um quórum deles.

Sistemas de quóruns podem ser utilizados para implementar uma gama variada de serviços na computação distribuída. Alguns deles são: réplicas de bancos de dados, exclusão mutua, leitura/escrita em registrador e comunicação de grupo. No presente trabalho adotaremos o uso da leitura/escrita em registrador.

Para um simples exemplo de como o algoritmo de leitura e escrita funciona, considere um sistema distribuído e um dado que está replicado em N servidores. Para fazer uma leitura, o cliente deve contactar pelo menos metade mais um dos servidores e conferir o dado enviado pelos mesmos. O dado correto é aquele que consta em pelo menos um quórum de servidores.

Para fazer uma escrita, o cliente entra em contato com metade mais um dos servidores e analisa qual o maior *timestamp* armazenado. O cliente então faz uma escrita em pelo menos um quórum de servidor com o dado desejado e o *timestamp* incrementado. O *timestamp* é utilizado pois como o sistema é assíncrono e utiliza a rede para a comunicação

não ha qualquer garantia da ordem de entrega das mensagens enviadas pelos clientes aos servidores. Com o uso do *timestamp*, o dado salvo no servidor sempre será o dado mais novo, ou seja, o dado de maior *timestamp*. Ao chegar um dado defasado, com *timestamp* menor que o salvo atualmente, o mesmo é descartado.

Destaca-se que nos sistemas de quóruns a comunicação ocorre apenas entre cliente e servidor, não havendo comunicação entre os servidores. Os sistemas de quóruns não exigem nenhuma técnica para sincronização dos servidores e estados. O próprio protocolo trata para que o sistema sempre avance, e caso um servidor fique desatualizado, é possível que o mesmo se recupere ao longo do tempo e da execução de novas requisições. Para isso alguns protocolos implementam uma fase de write-back após a leitura de um dado. Nessa fase, o cliente escreve nos servidores desatualizados o dado mais recente obtido na etapa anterior.

Nesses sistemas, falhas ocorridas em algumas das réplicas tornam-se transparentes para os clientes. Como os clientes necessitam de apenas um quórum de respostas para continuar seu processo, possíveis agentes faltosos não atrapalham o andamento do programa, pois considera-se o dado que está replicado em no mínimo um quórum de servidores.

2.6.1 Sistemas de Quóruns Bizantinos

Sistemas de quórum clássicos são aplicáveis no contexto de falhas por parada (Crash Faults). Em um contexto onde os nós podem falhar arbitrariamente, ou seja, podem apresentar falhas bizantinas, sistemas de quóruns simples não garantem a consistência dos dados [16]. Para suportar tais falhas é necessário implementar um sistema de quórum bizantino.

Em um sistema de quórum bizantino, é necessária uma fração maior de nós corretos em relação a um sistema com tolerância a falhas por parada. Os sistemas de quóruns bizantinos tem sido utilizados em aplicações de leitura/escrita assíncronas.

Malkhi e Reiter definem diversos tipos de sistemas de quóruns bizantinos, a depender do tipo de dado que a aplicação deve armazenar [3]. Os dois principais e que foram utilizados na implementação do presente trabalho foram Dissemination Quorum Systems e Masking Quorum Systems.

Dissemination Quorum Systems (DQS)

DQS [3] foram propostos com o objetivo de armazenar dados autenticados. Os dados autenticados são dados que não podem ser falsificados por servidores bizantinos. Na prática, pode-se alcançar esse recurso utilizando assinaturas digitais.

Ao fazer uma escrita no servidor, o cliente fornece além do dado e do *timestamp*, uma assinatura para o par dado e *timestamp*. Na leitura, o cliente deve conferir se a assinatura é válida para o par dado e *timestamp*. Para uma leitura retornar um resultado, a mesma deve possuir pelo menos f+1 dados válidos, onde f é o número de falhas suportadas pelo sistema.

Masking Quorum Systems (MQS)

MQS [3] foram propostos com o objetivo de armazenar dados não autenticados. Apesar de ser possível armazenar tais dados utilizando DQS, tem-se uma maior facilidade para implementar protocolos MQS para tal. Nesse caso não é necessário assinar os dados, porém os quóruns do sistema passam a ser de 2f+1.

2.7 Diversidade

Diferentes técnicas são utilizadas durante o desenvolvimento de software para minimizar os riscos de falhas e vulnerabilidades em um programa. Técnicas como verificação e validação de código, testes unitários e outros são utilizadas para mitigar os riscos. Porém necessita-se tempo e recursos para remover todas os problemas dos softwares.

Os softwares estão se tornando cada vez maiores e mais complexos. Mesmo que se utilize as melhores práticas de programação e testes, é inviável a construção de um programa imune a falhas e vulnerabilidades. Portanto, é fato que os softwares irão apresentar alguma falha e/ou vulnerabilidade durante seu uso. Os serviços devem ser entregues aos clientes mesmo que alguma parte do sistema apresente falhas. Devido a isso torna-se cada mais necessário utilizar técnicas para tolerar intrusões e falhas.

Um sistema é dito tolerante a intrusão se o mesmo consegue funcionar corretamente mesmo na falha de algum componente [6].Um exemplo de sistema tolerante a intrusão são os sistemas de quóruns bizantinos, que garantem o funcionamento correto mesmo na ocorrência de falhas arbitrárias. Porém, tais sistemas não garantem a segurança contra vulnerabilidades. Caso múltiplos componentes possuam as mesmas vulnerabilidades, um mesmo ataque pode comprometer todo o sistema.

Para reduzir a probabilidade da existência de vulnerabilidades em mais de um componente, pode-se utilizar a diversidade no design [17]. Nesse caso, cada componente utiliza diferentes softwares para executar a mesma função, com a expectativa que as diferenças irão reduzir a ocorrência de vulnerabilidades idênticas em mais de um componente.

A ideia central da diversidade de projeto é usar diferentes implementações de um mesmo sistema para alcançar a tolerância a falhas, partindo da premissa que implementações independente apresentarão falhas independentes [7]. Existem diversas formas de

implementar a diversidade, e cada uma possui suas vantagens e desvantagens. A seguir, apresenta-se as diversas formas de diversidade [7].

Implementação

A diversidade na implementação é a forma mais comum de diversidade em sistemas. Ela é mais fácil de ser feita pois o software é muitas vezes o único componente do qual tem-se total controle. Consiste em desenvolver o mesmo sistema utilizando diferentes linguagens. A grande desvantagem é que aumenta-se o custo de desenvolvimento, visto que será necessário implementar diferentes variantes de um software.

No presente trabalho utiliza-se a diversidade de implementação. As réplicas do sistema de quórum bizantino foram desenvolvidas em diferentes linguagens, mas utilizando os mesmos protocolos e algoritmos.

Administração

Na implementação de diversidade de administração os sistemas operam com diferentes administradores, os quais possuem diferentes políticas de gerenciamento de segurança. Isso busca dificultar o uso de engenharia social em ataques ao sistema.

Localização

Diversificar a localização consiste em utilizar diferentes locais físicos para instalação dos componentes do sistema. Possuir componente espalhados em diferentes áreas traz uma maior segurança contra ameaças físicas como queda de energia, roubo de equipamentos, enchentes, terremotos entre outros.

Commercial O-The-Shelf (COTS)

COTS, também conhecidos como componentes de prateleira ou componentes prontos, são bastante utilizados nos sistemas atuais. Os COTS apresentam uma boa oportunidade de implementação de diversidade a um bom custo benefício. Ela é barata e de fácil implementação, visto que a maioria dos componentes só precisam de alguns ajustes para integração no seu sistema. Alguns exemplos de COTS são: SGBDs, middlewares, VMs, compiladores e bibliotecas.

Sistema Operacional

O sistema operacional é a base de toda aplicação. Uma falha no mesmo pode fazer com que um atacante controle componentes do sistema mesmo que a aplicação não possua

vulnerabilidades. Consiste na distribuição da aplicação em diferentes sistemas operacionais. Possui facilidade de implementação caso exista alguma API padronizada que porte aplicações apenas recompilando-as no novo sistema, como o POSIX.

Métodos

Consiste em utilizar mais de um método para garantir maior segurança de algum atributo. Como exemplo, pode-se citar a autenticação em dois passos.

Hardware

A diversidade de hardware é alcançada ao se utilizar diferentes hardwares para execução do sistema. Tem como objetivo aumentar a segurança contra exploits.

Capítulo 3

Diversidade em Sistemas de Quóruns Bizantinos

O grande objetivo do presente trabalho é a implementação de diversidade em um sistema de quórum bizantino. Para isso tomou-se como referência os protocolos da seção 2.6.1 para a implementação dos sistemas de quóruns.

Esses algoritmos foram implementados de maneira idêntica, limitadas a particularidades de cada linguagem, em quatro linguagens de programação diferentes: C++, Java, Swift e Python. Definiu-se quatro linguagens, pois para os sistemas de quóruns MQS necessita-se de quóruns de tamanho 3 para tolerar 1 servidor faltoso. Para ter-se quóruns de tamanho 3, necessita-se 4 servidores executando.

Nesse capítulo aborda-se técnicas e ferramentas utilizadas para a implementação do sistema proposto no presente trabalho. O código fonte do trabalho pode ser acessado através do Github - https://github.com/Cayke/TCC - do projeto.

3.1 Visão Geral da Proposta

Propõe-se o desenvolvimento de um sistema onde diversos clientes possam fazer leituras e escrituras de dados em um registrador atômico para suportar o compartilhamento de dados abstratos. Para aumentar a confiabilidade do sistema iremos utilizar um sistema de quóruns com 4 servidores, onde cada servidor terá um registrador. Tal sistema deve sobreviver a eventuais falhas nos servidores e até mesmo clientes maliciosos.

Para isso implementou-se dois protocolos: Clientes Honestos e Clientes Desonestos. A principal diferença entre os dois é que no primeiro espera-se que os clientes sejam honestos para o bom andamento do sistema. Já no segundo, o protocolo consegue manter seu estado saudável mesmo na presença de clientes maliciosos.

Para implementar a diversidade no sistema, os servidores foram desenvolvidos para ambos os protocolos utilizando as 4 linguagens citadas acima. Durante a execução pode-se executar um servidor em cada linguagem ou qualquer combinação de uma ou mais linguagens, como por exemplo, dois servidores c++ e dois servidores java, sem qualquer perda de funcionalidade. Os clientes foram desenvolvidos em apenas duas linguagens, Java e Python, visto que o ponto chave de estudo são os servidores sendo os clientes apenas uma ferramenta para provar o funcionamento do protocolo.

Para que o sistema possa funcionar de maneira perfeita ao se utilizar diferentes linguagens, é necessário que o sistema utilize um protocolo de comunicação para que os programas possam se comunicar e avançar seus estados de aplicação. Foi definido um protocolo de comunicação e também a estrutura de dados que será enviada em toda comunicação cliente-servidor. Tais protocolos serão melhor descritos posteriormente.

3.2 Suposições

Um cliente ou servidor correto é aquele que atende a suas especificações. Aquele que desvia de suas especificações é considerado faltoso. Clientes ou servidores faltosos podem apresentar comportamentos bizantinos, incluindo colaborar com outros clientes ou servidores faltosos. Nesse trabalho assumimos que no máximo 1 servidor irá falhar.

É conveniente separar os clientes em dois grupos: os que falham benignamente e os desonestos. Os clientes desonestos são aqueles que falham e não se encaixam em falhas de crash, omissão ou timeout. Os clientes honestos são todos aqueles que não são desonestos - os corretos e os que falham benignamente.

Para se comunicar com os servidores, os clientes fazem uma Quorum Remote Procedure Call (Q-RPC). Na Q-RPC o cliente envia uma requisição para os servidores do sistema e a resposta retornada ao cliente pelo módulo Q-RPC é a resposta retorna pelo quórum necessário para dada operação.

Assume-se que cada cliente e servidor possuem um código único de identificação (ID) que é conhecido por todos e não pode ser alterado de forma alguma. Cada cliente e servidor também possui uma chave criptográfica privada a qual apenas ele tem acesso e uma chave pública criptográfica a qual todos possuem acesso. Com essas chaves os nós do sistema conseguem assinar suas mensagens quando necessário e os interessados conseguem validar a origem e autenticidade da mesma.

3.3 Protocolo Clientes Honestos

Quando todos os clientes são assumidos como honestos, o papel principal do protocolo é de garantir que servidores faltosos não irão induzir clientes honestos ao erro. É possível garantir isso ao fazer com que os clientes assinem digitalmente os seus valores de escrita. Dessa forma, ao fazer uma escrita, o cliente deverá passar tanto o dado que deseja armazenar, quanto a assinatura para aquele dado.

Esse tipo de sistema de quórum é conhecido como Dissemination Quorum Systems (DQS). Os protocolos para escrita e leitura sao apresentados com detalhes a seguir. Para o desenvolvimento deles, adotou-se como referência o protocolo "Honest Writers" [18].

Escrita

Para um cliente W escrever um valor V:

1. Faz uma Q-RPC para obter uma lista de *timestamps* de um quórum Q1 de servidores.
2. Escolhe um *timestamp* T maior que o maior encontrado na lista anterior.
3. Faz uma segunda Q-RPC para escrever um valor V e *timestamp* T, assinados pelo cliente W, num quórum Q2 de servidores.

Leitura

Para um cliente ler um valor V:

1. Faz uma Q-RPC para obter uma lista de timestamp/valor assinados de um quórum Q1 de servidores.
2. Escolhe o par cuja assinatura é valida e o *timestamp* é o maior.
3. Faz uma segunda Q-RPC para escrever (write-back) um valor V e *timestamp* T, assinados, num quórum Q2 de servidores.
4. Retorna o valor V como resultado da operação.

Um servidor salva a variável em seu registrador caso o *timestamp* da escrita seja maior que o *timestamp* armazenado atualmente. Caso o *timestamp* seja igual, o servidor irá armazenar o dado caso o ID do cliente seja maior que o ID do cliente do dado salvo atualmente. Para todos os outros casos, o dado não é escrito no servidor porém o retorno dado ao cliente é o equivalente ao de sucesso. Além disso, a operação de write-back pode ser otimizada enviando a operação de escrita apenas para os servidores que não enviaram o dado mais atual na ultima Q-RPC.

