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

TOLERÂNCIA A FALHAS PARA SISTEMAS LEGADOS: UM ESTUDO DE CASO NO EXÉRCITO BRASILEIRO

Fausto Andrade dos Santos Junior

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientadora
Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo von Paumgartten

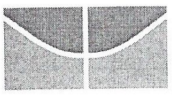
Brasília
2015

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

Junior, Fausto Andrade dos Santos
J95t TOLERÂNCIA A FALHAS PARA SISTEMAS LEGADOS: UM ESTUDO DE CASO NO EXÉRCITO BRASILEIRO / Fausto Andrade dos Santos Junior; orientador Aletéia Patrícia Favacho de Araújo von Paumgartten; co orientador Edward Ribeiro. -- Brasília, 2015.
118 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2015.

1. Sistemas Distribuídos. 2. Middleware Orientado a Mensagem. 3. Sistemas Legados. 4. Tolerância a Falhas. I. Patrícia Favacho de Araújo von Paumgartten, Aletéia, orient. II. Ribeiro, Edward, co-orient. III. Título.



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

Tolerância a Falhas para Sistemas Legados: um Estudo de Caso no Exército Brasileiro

Fausto Andrade Santos Júnior

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof.ª Dr.ª Aleteia Patrícia Favacho de Araújo von Paumgartten (Orientador)
Departamento de Ciência da Computação/UnB

Prof.ª Dr.ª Maristela Terto de Holanda
Departamento de Ciência da Computação/UnB

Prof. Dr. Cristiano Maciel
Universidade Federal de Mato Grosso

Prof. Dr. Marcelo Ladeira
Coordenador do Programa de Pós-graduação em Computação Aplicada

Brasília, 24 de agosto de 2015

Dedicatória

Dedico este trabalho aos meus pais, Fausto (*in memorian*) e Maria Aparecida (*in memorian*), que me ensinaram o justo e o correto, virtudes que me trouxeram até aqui. Dedico ao meu irmão Edilson (*in memorian*) por me auxiliar no caminho da honestidade e do caráter. Dedico também aos meus filhos Felipe, Manuela e Lívia, que tanto me apoiaram nesta pesquisa e são os motivos para eu continuar lutando.

Agradecimentos

Agradeço a Deus, por me permitir viver o suficiente para presenciar suas grandiosas obras. Agradeço também ao Exército Brasileiro, por me auxiliar no processo de ingresso e de estudos no mestrado, em reconhecimento pelos serviços prestados. Agradeço também a todos os familiares e amigos, que direta ou indiretamente prestaram apoio para que esta pesquisa ocorresse da melhor forma possível. Por fim, agradeço aos orientadores e membros da banca, e ao Prof. Dr. João Batista pelo incentivo em ingressar no programa de Mestrado.

Resumo

Soluções modernas de arquitetura de sistemas indicam, muitas vezes, a computação distribuída como meio de satisfazer características desejáveis de desempenho. Uma classe de arquitetura distribuída em especial é o MOM ou *Middleware* Orientado à Mensagem. Esta arquitetura é capaz de conceder um mecanismo de comunicação assíncrona, em que é possível implementar fatores como escalabilidade e tolerância à falhas, de maneira transparente. O Exército Brasileiro conta com diversos sistemas que proporcionam fatores similares aos proporcionados pela arquitetura MOM, embora boa parte dos sistemas legados não admita esta abordagem. O Departamento-Geral do Pessoal - DGP, responsável pelos processos que envolvem recursos humanos no Exército, utiliza sistemas legados ao prover vários serviços. Alguns destes sistemas participam de processos sensíveis, como é o caso observado do processo de concessão de benefícios do sistema previdenciário, em que o sistema Papiro cataloga e concentra a documentação comprobatória de direitos. Seu objetivo é prover o acesso aos documentos cadastrados para consulta e averiguação de direitos pelos beneficiários, e que deve ocorrer na região onde reside a pessoa interessada. A arquitetura do sistema Papiro utiliza dados do negócio armazenados diretamente em arquivos, e não possui mecanismos para oferecer escalabilidade e tolerância a falhas nativamente. Seu uso é limitado em função da disponibilidade do sistema no território nacional, tendo em vista que seus usuários o acessam a partir das organizações militares presentes nas regiões centrais e fronteiras do país. Esta pesquisa tem por objetivo compor uma proposta de infraestrutura de armazenamento para sistemas legados do DGP, que proporcione a tolerância à falhas e a escalabilidade, sem que para isso deva-se ajustar as aplicações legadas hospedadas. A arquitetura elaborada baseia-se no StackSync, um MOM capaz de sincronizar espaços de armazenamento em ambientes heterogêneos. Os serviços necessários para a construção da arquitetura são detalhados, de acordo com sua participação no ambiente. O estudo foi realizado utilizando dados extraídos do sistema Papiro. Com os resultados obtidos, foi possível avaliar a aplicação da infraestura e o custo de hardware necessário para sua implementação pelo Exército Brasileiro.

Palavras-chave: Sistemas legados, Sistemas distribuídos, Tolerância a falhas.

Abstract

Modern solutions of systems architecture indicates, often, the distributed computing as a mean to meet desired performance characteristics. An distributed architecture class in particular is MOM or Message Oriented Middleware. This architecture is able to provide a mechanism for asynchronous communication, in which it is possible to implement such factors as scalability and fault tolerance in a transparent manner. The Brazilian Army has several systems that provide similar factors to those provided by MOM architecture, although much of the legacy systems will not admit this approach. The Departamento-Geral do Pessoal - DGP, responsible for processes that involve human resources in the Army, uses legacy systems to provide various services. Some of these systems participate in sensitive cases, such as observed from the concession of benefits concerning the social security system process, in which the Papiro system catalogs and concentrates supporting documentation. Its goal is to provide access to registered documents for inquiry and investigation about recipient rights, and that should occur in the region where the person concerned resides. The Papiro system's architecture uses business data stored directly in files, and does not have mechanisms to provide scalability and fault tolerance natively. Its use is limited depending on system availability in the country, given that their users access from military organizations present in the central regions and the country's borders. This research aims to compose a storage infrastructure proposal to the DGP legacy systems, which provides the fault tolerance and scalability, without adjusting the hosted legacy applications. The developed architecture is based on StackSync, a MOM capable of storage synchronization between heterogeneous environments. The services needed for the construction of the architecture are detailed, according to their participation in the environment. The study was carried out using data extracted from Papiro system. With the results obtained, it was possible to evaluate the infrastructure's implementation and the cost of hardware needed to deliver this solution to the Brazilian Army.

Keywords: Legacy systems, Distributed systems, Fault Tolerance.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos últimos anos, a computação distribuída tem permeado os ambientes computacionais modernos. Todavia, nota-se nos cenários corporativos um grande número de aplicações legadas e diversos motivos que impedem sua adequação ao cenário da computação distribuída. Assim sendo, quanto mais as pesquisas nessas áreas avançam, mais percebe-se o grande custo em migrar soluções corporativas de sistemas de planejamento de recursos, de gestão de clientes e outros, com grande volume de funcionalidades e sensíveis aos ambientes em que operam.

Os órgãos públicos de maneira geral, e em particular o Exército Brasileiro, externam desafios devido à capilaridade de suas repartições nacionais e internacionais. Entre eles, destacam-se o de manter sistemas de informações disponíveis em todos os quartéis do Exército e, mesmo diante de intempéries operacionais, permitir a sincronia dos trabalhos realizados nas organizações militares. Assim, sobressai-se que diversos sistemas que necessitam dessas características não foram planejados, ou sequer dispõem de arquitetura compatível para computação distribuída, o que dificulta, e em alguns casos limita, a disponibilidade dos sistemas.

Para Tanenbaum e Steen [46], um sistema distribuído é uma coleção de computadores independentes que se apresentam ao usuário como se fossem um único sistema. Isso implica no que se espera deles – componentes autônomos que colaboram entre si e de forma transparente ao usuário. Uma das formas mais empregadas para implantar ambientes de computação distribuída é o uso de *middleware* orientado a mensagens ou MOM (*Message Oriented Middleware*). A função de um MOM é promover a troca de informações entre computadores de forma transparente para uma finalidade comum [11].

O Departamento-Geral do Pessoal – DGP, Órgão de Direção Setorial do Exército Brasileiro, tem por missão planejar, orientar, coordenar e controlar as atividades do Sistema de Pessoal do Exército, sendo responsável também pela execução das atividades de administração de pessoal que lhe são atribuídas pela legislação específica [23]. Neste

cenário, muitas aplicações críticas não possuem arquitetura compatível com programação distribuída, dificultando o funcionamento da organização. Além disso, o centro de dados do DGP encontra-se em Brasília, no Distrito Federal, e não possui equivalente em outra região do país. Assim sendo, em uma situação de catástrofe natural ou sinistro, o Departamento ficaria impossibilitado de prestar seus serviços até que todo o ambiente fosse reconstruído.

Diante do exposto, este trabalho propõe a implementação de uma plataforma transparente de *middleware* para hospedar sistemas legados. O objetivo da proposta é atribuir aos sistemas legados, de maneira transparente, a característica de tolerância a falhas, um fator importante e, atualmente, indisponível para a grande maioria dos sistemas do DGP.

1.1 Motivação

Em um país de dimensões continentais como é o Brasil, disponibilizar aplicações de forma distribuída é uma característica desejável, pois há processos em que há colaboração de usuários em diversas cidades do país. É o caso do sistema previdenciário, em que muitas vezes se observa que os dependentes de militares residem em cidades diferentes ao solicitar benefícios.

Para apoiar o processo de concessão de pensões, o Exército conta com o sistema Papiro, que cataloga e concentra os documentos comprobatórios de direitos de militares e seus dependentes. Este sistema atende atualmente mais de setenta mil beneficiários diretos, com a perspectiva de catalogação completa passando de um milhão de militares na reserva e pensionistas do sistema previdenciário do Exército. Os documentos catalogados no sistema Papiro são essenciais para o processo de concessão de benefícios, e sua falta causa atrasos nos processos. Os usuários do sistema previdenciário do Exército dependem dos benefícios, e muitas vezes num momento de fragilidade, em virtude do falecimento de familiares. No entanto, a celeridade e precisão dos processos depende, em grande parte das situações, de que os documentos sejam catalogados o mais breve possível e com documentos originados em várias cidades para o mesmo processo.

Segundo o portal institucional do DGP [23], os processos de negócio do DGP são realizados em boa parte nas Organizações Militares espalhadas pelo território nacional. Na condição em que o sistema Papiro se insere no contexto, sua execução depende da disponibilidade de acesso ao datacenter do DGP, o que muitas vezes é limitado por questões físicas, como por exemplo as linhas de transmissão que separam o DGP da região atendida.

O sistema Papiro não possui mecanismos que permitam sobrepor este tipo de limitação, e não há perspectiva de melhoria de seu código para implementar novas funcionalidades

ou recursos. Nesse sentido, este trabalho objetiva propor e implementar mecanismos para permitir a tolerância a falhas de aplicações legadas, de forma não intrusiva, e que o resultado garanta manter o código das aplicações.

1.2 Definição do Problema

Alguns sistemas legados do DGP operam sobre informações armazenadas em arquivos. Neste contexto, há diversos sistemas, com diferentes arquiteturas de hardware e de sistema operacional, módulos de software e outros componentes que incrementam a complexidade do ambiente. Alguns sistemas gerenciam diretamente os arquivos de dados como objeto de negócio, enquanto outros usam as informações em estruturas de dados dentro de arquivos. Como a arquitetura em que estes sistemas foram modelados não dispõe nativamente a computação distribuída, permitir tolerância a falhas e escalabilidade torna-se uma tarefa inviável sem a reescrever as aplicações, quer seja o próprio sistema, ou seja da plataforma operacional que o hospeda.

Cabe observar que não é objeto de pesquisa deste trabalho a sincronização dos bancos de dados das aplicações legadas. O Exército Brasileiro conta com projetos de distribuição das bases de dados corporativas, e mesmo as aplicações legadas utilizam esses dados. Além disso, a replicação de bancos de dados em arquiteturas obsoletas pode ser um propósito de pesquisas futuras, complementando este trabalho.

Diante do cenário de aplicações legadas, o objetivo deste trabalho é oferecer uma infraestrutura tolerante a falhas. A camada de infraestrutura que implementar a tolerância a falhas precisa garantir transparência e escalabilidade, e ainda precisa estar fortemente ligada à plataforma operacional que hospeda cada sistema legado.

1.3 Objetivos

1.3.1 Geral

O objetivo geral deste trabalho é implementar tolerância a falhas e escalabilidade à infraestrutura de armazenamento de sistemas legados. A abordagem se limita ao armazenamento de arquivos de dados manipulados por estas aplicações, tendo em vista que o DGP possui vários sistemas que manipulam arquivos de dados como objeto de negócio.

1.3.2 Específicos

Para que o objetivo geral deste trabalho seja atingido, faz-se necessário atender aos seguintes objetivos específicos:

- Identificar um MOM que satisfaça os parâmetros operacionais da proposta;
- Elaborar uma arquitetura baseada em MOM que permita a tolerância a falhas sob a perspectiva de armazenamento; e
- Identificar métricas de desempenho para a proposta, focando a sincronia do armazenamento distribuído, particularmente, das aplicações legadas.

1.4 Estrutura do Trabalho

Este trabalho está dividido em mais sete capítulos. No Capítulo 2 são apresentados os conceitos essenciais de sistemas distribuídos, as vantagens e as limitações de sua aplicação, e os principais desafios para esta opção de projeto. Também são apresentadas as principais arquiteturas e tipos de sistemas distribuídos, e situações em que cada opção é mais apropriada.

No Capítulo 3 são descritos os conceitos relacionados à tolerância a falhas, o tratamento de defeitos em sistemas computacionais, e as principais técnicas de redundância. No Capítulo 4, são apresentados os conceitos básicos acerca de *Middleware* Orientado a Mensagens (MOM), e da arquitetura *Remote Procedure Call* (RPC). Para o concreto entendimento dos assuntos, são apresentados os conceitos a respeito de filas de mensagens no contexto de MOM, e as filas mais utilizadas na arquitetura. Por fim, são apresentados os principais serviços da arquitetura MOM, e como podem ser aplicados no cenário de computação distribuída.

Por sua vez, o Capítulo 5 apresenta o *middleware* StackSync, voltado para a sincronia de espaços de armazenamento. São apresentados seus principais componentes e mecanisos básicos, bem como os elementos de sua estrutura que permitem as funcionalidades previstas em um MOM.

A atuação do DGP como órgão responsável pelo controle de recursos humanos no Exército Brasileiro está descrita no Capítulo 6. Em particular, observa-se a participação da Diretoria de Inativos, Civis e Pensionistas e de Assistência Social (DCIPAS) nos processos relacionados ao sistema previdenciário do Exército, e a importância do sistema Papiro nos processos de concessão de benefícios e direitos, em todo o território nacional.

O *middleware* StackSync é utilizado para compor a arquitetura proposta neste trabalho, e, no Capítulo 7, está descrita a infraestrutura em que os sistemas legados serão hospedados para garantir tolerância a falhas ao seu funcionamento. Para avaliar a arquitetura, é apresentado um ambiente de simulação, no qual realizaram-se testes de carga e operação do ambiente de armazenamento compartilhado. Os resultados dos testes são

discutidos, considerando-se as características esperadas para o funcionamento da arquitetura.

Por fim, o Capítulo 8 conclui o presente trabalho e sugere alguns trabalhos futuros.

Capítulo 2

Sistemas Distribuídos

Este capítulo apresenta os conceitos fundamentais acerca de sistemas distribuídos, necessários para a compreensão dos assuntos de interesse desta pesquisa. Para isso, são apresentados os objetivos da plataforma distribuída, as vantagens e limitações de sua aplicação, e por fim os principais desafios desta opção de projeto. Em seguida, são apresentados os principais tipos e arquiteturas de sistemas distribuídos, bem como as situações em que cada um é mais apropriado.

2.1 Conceitos Essenciais

Historicamente, durante a evolução de sistemas computacionais, a capacidade de processamento cresceu exponencialmente. Parte desse avanço se deve a pesquisas sobre a execução de tarefas em paralelo e a coordenação desse processamento em um ou mais processadores. Os primeiros computadores eram vistos como dispositivos sequenciais, com limitações físicas para implementar o paralelismo na execução de instruções. Contudo, com o surgimento de multiprocessadores, tornou-se possível utilizar de forma ainda melhor o potencial do hardware, implementando equipamentos e aplicativos capazes de executar instruções em paralelo. Segundo Deitel et al. [13], um sistema de multiprocessamento pode ser definido como qualquer sistema que contenha mais de um processador. Para agrupar e classificar sistemas computacionais com relação ao grau de paralelismo de suas arquiteturas, Michael Flynn [18] propôs um modelo, conhecido como Taxonomia de Flynn. Segundo ele, qualquer máquina pode ser classificada segundo o fluxo de instruções e o fluxo de dados. Assim, há quatro possíveis, as quais são:

- **SISD** (*Single Instruction Stream, Single Data Stream*): este tipo caracteriza a arquitetura mais elementar, que suporta um fluxo único de instruções e também um fluxo único de dados. Este é o modelo no qual se enquadram todos os computadores pessoais monoprocessados.

- **SIMD** (*Single Instruction Stream, Multiple Data Stream*): esta categoria de arquitetura engloba soluções de uma ou mais unidades de processamento, onde há grande paralelismo de dados, mas com operações sequenciais. Um caso típico é o de processamento de dados gráficos vetoriais, em que há grande quantidade de operações aritméticas sobre os dados gráficos, mas que também permite o paralelismo de instruções, sem ferir a sequência de execução.
- **MISD** (*Multiple Instruction Stream, Single Data Stream*): esta arquitetura apresenta múltiplos fluxos de instruções, porém somente um fluxo de dados. Sua implementação não é comumente encontrada, em especial porque é muito complexa tanto para construir quanto para controlar.
- **MIMD** (*Multiple Instruction Stream, Multiple Dada Stream*): esta categoria agrupa sistemas que dispõem de unidades de multiprocessamento atuando independentemente, e sobre fluxos de dados diversos e separados.

Para sintetizar, Gomes [25] apresenta graficamente a correlação entre fluxos de instruções e de dados, que serve de base para a taxonomia de Flynn, conforme mostrada na Figura 2.1.

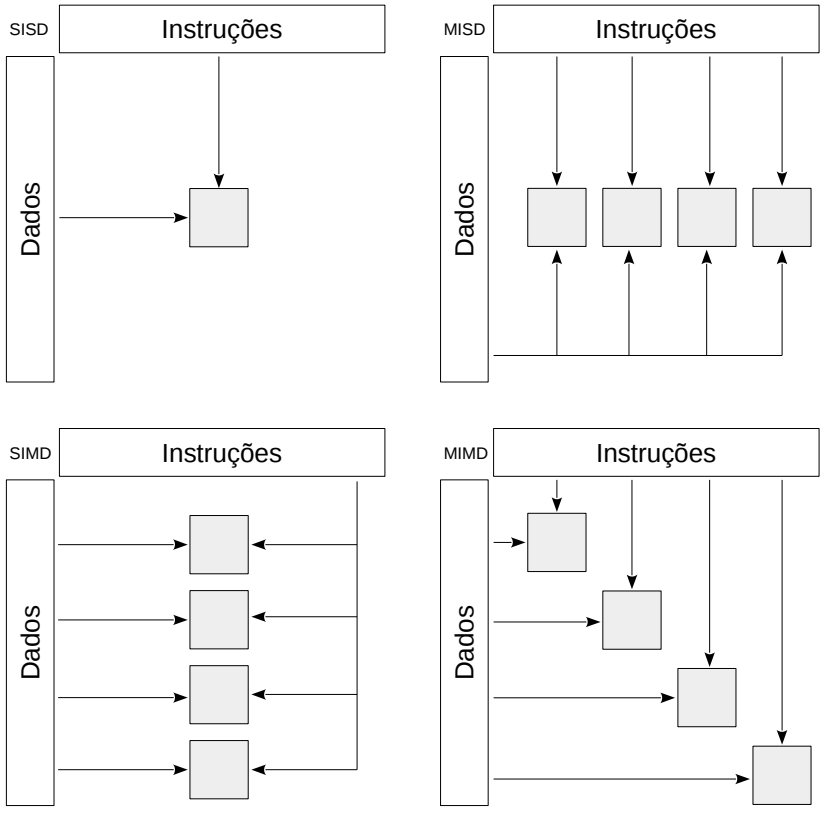


Figura 2.1: Taxonomia de Flynn e a Correlação entre Instruções e Dados [25].

Machado e Maia [37] expõem que as arquiteturas baseadas em MIMD também podem ser classificadas quanto ao seu grau de acoplamento. O acoplamento é a característica que define o quão interdependentes são os componentes de um sistema, no que se refere ao compartilhamento e acesso à memória principal, e ainda os métodos de comunicação, sincronia e a velocidade de processadores. Assim, a classificação proposta por Machado e Maia [37] é apresentada na Figura 2.2.

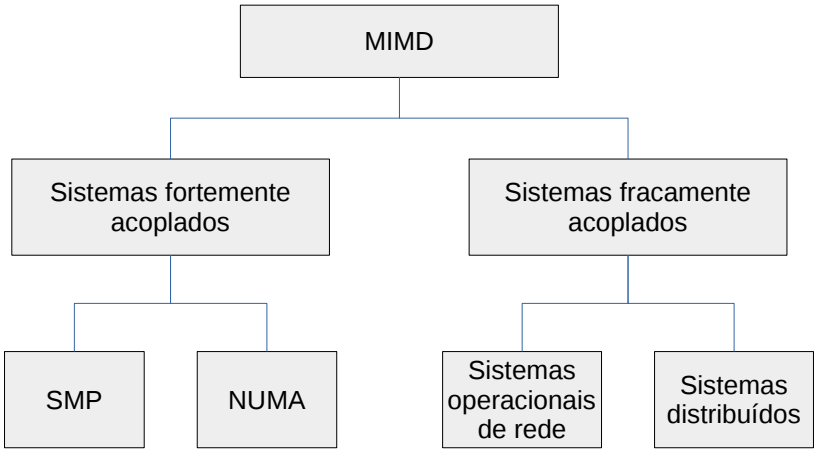


Figura 2.2: Classificação de Flynn sobre Sistemas com Múltiplos Processadores [37].

Sistemas fortemente acoplados (*tightly coupled system*) utilizam uma única memória principal, compartilhada por todos os processadores sob a regência de um único sistema operacional [10, 13, 46]. Em outra modelagem, a arquitetura de sistemas fracamente acoplados (*loosely coupled system)* interliga seus componentes por uma rede de comunicação, sob a gerência de diversos sistemas operacionais, processadores independentes e espaços de memória separados, em dois ou mais sistemas computacionais diferentes [13, 37].

Os sistemas fortemente acoplados, também conhecidos por arquitetura de multiprocessadores, se subdividem em SMP (*Symetric Multiprocessors* ou Multiprocessadores Simétricos) e NUMA (*NonUniform Memory Access* ou Acesso Não Uniforme à Memória). Os sistemas SMP, segundo Machado e Maia [37], "possuem dois ou mais processadores compartilhando um único espaço de endereçamento e gerenciados por apenas um sistema operacional". Um ponto importante é que o tempo de acesso à memória é uniforme entre os processadores, ou seja, todos possuem o mesmo desempenho de acesso à memória. Em sistemas NUMA, "o tempo de acesso à memória principal depende da localização do processador" [37]. Isso se deve ao fato de que, apesar da memória ser compartilhada, ela fica dividida em blocos associados a processadores formando conjuntos, e por fim conectados por um barramento. Assim, quando um processador acessa um bloco de memória local, dentro do conjunto a que pertence, o desempenho do acesso é superior em comparação ao acesso a um bloco de memória remoto.

Os sistemas fracamente acoplados, segundo Deitel et al. [13], "normalmente conectam componentes indiretamente através de canais de comunicação. Em alguns casos os processadores compartilham memória, mas frequentemente cada processador mantém sua memória local, para a qual o acesso é muito mais rápido do que para o resto da memória disponível ao sistema. Em outros casos, a troca de mensagens é a única forma de comunicação entre processadores, e a memória não é compartilhada".

Em geral, sistemas fracamente acoplados são muito mais flexíveis e escaláveis do que sistemas fortemente acoplados [10]. Outra designação empregada para este tipo de sistema é a de arquitetura multicomputadores. As principais características de projeto e operação agrupam sistemas fracamente acoplados em *clusters*, sistemas operacionais de rede e sistemas distribuídos.

Sistemas operacionais de rede, segundo Machado e Maia [37], são bons exemplos de sistemas fracamente acoplados. Cada computador na rede, identificado como nó, possui sistema operacional, processador, espaço de endereçamento e dispositivos periféricos próprios, e pode consumir recursos de outros nós de uma rede de comunicação. Todos os nós operam independentemente dos demais, e não há restrição de hardware ou software utilizado em cada nó, desde que se estabeleça um protocolo de comunicação para troca de informações. Entenda-se por "recurso", quaisquer elementos que podem ser compartilhados em um sistema de computadores interligados em rede, abrangendo componentes de hardware, como discos e impressoras, e ainda itens de software, como arquivos, bancos de dados e objetos de todos os tipos [10].

A definição de sistema distribuído é ampla, mas tangencia fatores elementares. Tanenbaum e Steen [46] conceituam como uma coleção de computadores independentes que o usuário visualiza ser um único sistema. Para Coulouris et al. [10], um sistema distribuído é uma plataforma constituída de computadores interligados em rede, que se comunicam e coordenam suas ações apenas enviando mensagens entre si. Lamport [32] complementa que a computação distribuída, que é a ação realizada por sistemas distribuídos, define-se por atividades executadas em um espaço fisicamente distribuído. Cabe enfatizar que não há limitações quanto à distância física que separa os diversos componentes do sistemas distribuídos, e os componentes, de hardware e de software, podem estar hospedados em quaisquer computadores, como computadores pessoais, ambientes de CPD (Centro de Processamento de Dados) ou equipamentos de automação industrial.

A seguir são apresentados os objetivos relacionados ao projeto de sistemas distribuídos, as vantagens de sua aplicação e as limitações a que estão sujeitos. Para atingir os objetivos desta pesquisa, também são explanados os tipos de sistemas distribuídos e suas subdivisões, provendo assim os elementos necessários para o entendimento deste trabalho.

2.2 Objetivos de Sistemas Distribuídos

O projeto de sistemas distribuídos visa contemplar a perspectiva do usuário em ter uma visão não-distribuída de um sistema distribuído, omitindo a complexidade de seu funcionamento, oriunda da localização e não uniformidade dos elementos que o compõe e da comunicação existente entre estes [45]. O projeto de um sistema baseado em plataforma distribuída visa contemplar os objetivos descritos a seguir [10, 46].

2.2.1 Compartilhamento de recursos

Um dos principais objetivos de um sistema distribuído é permitir o acesso a recursos remotos de forma ágil e simplificada [46]. Para isso, deve haver um padrão de acesso aos recursos, de forma que questões como localização física e quantidade de réplicas não sejam problemas no futuro. O compartilhamento favorece o intercâmbio de informações entre usuários e aplicações, mas também abre oportunidade para que intrusos e elementos mal intencionados capturem dados da comunicação entre entidades do sistema. O projeto de um sistema distribuído deve prover, na mesma medida que o compartilhamento de recursos, mecanismos de proteção dos canais de comunicação.

2.2.2 Abertura

Um sistema aberto implementa regras de acesso a seus serviços, padronizando semântica e sintaxe de chamada. Isso favorece substancialmente a expansão do sistema, seja durante a integração com aplicativos existentes ou legados, seja durante a implementação de novos módulos. A associação entre aplicações ocorre por meio de protocolos de comunicação, no qual interfaces definem a sintaxe de acesso a recursos. O conjunto de descrição sintática de interfaces de um sistema distribuído é comumente intitulado IDL (*Interface Definition Language* ou Linguagem de Definição de Interface). As interfaces definidas na IDL são definidas como "publicadas" [10]. Uma interface precisa identificar o seu objetivo, mas não necessariamente sua implementação interna. Isso significa que, mesmo que a implementação da interface mude, seu acesso deve continuar o mesmo, sem que isso reflita em aplicações remotas que a utilizam. Dessa forma, o sistema dito aberto torna-se flexível e extensível, pois as alterações na IDL ocorrerão de forma independente das demais interfaces existentes.

Como a definição das interfaces é de livre escolha do desenvolvedor, uma preocupação importante fica por conta da sua completude e neutralidade [46]. Uma interface é dita completa quando sua especificação apresenta todos os elementos necessários para que um desenvolvedor a utilize, sem tratar de questões específicas de implementação sobre como

aplicá-la. E uma interface é neutra quando não restringe a forma como o desenvolvedor a utilizará, ou seja, permite livre interoperabilidade e portabilidade com demais aplicações que requeiram seu uso. Sistemas interoperáveis trabalham em conjunto, sob um padrão comum de comunicação, mesmo provenientes de diferentes arquiteturas. A portabilidade refere-se ao fato de que uma interface deve prover acesso a um sistema A e, sem modificar a interface, um diferente sistema distribuído B deve conseguir acessá-la. Coulouris et al. [10] resumem as principais características de sistemas abertos:

- Suas principais interfaces são publicadas;
- Há um mecanismo de comunicação uniforme para acesso a interfaces e recursos compartilhados; e
- Podem ser construídos sobre hardware e software heterogêneos.

2.2.3 Heterogeneidade

Um sistema distribuído deve sobrepujar as diferentes plataformas de hardware e sistemas operacionais que hospedam recursos necessários para seu funcionamento, fazendo com que haja comunicação entre todos os elementos. Coulouris et al. [10] indicam que sistemas distribuídos encontram diversos desafios de heterogeneidade, conforme enumerados a seguir.

- **Redes**: a codificação de informações varia conforme os dados trafegam em diferentes redes, desde o formato de dado trafegado, até características como atraso de transmissão e a ordem em que a informação é entregue;
- **Hardware**: não há garantia ou restrição quanto ao tipo de hardware que hospeda aplicações, nem aos dispositivos de conexão entre elas;
- **Sistemas operacionais**: não há restrições quanto a sistemas operacionais que hospedam aplicações ou recursos;
- **Linguagens de programação**: desde que haja um protocolo e uma IDL bem definidos, a linguagem de programação utilizada em cada aplicação é livre; e
- **Código de diferentes desenvolvedores**: cada desenvolvedor tem liberdade de encontrar a solução que considerar melhor para determinados problemas, o que fatalmente gera diversidade de código, mesmo que para resolver os problemas mais elementares.

2.2.4 Escalabilidade

A escalabilidade, em computação distribuída, representa a habilidade de um sistema em aumentar ou diminuir sua capacidade de processamento e quantidade de recursos, sem que para isso seja necessário alterar sua estrutura ou reprogramar aplicativos. Segundo Tanenbaum e Steen [46], a escalabilidade pode ser medida sob três dimensões. A primeira é o tamanho, pois o sistema pode receber mais recursos ainda que ativo. A segunda dimensão é da escalabilidade geográfica, no qual os recursos utilizados podem ser distribuídos em localidades remotas, desde que alcançáveis através de uma rede de comunicação. O terceiro aspecto é que as aplicações podem ser administrativamente escaláveis, ou seja, mesmo que a aplicação alcance localidades remotas ou até outras organizações independentes, sua administração deve permanecer simples de executar. É muito importante frisar que, caso o objetivo seja a implementação de um sistema escalável, é bem provável que ocorra perda de desempenho conforme o sistema escala em qualquer uma das dimensões.

2.2.5 Concorrência

Tendo em vista que o compartilhamento de recursos é um fator chave em sistemas distribuídos, o que se tem como consequência é que vários clientes podem tentar acessar um recurso simultaneamente. E o acesso, quando realizado em diferentes ordenações, provavelmente resultará em saídas diferentes. Coulouris et al. [10] explica que "qualquer objeto que represente um recurso compartilhado em um sistema distribuído deve ser responsável por garantir que ele opere corretamente em um ambiente concorrente." Em conseguinte, para que uma entidade que ofereça recursos em um cenário distribuído mantenha coerência, suas operações devem ser sincronizadas de tal forma que seus dados permaneçam consistentes.

2.2.6 Transparência

A transparência pode ser definida como a ocultação, para um usuário final ou a um desenvolvedor de aplicativos, da separação de componentes em um sistema distribuído de modo que o sistema seja percebido como um todo, em vez de uma coleção de componentes independentes [10]. Em síntese, o uso de múltiplas réplicas de um mesmo recurso deve ser invisível ao usuário, fazendo crer que são apenas um. A transparência, quando percebida na condição de requisito de projeto de um sistema, tem grande influência em todas as etapas de sua construção.

A transparência de um sistema distribuído ocorre sob oito perspectivas, conforme sintetizado na Tabela 2.1. A transparência de acesso e de localização constituem as facetas que afetam mais fortemente o uso de recursos distribuídos. Isso se deve particularmente

Tabela 2.1: Diferentes Formas de Transparência em Sistemas Distribuídos [46].

Transparência	**Descrição**
Acesso	Ocultar diferenças na representação dos dados e como um recurso é acessado.
Concorrência	Ocultar que um recurso pode ter vários usuários acessando-o concorrentemente.
Falha	Ocultar falhas e recuperação de recursos.
Localização	Ocultar onde um recurso está localizado.
Migração	Ocultar que um recurso pode ser movido para outro local.
Replicação	Ocultar o fato de um recurso possuir réplicas.

pela proximidade que estas características têm em relação às definições centrais de sistemas distribuídos. Garantir transparência de acesso significa fornecer um único conjunto de operações para atuar em recursos locais e remotos.

A transparência de localização é um dos elementos centrais na computação distribuída. Para permitir esta característica é necessário que recursos permaneçam acessíveis sem que para isso seja necessário identificar sua localidade física. E em caso de realocação ou migração, qualquer que seja seu destino físico, o mecanismo de acesso deve permanecer o mesmo. Também como objetivo de sistemas distribuídos, a transparência por concorrência visa permitir que vários processos acessem recursos compartilhados, sem apresentar interferência entre eles.

A transparência de replicação é um dos pré-requisitos de sistemas modernos para garantir confiabilidade, desempenho e capacidade de recuperação. O objetivo neste caso é oferecer mecanismos automáticos de cópia de instâncias de recursos, assegurando assim que, em caso de falhas de comunicação, armazenamento ou outras falhas, o recurso ainda esteja disponível através da mesma operação de acesso inicial.

Outra perspectiva de transparência trata das falhas, que visa a conclusão de tarefas mesmo que encontrando falhas de hardware ou software durante sua execução. Para isso muitas vezes é necessário contar com mecanismos como reenvio de mensagens, de modo similar ao funcionamento de correio eletrônico – se algum dos componentes do sistema de envio de e-mail falhar, o servidor de origem agendará uma nova tentativa, até conseguir entregar a mensagem.

2.2.7 Tolerância a Falhas

Para um sistema distribuído, falhas podem ocorrer em um subconjunto de seus componentes, afetando apenas parte de suas funcionalidades. Por isso, tratar falhas distribuídas se torna um desafio significativo. O objetivo de um sistema distribuído é permitir que seus componentes e recursos funcionem normalmente, e na presença de falhas de um ou

mais elementos do sistema, seja capaz de recuperar-se para um estado de funcionamento de acordo com suas especificações [22]. Tendo em vista que o foco desta pesquisa é tratar a tolerância a falhas, este assunto será apresentado em detalhes no Capítulo 3.

Em complemento aos objetivos de sistemas distribuídos, ressaltam-se as vantagens de sua implementação. A seguir são apresentadas tais características e sua relevância para projetos de sistemas.

2.3 Vantagens

Os objetivos apresentados sobre a implementação de sistemas usando arquitetura distribuída correlacionam algumas vantagens, quando comparado à computação centralizada. Em especial, pode-se citar a escalabilidade, o compartilhamento de dados e dispositivos, e a flexibilidade com que se pode adaptar e estender as funções do sistema. Contudo, há outras vantagens expressivas que auxiliam na justificativa de se desenvolver sistemas distribuídos. Entre elas, destacam-se [46]:

- **Economia**: O compartilhamento de recursos, por si só, contribui no sentido da economia. Equipamentos caros, como supercomputadores e sistemas de armazenamento corporativo, podem ser compartilhados entre diversos usuários e aplicações.
- **Velocidade**: Conforme o sistema escala em tamanho e novos recursos são adicionados, o paralelismo confere mais vazão ao acesso de clientes, através do balanceamento de carga entre as diversas réplicas de recursos. De modo similiar, os recursos podem estar fisicamente próximos aos usuários, o que em termos de transmissão de dados torna-se mais rápido do que o acesso a dados remotos.
- **Confiabilidade**: A redundância de recursos torna o sistema mais confiável, pois a indisponibilidade de uma réplica não impede o sistema de continuar funcionando [10].

2.4 Limitações

Apesar das importantes vantagens obtidas ao implementar sistemas distribuídos, deve-se ter em mente as limitações a que a arquitetura distribuída está sujeita. A seguir são apresentadas as limitações inerentes a esse tipo de implementação, e de maneira geral, como lidar com elas.

2.4.1 Complexidade de Software

O desenvolvimento de sistemas distribuídos é mais complexo do que o desenvolvimento de sistemas centralizados. O hardware não pode ser um fator limitante, o sistema operacional não pode ser visto como somente um, e os canais de transmissão podem flutuar em questão do tempo de entrega de mensagens, ou mesmo pararem de funcionar. Todas essas características são relevantes ao se constituir um projeto de sistema distribuído [10].

2.4.2 Limitações de Rede

Ao escalar um sistema distribuído, presume-se que novas réplicas de recursos serão remotamente alocadas. Assim sendo, espera-se que a rede de transmissão ofereça suporte para tráfego de dados entre os recursos e os usuários. Para viabilizar a comunicação, deve-se respeitar as seguintes características:

- A rede não é confiável, pois pode falhar a qualquer momento, e em qualquer aspecto - hardware, canais físicos de transmissão, sobrecarga por alta demanda, e outros fatores;
- A rede não provê segurança inerentemente, o que torna a segurança uma responsabilidade da aplicação;
- Os canais de transmissão são heterogêneos, alternando desde redes de alto desempenho, até canais de longa distância congestionados e com alta latência;
- A topologia sofre alterações frequentes, tanto na rede local em que as aplicações estão hospedadas, quanto em canais de transmissão de longa distância;
- A latência de transmissão pode ser maior do que o esperado, e ainda variar durante a utilização do sistema;
- Conforme a distância física entre recursos aumenta, a largura de banda disponível tende a cair drasticamente;
- O custo de transmissão é diretamente proporcional à distância física entre recursos; e
- A distância física entre recursos torna necessária a administração por mais de uma pessoa, aumentando a complexidade da tarefa.

2.4.3 Segurança de Dados

Ao escalar um sistema geograficamente, a tramitação de dados entre componentes do sistema distribuído ocorrerá em segmentos de rede sob supervisão de várias equipes de

Tabela 2.2: Tipos de Sistemas Distribuídos e Subdivisões [46].