A assinatura digital feita pelo cliente serve para dois propósitos. Primeiramente, previne que um servidor faltoso perca um valor e convença um cliente a aceitar um valor arbitrário, pois um cliente só aceita valores assinados por um escritor. Segundo, previne que um leitor faltoso escreva um valor errado no servidor na fase de write-back.

Informalmente, nosso protocolo acima garante que as gravações no registrador sejam atômicas, de modo que as operações de leitura e gravação em todo o sistema pareçam ocorrer sequencialmente.

3.4 Protocolo Clientes Desonestos

Com esse protocolo conseguimos executar o sistema de forma correta mesmo que um ou mais clientes sejam desonestos. O objetivo desse protocolo é que o sistema consiga estar ou retornar a um estado consistente mesmo na presença de clientes faltosos.

O primeiro problema a ser enfrentado é que um escritor poderia escrever diferentes valores em diferentes servidores. Para evitar esse problema, foi desenvolvido um protocolo chamado de *echoe*.

No protocolo *echoe*, para fazer uma escrita, um cliente deve obter assinaturas do valor a ser escrito de um quórum de servidores. Após obter os *echoes*, o cliente então faz o processo de escrita passando o valor desejado conjuntamente com os *echoes* obtidos anteriormente. Como dois quóruns quaisquer se cruzam, não é possível para um escritor obter um quórum de *echoes* para dois diferentes valores com o mesmo *timestamp*, visto que os servidores só assinam um valor por *timestamp* para cada cliente. Por fim, o valor escrito por um cliente em um *timestamp* sempre vai ser único.

Apesar do protocolo acima assegurar escritas corretas, ele não assegura que os leitores identificarão o valor correto. No protocolo Honest Writers, os leitores utilizavam a assinatura digital para validar que o dado lido do servidor era correto. Porém no protocolo atual podemos ter clientes desonestos e com isso a assinatura digital se torna inútil. Por isso, confiaremos nos servidores corretos para mascarar os valores incorretos de servidores defeituosos. Isso é feito utilizando um Masking Quorum Systems (MQS).

A seguir os protocolos para escrita e leitura são apresentados, onde B é o número de servidores que podem falhar. Para o desenvolvimento deles, adotou-se como referência o protocolo "Dishonest Writers" [18].

Escrita

Para um cliente W escrever um valor V:

1. Faz uma Q-RPC para obter uma lista de *timestamps* de um quórum Q1 de servidores.

2. Escolhe um *timestamp* T maior que o maior encontrado na lista anterior.
3. Faz uma segunda Q-RPC para enviar (V,T) para os servidores e obter os *echoes* de (V,T) de um quórum Q2 de servidores.
4. Faz uma terceira Q-RPC para encaminhar os *echoes* e escrever (V,T) num quórum de servidores.

Leitura

Para um cliente ler um valor V:

1. Faz uma Q-RPC para obter uma lista de valor e *timestamp* (V,T) de um quórum Q1 de servidores, onde cada par é assinado pelo servidor que o mantém.
2. Descarta qualquer par (V,T) retornado por B ou menos servidores. Seleciona o par que foi retornado por B+1 servidores. Se tal par não existir, retorna NULL como resultado da operação de leitura.
3. Faz uma segunda Q-RPC para escrever (write-back) (V,T), conjuntamente com B+1 assinaturas de servidores.
4. Retorna o valor V como resultado da operação.

Na escrita, um servidor apenas assina (*echoe*) um valor V e *timestamp* T para o cliente C caso ele não tenha previamente assinado um valor diferente V2 para o *timestamp* T para o mesmo cliente C.

Um servidor salva a variável em seu registrador caso o *timestamp* da escrita seja maior que o *timestamp* armazenado atualmente e está acompanhado dos *echoes* de (V,T) de um quórum de servidores (escrita) ou de B+1 assinaturas de servidores (write-back).

Caso o *timestamp* seja igual, segue-se as mesmas regras descritas acima somando-se o fato que o servidor irá armazenar o dado caso o ID do cliente seja maior que o ID do cliente do dado salvo atualmente. Para todos os outros casos, o dado não é escrito no servidor porém o retorno dado ao cliente é o equivalente ao de sucesso.

3.5 Ferramentas Utilizadas

Abaixo lista-se algumas ferramentas utilizadas durante o desenvolvimento do projeto e uma breve descrição do que são e como foram utilizadas.

3.5.1 Socket

Para comunicação entre clientes e servidores utilizou-se sockets. O socket é uma porta entre o processo da aplicação e o protocolo da camada de transporte. Ele é responsável por transmitir as mensagens da rede para a aplicação e vice versa [19]. O socket é mapeado para uma porta do computador e permite que máquinas externas se comuniquem com a aplicação enviando mensagem para o socket, através do ip da máquina + porta.

Ao iniciar nossos clientes e servidores, um dos primeiros passos é instanciar seus respectivos sockets para comunicação com as demais partes do sistema. Por padrão, definiu-se que os servidores utilizariam as portas 5000 a 5003, onde o servidor com ID 0 escuta na porta 5000, o servidor com ID 1 escuta na porta 5001 e assim por diante. Para os clientes não foi definida uma porta especifica visto que não havia necessidade, o servidor retorna a mensagem para o socket que iniciou o contato com ele.

3.5.2 TCP

Ao utilizar Sockets, precisamos definir qual protocolo de transporte será utilizado, UDP ou TCP. Apesar de trazer um overhead para a aplicação, o TCP foi escolhido como protocolo de transporte pois com ele tem-se a garantia que a mensagem será entregue para o destinatário de forma confiável [19]. Dessa forma podemos focar em modelar nossa comunicação entre cliente e servidor tendo noção que as mensagens serão encaminhadas de maneira confiável pelo TCP. Toda comunicação no presente trabalho utiliza TCP.

3.5.3 Threads

Em geral, processos possuem um espaço de endereçamento de memória e uma única thread ou linha de execução. Porém, conforme afirma Tanenbaum, frequentemente há situações em que é desejável ter múltiplos threads de controle no mesmo espaço de endereçamento executando em quase-paralelo, como se eles fossem processos separados (exceto pelo espaço de endereçamento compartilhado) [20].

Os servidores são um desses casos. É desejável que um servidor execute as requisições tão logo elas cheguem aos seus sockets, porém em nossa aplicação temos diversas variáveis que devem ser compartilhadas entre essas execuções.

Para isso o processo do servidor cria uma thread filha a cada requisição que chega. Dessa forma um cliente não precisa esperar que outro que chegou primeiro tenha seu processamento encerrado para ter sua requisição processada. Também temos o compartilhamento de memória, podendo as diferentes threads acessar ao mesmo tempo o registrador, histórico de *echoes*, entre outros. Assim que a resposta é enviada para o cliente, a thread é finalizada.

3.5.4 Locks

Ao tornar o programa *multithread* ganha-se em tempo de resposta, mas nem tudo é perfeito. Há o risco das condições de corrida - situações onde dois ou mais processos estão lendo ou escrevendo um dado compartilhado e cujo resultado final depende de quem executa precisamente e quando [20]. Para evitá-las utilizou-se locks.

Os locks são travas para permitir a execução de apenas um processo em determinada parte de código por vez. Os locks foram utilizados majoritariamente nas regiões onde eram feitas leituras ou escritas nos registradores do sistema.

3.5.5 Semáforos

Assim como locks, semáforos são utilizados para controlar acesso a partes do código. Utilizou-se semáforos no código dos clientes. Para um operação de leitura ou escrita, um cliente faz um Q-RPC e após enviadas as requisições, a sua thread principal dá um DOWN num semáforo. Ao fazer o DOWN a thread fica bloqueada até que alguém faça uma operação UP. Uma thread secundária fica esperando pela resposta dos servidores, e ao obter um quórum de respostas, essa thread da um UP no semáforo, liberando a continuação da execução da thread principal, que poderá dar prosseguimento ao programa agora que possui um quórum de respostas.

3.5.6 JSON

Conforme o site oficial [21], "o JSON (JavaScript Object Notation - Notação de Objetos JavaScript) é uma formatação leve de troca de dados. Para seres humanos, é fácil de ler e escrever. Para máquinas, é fácil de interpretar e gerar. JSON é em formato texto e completamente independente de linguagem, pois usa convenções que são familiares às linguagens C e familiares, incluindo C++, C#, Java, JavaScript, Perl, Python e muitas outras. Estas propriedades fazem com que JSON seja um formato ideal de troca de dados."

É possível enviar diversos tipos de dado com o JSON, como: dicionários, listas, strings, números, booleanos. O JSON é constituído de duas estruturas básicas:

- Uma coleção de pares chave/valor. Em várias linguagens isto é conhecido como um object, record, struct, dicionário, etc.
- Uma lista ordenada de valores. Em várias linguagens isto é conhecido como uma array, vetor, lista ou sequência.

Em JSON, um dado pode ser representado da seguinte forma:

- **Objeto** - Figura 3.1 - Um objeto é um conjunto desordenado de pares chave/valor. Um objeto começa com (chave de abertura) e termina com (chave de fechamento). Cada chave é seguida por : (dois pontos) e os pares chave/valor são seguidos por , (vírgula).
- **Lista** - Figura 3.2 - Uma lista é uma coleção de valores ordenados. A lista começa com [(colchete de abertura) e termina com] (colchete de fechamento). Os valores são separados por , (vírgula).
- **Valor** - Figura 3.3 - Um valor pode ser uma string, um número, um booleano, null, um objeto ou uma lista. Estas estruturas podem estar aninhadas.

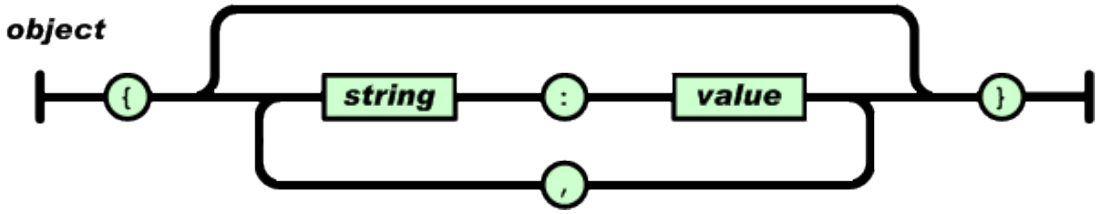


Figura 3.1: Estrutura de um objeto JSON. (Fonte: [21]).

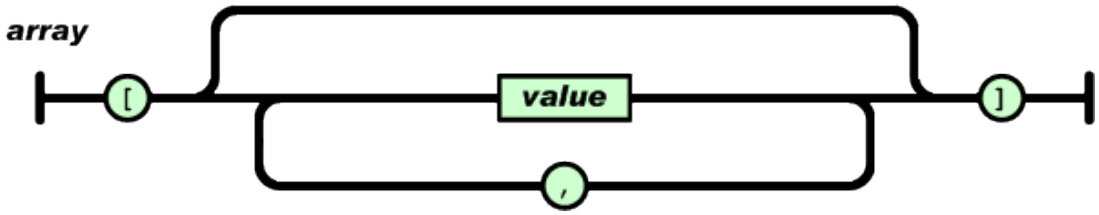


Figura 3.2: Estrutura de uma lista JSON. (Fonte: [21]).

Devido a facilidade de implementação e também de depuração, escolheu-se o JSON para ser o formato de envio de dados entre clientes e servidores no presente sistema. Toda comunicação é feita utilizando o JSON e também os dados/objetos a serem armazenados nos registradores são enviados em formato JSON. Na Figura 3.4 tem-se um exemplo de um JSON enviado por um servidor para um cliente que efetou uma requisicao de *timestamp*. Ao chegar na parte final, é feito o parse do JSON para objetos nas respectivas linguagens de programação.

3.5.7 Assinatura digital (RSA)

Uma assinatura digital é um esquema matemático que prova a autenticidade de uma mensagem ou dado. Com uma assinatura digital válida pode-se confiar que o dado em

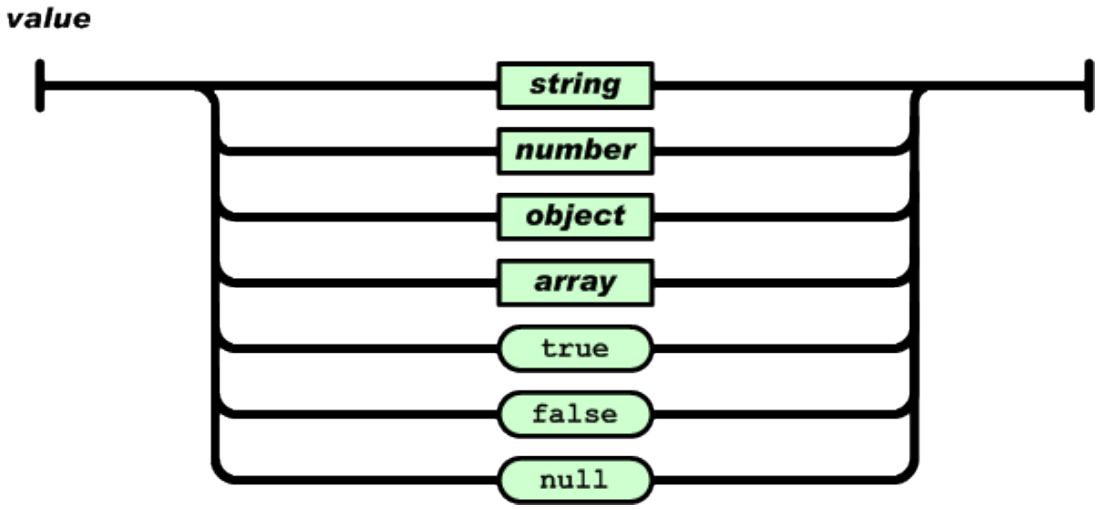


Figura 3.3: Estrutura de um valor JSON. (Fonte: [21]).

```
{
    server_id: 3,
    plataform: "java",
    request_code: 47,
    status: "success",
    msg: "read_timestamp",
    data: {
        timestamp: 22
    }
}
```

Figura 3.4: Resposta de um servidor em JSON.

questão foi escrito por determinado escritor e o dado não foi alterado até o presente momento. No presente trabalho utilizou-se assinatura digital com o algoritmo RSA, que utiliza chaves assimétricas, para assinar as mensagens no protocolo Honest Writers e também nos *echoes* do procolo Dishonest Writers.

Segue-se abaixo os passos para um cliente C assinar uma mensagem M [22]:

1. Utiliza-se alguma função de Hash na mensagem M, o resultado é conhecido como digest.
2. Encripta-se o digest com a chave privada de C. O resultado é a assinatura digital.