Sistemas de computação distribuídos	Sistemas de computação de *cluster*
	Sistemas de computação em grade.
Sistemas de informação distribuídos	Sistemas de processamento de transações.
	Integração de aplicações empresariais.
Sistemas distribuídos pervasivos	Sistemas domésticos.
	Sistemas eletrônicos de saúde.
	Redes de sensores.

administradores, ou seja, em outros domínios de administração. Dessa forma, não há garantia do nível de segurança atribuído para esse canal de transmissão, o que reverte a responsabilidade para ser tratada diretamente pelo sistema distribuído [46]. De forma equivalente, o trâmite de informações em ambiente corporativo também deve prever algum nível de segurança.

De posse das principais características de sistemas distribuídos, é possível visualizar como distinguem-se as diversas formas de uso da arquitetura. A seção a seguir apresenta a aplicabilidade e os cenários em que é possível utilizar cada variação da arquitetura distribuída.

2.5 Tipos de Sistemas Distribuídos

Para prover um melhor entendimento sobre quais situações o uso de sistemas distribuídos é mais adequado, convém delimitar os tipos existentes e suas principais características. Segundo Tanenbaum e Steen [46], os tipos de sistemas distribuídos podem ser classificados de acordo com sua principal finalidade, ou seja, em sistemas de computação distribuídos, sistemas de informações distribuídas e sistemas distribuídos pervasivos. As subdivisões em que estão segmentados os tipos de sistemas distribuídos estão apresentados na Tabela 2.2.

2.5.1 Sistemas de Computação Distribuída

Os sistemas de computação distribuída tem como principal finalidade oferecer mais capacidade e vazão de processamento. Para isso, o princípio básico é dividir a carga de trabalho entre vários computadores, que compartilham informações através de uma rede de comunicação. Contudo, embora pareça um objetivo direto e simples, segundo Tanenbaum e Steen [46] há pelo menos dois tipos de sistemas distribuídos que configuram este perfil, os sistemas em *cluster* e em grade.

Sistemas distribuídos em *cluster* tem como objetivo permitir que uma aplicação seja executada com o maior desempenho possível para o hardware disponível. Historicamente, a implementação de *clusters* usando computadores relativamente baratos tornou-se financeira e tecnicamente atrativa, frente ao uso de supercomputadores [46]. A computação em *cluster* é usada, essencialmente, para programação paralela em que um único aplicativo, que demanda muito processamento, é executado em múltiplas máquinas.

Uma característica marcante para sistemas de computação em *cluster* é a homogeneidade de seus componentes, já que todos os computadores do conjunto devem executar o mesmo sistema operacional, muitas vezes sobre plataformas de hardware também iguais, e em uma rede de alto desempenho. As funções administrativas e de coordenação do *cluster* podem ficar segregadas a um elemento destacado do conjunto, na função de "mestre", ou dividida entre os computadores do conjunto.

Apesar do objetivo de sistemas de computação em grade (em inglês, *grid*) estar centrado no processamento, a diferença mais expressiva em relação à computação em *cluster* é a heterogeneidade de sua arquitetura. Não há restrição ao hardware utilizado, nem ao quão compatíveis devem ser os sistemas operacionais, redes de interconexão, domínios administrativos, políticas de segurança e outros fatores [46].

A intenção de um sistema distribuído em grade é permitir a colaboração entre organizações e grupos de pessoas ou instituições, e para isso combina recursos computacionais de todos os envolvidos [46]. Os recursos aqui citados podem ser de natureza computacional, como capacidade de processamento, armazenamento e memória, mas também o compartilhamento de equipamentos de telemetria, sensores etc.

2.5.2 Sistemas de Informações Distribuídas

Os sistemas de informações distribuídas caracterizam uma classe definida por sistemas distribuídos encontrados em corporações e instituições, que necessitaram buscar soluções para integração em aplicações de rede. Muitas organizações iniciaram a pesquisa em sistemas distribuídos ao perceber que possuíam grandes volumes de dados e aplicações em rede, mas sem interoperabilidade.

Os sistemas de processamento de transações advém das plataformas baseadas em bancos de dados, nas quais uma aplicação em rede consiste de um servidor que executava a aplicação - e eventualmente hospeda também a base de dados - e que atende a requisições de aplicações clientes. As requisições, neste cenário, comportam-se de maneira ordenada e sistêmica, de forma a garantir que sua execução chegará em um estado estável e consistente de dados. As transações, muitas vezes, operam dados em mais de um banco de dados, e sua execução depende de regras estabelecidas pelo próprio sistema distribuído [46].

Ao passo que as aplicações foram se tornando mais sofisticadas, elas foram separadas em componentes independentes, o que, aos poucos, viabilizou sua integração direta. O resultado dessa abordagem culminou em modelos de integração entre aplicações empresariais definidos como EAI (*Enterprise Application Integration* ou Integração de Aplicações Empresariais) [46].

A EAI surgiu da necessidade de integrar aplicações destinadas a funções específicas, como controle de ambiente de produção, administração de recursos humanos e contabilidade. Conforme explica Martins [38], "as soluções de integração baseadas em EAI provem ferramentas, dispositivos e componentes para facilitar o processo de integração, por meio de adaptadores a sistemas legados".

2.5.3 Sistemas Pervasivos

Como descreve Martins [38], sistemas distribuídos pervasivos, também conhecidos como sistemas embutidos distribuídos, são sistemas que possuem como características a estabilidade em aplicações compactas. Integram equipamentos eletrônicos de consumo, típicos como equipamentos de áudio e vídeo, *smart phones* e outros equipamentos de uso pessoal. Por conta de sua concepção móvel, são inerentemente distribuídos e participam ativamente da vida cotidiana, integrados em um único sistema.

Martins [38] também indica que um dos requisitos para aplicações pervasivas é que devem prover mudanças de contexto durante sua execução, devido à natureza móvel do ambiente, possibilitando manter conectividade, embora em deslocamento. Também é importante incentivar a composição de rede não estruturada (*ad hoc*), para permitir maior alcance de todos as entidades conectadas. Por fim, indica que sistemas pervasivos devem prever o compartilhamento de recursos como padrão para as conexões, para habilitar o funcionamento de entidades pervasivas, de natureza automatizada.

Conforme expõem Tanenbaum e Steen [46], "na presença de mobilidade, tais dispositivos devem suportar a adaptação fácil e dependente de aplicação a seu ambiente local. Também devem ser capazes de descobrir serviços com eficiência e reagir de acordo".

Exemplos de sistemas distribuídos pervasivos são encontrados em sistemas domésticos, onde atualmente a conexão, é obtida por meio de padrões UPnP (*Universal Plug and Play*) [38]. No ambiente hospitalar, em sistemas e equipamentos voltados para o tratamento de saúde, pode-se encontrar sensores e dispositivos conectados ao corpo, em redes denominadas BAN (*Body-Area Network*) [38]. Tais dispositivos devem suportar o processamento de dados na rede, onde os dados de monitoramento devem ser agregados, armazenados e enviados a centros médicos especializados [46].

Outro exemplo de sistemas distribuídos pervasivos são as redes de sensores, que são utilizadas na aquisição de dados e processamento de informações. Tanenbaum e Steen [46]

definem-as como “as redes em malha, que em essência, formam um conjunto de nós que se comunicam por meio de ligações sem fio. Essas redes podem formar a base para muitos sistemas distribuídos de médio porte”. Em complemento, Martins [38] indica que tais dispositivos e sistemas são utilizados em redes sem fio para comunicação e troca de dados e informações em ambientes corporativos; no contexto industrial, dispositivos com estas características são utilizados para interligar equipamentos de automação em ambientes hostis e de difícil acesso.

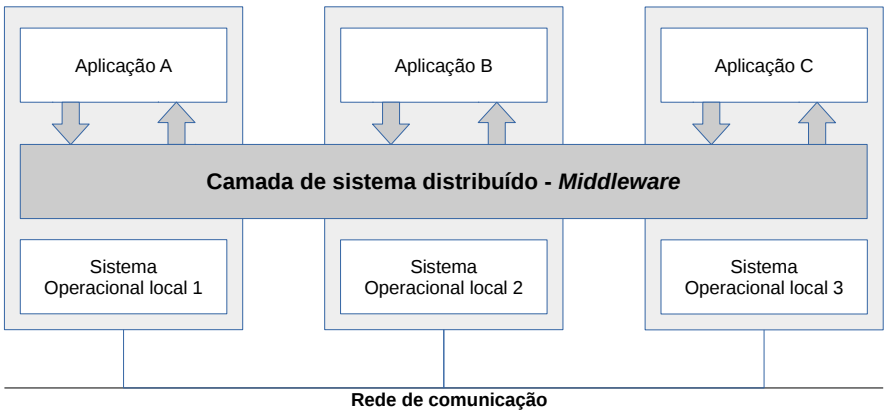


Figura 2.3: Sistema Distribuído Organizado como *Middleware* [46].

Para todos os tipos de sistemas distribuídos apresentados, a heterogeneidade de infraestrutura de hardware, sistemas operacionais e meios de comunicação, torna particularmente difícil a integração entre aplicações. Por este motivo, muitas implementações utilizam uma camada de software intermediária para realizar a troca de mensagens, de modo que as aplicações em um nível superior possam comunicar-se uniforme e eficientemente. Coulouris et al. [10] definem este tipo de camada intermediária de software como *middleware*, o qual “fornece uma abstração de programação, assim como o mascaramento da heterogeneidade das redes, do hardware, de sistemas operacionais e de linguagens de programação subjacentes”. Além de resolver o problema causado pela heterogeneidade, o *middleware* fornece um modelo computacional uniforme onde se pode basear o desenvolvimento de aplicações e de serviços distribuídos. Um modelo conceitual da aplicação de *middleware* pode ser visualizado na Figura 2.3. As principais características e aplicabilidade a respeito de *middleware* são tratadas no Capítulo 4.

Na próxima seção são apresentadas as arquiteturas fundamentais de sistemas distribuídos, e as características que as definem.

2.6 Arquitetura de Sistemas Distribuídos

Os investimentos em pesquisa sobre computação distribuída permitiram a comunidade científica encontrar soluções para diversas questões de compartilhamento de informação. A necessidade de compartilhar recursos e dispor de sistemas integrados e tolerantes às falhas tornou os sistemas distribuídos difundidos em diversas arquiteturas, tendo como propósito central tornar os sistemas mais gerenciáveis, adaptáveis, confiáveis e rentáveis [10].

A arquitetura de um sistema distribuído pode ser definida em função de sua estrutura, em termos de componentes especificados separadamente. O modelo arquitetônico simplifica e abstrai as funções dos componentes individuais e, em seguida, considera [10]:

- O posicionamento dos componentes em uma rede de computadores, para assim definir padrões para a distribuição de dados e da carga de trabalho; e
- Os papéis funcionais e o padrão de comunicação entre os componentes.

Coulouris et al. [10], definem o termo arquitetura de software como "algo que se refere à estruturação do software em camadas ou módulos em um único computador e, mais recentemente, em termos de serviços oferecidos e solicitados entre processos localizados em um mesmo computador ou em computadores diferentes". Assim sendo, a definição de arquitetura de software acompanha a definição de uma arquitetura de sistema.

2.6.1 Estilos Arquitetônicos

A definição de arquitetura de sistemas distribuídos atende ao que se pode chamar "estilos", especificados em função dos componentes envolvidos na arquitetura de sistemas distribuídos e pela forma como eles estão interconectados. Tanenbaum e Steen [46] classificam os estilos de arquiteturas de sistemas distribuídos em:

- **Arquitetura em Camadas**: Os componentes são organizados em uma estrutura na qual elementos da camada inferior oferecem serviços à camada superior. Dessa forma, o fluxo de requisições ocorre de cima para baixo, enquanto que as respostas fluem de baixo para cima, camada por camada. A Figura 2.4 ilustra os principais componentes de sistemas distribuídos em uma arquitetura de camadas.

  A plataforma define os níveis mais baixos da arquitetura, oferecendo serviços aos níveis superiores. O modelo em camadas facilita a comunicação e a coordenação entre processos. Os sistemas operacionais distribuídos fazem parte da plataforma, provendo o gerenciamento de recursos localizados em vários computadores em rede,

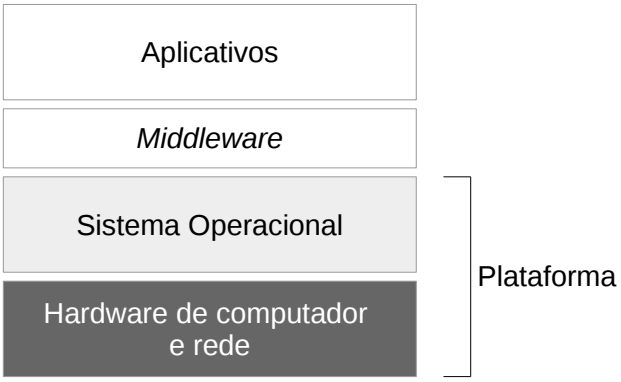


Figura 2.4: Camadas de Software e Hardware em Sistemas Distribuídos [10].

utilizando vários dos mesmos métodos de comunicação, estruturas e outros protocolos encontrados em sistemas operacionais de rede, o que torna a comunicação transparente [10].

- **Arquiteturas Baseadas em Objetos**: Objetos, representando entidades do sistema distribuído, conectam-se de forma livre, porém ordenada, requisitando e oferecendo recursos e serviços umas às outras, através de um mecanismo padrão de chamada remota [46].
- **Arquiteturas Centradas em Dados**: Operam sob a perspectiva que processos comunicam-se através de um repositório de dados, seja ele um repositório ativo ou não [46].
- **Baseado em Eventos**: Processos essencialmente se comunicam através da propagação de eventos, os quais opcionalmente carregam dados [46].

2.6.2 Classificação de Arquiteturas de Sistemas

Martins [38] explica que a distribuição de componentes de software determina a arquitetura de sistemas distribuídos em uma das três classificações: centralizadas, descentralizadas ou híbridas. A seguir são explanadas as três modelagens e suas principais características.

2.6.3 Arquiteturas Centralizadas ou Cliente-Servidor

A arquitetura centralizada define um sistema distribuído como fortemente acoplado e do tipo monolítico, ou seja, os serviços do sistema e do núcleo fazem parte de um mesmo programa. Conforme definem Deitel et al. [13], "o sistema operacional monolítico é a arquitetura de sistema operacional mais antiga e mais comum. Cada componente do sistema

operacional é contido no núcleo e pode comunicar-se diretamente com qualquer outro. O núcleo normalmente é executado com acesso irrestrito ao sistema de computador". De modo equivalente, aplicações em arquitetura cliente-servidor operam de forma monolítica, conforme a definição de Tanenbaum e Steen [46]: "nesta arquitetura distribuída o sistema é dividido em dois grupos. Servidor é um processo que implementa um serviço específico. Cliente é um processo que solicita um serviço de um servidor enviando uma requisição e, em seguida, aguardando uma resposta do servidor." Um servidor também pode requisitar serviços de outro servidor, ou seja, assumindo a posição de cliente. Isso pode ser visualizado na Figura 2.5.

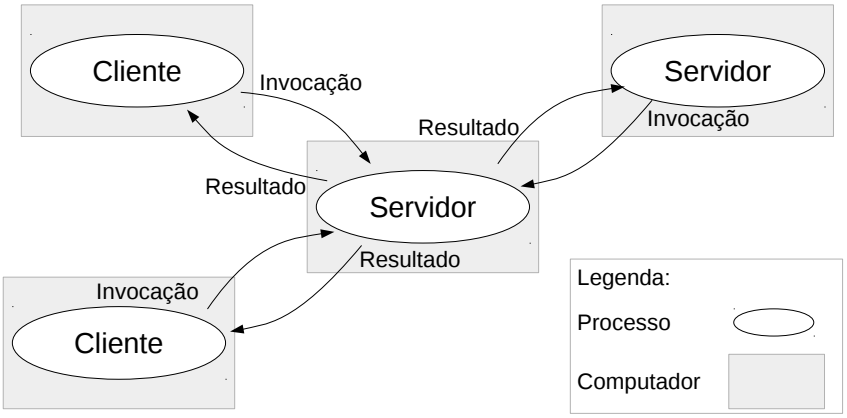


Figura 2.5: Fluxo de Requisições no Modelo Cliente-servidor [10].

Quando se aborda arquiteturas centralizadas, em sistemas distribuídos, a arquitetura cliente/servidor possui ampla utilização, tendo sido bastante implementada ao longo dos anos por diversas aplicações. Conforme expõem Tanenbaum e Steen [46], várias aplicações cliente-servidor podem ser construídas em conformidade com partes distintas da arquitetura: uma parte que manipula a interação com um usuário, outra parte que atua sobre um banco de dados e uma parte que contém as regras de negócio centrais à aplicação. Assim sendo, um sistema em arquitetura cliente-servidor pode ser implementado em camadas. Isso permite o melhor aproveitamento do hardware disponível, dividindo a carga de serviços entre distintos equipamentos, inclusive podendo delegar funções aos computadores clientes, atribuindo-lhes tarefas como manipulação de interface gráfica e filtros de formulários. Também com objetivo de divisão de carga, pode-se agregar vários servidores atendendo o mesmo serviço, seja por meio de *cluster*, seja por meio de servidores *proxy*, que distribuem as requisições entre servidores de forma ordenada e configurável [10].

Uma limitação significativa sobre a arquitetura centralizada, exposta por Coulouris et al. [10], é base para a pesquisa nos demais modelos arquitetônicos: "embora o modelo cliente-servidor ofereça uma estratégia direta e relativamente simples para o comparti-

lhamento de dados e outros recursos, ele não é flexível em termos de escalabilidade. A centralização do fornecimento e de gerenciamento de serviços, acarretada pela disposição de um serviço em um único computador, desfavorece um aumento de escala, além daquela limitada pela capacidade do computador que contém o serviço e da largura de banda de suas conexões de rede".

2.6.4 Arquiteturas Descentralizadas ou *Peer-to-Peer*

O conceito de arquitetura de sistema distribuído descentralizado pode ser considerado uma evolução do modelo de arquitetura distribuída cliente-servidor. O *peer-to-peer* define uma classe de arquitetura de sistemas que suporta distribuição horizontal, ou seja, que atribui a função de servidor a todas as entidades do sistema, denominados "pares". Segundo Coulouris et al. [10], "o objetivo da arquitetura *peer-to-peer* é explorar os recursos (tanto de dados quanto de hardware) de um grande número de computadores para o cumprimento de uma dada tarefa ou atividade. Tem-se construído, com sucesso, aplicativos e sistemas *peer-to-peer* que permitem a dezenas, ou mesmo, centenas de milhares de computadores, fornecerem acessos a dados e a outros recursos que eles armazenam e gerenciam coletivamente". Segundo Nowell [34], "sistemas *Peer-to-Peer*(P2P) são sistemas distribuídos sem controle centralizado ou organização hierárquica, onde o software que é executado em cada *peer* é equivalente em funcionalidade". Androutsellis-Theotokis e Spinellis [3] apontam as características básicas de um sistema P2P como sendo:

- *Peers* se conectam diretamente a outros pontos;
- *Peers* são responsáveis pelos seus próprios dados;
- *Peers* podem entrar e sair da rede a qualquer momento;
- *Peers* podem atuar tanto como clientes quanto como servidores; e
- *Peers* são autônomos com relação ao controle e à estruturação da rede, ou seja, não existe autoridade central.

2.6.5 Arquiteturas Híbridas

Os sistemas distribuídos com arquiteturas centralizadas apresentam certa simplicidade de implementação e gerenciamento. Contudo, a arquitetura centralizada é inerentemente pouco escalável, tendo em vista a expansão limitada do servidor central. Por outro lado, os sistemas descentralizados são escaláveis e robustos, ao custo da complexidade de implementação, principalmente, nas questões de tolerância à falhas e descoberta de recursos. Muitos sistemas distribuídos combinam características das duas arquiteturas: parte do

sistema no tradicional modelo cliente-servidor, e outra parte do modelo ponto-a-ponto, definindo assim arquiteturas híbridas.

Estruturas híbridas são muito úteis e largamente utilizadas em sistemas distribuídos colaborativos. Em uma arquitetura híbrida, os dados trafegam entre os pontos como na arquitetura ponto-a-ponto pura, porém a informação de controle é tratada por um servidor central, o qual realiza o monitoramento para os pontos e garante a coerência das informações [12].

2.7 Considerações Finais

Neste capítulo ficou clara a importância dos sistemas distribuídos no cenário de integração de sistemas de informações. O uso de arquitetura distribuída é o elo de ligação que permite atingir novos patamares de desempenho e confiabilidade, antes indisponíveis apenas em sistemas centralizados. A escalabilidade também é um fator inerente aos sistemas distribuídos, e a arquitetura visa oferecer este aspecto sem comprometer a complexidade de manutenção e administração.

Contudo, o custo da implementação de um sistema distribuído é superior ao custo de desenvolver sistemas centralizados para a mesma finalidade, tendo em vista que os aspectos relevantes da arquitetura devem ser levados em conta durante seu projeto e construção. E apesar das diversas opções arquitetônicas e finalidades de uso de sistemas distribuídos, não há um passo-a-passo que permita escolher qual tipo de sistema distribuído deve ser implementado. Assim, os requisitos de projeto devem ser avaliados, bem como suas peculiaridades e finalidades, para que o projetista tenha uma visão de qual caminho deve seguir. E a combinação de arquiteturas pode ser uma solução mais arrojada neste contexto, também ao custo da complexidade do resultado final.

O próximo capítulo trata das características de tolerância a falhas, retratadas inicialmente no contexto de sistemas distribuídos. É importante averiguar também sob a ótica de arquitetura de sistemas em geral, com foco nos objetivos desta pesquisa.

Capítulo 3

Tolerância a Falhas

A tolerância a falhas faz parte do suporte não funcional de um sistema que agrega segurança de funcionamento, qualidade importante, que faz com que usuários tenham confiança nos serviços do sistema [16]. Acerca do conceito de segurança de funcionamento é que se operam as técnicas e as tecnologias de tolerância a falhas, assunto central deste capítulo.

3.1 Defeitos, Erros e Falhas

Segundo Fernandes [17], falhas, erros e defeitos são mecanismos destrutivos que tentam impedir o correto funcionamento de um sistema, em função de uma sucessão de eventos indesejáveis. Meios de proteção podem ser estabelecidos para evitar que tais mecanismos alterem o comportamento de um sistema, ou ainda, que o sistema possa liberar o serviço especificado mesmo na presença desses mecanismos destrutivos.

Um serviço é dito correto quando executa a função do sistema dentro dos parâmetros especificados [4, 33]. Por sua vez, uma falha de serviço, é um evento que ocorre quando a entrega de serviço desvia do serviço correto. Uma falha de serviço é uma transição do estado de serviço correto para o estado de serviço incorreto, ou seja, deixa de executar o que foi especificado para a função do sistema. O período em que o sistema entrega serviço incorreto é uma interrupção de serviço. A transição de serviço incorreto para serviço correto é uma restauração de serviço.

Segundo Avizienis et al. [4], o desvio do serviço correto pode assumir diferentes formas que são chamados modos de falha de serviços e são classificados de acordo com a severidade das falhas, que serão detalhados mais adiante.

Uma vez que um serviço é uma sequência de estados externos de um sistema, uma falha de serviço significa que ao menos um (ou mais) estado(s) externo(s) do sistema desvia(ram) do estado de serviço correto. O desvio é chamado de erro. A causa de um

erro é chamada de defeito. Defeitos podem ser internos ou externos ao sistema. Na maioria dos casos, primeiro um defeito causa um erro no estado interno do serviço de um componente, e o estado externo não é imediatamente afetado. Contudo, se não houverem mecanismos de contenção desse erro interno, ele pode propagar sua ação em sucessivos erros, até atingir o estado externo do componente, culminando em uma falha, como será explicado na Subseção 3.1.3.

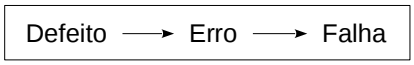


Figura 3.1: Correlação entre Defeitos, Erros e Falhas.

A definição de erro é a parte do estado total de um sistema que deve levar a uma falha de serviço subsequente [4, 33]. Segundo Nelson [40], um erro é uma manifestação de um defeito em um sistema, no qual o estado lógico de um elemento difere do seu valor pretendido. É importante notar que muitos erros não alcançam o estado externo do sistema, mas mesmo assim causam uma falha, por propagar-se no estado interno do sistema e assim influenciar outros processos. Um defeito é dito ativo quando causa um erro, caso contrário está inativo. A correlação entre defeitos, erros e falhas pode ser visualizada na Figura 3.1.

Avizienis et al. [4] apresentam, através da Figura 3.2, a maneira como um defeito propaga-se em um sistema distribuído, particularmente, clarificando os erros internos e a interação entre módulos em condição de falha. A partir da ativação de um defeito ou de uma falha externa, origina-se um erro. O erro propaga-se dentro do componente e, ao alcançar a interface de serviço, propaga-se para o componente cliente através de uma falha. A falha, por sua vez, propaga-se no componente cliente como um erro de entrada, e prossegue até a interface de serviço do componente em questão. Duante a propagação dos erros, quando a interface de serviço apresenta uma falha, ou seja, a interface visível do componente entrega um valor diferente do especificado, somente então caracteriza-se uma falha. O estado externo do componente pode ser assim considerado incorreto, visto que o erro interno propagou-se até atingir o estado externo em questão. As implicações da propagação de erros serão discutidas na Subseção 3.1.2.

3.1.1 Defeitos

O conceito de defeito sinaliza como a causa de um ou mais erros, e pode atuar no estado interno ou externo do sistema [22]. Segundo Gorender [26], um defeito ocorre quando algum componente do sistema não funciona corretamente. Avizienis et al. [4] esclarecem que há oito perspectivas básicas sob as quais um sistema pode apresentar defeitos,

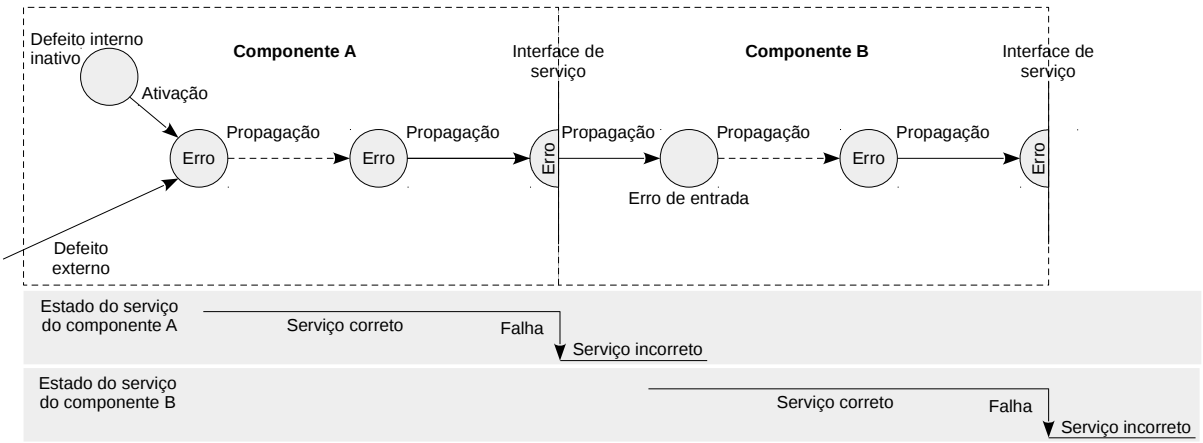


Figura 3.2: Propagação de Erros em Sistemas [4].

agrupando-as em defeitos decorrentes do desenvolvimento ou codificação; defeitos físicos, relacionados ao hardware; e defeitos de interação, incluindo todos os defeitos externos. Cada perspectiva básica desdobra-se em classes de defeitos elementares. A correlação dos defeitos pode ser visualizada na Tabela 3.1.

Defeitos decorrentes do estágio de criação, ou desenvolvimento, advém tanto da fase inicial de projeto e codificação, como de atualizações e de novos processos criados durante a operação do sistema. Também enquadram-se aqui os defeitos de operação na fase de uso do sistema, já por ocasião da entrega do serviço.

Em relação à perspectiva dos limites de atuação do sistema, os defeitos podem ser classificados como internos ou externos. Neste caso, os defeitos internos relacionam-se às ocorrências dentro dos limites de atuação do sistema. Em contrapartida, os defeitos externos ocorrem fora dos limites do sistema, e influenciam seu funcionamento por meio de interferência ou interação.

Defeitos causados sob a perspectiva de fenômenos naturais podem ter origem essencialmente em forças da natureza ou ação humana. Assim sendo, destacam-se para o primeiro caso as situações não previstas e em que não há interação humana, como falhas elétricas e incêndios acidentais, e ainda a deterioração de equipamentos pelo uso. A interação humana, nesta classificação, refere-se à participação humana, seja ela por omissão, como exemplo deixar de executar rotinas de manutenção preventiva, e por ação ou ordem, como desligar equipamentos acidentalmente.

A perspectiva de dimensão identifica duas linhas elementares. Por um lado, os defeitos de hardware, quando iniciam ou afetam diretamente os equipamentos físicos, como falha de memória, portas de comunicação ou setores de disco danificados. Por outro lado os defeitos de software, quando iniciam ou afetam diretamente programas ou dados, como

Tabela 3.1: As Classes Elementares de Defeitos [4].

Perspectiva	**Classe de defeito**
Estágio de criação	Desenvolvimento
	Operacional
Limites do sistema	Interno
	Externo
Fenômenos naturais	Natural
	Origem humana
Dimensão	Hardware
	Software
Objetivo	Malicioso
	Não malicioso
Propósito	Intencional
	Não intencional
Aptidão	Acidental
	Não acidental
Persistência	Persistente
	Transiente

falhas de sistema operacional e *drivers* de dispositivos [6].

A perspectiva de objetivo denota defeitos de origem maliciosa, em que há intenção de prejudicar o funcionamento do sistema, e os não maliciosos, em que decocorre por imperícia ou imprudência do usuário. Em ambos os tipos de defeito, é necessária a interação humana diretamente no sistema, diferentemente da perspectiva de fenômenos da natureza, que resume-se a ações ou fatos externos que interferem no funcionamento do sistema.

A perspectiva do propósito complementa a de objetivo, adicionando a questão da decisão. Se o usuário decidir por uma ação que ameace a estabilidade e o funcionamento do sistema, enquadra-se nesta perspectiva, sendo seu objetivo prejudicar ou não. Também enquadra-se aqui a situação de uma decisão que, embora nociva, que o usuário não perceba sua gravidade, ou seja, não tem intenção de danificar o sistema.

A perspectiva da aptidão remete a defeitos na maioria das situações ocasionados por interação humana. Sob esta ótica, os defeitos podem ser acidentais, como a escolha do comando errado na ocasião errada, ou ainda por imprudência ou imperícia, em que o usuário desconhece o funcionamento ou restrições de uso do sistema, mas mesmo assim decide operá-lo.

Por fim, a perspectiva de persistência, na qual defeitos podem ser permanentes ou

passageiros. Os defeitos passageiros ocorrem com início e fim delimitados, mas cabe observar que podem se repetir mais de uma vez, mesmo assim, sem se caracterizarem permanentes [4].

Para compor o entendimento acerca de defeitos, Fernandes [17] contribui indicando dois estados que um defeito pode assumir. Um defeito é dito inativo quando não produz um erro, e está ativo quando produz um erro como consequência de sua ação, por meio de desvio do estado desejado, pela violação do valor atribuído ao estado ou da transição entre estados.

3.1.2 Erros

Um erro foi definido anteriormente como parte do estado total do sistema que pode levar a uma falha. Segundo Gorender [26], um defeito pode provocar a ocorrência de um erro na execução do sistema, com a geração de resultados incorretos. Para contextualizar a causalidade dos fatos, uma falha ocorre quando um erro faz o serviço entregue desviar do serviço correto especificado [22, 33], e a causa do erro é chamada de defeito. Segundo Nelson [40], um erro somente ocorre quando um defeito é "sensibilizado", ou seja, ligado por uma entrada específica, em um estado particular do sistema.

Um erro é dito detectado se sua presença é indicada por uma mensagem ou sinal de erro. Erros que estão presentes, mas não detectados, são ditos erros latentes.

Segundo Avizienis et al. [4], à medida que um sistema consiste em um conjunto de componentes em interação, o estado total é um conjunto com o estado dos componentes. A definição implica que um defeito ativo, originalmente, causa um erro dentro do estado de um ou mais componentes, mas a falha de serviço não ocorrerá, desde que o estado externo de nenhum dos componentes com erro componha o estado externo do sistema. Sempre que o erro se torna parte do estado externo do sistema, isso implica em uma falha de serviço daquele sistema.

Ainda segundo Avizienis et al. [4], um erro levará a uma falha de serviço ou não, dependendo de dois fatores:

1. A estrutura do sistema e, especialmente, a natureza de qualquer redundância que exista nele; e
2. O comportamento do sistema, onde a parte do estado que contém o erro esteja fora do processo destinado ao provimento do serviço, ou o erro pode ser eliminado (por exemplo, sobrescrito) antes que leve a uma falha de serviço.

Segundo Fernandes [17], os erros podem ser distinguidos quanto à sua persistência. Erros *persistentes* ocorrem sem interrupção, enquanto que erros *transientes* param de

ocorrer, mas em condições não necessariamente controladas. Os erros também podem ser classificados quanto ao número de incidências, podendo ser erros *simples* ou isolados, e também pela incidência de *múltiplos* erros.

3.1.3 Falhas

A execução incorreta do sistema caracteriza a existência de uma falha [26]. Segundo Nelson [40], falha denota a inabilidade de um elemento de executar sua função projetada por causa de erros em seu ambiente, o que por sua vez é causado por vários defeitos.

Avizienis et al. [4] definem que a falha de serviço ocorre quando a entrega de serviço desvia do serviço correto. As diferentes formas nas quais o desvio é manifestado são chamados modos de falha de serviço. Cada modo pode ter vários níveis de severidade de falha de serviço, o que indica o quão prejudicial é a falha para o funcionamento do sistema. Os modos de falha de serviço caracterizam o serviço incorreto de acordo com as seguintes perspectivas:

- **Domínio**: percebe-se como falha de conteúdo, quando a informação entregue na interface de serviço desvia do esperado pela função do sistema, ou falha de temporização, quando o tempo de chegada, duração ou entrega da informação na interface de serviço ocorre fora das especificações da função do sistema;
- **Detectabilidade das falhas**: endereça a sinalização de falhas de serviço para o usuário, em que perdas detectadas e apontadas por um sinal de aviso apropriado indicam falhas sinalizadas. O mecanismo de detecção também falha, e isso ocorre em dois modos: 1) sinalizar a perda da função quando nenhuma falha realmente ocorreu, o que representa um alarme falso, e 2) não sinalizar uma perda de função real, o que é uma falha não sinalizada;
- **Consistência**: falhas consistentes apresentam o serviço incorreto identicamente para todos os usuários, o que por outro lado em falhas inconsistentes a percepção dos usuários é deferente, podendo inclusive apresentar o serviço correto para alguns usuários do mesmo sistema com falha; e
- **Consequências no ambiente**: as falhas podem apresentar resultados graduados em níveis de severidade, aos quais geralmente são associados níveis máximos de aceitação de ocorrência. O número, a nomenclatura, e a definição dos níveis de severidade, bem como da probabilidade aceitável de ocorrência, são relacionados à aplicação, o que significa que podem variar, dependendo principalmente do projeto do sistema e sua finalidade. De modo geral, dois níveis limitadores podem ser definidos de acordo com a relação entre o benefício geral, provido pelo serviço entregue

em ausência de falhas, e as consequências das falhas: 1) Falhas menores, quando as consequências prejudiciais são de custo similar aos benefícios providos pela entrega do serviço correto; e 2) falhas catastróficas, quando o custo das consequências prejudiciais é muito maior que o benefício da entrega do serviço correto.

Favarim [16] e Jeukens [31] complementam que falhas podem ocorrer, ainda sob domínios de tempo e valor, com as seguintes características:

- **Por parada (*crash*)**: falhas decorrentes de travamento ou perda do estado interno do componente com defeito, o que impede o sistema de mudar de estado de execução;
- **Por omissão**: o serviço pára de responder requisições, podendo ser de forma constante - definitivamente não atendendo nenhuma requisição - ou intermitente, situação em que o sistema responde eventualmente a requisições; e
- **Arbitrária**: o sistema passa a exibir um comportamento arbitrário e, em alguns casos, caracterizado como malicioso, pois responde a todas as requisições que lhe são feitas, dando sempre a impressão de estar funcionando corretamente. Contudo com respostas diferentes das especificações de funcionamento do sistema. Este tipo de classificação também é denominada falha bizantina [16, 26, 31].

A Tabela 3.2 sumariza os modos de falha de serviço, conforme a classificação proposta por Avizienis et al. [4], Favarim [16] e Jeukens [31].

Tabela 3.2: Modos de Falha de Serviços [4].

Modo	**Classificação**
Domínio	Falhas de conteúdo Falhas de temporiação adiantada Falhas de temporiação atrasada Falhas de travamento Falhas por omissão Falhas arbitrárias
Detectabilidade	Falhas sinalizadas Falhas não sinalizadas
Consistência	Falhas consistentes Falhas inconsistentes
Consequências	Falhas menores (*Outras classificações intermediárias*) Falhas catastróficas

3.2 O Tratamento de Defeitos em Sistemas Computacionais

Por melhor que os sistemas computacionais sejam desenvolvidos, sempre haverá possibilidade para a ocorrência de defeitos [4, 22, 33, 40]. Para permitir o funcionamento adequado de sistemas sob a perspectiva da ocorrência de defeitos, é necessário implementar meios de contorno e correção de defeitos. Avizienis et al. [4] indicam que há quatro categorias principais de tratamento de defeitos:

- **Prevenção de defeitos**: trata de prevenir a ocorrência ou introdução de defeitos;
- **Tolerância a defeitos**: refere-se a evitar falhas de serviço na presença de defeitos;
- **Remoção de defeitos**: presume reduzir a quantidade e a severidade de defeitos; e
- **Previsão de defeitos**: intenta estimar o número atual, a incidência futura, e as consequências gerais de defeitos.

Prevenção e tolerância a defeitos alinham-se em prover a habilidade de entregar um serviço confiável, enquanto que a remoção e a previsão de defeitos tem com alvo alcançar confiança por justificar que o sistema alcançará as especificações funcionais, mesmo após a ativação de defeitos. Esta pesquisa direciona esforços para atender a perspectiva de tolerância a defeitos em sistemas computacionais, que será explicada com mais detalhes na subseção a seguir.