Para validar uma assinatura A de uma mensagem M feita por um cliente C:

1. Utiliza-se a mesma função de Hash utilizada anteriormente na mensagem M, o resultado é conhecido como digest.
2. Encripta-se o digest com a chave pública de C.
3. Eleva-se o digest a uma potência matemática definida quando criou-se ambas chaves. O resultado é comparado com a assinatura digital. Caso sejam iguais, a mensagem é considerada válida.

3.5.8 Base64

Base64 é uma forma de codificação de dados que representa dados binários em um formato ASCII. Base64 é um método focado para transferência na Internet (codificação MIME para transferência de conteúdo). É utilizado frequentemente para transmitir dados binários por meios de transmissão que lidam apenas com texto [23], como no caso do presente trabalho.

Utilizou-se Base64 para codificar as assinaturas digitais. Todas as assinaturas são convertidas para base64 antes de serem enviadas entre os sockets da aplicação. Essa atitude foi necessária pois com a criptografia dos dados, frequentemente eram criados caracteres especiais que causavam problemas de leituras entre as diferentes linguagens do sistema desse trabalho. Com a base64 isso foi evitado, pois os dados trafegados estão em formato ASCII.

3.6 Clientes e servidores

Apesar de implementar-se dois protocolos, a linha de raciocínio para a execução dos programas é a mesma tanto no protocolo honesto quanto no desonesto e segue a Figura 3.5. Desta forma, a explicação contida nessa seção demonstra o projeto para ambos protocolos.

3.6.1 Servidor

O servidor é criado e fica sempre a espera de um contato de um cliente em sua thread principal. Ao receber uma mensagem em seu socket, é estabelecida uma conexão TCP com o cliente e o servidor criar uma nova thread para executar os procedimentos necessários

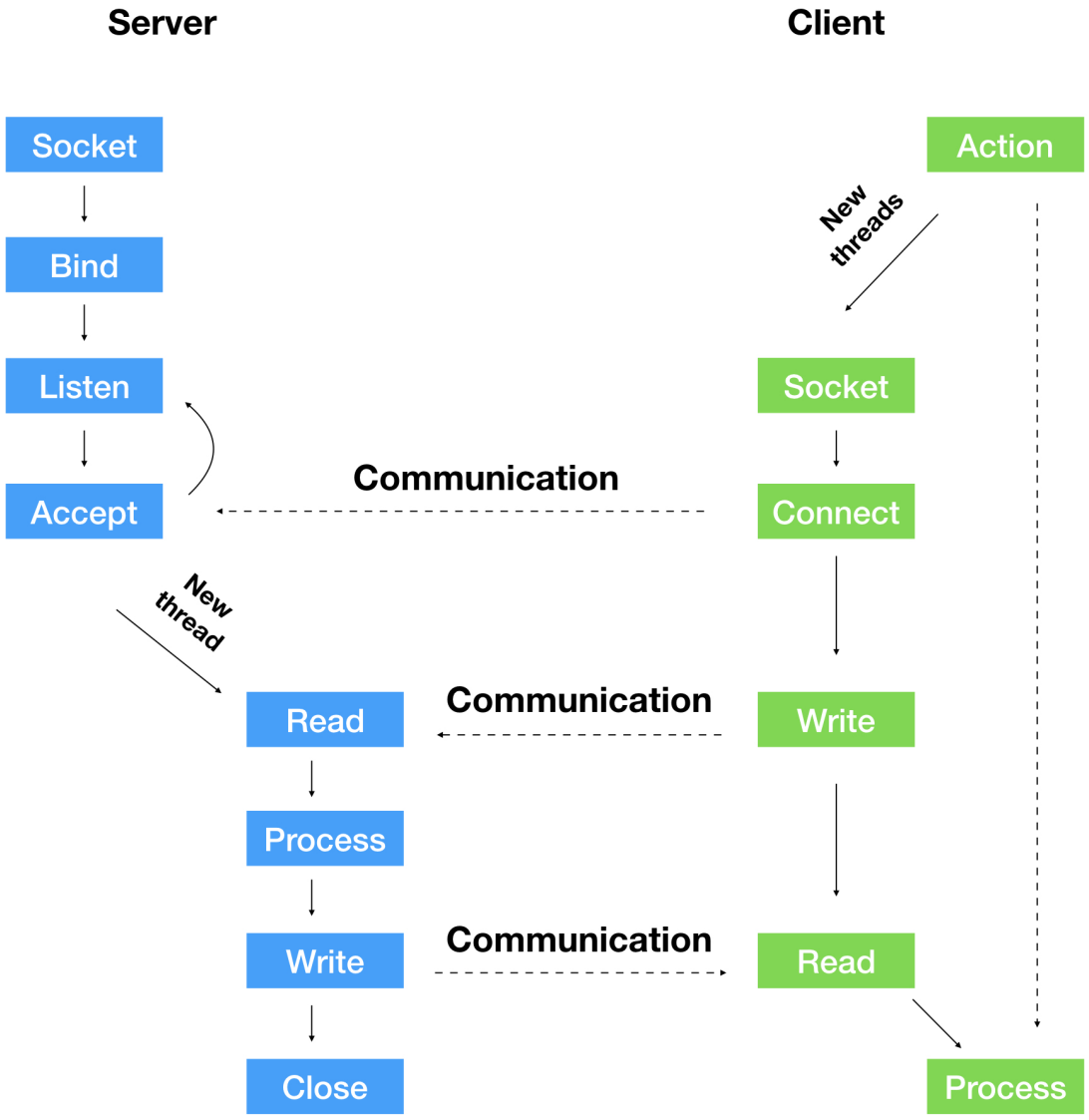


Figura 3.5: Estruturação da execução de clientes e servidores.

para retornar a resposta esperada pelo cliente. Enquanto está em sua linha de execução paralela, o servidor pode receber novas requisições de outros clientes.

Após executar e obter o dado necessário para o cliente, a thread secundária criada retorna o valor para o cliente e encerra a conexão TCP e também se destrói. Dessa forma o servidor sempre trabalha de uma forma assíncrona, sendo possível atender a diversos clientes de forma simultânea.

3.6.2 Cliente

O cliente é aquele que inicia o contato entre as partes. Após decidir que precisa fazer uma operação, como por exemplo uma leitura, a thread principal cria N threads secundárias. - N é o número de servidores conhecidos - as quais irão entrar em contato com um servidor específico.

Uma thread secundária estabelece uma conexão TCP com um servidor e após isso faz a requisição para o mesmo. Ao receber a resposta, o valor é retornado para a thread principal e a thread secundária é finalizada. A thread principal fica num estado bloqueado até que um quórum de threads tenha lhe retornado um valor. Após obter o quórum necessário de respostas, a thread principal ignora eventuais valores atrasados. A execução do cliente então continua de acordo com o protocolo e fase que está executando, conforme descrito em seções anteriores.

3.7 Comunicação

Como dito, toda comunicação foi feita utilizando JSON. Todas as mensagens enviadas entre clientes e servidores seguem um padrão. Abaixo define-se o escopo de parâmetros e repostas esperadas no sistema. As Figuras 3.1 a 3.9 definem os objetos utilizados para fazer requisições e respostas no sistema. Esses objetos são convertidos para JSON e enviados pela rede. Ao receber o JSON, o cliente/servidor faz o parse do JSON para algum dos objetos descritos nas imagens citadas, a depender do tipo de requisição.

3.7.1 Protocolo Clientes Honestos

Pode-se observar nas Figura 3.6 e Figura 3.7, que para fazer as leituras basta enviar o tipo da requisição apropriado e o identificador da requisição.

Figura 3.6: Requisição para leitura do registrador - Protocolo Honesto.

Figura 3.7: Requisição para leitura do *timestamp* - Protocolo Honesto.

Já para fazer uma escrita, é necessário mais dados conforme apresentado na Figura 3.8. O cliente deve enviar seu identificador, além dos dados para escrita (mensagem/dado e timestamp) + assinatura digital.

Figura 3.8: Requisição para escrita no registrador - Protocolo Honesto.

Na Figura 3.9 foram mapeadas as chaves retornadas pelos servidores para as requisições. Todas as respostas possuem uma chave status - indica se a requisição foi tratada com sucesso ou apresentou erro na execução -, além da chave msg que define o tipo do dado presente na chave data. Para uma msg igual a "read"o campo data conterá o dado escrito no registrador + *timestamp*. Para uma msg igual a "read timestamp"o campo data conterá o *timestamp* do dado escrito no registrador.

3.7.2 Protocolo Clientes Desonestos

A comunicação do protocolo desonesto é parecida com a anterior. O grande diferencial é no fato de que esse protocolo necessita dos *echoes* dos servidores, o que levou a criação de uma requisição adicional - para obter os *echoes*, conforme Figura 3.10 - e alguns campos em outras requisições já existentes.

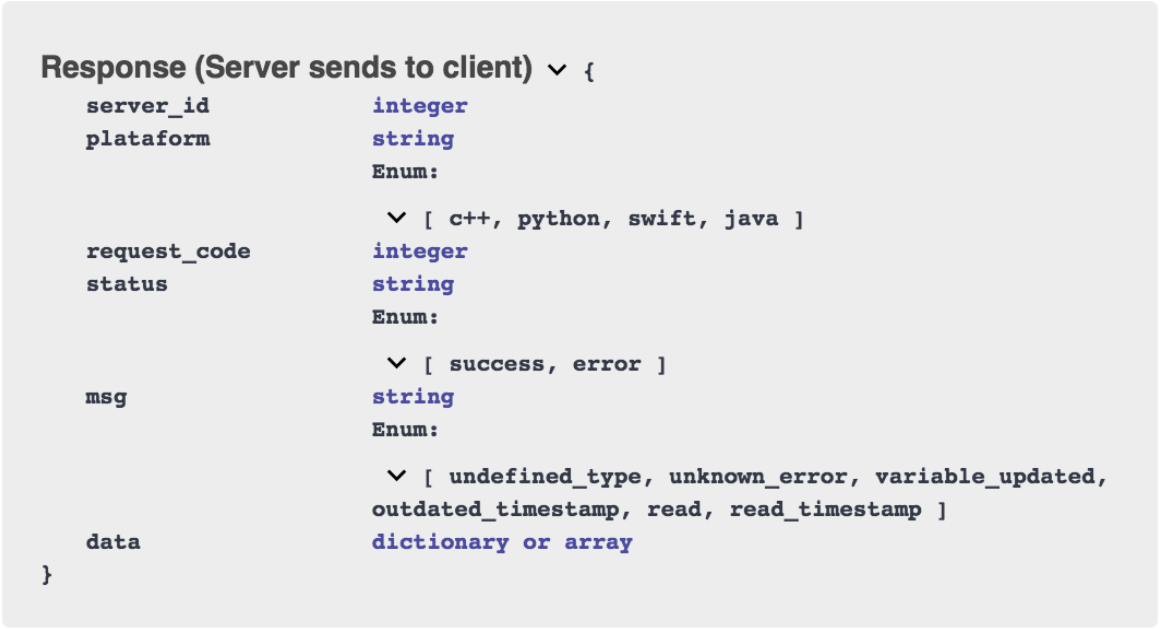


Figura 3.9: Resposta do servidor - Protocolo Honesto.

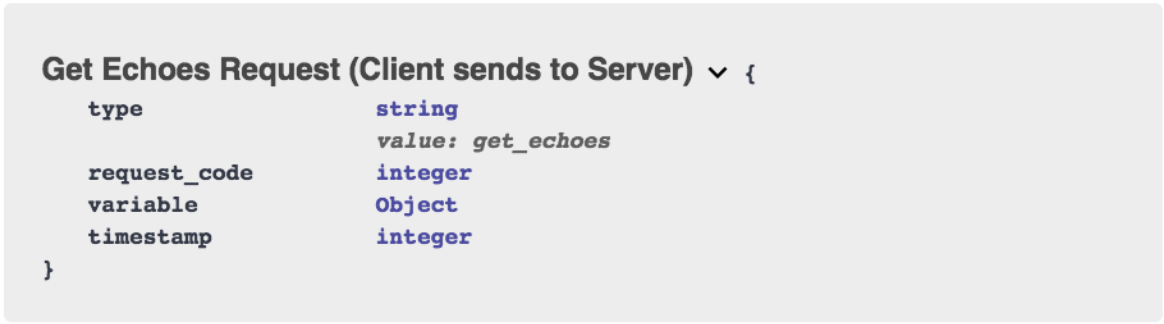


Figura 3.10: Requisição para obtenção dos *echoes* dos servidores - Protocolo Desonesto.

As requisições de leitura, Figura 3.11 e Figura 3.12, mantem-se iguais ao protocolo anterior. Já para a escrita, Figura 3.13, faz-se necessário enviar um parâmetro a mais, uma lista de *echoes* do dado a ser escrito por um quórum de servidores.

As respostas dos servidores, Figura 3.14, são parecidas com as do protocolo anterior, adicionando-se 3 mensagens relacionadas aos *echoes* e o próprio echo que pode ser retornado. Fora isso, mantem-se o padrão descrito anteriormente.

Figura 3.11: Requisição para leitura do registrador - Protocolo Desonesto.

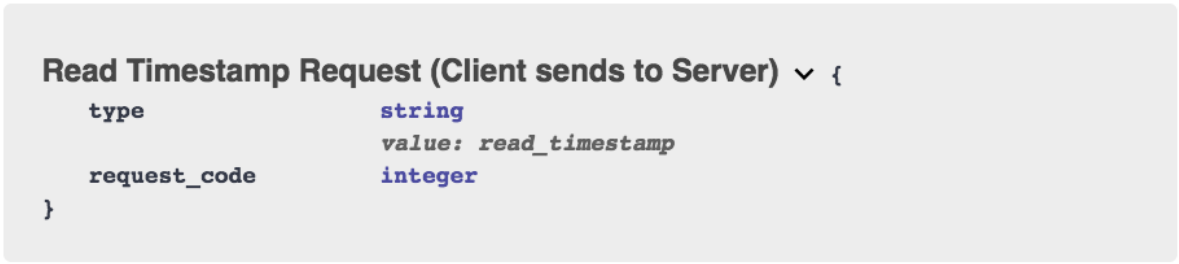


Figura 3.12: Requisição para leitura do *timestamp* - Protocolo Desonesto.

Figura 3.13: Requisição para escrita no registrador - Protocolo Desonesto.