3.2.1 Tolerância a Defeitos

Segundo Favarim [16], tolerância a defeitos é uma propriedade ou qualidade do sistema que tenta garantir que serviços operem de forma ininterrupta, de acordo com a sua especificação, mesmo na presença de defeitos, através do uso de redundância. Apesar desta pesquisa se contextualizar na tolerância a falhas, Gärtner [22] menciona que "usa-se o termo *falha* para denotar o fato de um sistema não se comportar adequadamente de acordo com sua especificação", e os mecanismos de tolerância a defeitos apresentados congregam o objetivo de manter sistemas operantes e protegidos, embora na presença de defeitos.

Dessa forma, faz-se necessário distinguir que a tolerância a defeitos contempla um subconjunto dos mecanismos de tolerância a falhas, pois a ativação de defeitos incorre em erros, que por sua vez propagam-se até ocasionarem falhas em serviços. A tolerância a defeitos não sobrepõe por completo a tolerância a falhas porque não há garantia de que determinado defeito, quando ativado, resulte em falhas limitadas no tempo (uma

falha pode ativar-se no futuro) ou no espaço (falhas podem ocorrer em todas as cópias redundantes, sendo essa uma falha significativa de confiabilidade e proteção do sistema) [22]. Sendo assim, a tolerância a falhas compõe-se de mecanismos mais arrojados do que a tolerância a defeitos, que serão esclarecidos nas próximas seções.

As redundâncias (ou réplicas) segundo Gärtner [22], são fundamentais para a tolerância a defeitos, e podem ser de software, de hardware, ou de número de execuções (isto é, um processo pode ser executado novamente, se necessário). Para permitir a tolerância a defeitos, é necessário atuar em diversos estágios, desde a detecção, o confinamento, o processamento e o tratamento dos erros e defeitos, que serão explicados a seguir [16, 33]:

- **Detecção de erros**: por ser a etapa que inicia o processo de tolerância a defeitos, torna-se a mais importante. Segundo Favarim [16], as técnicas de detecção envolvem testes, e mesmo assim pode não ser imediata ou completa em um sistema. A detecção de erros também pode ocorrer com atraso.
- **Confinamento de erros**: neste estágio, delimita-se a propagação de erros, para que outros componentes do sistema não sejam afetados. Segundo Favarim [16], a propagação do erro é quase certa, pois há um atraso na detecção do erro, e informações incorretas continuam sendo disseminadas até o confinamento do evento de erro. Neste estágio, é necessário identificar a extensão dos danos causados ao sistema, concomitante ao processo de recuperação.
- **Processamento de erros**: de posse do erro delimitado e dos fatores por ele desencadeados, é necessário remover do estado do sistema todas as implicações causadas, para que o sistema possa retornar para um estado correto de execução, e assim continuar a operar. Segundo Favarim [16] e Laprie [33], o processamento de erros pode ser feito por duas técnicas:
  - **Compensação de erros**: corresponde à redundância da informação ou de processamento, para mascarar os efeitos de elementos eventualmente defeituosos. Fernandes [17] e Gärtner [22] citam esta etapa como mascaramento de defeitos (*fault masking*).
  - **Recuperação de erros**: consiste em substituir um estado incorreto do sistema por um estado livre de erros. A recuperação pode ser por retrocesso (*backward error recovery*), para um ponto específico no tempo em que o sistema funcionava corretamente, ou por avanço (*forward error recovery*), em que o sistema remove as informações decorrentes de erros e atribui ao sistema um estado correto de execução. Fernandes [17] e Gärtner [22] citam esta etapa como sendo a manipulação de erros (*error handling*).Uma visão gráfica da recuperação de erros pode ser visualizada na Figura 3.3.

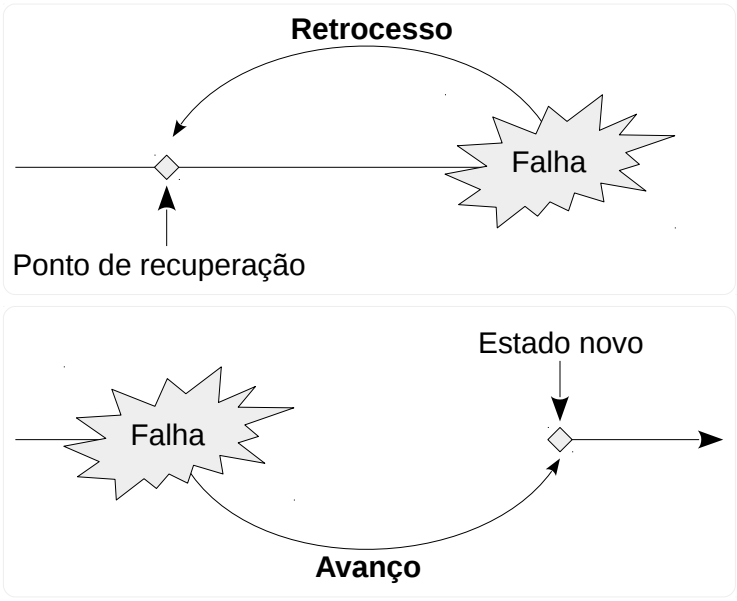


Figura 3.3: Recuperação em Retrocesso e em Avanço [16].

Na recuperação por retrocesso, o estado do sistema é retornado a um estado anterior. Desta forma, a recuperação por retrocesso do erro exige que informações de estado do sistema sejam regularmente salvas em pontos específicos de execução do serviço, denominados de pontos de recuperação (*checkpoint*), em um repositório de memória estável. Quando ocorre a detecção de um erro, o estado do sistema é restaurado com os valores do último ponto de recuperação estabelecido.

Na recuperação por avanço, nenhum estado anterior está disponível. A recuperação do erro consiste da tentativa de transformar o estado falho em um estado futuro, a partir do qual o sistema retorna a sua operação normal.

- **Tratamento de defeitos**: após a recuperação de erros, os defeitos apresentados no início do tratamento voltam ao estado inativo, porém ainda podem ser reativados. É necessário que os defeitos sejam tratados, para não culminarem em novos erros. Segundo Fernandes [17], esta solução envolve quatro passos: 1) diagnóstico do defeito; 2) isolamento do defeito por meio da exclusão física ou lógica do componente defeituoso; 3) reconfiguração do sistema utilizando componentes redundantes, ou redistribuir a tarefa entre componentes ativos; e 4) reinicialização do sistema para operar sob a nova configuração, já restaurada.

Um sistema, que se queira tolerante a defeitos, precisa promover algum tipo de redundância em seus componentes. A incorporação de elementos redundantes a um sistema pode ocorrer de várias formas:

- **Redundância de hardware ou software**: consiste na replicação dos componentes de um sistema, de modo que mais de um componente desempenhe a mesma função [31];
- **Redundância da informação**: consiste em adicionar dados de recuperação à informação, para permitir que funções de validação atuem em caso de falhas de armazenamento ou transmissão [17]; e
- **Redundância temporal**: incrementa-se o tempo de execução de funções, para permitir a detecção e processamento de defeitos.

3.3 Técnicas de Redundância

O uso de técnicas de redundância requer o uso de protocolos de coordenação, no sentido de manter a consistência de estado e a transparência do conjunto. Estes protocolos também são responsáveis pelo controle da concorrência e pela recuperação em situação de falha parcial ou total das réplicas. Cada técnica de redundância exige o uso de um diferente protocolo.

As principais abordagens de replicação envolvem três padrões [16]: Replicação Passiva, Ativa e Semi-Ativa. As abordagens de replicação passiva e semi-ativa isolam elementos faltosos de um serviço replicado do sistema, enquanto que a abordagem de replicação ativa é capaz de mascarar falhas parciais de réplicas do conjunto. Segundo Favarim [16], Para a escolha de cada uma das abordagens é necessário levar em conta o tipo de aplicação, a classe de faltas que se deseja tolerar e as características do sistema distribuído.

3.3.1 Replicação Ativa

Nesta abordagem todas as réplicas operando no estado correto recebem as requisições, processam de forma paralela e produzem as mesmas saídas, conforme ilustrado na Figura 3.4. Desta forma, em caso de ocorrer um defeito em uma das réplicas, o resultado correto pode ser obtido no mesmo instante, uma vez que as réplicas que não apresentam defeitos já realizaram o processamento, não havendo a necessidade de recuperar o estado das réplicas.

Segundo Favarim [16], diferentes estratégias podem ser usadas para enviar o resultado para o cliente. Pode ser enviado o primeiro resultado que chegar; ou todos os resultados podem ser concatenados em seqüência e enviados ao cliente; ou os resultados passam por um votador que seleciona o resultado que a maioria votou.

Esta abordagem é capaz de tolerar todos os tipos de defeitos e, por extensão, os modos de falha por eles gerados (conteúdo, temporização, travamento, omissão ou arbitrária).

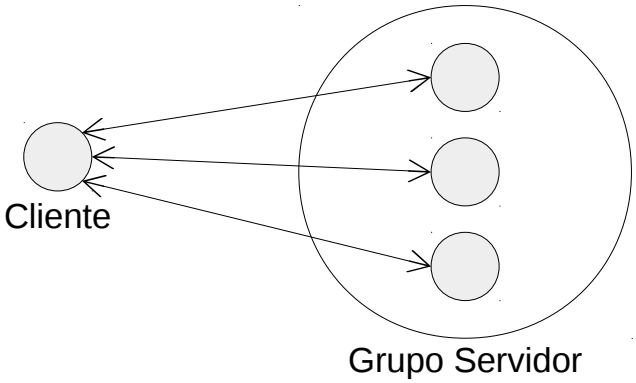


Figura 3.4: Arquitetura de Replicação Ativa [16].

É a abordagem mais apropriada para aplicações que necessitam de serviços ininterruptos com sobrecarga mínima em situações de falha, como as aplicações de tempo real, pois as falhas parciais são mascaradas.

3.3.2 Replicação Passiva

Na replicação passiva apenas uma réplica, tratada como primária, recebe, processa e responde as requisições. As outras réplicas são passivas (*backup*) e têm seus estados atualizados periodicamente a partir da réplica primária, através de mecanismos de sincronia, conforme exibido na Figura 3.5. Caso a réplica primária falhe, uma réplica *backup* deve assumir a sua função. Isso ocorrerá retrocedendo o estado para a última sincronia realizada entre a réplica primária e as réplicas de *backup*.

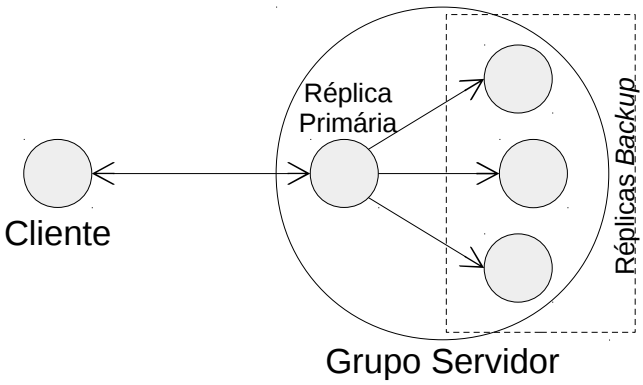


Figura 3.5: Arquitetura de Replicação Passiva [16].

Por um lado, esta abordagem apresenta um custo de processamento e disponibilidade menor do que a abordagem ativa, mas que por sua vez confere recuperação a um conjunto menor de defeitos e falhas, somente em casos de travamento e falhas por omissão.

Por este motivo, não é a replicação mais apropriada para aplicações que necessitam de serviços ininterruptos, pois existe uma sobrecarga relevante em tempo de execução para a recuperação do estado da réplica primária, no caso de ocorrerem defeitos na mesma. Também deve-se levar em conta que o estado recuperado é anterior ao instante atual de execução, e isso implica na perda efetiva de operações ocorridas desde o último *backup*.

3.3.3 Replicação Semi-Ativa

A abordagem de replicação semi-ativa apresenta características de ambos os modelos anteriores, pois as réplicas são todas ativas, com apenas uma sendo considerada privilegiada [16]. Uma representação gráfica dessa abordagem é apresentada na Figura 3.6. A replicação semi-ativa tenta diminuir o impacto das limitações das outras técnicas apresentadas, reduzindo o custo de processamento e transmissão de todas as réplicas ativas, e ao mesmo tempo diminuindo o tempo de recuperação da réplica privilegiada.

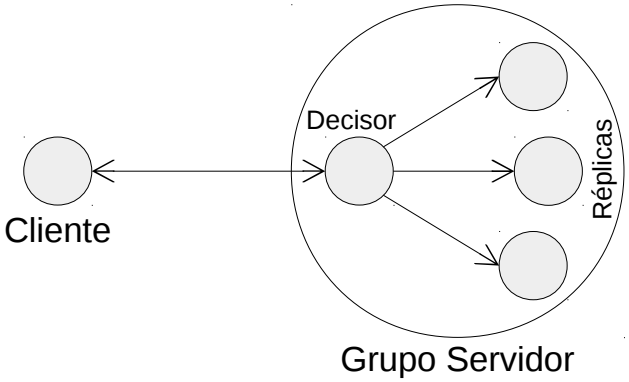


Figura 3.6: Arquitetura de Replicação Semi-Ativa [16].

Nesta abordagem todas as réplicas executam os pedidos de serviço e somente a réplica privilegiada é responsável por impor a ordem de execução dos pedidos como também pela resposta aos clientes. Quando a réplica privilegiada falha, uma das réplicas restantes assume o seu papel. Como nesta abordagem todas as réplicas são ativas, não ocorre a recuperação de estado baseado em retrocesso. Esta abordagem suporta os modos de falha por conteúdo, temporização, travamento e omissão. No entanto, o tratamento de falhas arbitrárias no mecanismo de votação de réplica fica comprometido, sendo a principal limitação deste modelo de replicação.

3.4 Considerações Finais

Neste capítulo foram explanadas as principais características sobre tratamento de defeitos, erros e falhas em sistemas distribuídos. O foco da pesquisa é permitir tolerância a falhas, que por sua vez é baseada na tolerância a defeitos, como apresentado neste capítulo. Os meios para se alcançar a tolerância a defeitos advém fundamentalmente da presença de réplicas de ativos, sejam de hardware, software, execução de instruções e da informação tratada. Para esta pesquisa, é importante tratar da replicação de software, as variações viáveis de arquitetura e os mecanismos de apoio a este fim.

O próximo capítulo apresentará as características de uma variação de arquitetura de sistemas distribuídos, a de *Middleware* Orientado a Mensagens (MOM), voltada para a integração ágil e simplificada entre entidades de software. O objetivo dessa arquitetura é permitir que sistemas distribuídos heterogêneos troquem informações, abstraindo as questões de integração mais comuns, permitindo que os sistemas foquem seu código no negócio a que se destinam.

Capítulo 4

Middleware Orientado a Mensagem

Neste capítulo são apresentadas as principais características da arquitetura MOM. Para isso, é fundamental compreender os modelos de interação entre processos, e o funcionamento da arquitetura RPC, base para a computação distribuída. Também é apresentada a conceituação a respeito de filas de mensagens e sua função no contexto de MOM, bem como os tipos de filas mais utilizadas na arquitetura. Por fim, é importante compreender os principais serviços da arquitetura MOM, e como eles podem ser aplicados no cenário de computação distribuída.

4.1 Conceitos Essenciais

Ao passo que sistemas continuam a ser empregados de modo distribuído e em escala cada vez maior, transcendendo limites geográficos, organizacionais e comerciais, a demanda por infraestrutura de comunicação cresce exponencialmente. Sistemas modernos operam em ambiente complexo com múltiplas linguagens de programação, plataformas de hardware e sistemas operacionais. Se não bastasse a complexidade da infraestrutura, ainda se deve lidar com os requisitos de rápida instalação e flexível, alta taxa de transferência de dados e alto nível de segurança, enquanto mantendo um alto nível de qualidade de serviço (QoS). Se qualquer sistema pretende prover as características citadas, a interface de apresentação ao usuário deve ser a de serviços e não de servidores. O usuário deve ser capaz de solicitar uma ação especificando o que deve ser feito e não precisar citar qual recurso físico ou lógico proverá o serviço [15]. Nestes ambientes, o tradicional mecanismo de RPC (*Remote Procedure Call*, ou Chamada de Procedimento Remoto) falha ao buscar os objetivos apresentados [11].

Com o propósito de cooperar com a implementação das demandas para tais sistemas, uma alternativa para o mecanismo de distribuição RPC emergiu. O mecanismo chamado MOM (*Message-Oriented Middleware* ou *Middleware* Orientado a Mensagens) provê um

método claro de comunicação entre entidades de software díspares. MOM provê um serviço que permite o provedor de conteúdo e consumidores concentrarem-se na produção e consumo de informações transmitidas [1]. A plataforma MOM permite criar sistemas flexíveis e coesos. Como visto no Capítulo 2, o *middleware* é uma plataforma de computação distribuída, que isola desenvolvedores e usuários dos sistemas operacionais e protocolos de rede heterogêneos [2].

Um sistema de processamento de dados distribuídos deve ser projetado para que as operações de todos os componentes ou recursos, tanto físicos quanto lógicos, sejam altamente autônomos [15]. Sistemas multi-camada e distribuídos são virtualmente complexos de implementar sem o uso massivo de infraestrutura de mensagens. A variante de sistema mais comum é o esquema *publish/subscribe* [1], que é um modelo de troca de mensagens poderoso, por permitir que a transmissão de uma mensagem se propague a mais de um destinatário, com as variações de entrega de um-para-muitos ou de muitos-para-muitos, como será descrito mais adiante.

4.2 Modelos de Interação

Para compreender o funcionamento dos mecanismos de computação distribuída, faz-se necessário entender antes os modelos de interação e troca de informações entre processos. Esta seção apresentará dois modelos de interação, síncrono e assíncrono, suas características de uso, vantagens e limitações. Em seguida serão comparadas as técnicas de interação de RPC e MOM, e os cenários mais adequados para cada uma das técnicas.

Quando um procedimento/função/método é chamado usando o modelo de interação síncrono, o código invocador deve bloquear e esperar (suspendendo, assim, o processamento) até que o código invocado complete a execução e devolva o controle; o código invocador somente continuará processando após isso [11]. A Figura 4.1 apresenta o modelo de interação síncrono, onde fica claro como o processo invocador depende da resposta do processo invocado para continuar seu processamento.

O modelo de interação assíncrono, ilustrado na Figura 4.2, mantém o controle do processamento sob a ordem do processo invocador. Este modelo torna o controle de processamento do invocador independente do procedimento/função/método invocado [11]. Com interação assíncrona, o código invocado também não precisa executar imediatamente. Este modelo de interação requer um controle adicional para intermediar requisições, o que aumenta sua complexidade de implementação, quando comparado ao modelo síncrono de troca de mensagens. Normalmente, este controle fica sob responsabilidade de uma fila de mensagens [11].

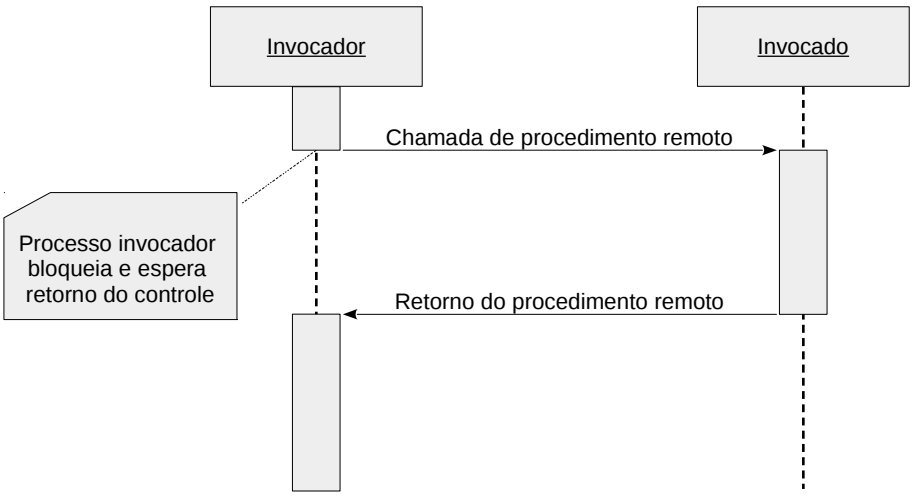


Figura 4.1: Modelo de Interação Síncrono [11].