3.8 Implementação

Esta seção discute detalhes da implementação dos programas e apresenta interfaces dos mesmos. Os clientes foram implementados em Java e Python. Já os servidores foram implementados em C++, Java, Python e Swift. Buscou-se implementar os códigos da maneira mais próxima possível nas diferentes linguagens de programação. Analisando os algoritmos que seguem abaixo percebe-se que as interfaces mantêm-se quase idênticas nas

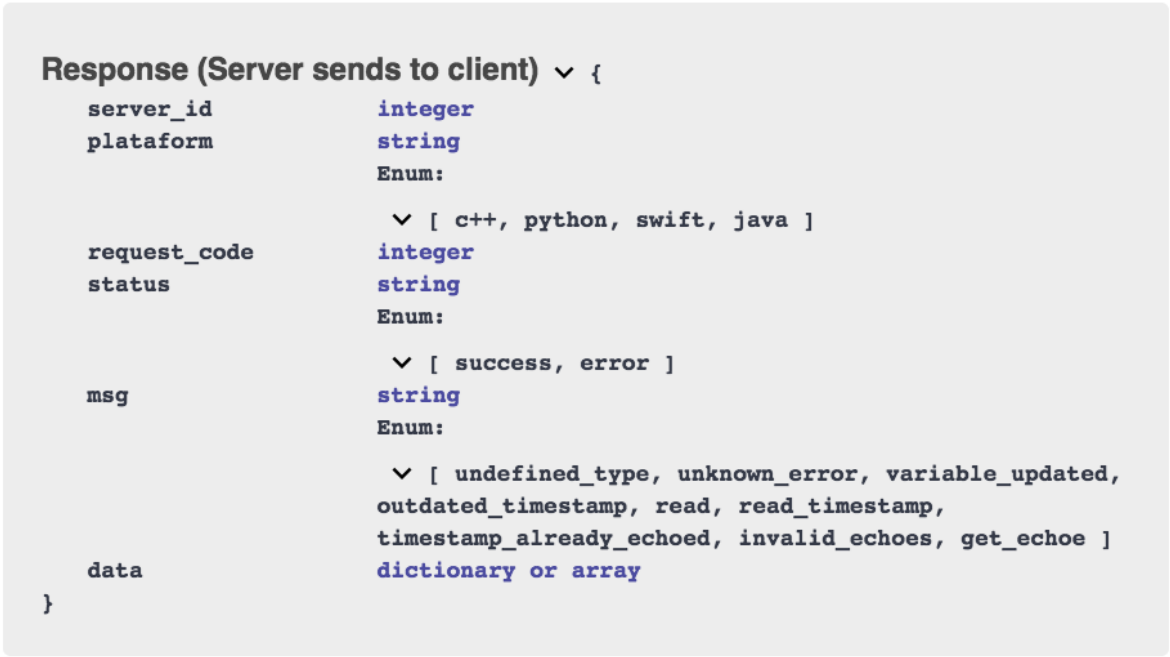


Figura 3.14: Resposta do servidor - Protocolo Desonesto.

diferentes linguagens.

3.8.1 Servidores Protocolo Honesto

No protocolo honesto, os servidores possuem o papel de basicamente registradores. Primeiramente, deve-se instanciar uma classe **Server** passando os parâmetro: id, ip, porta e verbose. Ao instanciar-se um **Server** o mesmo chama o método **waitForConnection()**. A seguir detalha-se como funcionalidade de cada método das interfaces descritas nos algoritmos 1, 2, 3 e 4.

- **waitForConnection()** - Inicia o socket do servidor que irá receber o contato inicial dos clientes. Fica em um loop infinito aguardando contato de clientes. Ao receber um contato, cria uma nova thread que irá executar o método **getRequestStatus()**.
- **getRequestStatus()** - Analisa a mensagem enviada pelo cliente e chama o metódo correto: write, read ou readTimestamp.
- **write()** - Salva um dado no registrador se os requerimentos são satisfeitos(ex: *timestamp* do dado maior que o salvo atualmente).
- **read()** - Lê o dado salvo no registrador e retorna para o cliente.

- **readTimestamp()** - Lê o *timestamp* do dado salvo no registrador e retorna para o cliente.

```
1  class Server(object):
2      HOST = ''
3      PORT = -1
4      ID = -1
5      VARIABLE = ''
6      TIMESTAMP = -1
7      DATA_SIGNATURE = ''
8      CLIENT_ID = -1
9      LOCK = threading.Lock();
10     VERBOSE = 0
11
12     def __init__(self, id, ip, port, verbose)
13     def waitForConnection(self)
14     def getRequestStatus(self, request, socketTCP)
15     def write(self, request, socketTCP)
16     def read(self, request, socketTCP)
17     def readTimestamp(self, request, socketTCP)
```

Algoritmo 1: Interface servidor Python - Protocolo honesto

```
public class Server
{
    public String host = "";
    public int port = -1;
    public int id = -1;
    public String variable = "";
    public int \emph{timestamp}  = -1;
    public String data_signature = "";
    public int client_id = -1;
    Lock lock = new ReentrantLock();
    public int verbose = 0;

    public Server (int id, String ip, int port, int verbose);
    public void waitForConnection() throws IOException;
    private boolean getRequestStatus(Map<String, Object> request, Socket socket);
    private boolean read(Map<String, Object> request, Socket socket);
    private boolean readTimestamp(Map<String, Object> request, Socket socket);
    private boolean write(Map<String, Object> request, Socket socket);
}
```

Algoritmo 2: Interface servidor Java - Protocolo honesto

```cpp
namespace server{
    std::string HOST = "";
    int PORT = -1;
    int ID = -1;
    int VERBOSE = 0;
    std::string VARIABLE = "";
    unsigned int TIMESTAMP = 0;
    std::string DATA_SIGNATURE = "";
    int CLIENT_ID = -1;
    std::mutex LOCK;

    void init (int id, std::string ip, int port, int verbose);
    void waitForConnection();
    void error(std::string msg);
    void getRequestStatus(rapidjson::Document *request, int socketTCP);
    void write(rapidjson::Document *request, int socketTCP);
    void readData(rapidjson::Document *request, int socketTCP);
    void readTimestamp(rapidjson::Document *request, int socketTCP);
}
```

Algoritmo 3: Interface servidor C++ - Protocolo honesto

```
class Server: NSObject {
    var HOST = "";
    var PORT = -1;
    let ID : Int
    var VARIABLE = "";
    var TIMESTAMP = -1;
    var DATA_SIGNATURE = "";
    var CLIENT_ID = -1;
    var LOCK = pthread_mutex_t();
    var VERBOSE = 0;

    init(id: Int, ip: String, port: Int, verbose: Int)
    func waitForConnection()
    func getRequestStatus(request: Dictionary<String, Any>, clientSocket: TCPClient)
    func write (request: Dictionary<String, Any>, clientSocket: TCPClient)
    func read (request: Dictionary<String, Any>, clientSocket: TCPClient)
    func readTimestamp (request: Dictionary<String, Any>, clientSocket: TCPClient)
}
```

Algoritmo 4: Interface servidor Swift - Protocolo honesto

3.8.2 Clientes Protocolo Honesto

No protocolo honesto, os clientes devem fazer as operações de leitura e escrita no servidor e também a assinatura digital e validação das assinaturas. Primeiramente, deve-se instanciar uma classe **Client** passando os parâmetro: id, lista de ip e porta dos servidores, modo verbose, path dos certificados. Ao instanciar-se um **Client** o mesmo chama o método **initUserInterface()**. A seguir detalha-se cada método das interfaces descritas nos algoritmos 5 e 6.

- **initUserInterface()** - Inicializa a interface com usuário. Nela o usuário poderá selecionar qual tipo de operação deseja fazer: leitura ou escrita. Para leitura, é chamado o método **read**. Já ao clicar na escrita, o usuário deve inserir o dado a ser salvo e este é passado como parâmetro na chamado do método **write**.
- **write()** - Chama **readTimestamp** para obter o maior *timestamp*. Incrementa o maior *timestamp* obtido. Faz a assinatura digital do dado e envia dado + assinatura + *timestamp* para os servidores.

- **readTimestamp()** - Obtém os *timestamps* dos dados salvos nos servidores.
- **read()** - Obtém os dados salvos nos servidores. Mapeia possíveis servidores com dados defasados e faz uma chama de **writeBack** para eles.
- **writeBack()** - Atualiza os dados salvos em servidores defasados.

```
1  class Client (object):
2      SERVERS = []
3      QUORUM = 2
4      ID = -1
5      LOCK = threading.Lock()
6      REQUEST_CODE = 0
7      RESPONSES = []
8      OUT_DATED_SERVERS = []
9      SEMAPHORE = threading.Semaphore(0)
10     VERBOSE = 0
11     CERT_PATH = ''
12
13     def __init__(self, id, servers, verbose, cert_path):
14     def initUserInterface(self)
15     def write(self, value)
16     def readTimestamp(self)
17     def read(self)
18     def writeBack(self, value, timestamp, data_signature, client_id)
```

Algoritmo 5: Interface cliente Python - Protocolo honesto

```java
public class Client {
    List<Pair<String, Integer>> servers;
    int quorum = 2;
    int id = -1;
    Lock lock = new ReentrantLock();
    private int request_code = 0;
    List<ResponseData> responses;
    List<Pair<String, Integer>> out_dated_servers;
    Semaphore semaphore = new Semaphore(0);
    public int verbose = 0;
    public String cert_path = "";
    boolean exit = false;


    public Client(int id, List<Pair<String, Integer>> servers, int verbose,
    String cert_path);
    private void initUserInterface();
    private void write(String value);
    private int readTimestamp();
    private void read();
    private void writeBack(ResponseData data);
}
```

Algoritmo 6: Interface cliente Java - Protocolo honesto

3.8.3 Servidores Protocolo Desonesto

No protocolo desonesto, os servidores possuem o papel de registradores e devem implementar os *echoes* dos dados. Primeiramente, deve-se instanciar uma classe **Server** passando os parâmetro: id, ip, porta e verbose. Ao instanciar-se um **Server** o mesmo chama o método **waitForConnection()**. A seguir detalha-se para as interfaces descritas nos algoritmos 7, 8, 9 e 10, os novos métodos ou métodos que sofreram alterações em relação ao protocolo honesto. Para os métodos presentes na interface, mas não descritos a seguir, deve-se considerar o comportamento descrito na seção do protocolo honesto.

- **getRequestStatus()** - Analisa a mensagem enviada pelo cliente e chama o metódo correto: write, read ou readTimestamp ou getEchoe.

- **write()** - Salva um dado no registrador se os requerimentos são satisfeitos (Timestamp do dado maior que o salvo atualmente e *echoes* válidos).
- **getEchoe()** - Retorna um *echoe* para um dado + *timestamp* caso não tenha retornado um *echoe* para o *timestamp* em questão e um dado diferente.
- **isEchoValid()** - Confere se os *echoes* são válidos para o dado + *timestamp* em questão.

```
1  class Server(object):
2          HOST = ''
3      PORT = -1
4      ID = -1
5      FAULTS = 1
6      QUORUM = 2 * FAULTS + 1
7      VARIABLE = ''
8      DATA_SIGNATURE = ''
9      TIMESTAMP = -1
10     CLIENT_ID = -1
11     ECHOED_VALUES = []
12     LOCK = threading.Lock()
13     VERBOSE = 0
14     CERT_PATH = ''
15
16     def __init__(self, id, ip, port, verbose)
17     def waitForConnection(self)
18     def getRequestStatus(self, request, socketTCP)
19     def write(self, request, socketTCP)
20     def read(self, request, socketTCP)
21     def readTimestamp(self, request, socketTCP)
22     def getEchoe(self, request, socketTCP):
23     def isEchoValid(self, echoes, value, timestamp, type):
```

Algoritmo 7: Interface servidor Python - Protocolo desonesto

```
1  public class Server
2  {
3      public String host = "";
4      public int port = -1;
5      public int id = -1;
6      public int faults = 1;
7      public int quorum = 2*faults + 1;
8      public String variable = "";
9      public int \emph{timestamp}  = -1;
10     public String data_signature = "";
11     public int client_id = -1;
12     public Map<Integer, List<Pair<Integer, String>>> echoed_values =
13             new HashMap<Integer, List<Pair<Integer, String>>>();
14     Lock lock = new ReentrantLock();
15     public int verbose = 0;
16     public String cert_path = "";
17
18     public Server (int id, String ip, int port, int verbose);
19     public void waitForConnection() throws IOException;
20     private boolean getRequestStatus(Map<String, Object> request, Socket socket);
21     private boolean read(Map<String, Object> request, Socket socket);
22     private boolean readTimestamp(Map<String, Object> request, Socket socket);
23     private boolean write(Map<String, Object> request, Socket socket);
24     private boolean getEchoe(Map<String, Object> request);
25     private boolean isEchoValid(List<Pair<Integer, String>> echoes, String value,
26             int timestamp, String type)
27 }
```

Algoritmo 8: Interface servidor Java - Protocolo desonesto

```
namespace server{
        std::string HOST = "";
    int PORT = -1;
    int ID = -1;
    int VERBOSE = 0;
    std::string CERT_PATH = "";
    int FAULTS = 1;
    int QUORUM = 2*FAULTS + 1;
    std::string VARIABLE = "";
    unsigned int TIMESTAMP = 0;
    std::string DATA_SIGNATURE = "";
    int CLIENT_ID = -1;
    std::vector<std::pair<int, std::pair<int, std::string>>> ECHOED_VALUES;
    std::mutex LOCK;

    void init (int id, std::string ip, int port, int verbose);
    void waitForConnection();
    void error(std::string msg);
    void getRequestStatus(rapidjson::Document *request, int socketTCP);
    void write(rapidjson::Document *request, int socketTCP);
    void readData(rapidjson::Document *request, int socketTCP);
    void readTimestamp(rapidjson::Document *request, int socketTCP);
    void getEchoe(rapidjson::Document *request, int socketTCP);
    bool isEchoValid(std::vector<std::pair<int, std::string>> echoes,
            std::string value, int timestamp, std::string type);
}
```

Algoritmo 9: Interface servidor C++ - Protocolo desonesto

```
class Server: NSObject {
    var HOST = "";
    var PORT = -1;
    let ID : Int
    let FAULTS = 1
    let QUORUM = 2 * FAULTS + 1
    var VARIABLE = "";
    var TIMESTAMP = -1;
    var DATA_SIGNATURE = "";
    var CLIENT_ID = -1;
    var LOCK = pthread_mutex_t();
    var ECHOED_VALUES : Array<(Int, Int, String)> = []
    var VERBOSE = 0;
    var CERT_PATH = "";

    init(id: Int, ip: String, port: Int, verbose: Int)
    func waitForConnection()
    func getRequestStatus(request: Dictionary<String, Any>, clientSocket: TCPClient)
    func write (request: Dictionary<String, Any>, clientSocket: TCPClient)
    func read (request: Dictionary<String, Any>, clientSocket: TCPClient)
    func readTimestamp (request: Dictionary<String, Any>, clientSocket: TCPClient)
    func getEchoe(request: Dictionary<String, Any>, clientSocket: TCPClient)
    func isEchoValid(echoes : [(Int, String)], value : String, \emph{timestamp}  : In
            type : String) -> Bool
}
```

Algoritmo 10: Interface servidor Swift - Protocolo desonesto

3.8.4 Clientes Protocolo Desonesto

No protocolo desonesto, os clientes devem fazer as operações de leitura e escrita no servidor e obter os *echoes* dos servidores como parte da operação de escrita. Primeiramente, deve-se instanciar uma classe **Client** passando os parâmetro: id, lista de ip e porta dos servidores, modo verbose, path dos certificados. Ao instanciar-se um **Client** o mesmo chama o método **initUserInterface()**. A seguir detalha-se para as interfaces descritas nos algoritmos 11 e12, os novos métodos ou métodos que sofreram alterações em relação

ao protocolo honesto. Para os métodos presentes na interface, mas não descritos a seguir, deve-se considerar o comportamento descrito na seção do protocolo honesto.

- **write()** - Chama **readTimestamp** para obter o maior *timestamp*. Incrementa o maior *timestamp* obtido. Chama **getEchoes** para obter *echoes*. Envia *echoes* + dado + *timestamp* para os servidores.
- **writeBack()** - Atualiza os dados salvos em servidores defasados, enviando *echoes* + dado + *timestamp*.
- **getEchoes()** - Obtém *echoes* de um quórum de servidores para um dado + *timestamp*.

```
1  class Client (object):
2      SERVERS = []
3      FAULTS = 1
4      QUORUM = 2*FAULTS + 1
5      ID = -1
6      LOCK = threading.Lock()
7      REQUEST_CODE = 0
8      RESPONSES = []
9      ECHOES = []
10     OUT_DATED_SERVERS = []
11     INCREMENT_TIMESTAMP_BY = 1
12     TIMESTAMP_ALREADY_ECHOED_BY_ANY_SERVER = False
13     TIMESTAMP_ALREADY_ECHOED_POWER = 0
14     SEMAPHORE = threading.Semaphore(0)
15     VERBOSE = 0
16     CERT_PATH = ''
17
18     def __init__(self, id, servers, verbose, cert_path):
19     def initUserInterface(self)
20     def write(self, value)
21     def readTimestamp(self)
22     def read(self)
23     def writeBack(self, value, timestamp, echoes)
24     def getEchoes(self, value, timestamp)
```

Algoritmo 11: Interface cliente Python - Protocolo desonesto

```
1  public class Client {
2      List<Pair<String, Integer>> servers;
3      int quorum = 2;
4      int id = -1;
5      Lock lock = new ReentrantLock();
6      private int request_code = 0;
7      List<ResponseData> responses;
8      List<Pair<String, Integer>> out_dated_servers;
9      Semaphore semaphore = new Semaphore(0);
10     public int verbose = 0;
11     public String cert_path = "";
12     boolean exit = false;
13
14
15     public Client(int id, List<Pair<String, Integer>> servers, int verbose,
16             String cert_path);
17     private void initUserInterface();
18     private void write(String value);
19     private int readTimestamp();
20     private void read();
21     private void writeBack(ResponseData data);
22 }
```

Algoritmo 12: Interface cliente Java - Protocolo desonesto

3.9 Execução

A execução dos clientes e servidores foi automatizada de forma a diminuir o número de passos necessários para executar as aplicações desse projeto. Nesta seção detalha-se o passo a passo para a execução de clientes e servidores.

Certificados

Primeiramente, deve-se gerar os certificados que serão utilizados nas aplicações. No protocolo honesto, deve-se gerar os certificados públicos e privados para cada cliente que irá participar da execução em questão. Os certificados são utilizados para fazer as assinaturas digitais. Já no protocolo desonesto, deve-se gerar os certificados públicos e privados para

cada servidor que irá processar as requisições, é com que eles que gera-se os *echoes* das requisições de escrita.

Para gerar os certificados, utilizou-se a biblioteca OpenSSL [24]. Por meio dela criou-se certificados RSA com tamanho de 1024 bytes. Os certificados devem ser salvos em uma única pasta e devem seguir a seguinte nomenclatura: identificador se cliente ou servidor + ID + identificador se privado ou público + extensão. Por exemplo, um certificado privado do servidor de ID 0 deve-se chamar server0_private.pem e um certificado público de um cliente de ID 1 deve-se chamar client1_public.pem.

Servidores

Para executar os servidores deve-se iniciar o programa passando-se os parâmetros na ordem indicada a seguir, independentemente da linguagem:

- **IP de entrada dos dados.** IP da máquina em que o servidor está rodando. Pode-se utilizar ifconfig no terminal para obter o mesmo.
- **ID do servidor.** Identificador único do servidor. Valor inteiro que identifica o servidor e também a porta para acesso do socket. A porta é definida como 5000 + ID.
- **Modo verbose.** 0 para não printar nada no console, 1 para printar apenas os dados principais, 2 para printar todos os dados e todas as requests.
- **Path dos certificados.** Caminho completo para pasta que contém os arquivos dos certificados públicos e privados.

A seguir um exemplo de comando para iniciar um servidor na linguagem python:
$ python3 server.py 192.168.0.199 0 2 /users/cayke/certs

Clientes

Para executar os clientes deve-se iniciar o programa passando-se os parâmetros na ordem indicada a seguir, independentemente da linguagem:

- **ID do cliente.** Identificador único do cliente
- **Modo verbose.** 0 para não printar nada no console, 1 para printar apenas os dados principais, 2 para printar todos os dados e todas as requests.
- **Path dos certificados.** Caminho completo para pasta que contém os arquivos dos certificados públicos e privados.

- **IP e porta dos servidores.** IP e porta dos servidores para comunicação, separados por espaço.

A seguir um exemplo de comando para iniciar um cliente na linguagem python:
$ python3 client.py 0 2 /users/cayke/certs node0.caykequoruns.freestore.emulab.net 5000 node1.caykequoruns.freestore.emulab.net 5001 node2.caykequoruns.freestore.emulab.net 5002

Testes

Para realização dos testes, automatizou-se ainda mais a criação dos clientes. Foi criado um script que recebe alguns parâmetros e executa os testes para o ambiente descrito no arquivo bash. A seguir descreve-se os campos necessários no arquivo bash:

- **Número de clientes** Quantidades de clientes que deve ser executada simultaneamente na mesma máquina.
- **Tipo de operação** Tipo de operação que os clientes executaram, escrita ou leitura.
- **Número de operações** Quantidade de vezes que cada cliente irá executar a operação de escrita ou leitura.
- **Path dos resultados.** Caminho completo para pasta que contém os resultados das execuções.

Após a execução dos testes seguindo os parâmetros descritos acima, cada cliente cria um arquivo texto que contém os dados relativos a sua execução como tempo de início e término das operações, número de operações executadas. Ao fim da execução de todos os clientes, é executado um novo programa que irá analisar todos os arquivos criados e irá calcular a latência e throughput do sistema durante a execução dos clientes em questão.

Capítulo 4

Experimentos

Nesse capítulo aborda-se a metodologia empregada para executar os testes feitos no sistema proposto nos capítulos anteriores e analisa-se os resultados de desempenho obtidos através dos testes.

4.1 Ambiente de execução

Para executar os programas utilizou-se diversas máquinas. As máquinas foram disponibilizadas pelo projeto Emulab [25]. O Emulab oferece aos desenvolvedores diversos ambientes e possibilidades para desenvolver e testar novos sistemas. É utilizado por diversos pesquisadores da computação nas áreas de redes e sistemas distribuídos [25].

Para esse projeto foram utilizadas 8 máquinas do tipo d710, as quais 4 são servidores e 4 são clientes. Em cada máquina servidor há a execução de apenas 1 servidor por vez. Já nas máquinas cliente, pode-se executar diversos clientes simultaneamente conforme descrito nas próximas seções.

A máquina d710 possui a seguinte configuração de hardware: processador 2.4 GHz 64-bit Intel Quad Core Xeon E5530, 12GB de memória RAM e interface de rede gigabit conectadas a um switch de 1Gbps. O ambiente de software utilizado foi o sistema operacional Ubuntu 14.04.5 LTS 64-bit, g++ 4.6.0 (C++), JVM Oracle JDK 1.8 171 (Java), Python 3.5.2 (Python), Swift 4.1 (Swift).

4.2 Metodologia

Os testes foram executados tendo como objetivo mensurar a latência e o throughput do sistema proposto em determinado tipo de operação, leitura ou escrita, de acordo com o tipo do protocolo, honesto ou desonesto. Para isso foram feitas diversas execuções e combinações de clientes e servidores.

Latência e throughput

Cada cliente executa mil vezes uma operação de leitura ou escrita (de acordo com o teste sendo executado) de forma sequencial. Antes de iniciar uma operação o cliente registra o tempo atual. Ao receber a resposta do servidor para a operação desejada, o cliente registra o tempo total de execução da operação em memória. Ao fim das mil operações, o cliente calcula a média de tempo para executar uma operação (latência) e registra em um arquivo de texto conjuntamente com o tempo de inicio da primeira operação e o tempo final da última operação.

Após o término da execução de todos os clientes, um programa é executado para calcular a latência e o throughput do sistema. A latência do sistema é a média da latência dos clientes, (somatório de todas as latências dividido pelo número de cliente). O throughput é o número de operações por segundo, ou seja, (mil x número de clientes) / (maior tempo final da última operação de um cliente Y - menor tempo inicial da primeira operação de um cliente X).

Nota-se que a latência é calculada como o tempo necessário para obter-se uma resposta do servidor para uma operação completa do protocolo (leitura ou escrita). Portanto são feitas várias requisições por operação. A latência calculada não é a latência de uma requisição, e sim a latência das operações proposta nos algoritmos descritos no Capítulo 3. Durante os testes foram descartados os 10% dos resultados de maior latência e os 10% dos resultados de menor latência.

Operações: leitura e escrita

Para os testes, os clientes fazem operações de leitura e escrita de forma automática. Tao logo a resposta da operação chega para si, uma nova operação é iniciada. Como padrão adotou-se tanto para escrita como para leitura um dado de tamanho 200 bytes. Na escrita esse dado é gerado de forma aleatória a cada nova escrita do cliente, já na leitura o dado é sempre o mesmo visto que todos os clientes estarão fazendo apenas operações de leitura.

Número de clientes

Os clientes foram distribuídos em 4 máquinas. Os testes iniciam com 1 cliente sendo executado em cada máquina, e após cada rodada de execução o número de clientes é dobrado em todas as máquinas. Para cada número X de clientes os testes são executados com 1,2,3 e 4 máquinas simultâneas. Os resultados são apresentados de acordo com o número de máquinas clientes, sendo que dentro de cada gráfico contem-se o número total de clientes (número de clientes em uma máquina x número de máquinas)

Servidores

Durante os testes, testou-se as operações de leitura e escrita nos protocolos honestos e desonestos em 5 conjuntos de servidores:

1. Mix (com diversidade) - 1 servidor C++, 1 servidor Java, 1 servidor Python, 1 Servidor Swift
2. C++ - 4 servidores C++
3. Java - 4 servidores Java
4. Python - 4 servidores Python
5. Swift - 4 servidores Swift

Critério de parada

O teste é finalizado para um tipo de operação/servidor quando ao dobrar o número de clientes ocorre uma diminuição ou estabilização no throughput do sistema, pois entende-se que o sistema já chegou ao seu limite na execução anterior. Esse limite pode ser causado pelo servidor ou pelo cliente. Deve-se analisar caso a caso.

4.3 Resultados

A seguir analisa-se os resultados obtidos para os dois protocolos propostos no presente trabalho. Para ambos foi possível executar os testes de forma satisfatória, o que prova que os protocolos propostos funcionam na prática. Para os gráficos a seguir, menor valor de latência e maior valor de throughput representam melhores resultados.

4.3.1 Protocolo Honesto

No protocolo honesto, o servidor tem uma carga muito pequena de processamento, funcionando basicamente como registradores. Devido a esse fato esperava-se que não seria possível alcançar os valores máximos de throughput de escrita e leitura dos servidores pois o protocolo exige muito mais poder de processamento dos clientes, que devem fazer operações de criptografia (custosas computacionalmente).

De fato, isso foi observado na maioria das execuções. Em muitas delas, acabou-se chegando ao limite do sistema dos clientes e não dos servidores. Analisando os resultados de leitura - Figura 4.1, Figura 4.3, Figura 4.5, Figura 4.7 - e os de escrita - Figura 4.2, Figura 4.4, Figura 4.6, Figura 4.8 - pode-se chegar a algumas conclusões:

- A latência aumenta com o acréscimo de clientes não pela sobrecarga dos servidores, e sim pela sobrecarga da máquina que está executando os clientes. Analisando os casos extremos onde temos 1 e 4 máquinas de clientes, percebe-se que a latência de 8 clientes (Figura 4.1) é praticamente a mesma de 32 clientes (Figura 4.7, 8 clientes por máquina) e para todos os outros casos de números de clientes múltiplos entre si.
- Quanto maior o throughput, maior a latência. É normal que um sistema demore mais a responder quando está recebendo mais requisições. Isso foi visto em todos os resultados, tanto na escrita quanto na leitura.
- 8 clientes por máquina é o limite ideal para uma execução sem sobrecarga nos clientes. Percebe-se que aumentando os clientes até 8 clientes por máquina obtém-se resultados satisfatórios de throughput sem comprometer a latência. Ao ultrapassar essa marca, a latência começa a subir de forma exponencial e ocorrem até diminuições no throughput devido a sobrecarga das máquinas de clientes.
- O sistema apresentou um bom desempenho com a presença de diversidade (mix). No geral, foi mais lento apenas que o C++ e o Java.
- As leituras apresentaram menor latência e maior throughput que as escritas. Isso era esperado visto que o processo de leitura leva um passo a menos que a escrita conforme descrito na seção 3.3.