4.2.1 Chamada de Procedimento Remoto (RPC)

O modelo tradicional de RPC (*Remote Procedure Call*) é um conceito fundamental de computação distribuída. O Objetivo de RPC é permitir a interação entre dois processos. O mecanismo de RPC cria uma interface entre processos para fazer com que ambos acreditem estar no mesmo espaço de execução, ou seja, como se fossem um único processo [11, 46].

Baseado no modelo de interação síncrono, o RPC opera de forma similar a uma chamada de procedimento local, pelo qual o controle é passado para o procedimento invocado de forma sequencial e síncrona, enquanto o procedimento que invoca bloqueia e espera por uma resposta à chamada. Um exemplo de chamada baseada em RPC pode ser vista no modelo retratado na Figura 4.3. Hohpe e Woolf [29] indicam que o RPC aplica-se ao princípio de encapsulamento para integrar aplicações. Se uma aplicação precisa de informações de propriedade de outra, a primeira deve solicitar as informações diretamente. O mesmo se a necessidade for de alterar informações de propriedade de outra aplicação; uma invocação remota deverá ser executada para completar o objetivo.

O RPC é projetado para trabalhar com interfaces de objetos ou funções, resultando em um modelo que produz sistemas fortemente acoplados, já que qualquer modificação nas interfaces resultará em mudanças ao código base de ambos os sitemas. Segundo Curry e Mahmoud [11], isso faz do RPC um mecanismo de distribuição muito invasivo, e ainda indicam que o custo de implementação aumenta significativamente à mesma proporção do volume de mudanças nos sistemas de origem e de destino. Dessa forma, o RPC provê um método inflexível de integração entre múltiplos sistemas [46].

Meios de comunicação confiáveis são muito importantes quando se constrói aplicações

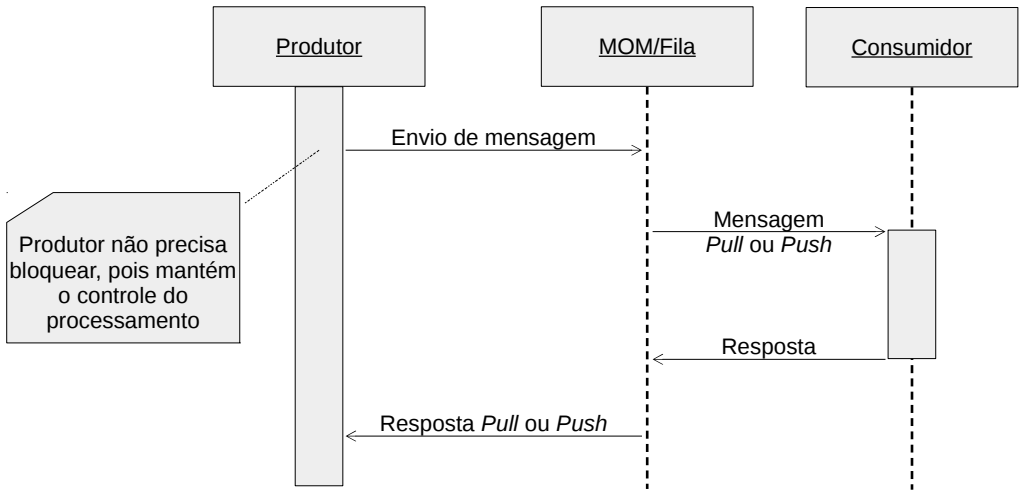


Figura 4.2: Modelo de Interação Assíncrono [11].

distribuídas. Qualquer falha fora da aplicação - código, rede, hardware, serviço, outro software ou serviços indisponíveis de vários tipos (provedor de rede, energia elétrica, etc) - podem afetar a confiabilidade de transporte de dados entre sistemas [11]. Segundo Ibrahim e Hassan [30], muitas implementações de RPC proveem pouca ou nenhuma garantia de capacidade de comunicação confiável, ou seja, elas são muito vulneráveis à indisponibilidade dos serviços.

Em um sistema distribuído construído com RPC, a natureza bloqueante pode afetar negativamente o desempenho em sistemas onde os subsistemas participantes não escalam na mesma proporção [11]. Isto efetivamente diminui o desempenho global de todo o sistema para a velocidade máxima de seu participante mais lento. Em tais condições, técnicas de comunicação síncrona como RPC podem ter problemas de paralisação do sistema quando elementos são sujeitos a rajadas de chamadas concentradas.

Segundo Curry e Mahmoud [11], o modo de interação síncrono de RPC usa mais banda de transmissão porque várias chamadas podem ser feitas através da rede para apoiar uma chamada síncrona de função. Assim, sistemas construídos sobre o modelo RPC são interdependentes, requerindo a simultânea disponibilidade de todos os subsistemas [46]. A falha em um subsistema pode causar a falha do sistema todo. Em um ambiente RPC, a indisponibilidade de um subsistema, mesmo que temporária, seja por indisponibilidade de serviços ou atualização de sistemas, pode causar erros que refletirão em todo o sistema distribuído.

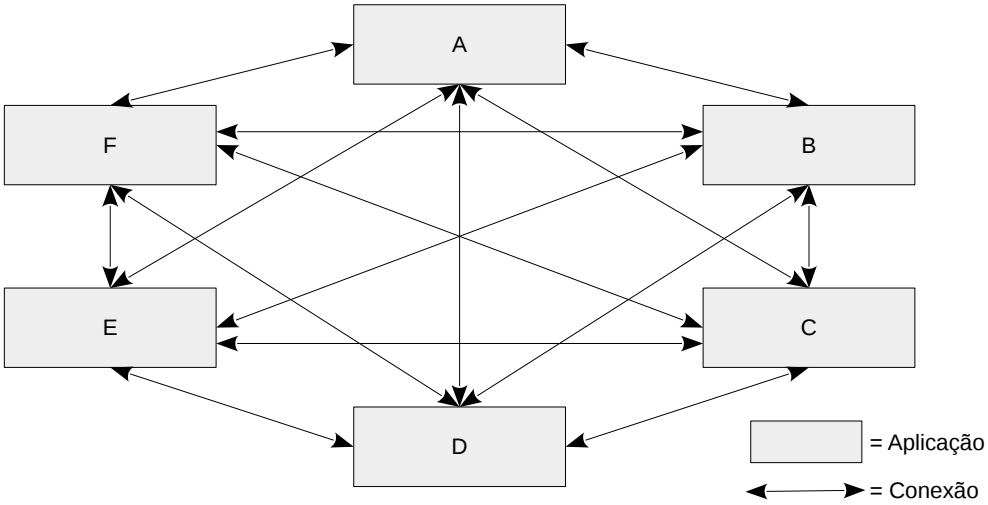


Figura 4.3: Exemplo de Infraestrutura RPC [11].

4.2.2 *Middleware* Orientado a Mensagens - MOM

Sistemas MOM proveem comunicação distribuída baseada no modelo de interação assíncrono. Este modelo não-bloqueante permite ao MOM solucionar muitas das limitações encontradas em RPC [11]. Participantes de um sistema baseado em MOM não são requeridos a bloquear e esperar ao enviar uma mensagem, eles podem continuar processando assim que a mensagem for enviada. Isso permite a entrega de mensagens quando o emissor ou destinatário não estiverem ativos ou disponíveis para responder no momento da execução [46].

A plataforma MOM permite liberdade de tempo para entregar mensagens, ao contrário de mecanismos como RPC que forçam a necessidade de entrega em milisegundos ou segundos [11]. Contudo, é relevante ressaltar que quando se usa MOM, uma aplicação emissora não tem garantia de que a mensagem será lida pela aplicação destinatária, e nem lhe é dada garantia sobre o tempo exato que levará para a mensagem ser entregue. Estes aspectos são determinados fundamentalmente pela aplicação destinatária.

A implantação de sistemas distribuídos baseados em MOM, como mostrado na Figura 4.4, oferece uma abordagem baseada em serviço para comunicação entre processos. MOM é focado em infraestrutura no sentido de enviar e receber mensagens que permitam a módulos de aplicação se tornarem distribuídos em plataformas heterogêneas [2]. Para isso, a plataforma MOM injeta uma camada entre emissores e receptores, o que permite-lhes usar essa camada intermediária para intercâmbio de mensagens. Um benefício primário do uso de MOM é o fraco acoplamento entre participantes do sistema - a habilidade de ligar aplicações por meio de interfaces de comunicação, e que a mudança de implementação de uma entidade não implique na adaptação de outras, resultando em sistemas muito

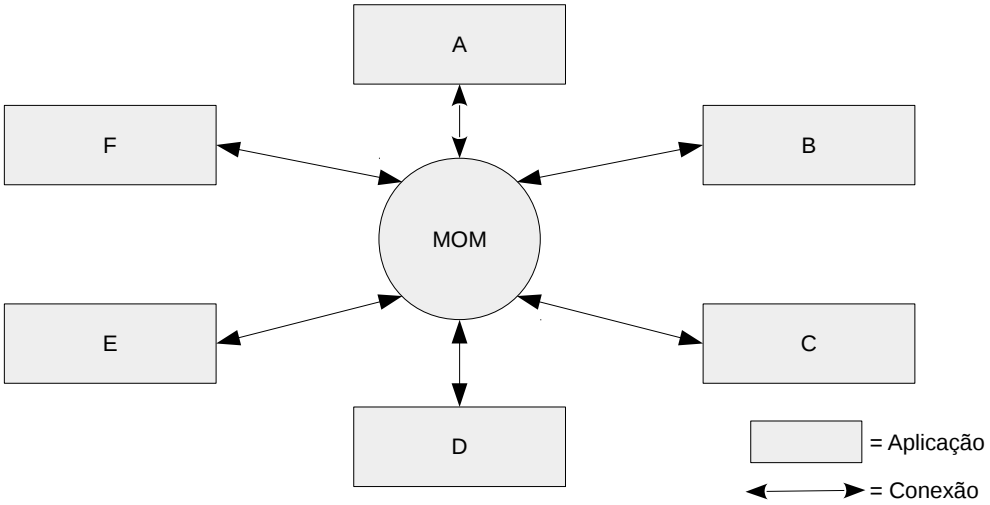


Figura 4.4: Exemplo de Infraestrutura de *Middleware* Orientado a Mensagens [11].

coesivos e desacoplados. Tanenbaum [46] apresenta uma visão gráfica de fraco acoplamento apoiado por uma fila de mensagens, mostrada na Figura 4.5: na situação (a), as entidades transmissora e receptora estão em execução, o que permite que a transmissão se complete, desde o envio da mensagem à fila, e posteriormente ao destinatário (receptor). Na situação (b), o transmissor está ativo, porém o receptor não está; por isso, a mensagem será entregue na fila, a qual aguardará o receptor ficar ativo para enviar-lhe. Na situação (c), o transmissor está desativado, porém o receptor está ativo; então o receptor somente receberá as mensagens que foram armazenadas na fila antes do transmissor ficar inoperante. Por fim, na situação (d), ambos transmissor e receptor estão desativados, o que implica que nenhum evento pode ocorrer.

Segundo Curry e Mahmoud [11], com MOM a perda de mensagens na rede ou por falha de sistemas é prevenida usando um mecanismo *store-and-forward* (armazena e encaminha) para persistência de mensagens. Isto garante alto nível de confiabilidade no mecanismo de distribuição do MOM, por previnir a perda de mensagens quando partes do sistema ficarem indisponíveis ou ocupadas. O nível de confiabilidade é configurável, mas sistemas de mensagens MOM conseguem garantir que uma mensagem pode ser entregue, e que ela será entregue a cada destinatário exatamente uma vez, como será explicado adiante.

Segundo Ibrahim e Hassan [30], adicionalmente ao desacoplamento da interação entre subsistemas, a arquitetura MOM também desacopla as características de desempenho de subsistemas um do outro. Subsistemas podem escalar independentemente, com pouco ou nenhum impacto aos demais. A arquitetura MOM também permite que sistemas lidem com picos de atividade não previstos em um subsistema, sem afetar outros processos ou funcionalidades.

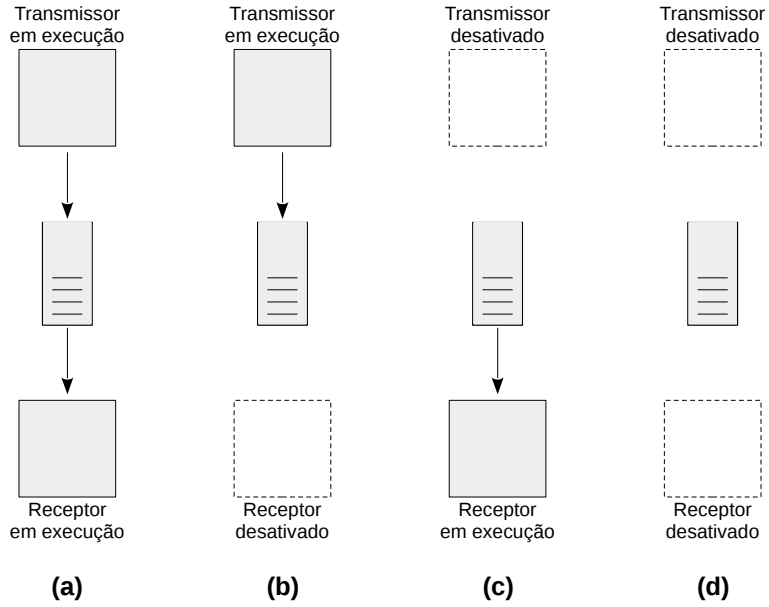


Figura 4.5: Quatro Combinações de Comunicação Fracamente Acoplada usando Filas [46].

Conforme explicam Curry e Mahmoud [11], a plataforma MOM permite alta disponibilidade em sistemas e redução no tempo de resposta do sistema como um todo. Como as mensagens serão entregues tão logo as entidades de destino estejam disponíveis, pode-se afirmar que o tempo de execução de processamento será o mais breve possível, considerando-se a disponibilidade de subsistemas envolvidos.

4.2.3 Cenários de Aplicação de MOM e RPC

Segundo Curry e Mahmoud [11], dependendo do cenário em que estão inseridos, tanto MOM quanto RPC tem vantagens e desvantagens. O modelo RPC provê uma abordagem mais direta para mensagens usando a interação síncrona. Entretanto, o mecanismo RPC cria restrições em virtude do forte acoplamento, que leva a um potencial crescimento geométrico de interfaces. Também é problemático escalar partes do sistema e lidar com indisponibilidades de serviço. O modelo RPC assume que todas as partes do sistema estarão simultaneamente disponíveis; se uma parte do sistema falhar, o sistema todo falhará como resultado.

Curry e Mahmoud [11] indicam que há uma grande sobrecarga associada à interação de RPC, já que requer mais largura de banda que uma interação similar em MOM. O modelo RPC é projetado para comunicar um único cliente e um único servidor por vez. O RPC tradicional não permite suporte à comunicação um-para-muitos. Contudo, a vantagem de um sistema RPC é a simplicidade do mecanismo e sua implementação direta e clara.

Outra vantagem é a garantia de processamento sequencial que o modelo síncrono oferece. Assim sendo, se o objetivo do sistema necessita da ordem de processamento garantida, a solução de RPC pode ser mais adequada.

Por outro lado, segundo Abie et al. [1], a plataforma MOM aumenta a interoperabilidade, a portabilidade e a flexibilidade de arquiteturas por habilitar a troca de mensagens entre programas sem precisar saber qual plataforma ou processador hospedam as outras aplicações em rede. A abordagem desacoplada de MOM permite a integração flexível de clientes em um sistema e o suporte de um grande número de consumidores/clientes e produtores/consumidores anonimamente [11].

Conforme explanam Curry e Mahmoud [11], o método RPC é ideal para sistemas fortemente tipados/orientados a objetos e fortemente acoplados, com verificação semântica em tempo de compilação e uma implementação geral direta. Entretanto, se o sistema distribuído será geograficamente disperso, com recursos físicos limitados para conectividade, porém severa demanda por confiabilidade, flexibilidade e escalabilidade, então MOM é a solução mais apropriada. Nesse contexto, dada a caracterização do sistema tratado neste trabalho, MOM foi a solução adotada para atingir os objetivos da pesquisa, ao tratar a troca de informações entre clientes para assegurar a sincronia de operações e dados, conforme é detalhado no Capítulo 5.

4.3 Filas de Mensagens

A fila de mensagens é um conceito fundamental em MOM. Filas proveem a habilidade de armazenar mensagens em uma plataforma MOM, o que viabiliza o modelo de interação assíncrono. Usualmente, as mensagens de uma fila são ordenadas sob um critério em particular. A fila padrão encontrada em sistemas de mensagem é a do tipo FIFO (*First-In-First-Out* ou primeiro que entra é o primeiro que sai) [11, 29].

Conforme explicam Tanenbaum e Steen [46], atributos de uma fila podem ser configurados. Isso inclui o nome da fila, seu tamanho, a quantidade de mensagens gravadas e mantidas, o algoritmo de ordenação, entre outros atributos. O enfileiramento é um benefício em particular para clientes onde a rede de comunicação é precária ou intermitente. Potencialmente, cada aplicação pode ter sua própria fila, ou aplicações podem compartilhar uma fila, pois não há restrição para essa configuração. Tipicamente, plataformas de MOM suportam múltiplas filas de diversos tipos, cada uma com um propósito diferente.

4.4 Modelos de Troca de Mensagens

Conforme explicam Curry e Mahmoud [11], um entendimento sólido dos modelos de troca de mensagens disponíveis usando MOM é a chave para visualizar as capacidades excepcionais que a arquitetura provê. Dois modelos principais são comumente utilizados, o modelo ponto-a-ponto e o modelo *publish/subscribe* (publicar/assinar). Ambos são modelos baseados no intercâmbio de mensagens através de um canal (fila), e serão descritos em detalhes nas próximas seções.

4.4.1 Ponto-a-ponto

O modelo de troca de mensagens ponto-a-ponto provê intercâmbio assíncrono e direto de mensagens entre entidades de software. Neste modelo, as mensagens originadas em clientes produtores são roteadas para clientes consumidores por meio de uma fila. O tipo de fila mais usado é a fila FIFO, em que as mensagens são ordenadas na sequência em que são recebidas e, assim que consumidas, as mensagens são removidas do início da fila. A característica central na fila ponto-a-ponto está na entrega da mensagem, que sempre ocorrerá a somente um destinatário por mensagem, ou seja, não haverão cópias entregues a outros clientes do sistema. A próxima solicitação de mensagem na fila, entregará a próxima mensagem disponível, que também será imediatamente descartada do topo da fila. No modelo ponto-a-ponto, mensagens são sempre entregues e armazenadas na fila até que um consumidor esteja pronto para recuperá-las.

Uma variação importante do modelo ponto-a-ponto é o modelo requisição-resposta. Este modelo é usado para a World Wide Web (WWW), onde um cliente requisita uma página a um servidor, e o servidor responde com a página requisitada. Neste cenário, o modelo requer que qualquer produtor (cliente web) que envie uma mensagem esteja pronto para receber a resposta de consumidores (servidor web) em algum momento no futuro. Este modelo baseia-se no mecanismo de troca de mensagens ponto-a-ponto, com a diferença de operar sobre um passo adicional que é a resposta, também como mensagem ponto-a-ponto [11].