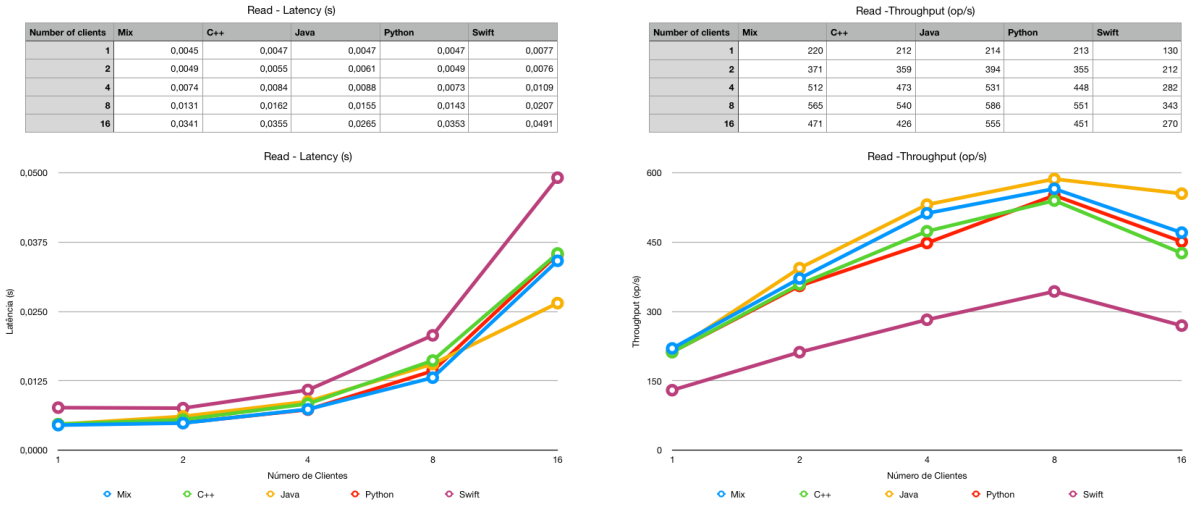


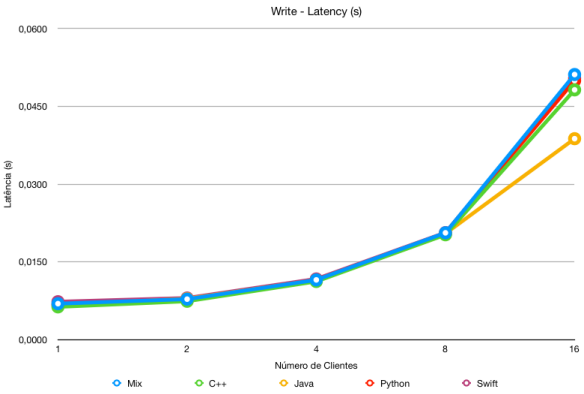
Figura 4.1: Resultados Leitura - 1 máquina cliente - Protocolo Honesto.

Write - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
1	0,0069	0,0063	0,0066	0,0067	0,0073
2	0,0078	0,0074	0,0079	0,0075	0,0080
4	0,0115	0,0112	0,0115	0,0115	0,0117
8	0,0206	0,0202	0,0205	0,0204	0,0206
16	0,0511	0,0482	0,0388	0,0500	0,0504

Write -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
1	145	159	151	149	137
2	254	268	252	264	249
4	343	354	344	343	338
8	383	377	384	384	380
16	303	323	403	311	308



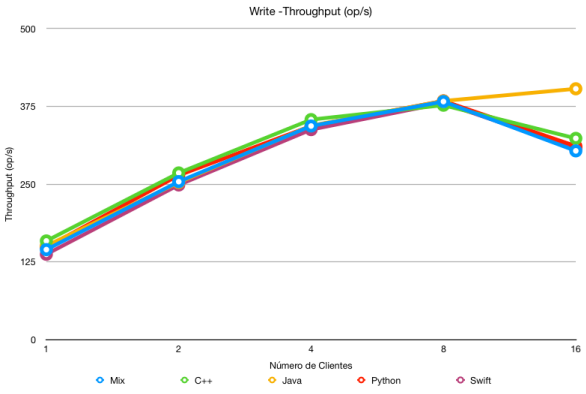


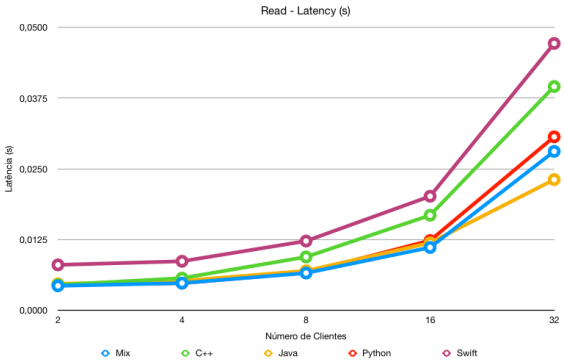
Figura 4.2: Resultados Escrita - 1 máquina cliente - Protocolo Honesto.

Read - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
2	0,0043	0,0045	0,0046	0,0045	0,0080
4	0,0048	0,0057	0,0052	0,0048	0,0086
8	0,0066	0,0094	0,0070	0,0066	0,0122
16	0,0111	0,0168	0,0119	0,0123	0,0201
32	0,0281	0,0395	0,0231	0,0306	0,0472

Read -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
2	416	400	376	395	221
4	729	744	631	639	410
8	969	1133	772	775	572
16	1038	1226	864	1001	615
32	843	1021	924	868	476



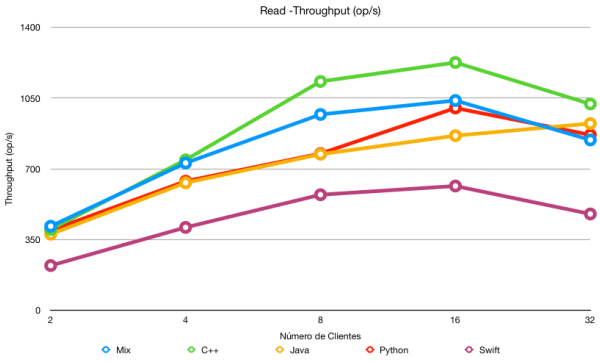


Figura 4.3: Resultados Leitura - 2 máquinas clientes - Protocolo Honesto.

4.3.2 Protocolo Desonesto

No protocolo desonesto o servidor passa a ter uma carga de processamento bem maior em relação ao protocolo honesto. Devido a fase de echo, é necessário que os servidores assinem os dados e validem as assinaturas de outros servidores ao menos uma vez para cada escrita feita neles.

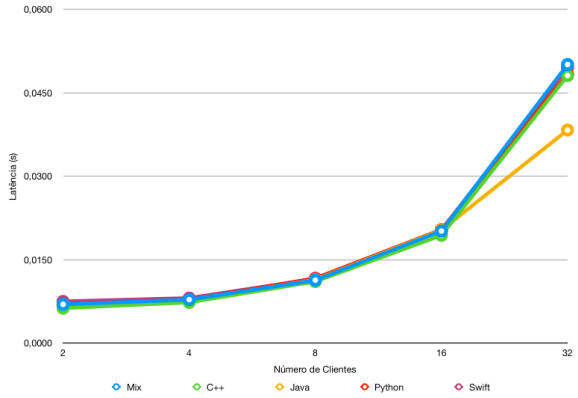
A carga de processamento maior se inverte nesse protocolo, e podemos perceber isso nos resultados de escrita - Figura 4.10, Figura 4.12, Figura 4.14, Figura 4.16 - que diferem bastantes dos resultados obtidos para o protocolo honesto. Já para a leitura não há

Write - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
2	0,0070	0,0063	0,0067	0,0069	0,0075
4	0,0078	0,0073	0,0076	0,0078	0,0081
8	0,0113	0,0111	0,0114	0,0117	0,0116
16	0,0202	0,0194	0,0203	0,0204	0,0203
32	0,0501	0,0481	0,0383	0,0484	0,0496

Write -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
2	266	287	279	269	246
4	472	490	478	467	457
8	661	678	650	610	644
16	745	719	758	743	737
32	593	630	787	614	595



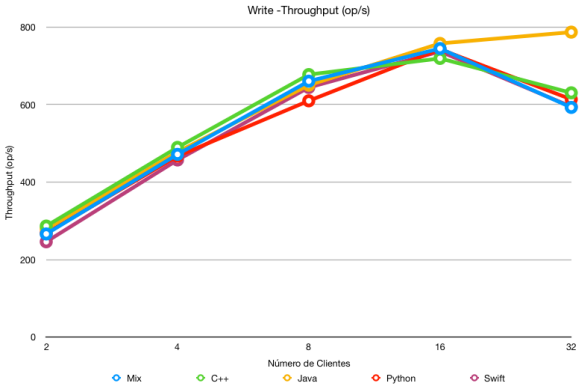


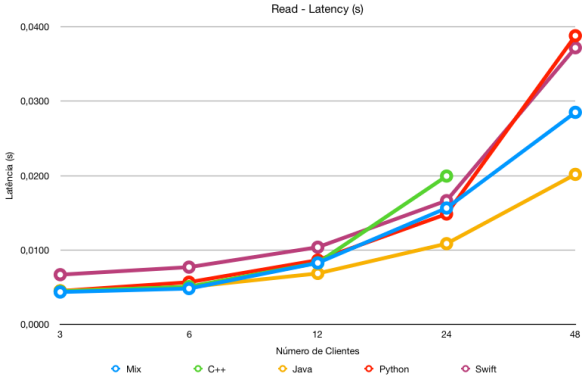
Figura 4.4: Resultados Escrita - 2 máquinas clientes - Protocolo Honesto.

Read - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
3	0,0044	0,0044	0,0045	0,0045	0,0067
6	0,0048	0,0051	0,0050	0,0057	0,0077
12	0,0083	0,0083	0,0069	0,0086	0,0104
24	0,0156	0,0199	0,0109	0,0148	0,0166
48	0,0285		0,0202	0,0388	0,0372

Read -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
3	506	529	522	522	341
6	919	858	928	972	661
12	1050	1292	1133	1256	884
24	1087	1038	1202	1277	962
48	1118		1261	1055	695



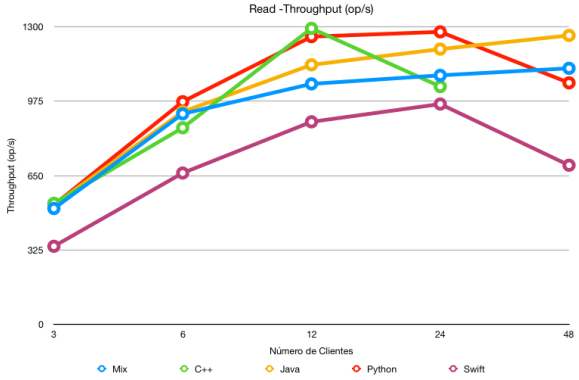


Figura 4.5: Resultados Leitura - 3 máquinas clientes - Protocolo Honesto.

nenhuma carga grande nos servidores, os mesmos atuam apenas como registradores. Dessa maneira os resultados - Figura 4.9, Figura 4.11, Figura 4.13, Figura 4.15- lembram aqueles obtidos para os clientes honestos.

Analisando os resultados de leitura - Figura 4.9, Figura 4.11, Figura 4.13, Figura 4.15 - e os de escrita - Figura 4.10, Figura 4.12, Figura 4.14, Figura 4.16 - pode-se chegar a algumas conclusões:

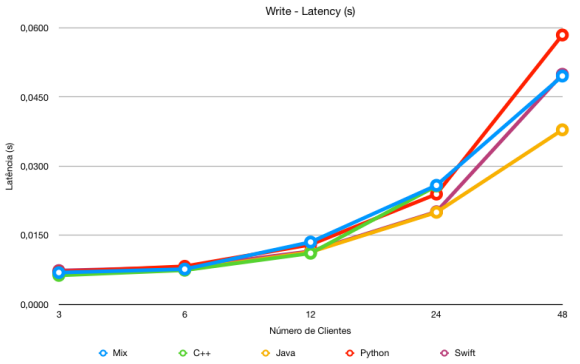
- Alterações no protocolo, como a fase de obter echo, trouxeram um overhead para o protocolo e fizeram com que houvesse um aumento geral da latência e um decréscimo

Write - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
3	0,0069	0,0063	0,0066	0,0071	0,0074
6	0,0077	0,0074	0,0075	0,0083	0,0081
12	0,0135	0,0111	0,0114	0,0130	0,0116
24	0,0258	0,0257	0,0200	0,0239	0,0201
48	0,0495		0,0379	0,0585	0,0499

Write -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
3	362	415	377	352	350
6	657	694	683	606	645
12	774	958	926	808	919
24	809	740	1092	883	1086
48	876		1169	681	872



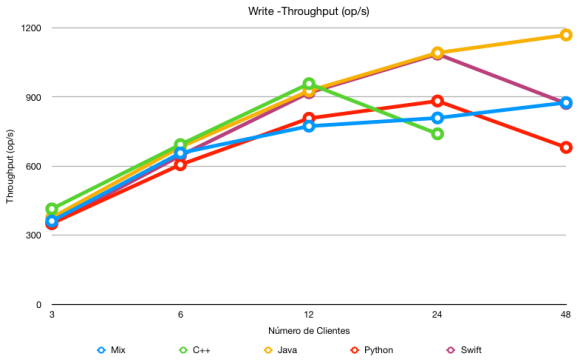


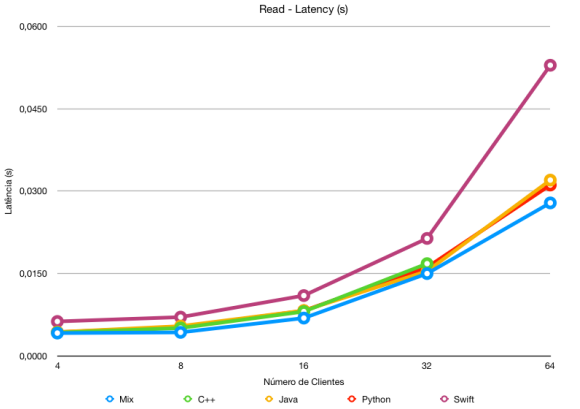
Figura 4.6: Resultados Escrita - 3 máquinas clientes - Protocolo Honesto.