4.4.2 *Publish/Subscribe*

O modelo de mensagens *publish/subscribe* é um mecanismo valioso usado para disseminar informação entre consumidores e produtores anônimos [11]. Este mecanismo de distribuição um-para-muitos e muitos-para-muitos permite que um simples produtor envie uma mensagem para um usuário ou até centenas de milhares de consumidores.

Segundo Abie et al. [1], o modelo de mensagens *publish/subscribe*, ou pub/sub, possui algumas propriedades chave oferecendo meios eficazes para modelagem de sistemas:

- Pode ser modelado e re-fatorado durante a execução como se estivesse em fase de desenvolvimento, já que a troca de mensagens é assíncrona e sem conexão fim-a-fim (*connectionless*);
- Sistemas ao redor de arquiteturas MOM que implementam o modelo pub/sub são inerentemente extensíveis; e
- O modelo pub/sub é uma base poderosa para implementação de recursos escaláveis e resilientes.

Segundo Curry e Mahmoud [11], no modelo pub/sub, as aplicações emissoras e receptoras não precisam conhecer umas às outras, nem conhecer o funcionamento alheio. Somente é necessário enviar a informação para um destino dentro do mecanismo pub/sub, que será repassado aos destinatários consumidores. Clientes produzindo mensagens "publicam" a um tópico ou canal específico, onde tais canais são "assinados" por clientes interessados em consumir as mensagens. O serviço roteia as mensagens para consumidores baseando-se nas assinaturas em tópicos de interesse dos clientes. No modelo pub/sub não há restrição quanto ao papel de um cliente, cada um pode tanto produzir quanto consumir do mesmo tópico/canal.

Inúmeros métodos de intercâmbio de mensagens pub/sub foram desenvolvidos, os quais suportam diferentes características, técnicas, e algoritmos para filtro de mensagens, publicação, assinatura, e gerenciamento de assinaturas. Quando se usa este modelo de troca de mensagens, um cliente consumidor tem duas formas de receber mensagens de um provedor de MOM, os quais são [11]:

- *Pull*: um consumidor pode solicitar ao provedor para verificar por novas mensagens, efetivamente "*puxando-as*" do provedor.
- *Push*: alternativamente, um consumidor pode requisitar que o provedor envie mensagens relevantes tão logo as receba; o cliente instrui o provedor a "*empurrar-lhe*" as mensagens.

Os dois modelos apresentados possuem capacidades similares e muitos dos objetivos de intercâmbio de mensagens podem ser obtidos usando qualquer um dos dois modelos, ou ambos.

A diferença fundamental entre eles remete ao fato que, no modelo pub/sub, cada assinante de um canal receberá uma cópia de cada mensagem lá publicada, enquanto

que no modelo ponto-a-ponto somente um consumidor a receberá. O modelo pub/sub é normalmente usado em cenários de difusão onde um publicador deseja enviar uma mensagem a diversos clientes. O publicador não tem controle real sobre quantos clientes receberão a mensagem, nem qualquer garantia de que qualquer um deles receberá de fato uma cópia. Mesmo no cenário de mensagens um-para-um, tópicos podem ser úteis para categorizar diferentes tipos de mensagens. O modelo pub/sub é o mais poderoso modelo de troca de mensagens por sua flexibilidade; a desvantagem neste caso é a sua complexidade.

4.5 Principais Serviços de MOM

A arquitetura MOM permite a integração de mais serviços básicos, em extensão aos recursos de filas de mensagens. Esta seção apresentará uma visão geral sobre tais fatores.

- **Filtro de Mensagens**: Fitros de mensagens tornam emissores e receptores mais seletivos face ao conteúdo que receberão de canais ou tópicos. A filtragem pode operar em vários níveis, assim os filtros podem usar expressões de lógica booleana para declarar mensagens de interesse do cliente, mas o formato das expressões depende diretamente da implementação.

- **Transações**: Transações proveem a habilidade de agrupar tarefas em uma unidade de trabalho. Segundo Curry e Mahmoud [11], a definição básica mais direta sobre transação é a de que todas as tarefas devem completar com sucesso, ou todas falharão juntas.

  Quando um cliente deseja enviar ou receber mensagens dentro de uma transação, isso é referenciado como Troca de Mensagens Transacional. Esta modalidade é usada quando se deseja executar várias tarefas com mensagens, enviando 1 a N mensagens, de forma que todas as tarefas executem com sucesso, ou em caso contrário todas falharão. Quando a troca transacional é usada, o envio ou o recebimento de mensagens tem a oportunidade de completar com sucesso a transação (*commit*), ou abortar a transação em caso de falha de uma ou mais operações, de modo que todas as ações da transação são desfeitas, voltando todas as entidades envolvidas ao estado anterior a seu início [11].

- **Garantia de Entrega de Mensagens**: Um serviço MOM tipicamente permite a configuração de perfis de qualidade de serviço (QoS - *Quality of Service*) para a entrega de mensagens [11]. De modo geral, é possível realizar a entrega de mensagens em modo *um-para-muitos*, *no-mínimo-um*, ou *somente-uma-vez*. A confirmação de

entrega das mensagens pode ser configurada, adicionalmente ao número de tentativas em caso de falha de entrega. Com a comunicação assíncrona e persistente, a mensagem pode ser enviada ao serviço de troca de mensagens que a armazenará pelo tempo que for necessário para a entrega, a menos que o tempo de vida (TTL - *Time-To-Live*) da mensagem expire.

- **Formatos de Mensagens**: Dependendo da implementação de MOM, inúmeros formatos de mensagens pode ser usados pelo usuário. Alguns dos tipos mais comuns incluem formato texto (como XML), objetos, fluxos de dados (*stream*), *HashMaps*, fluxos multimídia, entre outros [11]. Provedores MOM podem oferecer mecanismos para transformar mensagens de um formato para outro, alterando o conteúdo da mensagem. Além disso, algumas implementações de MOM permitem transformação XSL transportada junto ao conteúdo de mensagens XML. Tais plataformas MOM são encontradas em agentes de mensagens que tratam também as diferenças entre diversos sistemas.

- **Balanceamento de Carga**: O balanceamento de carga é um processo de espalhar a carga do sistema em vários servidores (neste cenário, um servidor pode ser uma máquina física ou software servidor ou até ambos). Um sistema corretamente balanceado deve distribuir dinamicamente o trabalho entre servidores, dinamicamente alocando tarefas para o servidor menos sobrecarregado.

  Segundo Curry e Mahmoud [11], duas abordagens podem ser adotadas para o balanceamento, "*push*" e "*pull*". No modelo "*push*", um algoritmo é usado para balancear a carga entre múltiplos servidores. Inúmeros algoritmos existem para este fim, os quais buscam prever qual servidor está menos sobrecarregado e repassar-lhe a requisição. O algoritmo, em conjunto com a previsão prognóstica de carga, baseiam-se nos registros de desempenho de cada servidor participante; caso não tenha tais informações, somente poderá presumir o servidor menos ocupado.

  No modelo "*pull*", a carga é balanceada ao se colocar as mensagens de requisição em uma fila ponto-a-ponto, de modo que os servidores consumam as mensagens pela fila. Isto permite um verdadeiro balanceamento de carga, já que um servidor somente requisitará uma mensagem se tiver capacidade de processamento disponível. Assim, o modelo "*pull*" provê o mecanismo ideal de balanceamento, de forma a distribuir igualmente a carga no sistema.

- ***Clustering***: Com o objetivo de recuperar-se de falhas, o estado do servidor precisa ser replicado através de múltiplos servidores. Isso permite a um cliente migrar transparentemente para um servidor alternativo, se o servidor atual falhar. *Clustering* é a distribuição de um aplicativo sobre múltiplos servidores para escalar o

sistema além dos limites físicos locais, oferecendo mais confiabilidade e desempenho se comparado ao uso de um único servidor. A arquitetura MOM oferece esse tipo de integração nativa ao modo de operação, pela opção de entrega de mensagens a destinatários ativos dentro de um conjunto pré-determinado.

4.6 Considerações Finais

Este capítulo esclareceu a fundamentação teórica e prática da arquitetura baseada em MOM. Para esta pesquisa, é fundamental compreender os modelos de interação entre processos, tendo em vista sua aplicabilidade no cenário de sistemas corporativos.

O próximo capítulo tratará de um *middleware* específico, o *Stacksync* [35]. Esta plataforma permite a sincronização de informações entre dispositivos de armazenamento, e serão apresentadas suas características de funcionamento, cenários de aplicação e requisitos operacionais.

Capítulo 5

Plataforma *StackSync*

Neste capítulo é apresentada a plataforma StackSync, parte do projeto OpenStack [19] de computação distribuída. Inicialmente, serão apresentadas as características do projeto StackSync, e seus principais componentes de arquitetura e mecanismos básicos. O *framework* ObjectMQ é detalhado na Seção 5.2, e compreende as principais funcionalidades da plataforma StackSync. Em seguida, na Seção 5.3, é apresentado o MOM RabbitMQ e sua participação no apoio ao StackSync. Na Seção 5.5 é apresentada a plataforma e seu protocolo de comunicação entre clientes e servidores, de modo a permitir a escalabilidade e tolerância a falhas inerentemente. Por fim, na mesma seção, o mecanismo de sincronização é detalhado, com suas funcionalidades e o modo como as trocas de mensagens se processam no *middleware*.

5.1 Características do StackSync

A plataforma StackSync é um *middleware* que colabora com o armazenamento de arquivos em sistemas distribuídos, pois permite que o mesmo sistema de arquivos seja compartilhado entre diversos sistemas operacionais, de forma transparente. Assim sendo, vários componentes distribuídos podem trocar informações gravando arquivos, de forma colaborativa, e abstrair a complexidade do compartilhamento, que fica por conta do *middleware* StackSync.

A tarefa de sincronização de arquivos entre diversos espaços de armazenamento é uma tarefa complexa. No cenário de computação distribuída, segundo Lopez et al. [36], há dois desafios que se destacam, os quais são o desafio de promover a escalabilidade de sistemas, de maneira automatizada e transparente, por meio de implementação na plataforma de hospedagem; e também o desafio de notificar eficientemente todas as cópias de um sistema de arquivos sincronizado, pensando na escala de milhões de clientes concorrentes.

O primeiro desafio é relacionado ao fato de que escalar alguns tipos de aplicações distribuídas não é direto ou trivial, fundamentalmente porque algumas aplicações são intensivas em processamento ou uso de memória, enquanto que outras são intensivas em leitura e gravação em disco. Nestes casos, onde leitura e gravação em disco são o foco da aplicação, Lopez et al. [36] indicam que a arquitetura precisa melhorar índices de desempenho específicos, como tempo médio de entrega de mensagens, ou tempo de processamento de mensagens, pois são fatores mais diretamente relacionados ao desempenho global da aplicação distribuída, particularmente à escalabilidade.

Outro desafio é que a alta taxa de leitura e escrita para sincronização de arquivos faz com que a comunicação *push* no modo um-para-muitos seja mais adequada para rápida notificação. Segundo Lopez et al. [36] para manter de forma eficiente a consistência dos arquivos, qualquer alteração ocorrida deve ser notificada o mais breve possível para reduzir conflitos, em particular, quando um arquivo está suscetível a ser modificado por mais de um cliente ao mesmo tempo. Isso requer que o serviço de sincronização opere o mais breve possível para enviar as mudanças, com a colaboração de um serviço eficiente de notificações para informar aos clientes sobre as alterações em arquivos.

Dois aspectos são relevantes em sistemas de sincronização de arquivos: 1) desacoplamento dos fluxos de controle em relação aos fluxos de dados; e 2) notificação de mudanças usando o método *push*. Isso exige uma conexão TCP constantemente aberta com o servidor, usada para receber informações sobre mudanças ocorridas em qualquer nó replicado.

5.2 O *Framework* ObjectMQ

O ObjectMQ é um *framework* que provê escalabilidade programática para objetos distribuídos, e que utiliza um MOM em seu projeto. O ObjectMQ faz uso de técnicas de RPC para comunicação entre componentes do sistema, descritas no Capítulo 2 deste trabalho, aliado à infraestrutura de MOM para permitir as características desejáveis explicadas no Capítulo 4.

Segundo Lopez et al. [36], para permitir o correto funcionamento de aplicações baseadas em ObjectMQ, conforme apresentado na Figura 5.1, é necessária a atuação de alguns componentes fundamentais, descritos a seguir:

- ***Stub* cliente**: a extremidade de conversão ou *stub* do cliente permite a chamada remota utilizando a camada de comunicação do MOM. Para fazer a chamada remota, o *stub* envia uma mensagem para uma fila, que por sua vez foi assinada pelo objeto remoto. Cada *stub* também precisa de sua própria fila para receber respostas do lado servidor.

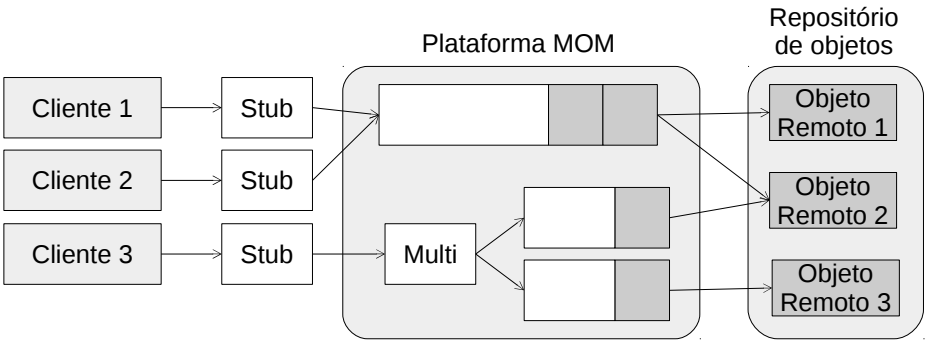


Figura 5.1: Arquitetura do *Framework* ObjectMQ [36].

- **Plataforma MOM**: é a camada entre *stubs* e objetos remotos. Cada *stub* tem sua própria fila para receber respostas de objetos remotos. A Figura 5.1 mostra dois tipos de filas que um objeto remoto pode assinar. Especificamente, a fila superior é uma fila *global*, compartilhada entre os diferentes objetos remotos. As duas filas inferiores correspondem a filas *privadas*, nas quais cada objeto escuta e recebe chamadas.

- **Objetos remotos**: são os objetos que escutam as filas e executam as chamadas remotas (RPC). Para adicionar um objeto remoto ao sistema, o ObjectMQ provê um método específico, que pode ser executado pelo sistema baseado nesta arquitetura. O método cria uma fila, baseada no identificador do objeto executante, ligando-o assim ao objeto remoto por meio de uma fila no MOM. Se esta fila já existir, a nova instância de objeto remoto simplesmente a assinará, ou seja, receberá notificações através da fila. Segundo Lopez et al. [36], este mecanismo de conexão auxilia a escalar o sistema por criar objetos dinamicamente e fazê-los assinar filas nomeadas específicas, com o MOM provendo automaticamente o balanceamento de carga para todos os objetos que assinam a fila. Pelo fato de que somente um objeto remoto pode consumir uma mensagem específica, isto significa que a mesma mensagem não será entregue a nenhum outro objeto remoto. Se for o caso de entregar a mesma mensagem a mais de um objeto remoto, então deve-se utilizar uma fila privada para cada destino, com o uso de um mecanismo de *multicast* como exibido na Figura 5.1.

Além disso, a Figura 5.1 ilustra os dois tipos de invocação remota suportados por ObjectMQ: *unicast* e *multicast*. As invocações *unicast*, apresentadas pelos clientes 1 e 2, são processadas pela fila global. Para este tipo de chamada, o MOM entregará a mensagem de chamada remota para o primeiro objeto remoto que estiver ocioso. Em invocações *multicast*, apresentadas no exemplo pelo cliente 3, a mesma mensagem de

chamada remota será enviada para todas as filas privadas conectadas pelo mesmo, criando várias cópias da mesma chamada remota.

Dessa forma, o principal objetivo do *framework* ObjectMQ foi criar uma camada minimalista de comunicação, delegando a complexidade das comunicações para a camada de mensagens. O *middleware* implementado sob o conjunto de padrões de ObjectMQ delega as responsabilidades, tanto quanto possível, para o sistema MOM. Assim sendo, o MOM fica responsável pelo balanceamento de carga, ao mesmo tempo que evita a perda de mensagens. O MOM também auxilia no serviço de nomes para os objetos, pois o *framework* permite a conexão aos objetos remotos por meio de nomes de identificação. Assim, é possível utilizar mecanismos de pesquisa por nomes entre os objetos disponíveis. Para isso, ao invés de usar um registro de nomes centralizado, são usadas as filas como meio de conexão entre objetos e identificadores. Como resultado, quando um *stub* quiser interagir com um objeto remoto, não é necessário pesquisar um registro. Ao invés disso, é suficiente saber o nome da fila para onde se queira enviar uma mensagem.

Para permitir a troca de informações entre entidades clientes e objetos remotos, a plataforma ObjectMQ implementa uma classe de objetos para agentes de troca de mensagens, chamada *Broker*. Essa classe possui os mecanismos fundamentais para esta finalidade, através de dois métodos. O primeiro método é o *bind*, que conecta um objeto remoto a um identificador, que por sua vez referencia uma fila no MOM. O outro método é o *lookup*, o qual conecta um cliente a uma fila no MOM através de seu identificador, e internamente conecta uma fila privativa para o cliente receber respostas do objeto remoto.

Cabe observar que ligar mais do que um objeto remoto com o mesmo identificador também significa que a carga de clientes será distribuida igualmente entre múltiplos objetos remotos. Isso ajudará a escalar o serviço, para cima ou para baixo, adicionando ou removendo objetos remotos dinamicamente. Neste caso, não há necessidade de modificar os *stubs* clientes, e eles não precisam de aviso quanto às modificações no conjunto de objetos remotos que oferecem o serviço, ou seja, a escalabilidade é efetivamente transparente.

Conforme apresentado por Lopez et al. [36], em ObjectMQ a transparência de objetos remotos não é desejada, porque ao se utilizar o *framework*, pode ser necessário ter acesso às informações de entidades remotas ou locais para programar de forma a refletir as restrições de indeterminância e concorrência inerentes ao uso de objetos remotos. Por esta razão, o ObjectMQ oferece mecanismos explícitos para definir primitivas de invocação de métodos. Em particular, são oferecidas três principais abstrações de invocação, as quais são *assíncrona*, *síncrona* e *multi-chamada*, conforme descritas a seguir [36]:

- **Método Assíncrono**: esta é a forma de invocação assíncrona e não-bloqueante, onde o cliente publica uma mensagem em uma fila de requisições para o objeto alvo.

Por padrão, o cliente não espera receber resposta ou notificação de entrega de sua mensagem, ou mesmo se sua mensagem foi encaminhada corretamente.

- **Método Síncrono**: esta chamada remota é da forma síncrona bloqueante, na qual o cliente publica uma mensagem na fila de requisições do objeto alvo, bloqueando sua execução até que uma resposta seja recebida em sua fila de resposta de cliente. Esta chamada pode ser configurada com um tempo limite (*timeout*), e um número máximo de tentativas, antes de lançar uma exceção de transmissão.
- **Método Multi-chamada**: esta é a invocação um-para-muitos de um cliente para múltiplos objetos alvo, e para isso combina invocações síncronas e assíncronas. A entidade intermediária gera múltiplas invocações não-bloqueantes para vários servidores, nos quais, posteriormente, é produzida uma invocação múltipla e bloqueante, que coleta os resultados recebidos dos vários servidores em um determinado *timeout*.