Read - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
4	0,0042	0,0042	0,0043	0,0043	0,0063
8	0,0043	0,0051	0,0054	0,0052	0,0071
16	0,0069	0,0081	0,0083	0,0082	0,0110
32	0,0150	0,0168	0,0154	0,0161	0,0214
64	0,0278		0,0320	0,0311	0,0529

Read -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
4	631	624	617	625	413
8	1195	1081	1168	946	721
16	1689	1548	1563	1179	1115
32	1394	1227	2060	1151	1458
64	1437		2557	1029	1224



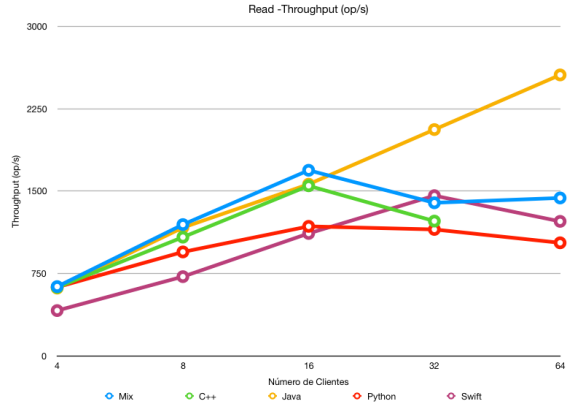


Figura 4.7: Resultados Leitura - 4 máquinas clientes - Protocolo Honesto.

do throughput.

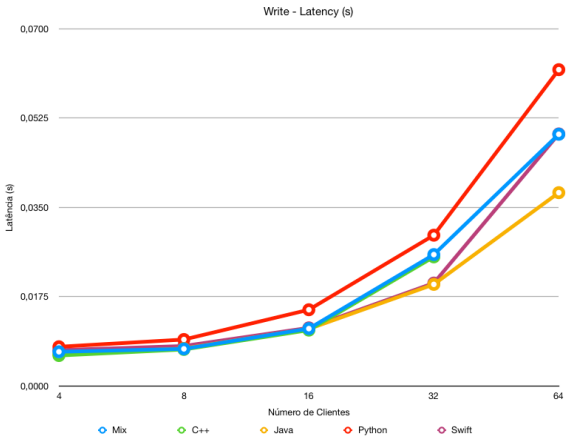
- Na operação de escrita, o servidor Python apresentou um desempenho muito atípico e inesperado. Atribui-se esse desempenho ao uso das bibliotecas de criptografia nativas da linguagem que não devem estar otimizadas.
- Nas operações de leitura e escrita o sistema com diversidade (mix) manteve seu desempenho próximo aos de melhor desempenho (C++ e Java) mesmo com o desempenho baixo do Swift na leitura e do Python na escrita. Isso é possível visto

Write - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
4	0,0067	0,0060	0,0063	0,0077	0,0070
8	0,0073	0,0072	0,0073	0,0091	0,0078
16	0,0112	0,0109	0,0112	0,0149	0,0114
32	0,0257	0,0253	0,0199	0,0295	0,0202
64	0,0493		0,0379	0,0620	0,0494

Write -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
4	434	506	476	434	458
8	867	877	843	645	793
16	1178	1209	1127	837	1145
32	1023	1064	1415	825	1362
64	1087		1612	802	1132



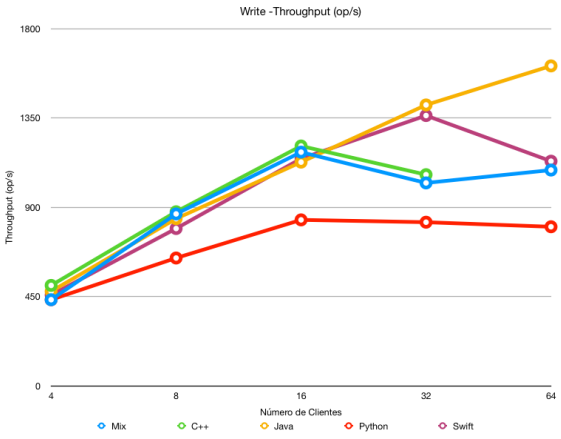


Figura 4.8: Resultados Escrita - 4 máquinas clientes - Protocolo Honesto.

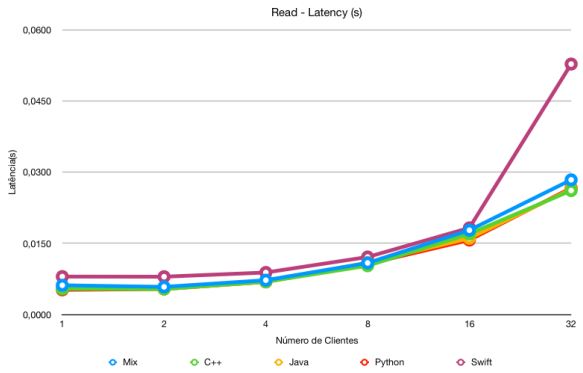
que o protocolo permite até uma falha nos servidores. Dessa forma, as respostas do servidor Swift e Python (de acordo com cada tipo de operação) provavelmente eram descartadas no sistema com diversidade, pois o quórum de respostas era de tamanho 3.

Read - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
1	0,0062	0,0054	0,0056	0,0052	0,0080
2	0,0058	0,0054	0,0054	0,0054	0,0080
4	0,0072	0,0069	0,0069	0,0069	0,0088
8	0,0109	0,0103	0,0109	0,0106	0,0121
16	0,0178	0,0170	0,0161	0,0157	0,0183
32	0,0284	0,0261	0,0264	0,0266	0,0528

Read -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
1	161	182	178	190	124
2	286	308	310	314	219
4	449	466	468	467	390
8	545	558	551	560	524
16	580	597	575	596	579
32	596	589	591	600	513



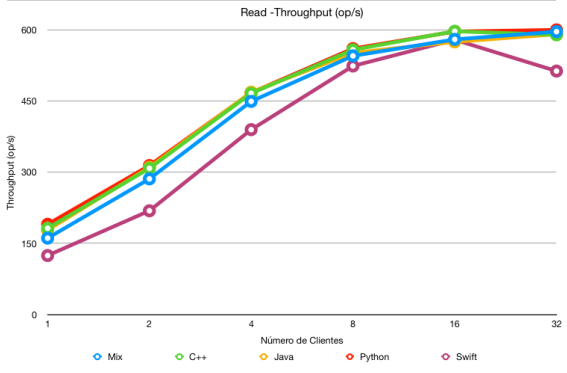


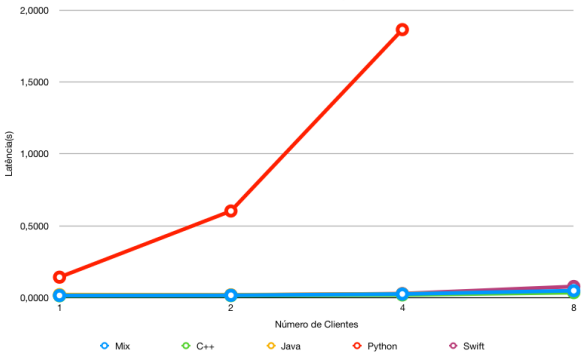
Figura 4.9: Resultados Leitura - 1 máquina cliente - Protocolo Desonesto.

Write - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
1	0,0143	0,0120	0,0201	0,1415	0,0158
2	0,0156	0,0132	0,0187	0,6023	0,0171
4	0,0245	0,0181	0,0258	1,8648	0,0273
8	0,0476	0,0342	0,0430		0,0768

Write -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
1	69	83	49	7	63
2	120	139	100	3	110
4	150	202	144	2	141
8	135	221	174		102



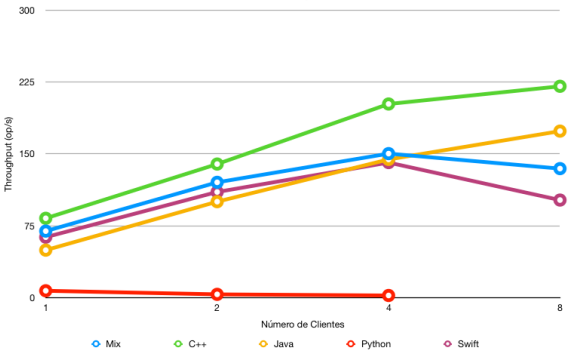


Figura 4.10: Resultados Escrita - 1 máquina cliente - Protocolo Desonesto.

Read - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
2	0,0058	0,0053	0,0054	0,0052	0,0079
4	0,0058	0,0055	0,0055	0,0054	0,0079
8	0,0071	0,0068	0,0068	0,0068	0,0088
16	0,0100	0,0091	0,0093	0,0092	0,0111
32	0,0171	0,0134	0,0130	0,0131	0,0152
64	0,0317	0,0209	0,0205	0,0199	0,1068

Read -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
2	308	337	338	344	234
4	505	529	532	537	407
8	807	835	797	825	707
16	945	980	973	1021	951
32	1017	1065	1049	1051	1032
64	1066	1060	1041	1018	572

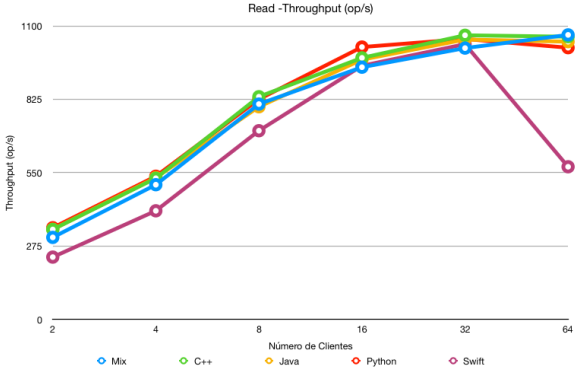


Figura 4.11: Resultados Leitura - 2 máquinas clientes - Protocolo Desonesto.

Write - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
2	0,0172	0,0131	0,0206	0,8273	0,0168
4	0,0210	0,0144	0,0255	1,8202	0,0210
8	0,0322	0,0206	0,0402	3,4652	0,0542
16	0,1098	0,0555	0,0735		

Write -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
2	108	140	91	2	112
4	160	235	143	2	172
8	192	337	183	2	141
16	128	273	203		

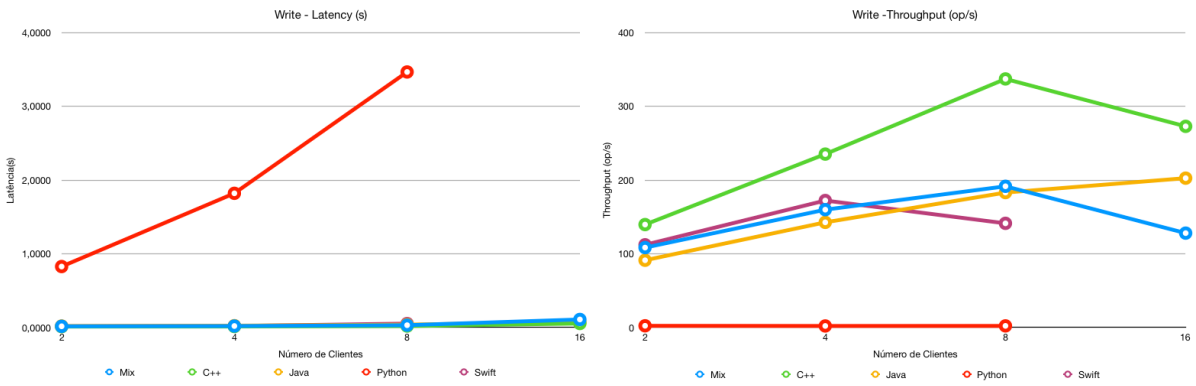


Figura 4.12: Resultados Escrita - 2 máquinas clientes - Protocolo Desonesto.

Read - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
3	0,0056	0,0052	0,0053	0,0053	0,0079
6	0,0057	0,0055	0,0054	0,0055	0,0080
12	0,0070	0,0065	0,0066	0,0067	0,0089
24	0,0117	0,0079	0,0078	0,0084	0,0106
48	0,0178	0,0105	0,0104	0,0138	0,0127
96	0,0349	0,0153	0,0160	0,0182	0,1609

Read -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
3	429	463	461	454	325
6	694	732	737	735	563
12	1050	1095	1105	1104	952
24	1120	1123	1191	1170	1157
48	1207	1239	1256	1211	1182
96	1192	1204	1185	1201	578

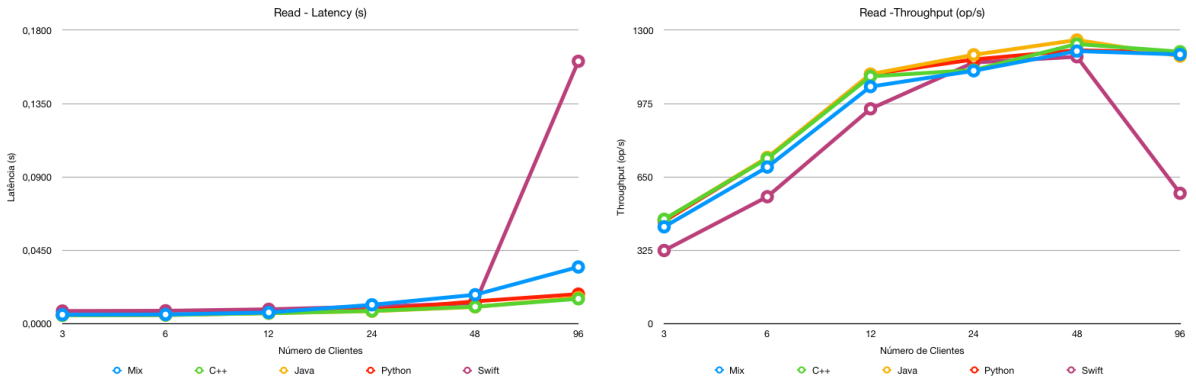


Figura 4.13: Resultados Leitura - 3 máquinas clientes - Protocolo Desonesto.

Write - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
3	0,0187	0,0132	0,0229	1,3500	0,0191
6	0,0269	0,0158	0,0321	2,6511	0,0299
12	0,0662	0,0284	0,0561	5,1539	0,0891
24	0,1878	0,0964	0,1062		

Write -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
3	115	184	117	2	135
6	196	303	166	2	180
12	154	368	197	2	130
24	114	239	214		

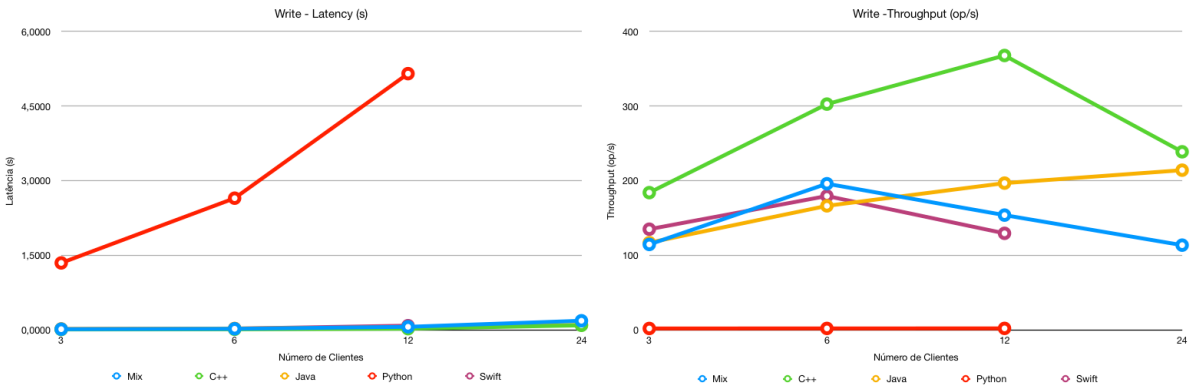


Figura 4.14: Resultados Escrita - 3 máquinas clientes - Protocolo Desonesto.

Read - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
4	0,0056	0,0052	0,0053	0,0052	0,0080
8	0,0058	0,0055	0,0055	0,0055	0,0081
16	0,0072	0,0062	0,0060	0,0063	0,0087
32	0,0120	0,0069	0,0071	0,0096	0,0104
64	0,0204	0,0089	0,0090	0,0151	0,0996
128	0,0282	0,0141	0,0135	0,0249	0,2017

Read -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
4	523	557	552	559	400
8	849	895	889	775	716
16	1133	1159	1151	1144	1071
32	1165	1209	1223	1221	1273
64	1207	1228	1223	1278	618
128	1206	1243	1211	1098	616

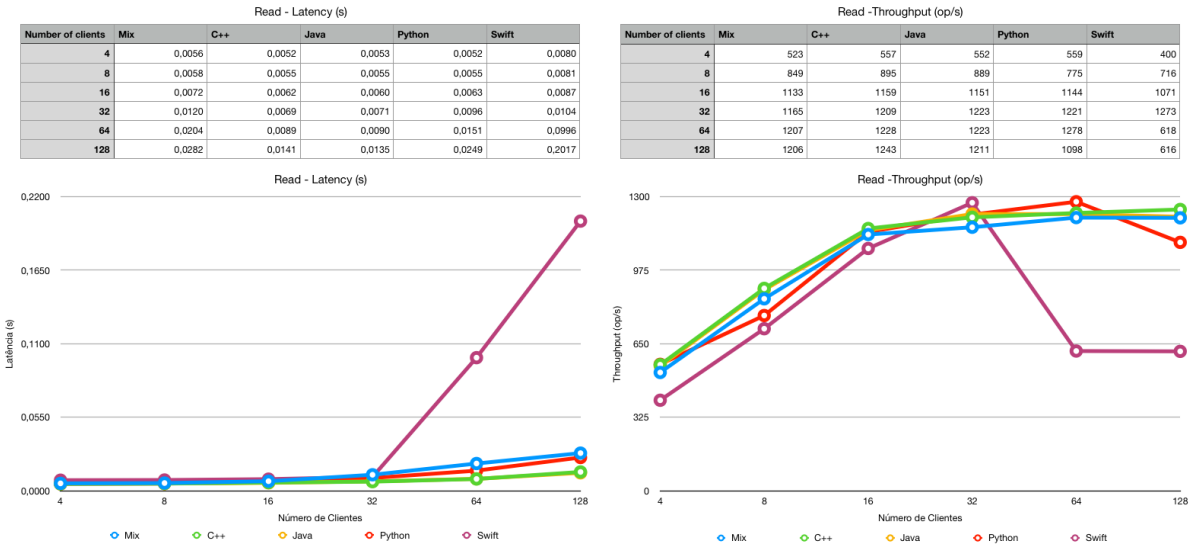


Figura 4.15: Resultados Leitura - 4 máquinas clientes - Protocolo Desonesto.

Write - Latency (s)

Number of clients	Mix	C++	Java	Python	Swift
4	0,0249	0,0143	0,0251	1,8485	0,0223
8	0,0340	0,0174	0,0352	3,4438	0,0435
16	0,0627	0,0429	0,0712	6,8726	0,1342
32		0,1546	0,1364		

Write -Throughput (op/s)

Number of clients	Mix	C++	Java	Python	Swift
4	138	223	135	2	149
8	192	350	173	2	167
16	220	333	205	2	115
32		199	222		

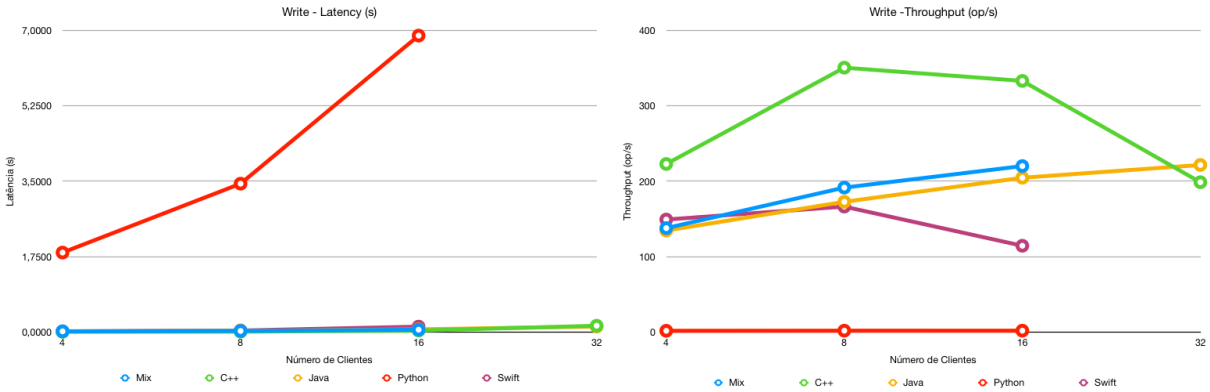


Figura 4.16: Resultados Escrita - 4 máquinas clientes - Protocolo Desonesto.

Capítulo 5

Conclusões

Nessa seção, apresenta-se um resumo sobre o que foi desenvolvido e alcançado no presente trabalho.

5.1 Visão Geral do Trabalho

O presente trabalho apresentou uma nova maneira de se implementar um sistema de quóruns. Ao utilizar-se a técnica de diversidade na implementação das réplicas aumenta-se a segurança de todo o sistema. Neste caso, é improvável que uma mesma vulnerabilidade esteja presente em mais de uma réplica, pois diferentes linguagens e implementações tendem a ter diferentes falhas.

Através dos experimentos foi possível demonstrar que o protocolo proposto e a implementação feita funcionam e entregam um desempenho satisfatório. O desempenho do sistema com diversidade foi muito próximo ao sistema quando utilizando programas em uma só linguagem. Contudo, deve-se pontuar que ao utilizar diversidade de linguagens no sistema, aumentou-se de forma considerável o trabalho de desenvolvimento e manutenção.

5.2 Revisão dos Objetivos e Contribuições

Todos os objetivos específicos foram alcançados. A seguir discute-se cada um deles:

- **Estudar os conceitos de sistemas de quóruns, falhas bizantinas e diversidade.** Foi alcançado por meio do estudo acerca do assunto e uma revisão bibliográfica, que fundamentam essa monografia, conforme discutido no Capítulo 2.
- **Proposta de uma arquitetura capaz de suportar diversidade na implementação das réplicas de um sistema de quóruns bizantino.** Objetivo alcançado através da arquitetura e protocolos propostos no Capítulo 3.

- **Implementação dos servidores em diversas linguagens.** Alcançado através do desenvolvimento dos servidores em C++, Java, Swift e Python, conforme código no Github do projeto.
- **Implementação de clientes em diversas linguagens.** Alcançado através do desenvolvimento dos clientes em Java e Python, conforme código no Github do projeto.
- **Análise Experimental.** Apresentada no Capítulo 4, apresentou os resultados do sistema proposto no presente trabalho, concluindo que a implementação é viável e apresenta bom desempenho.

5.3 Perspectivas Futuras

Como trabalhos futuros, pretende-se implementar servidores em outras linguagens de programação, além de refatorar o código de criptografia utilizado pelo Python. No protocolo desonesto, os servidores implementados em Python apresentaram resultados muito abaixo do esperado. Isso pode ter sido causado pelo processo de criptografia de dados que é feito pelo servidor nesse protocolo. Acredita-se que a biblioteca utilizada não está otimizada e uma outra implementação levaria a melhores resultados para a linguagem. Também pretende-se realizar outros experimentos com mais máquinas de clientes, para entender melhor as limitações de desempenho do sistema. Por fim, seria interessante a implementação e testes de diversidade de hardware no sistema.

Referências

[1] Avizienis, Algirdas, J C Laprie, Brian Randell e Carl Landwehr: *Basic concepts and taxonomy of dependable and secure computing.* IEEE transactions on dependable and secure computing, 1(1):11–33, 2004. 1, 14, 15, 16, 18

[2] Driscoll, Kevin, Brendan Hall, Håkan Sivencrona e Phil Zumsteg: *Byzantine fault tolerance, from theory to reality.* Em *International Conference on Computer Safety, Reliability, and Security*, páginas 235–248. Springer, 2003. 1

[3] Malkhi, Dahlia e Michael Reiter: *Byzantine quorum systems.* Distributed computing, 11(4):203–213, 1998. 1, 20, 21

[4] Bessani, Alysson, Alessandro Daidone, Ilir Gashi, Rafael Obelheiro, Paulo Sousa e Vladimir Stankovic: *Enhancing fault/intrusion tolerance through design and configuration diversity.* Em *Proceedings of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems–WRAITS*, volume 9, 2009. 2

[5] Garcia, Miguel, Alysson Bessani, Ilir Gashi, Nuno Neves e Rafael Obelheiro: *Os diversity for intrusion tolerance: Myth or reality?* Em *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, páginas 383–394. IEEE, 2011. 2

[6] Garcia, Miguel, Alysson Bessani, Ilir Gashi, Nuno Neves e Rafael Obelheiro: *Analysis of operating system diversity for intrusion tolerance.* Software: Practice and Experience, 44(6):735–770, 2014. 2, 21

[7] Obelheiro, Rafael R, Alysson Neves Bessani e Lau Cheuk Lung: *Analisando a viabilidade da implementaçao prática de sistemas tolerantes a intrusoes.* Anais do V Simpósio Brasileiro em Segurança da Informaçao e de Sistemas Computacionais-SBSeg 2005, 2005. 2, 21, 22

[8] Platania, Marco, Daniel Obenshain, Thomas Tantillo, Ricky Sharma e Yair Amir: *Towards a practical survivable intrusion tolerant replication system.* Em *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*, páginas 242–252. IEEE, 2014. 2

[9] Tanenbaum, Van Steen: *Distributed Systems: Principles and Paradigms.* Pearson, segunda edição, 2006, ISBN 0132392275. 4, 5, 6, 8, 10, 11, 12, 13

[10] Coulouris, George F, Jean Dollimore e Tim Kindberg: *Distributed Systems Concepts and Design.* Pearson, quinta edição, 2011, ISBN 0132143011. 4, 5

[11] Neuman, B Cliord: *Scale in distributed systems*. ISI/USC, 1994. 9

[12] Netzer, Robert HB e Barton P Miller: *What are race conditions?: Some issues and formalizations*. ACM Letters on Programming Languages and Systems (LOPLAS), 1(1):74–88, 1992. 11

[13] Chandy, K Mani, Jayadev Misra e Laura M Haas: *Distributed deadlock detection*. ACM Transactions on Computer Systems (TOCS), 1(2):144–156, 1983. 11

[14] Vukolić, Marko *et al.*: *The origin of quorum systems*. Bulletin of EATCS, 2(101), 2013. 19

[15] Thomas, Robert H: *A majority consensus approach to concurrency control for multiple copy databases*. ACM Transactions on Database Systems (TODS), 4(2):180–209, 1979. 19

[16] Lamport, Leslie, Robert Shostak e Marshall Pease: *The byzantine generals problem*. ACM Transactions on Programming Languages and Systems (TOPLAS), 4(3):382–401, 1982. 20

[17] Chen, Liming e Algirdas Avizienis: *N-version programming: A fault-tolerance approach to reliability of software operation*. Em *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, páginas 3–9, 1978. 21

[18] Malkhi, Dahlia e Michael K Reiter: *Secure and scalable replication in phalanx*. Em *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, páginas 51–58. IEEE, 1998. 26, 27

[19] Kurose, James F: *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India, 2005. 29

[20] Tanenbaum, Andrew S e Nery Machado Filho: *Sistemas operacionais modernos*, volume 3. Prentice-Hall, 1995. 29, 30

[21] *Introdução ao json*. `https://www.json.org/json-pt.html`, acesso em 2018-05-01. 30, 31, 32

[22] Somani, Uma, Kanika Lakhani e Manish Mundra: *Implementing digital signature with rsa encryption algorithm to enhance the data security of cloud in cloud computing*. Em *Parallel Distributed and Grid Computing (PDGC), 2010 1st International Conference on*, páginas 211–216. IEEE, 2010. 32

[23] Josefsson, Simon: *The base16, base32, and base64 data encodings*. 2006. 33

[24] Viega, John, Matt Messier e Pravir Chandra: *Network Security with OpenSSL: Cryptography for Secure Communications*. " O'Reilly Media, Inc.", 2002. 52

[25] White, Brian, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb e Abhijeet Joglekar: *An integrated experimental environment for distributed systems and networks*. ACM SIGOPS Operating Systems Review, 36(SI):255–270, 2002. 54