Por um lado, invocações assíncronas encaixam-se à camada de troca de mensagens imediatamente abaixo da aplicação, na arquitetura MOM. Elas reduzem o fardo de manipular mensagens e filas, e não aumentam demasiadamente a carga nas comunicações. Por outro lado, invocações síncronas implicam que o intermediador bloqueará durante um período, até o limite estabelecido pelo *timeout*, para esperar por um resultado. Chamadas síncronas neste modelo devem atravessar um agente intermediário (servidor de mensagens) que não é um elemento necessário em modelos cliente-servidor. Deste modo, ocorre uma efetiva carga adicional na comunicação, apesar de relativamente pequena, pois as mensagens devem trafegar através de filas de objetos de servidores e de clientes. O benefício é que, ao delegar a comunicação para a camada de mensagens, o servidor que implementa ObjectMQ não pode ser saturado com mensagens, já que receberá somente as que conseguirá processar.

Um dos objetivos do *framework* ObjectMQ é prover uma plataforma tolerante a defeitos. Na subseção a seguir são indicadas as técnicas disponíveis para permitir este tipo de plataforma, e como o *middleware* construído a partir deste *framework* pode fazer uso destes recursos.

5.2.1 Tolerância a Defeitos

Segundo Lopez et al. [36], o mecanismo implementado pelo ObjectMQ não perde informações, particularmente porque não guarda os dados de objetos somente na memória principal. Em uma situação de falha de processamento em um objeto remoto, a operação será enviada para outra instância ligada à mesma fila do objeto com defeito. Desta forma, nenhuma invocação remota pode ser perdida. Isto é possível porque toda mensagem enviada para um objeto remoto fica armazenada no sistema de filas até que o objeto remoto

envie uma confirmação do sucesso da operação. Usando esta abordagem, o sistema de mensagens pode saber também quais instâncias estão ocupadas ou não para balancear a carga.

Outra importante propriedade é que o *framework* permite o uso de objetos *supervisores*. Quando um objeto remoto falha, se ele estiver usando uma multichamada, ou seja, há outros objetos conectados à mesma fila, basta remover a ligação do objeto defeituoso à fila, e o sistema continuará funcionando sem maiores perdas.

O *supervisor* também pode falhar. Se isso ocorrer, será impossível saber quais objetos estão ativos e em estado correto. Para endereçar esta questão, cada agente intermediário no sistema verificará periodicamente se o *supervisor* está executando. Em qualquer situação em que o *supervisor* falhar, um algoritmo de eleição de líder será executado usando os identificadores únicos dos agentes.

Finalmente, para tolerar defeitos nos agentes intermediários, o sistema de mensagens pode ser configurado para armazenar todas as mensagens presentes nas filas, de modo que quando o sistema for reiniciado, as mensagens não processadas possam ser recuperadas. De ambos os modos, a alta disponibilidade pode ser alcançada usando *clusters* de agentes de mensagens.

A próxima seção apresenta o MOM RabbitMQ, utilizado nesta pesquisa, e que é capaz de colaborar com a implementação de ObjectMQ. A implementação de RabbitMQ, segundo Lopez et al. [36], compreende um conjunto de mecanismos de interesse ao funcionamento do sistema StackSync, que é apresentado com mais detalhes na Seção 5.5.

5.3 O MOM RabbitMQ

O RabbitMQ [44] é um MOM que implementa eficazmente a comunicação por meio de filas de mensagens. É um projeto de código aberto, implementado na linguagem de programação Erlang [49], e que pode conectar-se a diversas linguagens de programação, por meio de bibliotecas de conexão apropriadas. Segundo Smith [43], O RabbitMQ possui uma interface de interação compatível com o padrão AMQP (*Advanced Message Queueing Protocol*, ou Protocolo Avançado de Enfileiramento de Mensagens) [50], o que faz dele um MOM adaptável a várias implementações, especialmente, por poder substituir ou ser substituido por outros MOMs que apresentem características desejáveis ao objetivo almejado.

A implementação de RabbitMQ oferece todos os mecanismos necessários ao funcionamento do *framework* ObjectMQ, pois permite o controle de filas com identificação por rótulo, inclui os protocolos ponto-a-ponto e *publish/subscribe*, e ainda define mecanismos de segurança de acesso a filas. Também possui acesso aos mecanismos de invocação de no

mínimo um, somente uma vez, e no máximo um, o que implica em confiabilidade para a plataforma.

Sua escolha, segundo Lopez et al. [36], deve-se principalmente ao fato de ser um projeto de código livre, em que os ajustes necessários para sua adaptação poderiam ocorrer durante a construção do projeto StackSync. Deste modo, o critério de código aberto foi o fator preponderante para sua escolha neste trabalho. Contudo, com o uso do *framework* ObjectMQ, foi possível garantir que outras plataformas de MOM possam ser utilizadas no futuro, tendo em vista a compatibilidade efetiva do RabbitMQ com o protocolo *Advanced Message Queueing Protocol*, assegurado também pelo *framework* em questão.

A próxima seção apresenta informações sobre a plataforma de armazenamento OpenStack Swift, e seu papel no apoio ao software StackSync.

5.4 A Plataforma OpenStack Swift

Segundo Biswas et al. [5], a plataforma OpenStack Swift [20] compreende um sistema de armazenamento de objetos, de código aberto, e que permite a construção de *clusters* de armazenamento, para combinar o desempenho e a alta disponibilidade em sistemas distribuídos. Sua arquitetura comporta a hospedagem de múltiplos usuários e permite a colaboração, pois sua concepção foca no suporte ao desenvolvimento de sistemas, ao invés de simplesmente entregar espaço de armazenamento. Para isso, as interfaces de acesso são voltadas para o rápido provisionamento, e o uso do protocolo HTTP para comunicação permite a recuperação de forma eficiente e rápida na troca de mensagens.

A plataforma Swift aloca múltiplas cópias de objetos em espaços de armazenamento, chamados de *conteiners*, de modo que a recuperação é imediata quando qualquer uma das réplicas torna-se indisponível. Para isso, sua concepção sugere fortemente o uso de diversos dispositivos de hardware, alocando réplicas de objetos em dispositivos distintos. A organização interna dos dispositivos de armazenamento ocorre por regiões, zonas, servidores e *drives*, para permitir que a alocação de *conteiners* possa ocorrer inclusive em localidades geograficamente distantes, mas de uma forma relativamente simples de administrar.

Segundo Biswas et al. [5], o controle de acesso de usuários é feito por meio de ACL (*Access Control List*, ou Lista de Controle de Acesso), em que usuários são associados a *conteiners* em que possuem permissões, classificadas em níveis. O controle de usuários se dá pelo provedor de identificação OpenStack Keystone, que centraliza a responsabilidade de autorizar usuários na plataforma.

A próxima seção apresenta uma implementação que utiliza o *framework* ObjectMQ, e que tem como objetivo a sincronia de arquivos em diferentes espaços de armazenamento.

O sistema StackSync, mecanismo utilizado nesta pesquisa, oferece recursos que constituem um *middleware* para computação distribuída, que implementa as características de transparência e escalabilidade de forma muito versátil.

5.5 StackSync: Serviço de Sincronia de Arquivos

O StackSync é uma implementação de código livre de um sistema de sincronia de arquivos. Segundo Lopez et al. [36], em alto nível, sua arquitetura é similar a uma aplicação Web de três camadas, onde as filas são a camada de apresentação e balanceamento de carga, o serviço de sincronia é a camada lógica de negócio, e o banco de metadados é a camada persistente.

O StackSync é caracterizado por dois principais componentes: um aplicativo cliente, que executa em dispositivos clientes, e um processo servidor *back-end* com duas funcionalidades principais: armazenagem de arquivos, e o gerenciamento dos metadados associados aos arquivos, incluindo versionamento, histórico de modificação de atributos, tempo de última modificação, e outros atributos. O StackSync desacopla metadados dos fluxos de dados armazenados, dividindo o serviço de *back-end* em dois componentes separados: o *Back-end de Armazenamento*, que hospeda os arquivos, e o *Back-end de Metadados*, que é responsável por gerenciar os metadados de sincronia de arquivos. Para os experimentos realizados nesta pesquisa, foi utilizada uma versão estável da infraestrutura, apoiada por um banco PostgreSQL [27] como *Back-end de Metadados*, e o armazenamento de objetos OpenStack Swift [20] como *Back-end de Armazenamento*. O projeto interno do StackSync permite utilizar outras tecnologias e protocolos para as finalidades de *Back-end*, como FTP (*File Transfer Protocol*) [41] para armazenamento de arquivos e objetos, e outros bancos de dados para os metadados. O controle de acesso aproveitou a infraestrutura necessária para autenticação do OpenStack Swift, e utilizou como controle de usuários o OpenStack Keystone, por sua robustez e versatilidade [36].

O componente chave que interage com o *Back-end de Metadados* é o serviço de sincronia de arquivos, referenciado como *SyncService*. Este serviço processa requisições de sincronia de clientes. Basicamente, o serviço verifica se as mudanças propostas pelo cliente são consistentes, e depois aplica as mudanças em caso afirmativo. A arquitetura do StackSync e os canais de troca de informações podem ser visualizados na Figura 5.2.

Lopez et al. [36] indicam que a estrutura do StackSync cliente é baseada no projeto Syncany [28], enquanto que o servidor ou *SyncService* foi implementado com o *framework* ObjectMQ, ambos em Java.

O aplicativo cliente monitora arquivos e diretórios locais, delimitados em suas configurações, e envia todas as alterações para o repositório remoto. A interação ocorre com

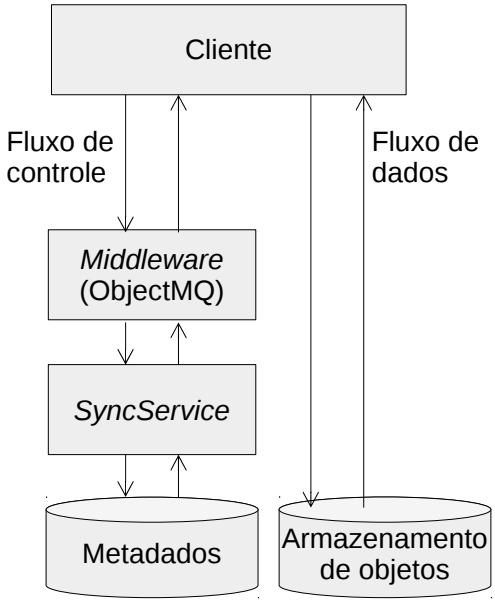


Figura 5.2: Arquitetura do StackSync [36].

os dois *back-ends* citados: a coordenação das mudanças em arquivos é sinalizada pelo *SyncService*; e os objetos correspondentes às mudanças, que devem ser armazenados, são encaminhados ao *Storage back-end*. Para conseguir atingir os objetivos da plataforma, foi necessário manter uma base de dados local em cada cliente, com os metadados dos arquivos sincronizados [36].

O aplicativo cliente processa periodicamente as mudanças em arquivos locais, recuperadas por um processo de monitoramento ao sistema operacional e as sinalizações específicas de manipulação de arquivos. Qualquer evento de modificação em arquivos monitorados, seja de criação, alteração ou remoção de arquivos e diretórios, gerará um evento que será registrado localmente e encaminhado ao *SyncService*.

Um mecanismo paralelo a esse serviço de monitoramento fragmenta o arquivo em blocos, de modo que as alterações possam ser manipuladas pelo tamanho do bloco, e não no arquivo completo. Isto torna o processo de transferência de arquivos modificados muito mais otimizado, pois somente os dados efetivamente modificados serão copiados para o *Storage back-end*. O tamanho utilizado para os blocos foi de 512 kilobytes nesta pesquisa, em compatibilidade ao ambiente de testes e dos arquivos utilizados. Os blocos são indexados na base local por meio de um *hash*, tornando mais fácil a consulta por mudanças no coneúdo dos blocos, pois basta comparar o *hash* do bloco atual com o *hash* armazenado no banco de dados local.

Quando o processo indexador de arquivos identificar que algum bloco de dados foi efetivamente modificado, ele iniciará o processo de envio dos novos blocos para o *Storage back-end*. Os blocos de dados que não foram modificados permanecerão inalterados. Após

a transferência dos novos blocos, o processo indexador envia uma mensagem assíncrona ao *SyncService*, avisando sobre os novos blocos, e em seguida enviando as atualizações de arquivos.

Durante o trâmite de arquivos, podem surgir novas modificações ocorridas remotamente, aplicadas ao conjunto de arquivos monitorados. Neste caso, segundo Lopez et al. [36], deve-se definir um modo de tratamento específico. No caso da implementação utilizada de StackSync, o mecanismo prevê a criação de um novo arquivo como cópia do conflito, renomeado por meio do acréscimo da data/hora de ocorrência do conflito. Esta abordagem permite o rastreamento da causa dos conflitos em análise posterior.

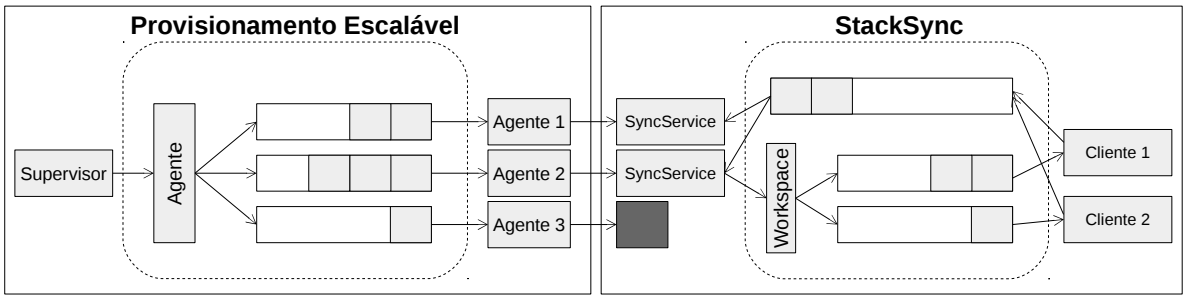


Figura 5.3: Fluxo de Mensagens do StackSync [36].

O *SyncService* participa como o lado servidor, e é uma implementação do *framework* ObjectMQ. Como pode ser percebido na Figura 5.3, uma fila global é utilizada para enviar mensagens dos clientes às instâncias do *SyncService*, e há uma fila de resposta para cada cliente, com um mecanismo de multichamada para replicar mensagens. O objeto denominado *Workspace* permite o compartilhamento e a colaboração em espaços de armazenamento entre clientes, e para isso necessita da fila multichamada.

Quando um cliente inicia sua execução, ele solicita ao *SyncService* o estado atual do *Workspace* no qual tem permissão de acesso registrada. Então, o *SyncService* entrega o estado atual do *Workspace*, com todas as modificações recentes desde a última sincronização. Após esta sincronia de inicialização, o cliente registra interesse em receber as mudanças no *Workspace*, através de uma fila privativa e o *SyncService* usa o método *push* para enviar mensagens.

O cliente possui um método em que avisa o *SyncService* de alterações em seu *Workspace*, por meio de um método assíncrono. O *SyncService*, por sua vez, executa um *push* assíncrono, numa operação um-para-muitos através da fila multichamada, informando todos os clientes conectados ao *Workspace* sobre as mudanças recentes. Toda vez que um conflito de arquivos for detectado, o *SyncService* resolve usando a versão do cliente que for processada primeiro. Os clientes seguintes gerarão cópias renomeadas do arquivo em conflito, indicando no nome do arquivo o momento em que o conflito ocorreu.

Os mecanismos apresentados no provisionamento escalável são oferecidos pelo *framework* ObjectMQ. Novos objetos remotos do tipo *SyncService* podem ser alocados ao StackSync, e isso ocorre automaticamente conforme a demanda de clientes aumenta. Segundo Lopez et al. [36], o *framework* permite a análise automática das filas, considerando o volume de mensagens na fila global, bem como o tempo de espera de cada mensagem processada. Assim, o sistema consegue definir se precisa alocar novos objetos remotos para atender os clientes conectados. O *Supervisor* envia mensagens de verificação aos objetos *SyncService* e, conforme previsto pelo *framework*, se algum deles apresentar defeitos, atua em sua desativação e substituição.

5.6 Considerações Finais

Este capítulo apresentou o mecanismo de sincronização de arquivos StackSync e uma visão geral do seu funcionamento. O *framework* ObjectMQ aparece como viabilizador da infraestrutura, associado ao RabbitMQ, um MOM de código aberto que implementa o protocolo AMQP. Todos esses elementos são fundamentais para a implementação do *middleware* StackSync, cujo objetivo é sincronizar arquivos entre ambientes de armazenamento heterogêneos. As funcionalidades que implementam a sincronia de arquivos apresentam particularidades que serão exploradas nesta pesquisa, por isso foram apresentadas com mais detalhes.

O próximo capítulo apresenta as características do Departamento-Geral do Pessoal do Exército Brasileiro - DGP, e sua missão como órgão da administração pública e da Força Terrestre. Algumas das atribuições do DGP estão no apoio à família militar, entre dependentes e pensionistas do sistema previdenciário. E para este fim, destinam-se alguns sistemas legados, fundamentais para sua perfeita execução e para a celeridade em processos, tanto de concessão de benefícios, quanto para geração de direitos. Dessa forma, o próximo capítulo apresenta o Sistema de Controle Documental de Acesso à Reserva, o *Papiro*, que tem primordial importância neste cenário.

Capítulo 6

Controle de Pessoal no Exército

Neste capítulo é apresentado o cenário de atuação do Departamento-Geral do Pessoal (DGP) e sua missão constitucional. Em seguida será apresentada a visão geral da Diretoria de Civis, Inativos e Pensionistas (DCIPAS), em que o Sistema de Controle de Documentação Funcional (Papiro) participa como facilitador na concessão de benefícios e direitos aos militares da reserva e seus dependentes. Por fim, é apresentada a importância do sistema Papiro no território nacional, apoiando processos em todas as regiões militares.

6.1 O Departamento-Geral do Pessoal

O Departamento-Geral do Pessoal (DGP) é o Órgão de Direção Setorial do Exército Brasileiro [8] responsável, fundamentalmente, pelo controle de servidores civis e de militares que o compõem. Segundo o Portal de Informações do DGP [14], sua missão é "Planejar, orientar, coordenar e controlar as atividades de pessoal decorrentes da Legislação de Pessoal vigente e do Sistema de Planejamento do Exército (SIPLEx), a fim de assegurar ao Exército Brasileiro condições para cumprir sua destinação constitucional e as atribuições subsidiárias explicitadas em Lei Complementar e participar de Operações Internacionais."

No conjunto de atribuições do DGP, o apoio à família militar se inicia durante o tempo de serviço ativo, compreendido como o tempo em que o militar presta serviços ao Exército, nas condições previstas pelo Estatuto dos Militares [7]. Esse apoio ao militar e seus dependentes continua mesmo após sua aposentadoria, que, ainda segundo o Estatuto dos Militares, pode ser sob a forma de:

- Reserva remunerada, quando pertençam à reserva das Forças Armadas e percebam remuneração da União, porém sujeitos, ainda, à prestação de serviço na ativa, mediante convocação ou mobilização; e

- Reforma, quando, tendo passado por uma das situações anteriores estejam dispensados, definitivamente, da prestação de serviço na ativa, mas continuem a receber remuneração da União.

Outrossim, o Exército possui um sistema previdenciário regulado separadamente dos demais órgãos federais, e o apoio aos dependentes continua mesmo após o falecimento do militar, sendo da ativa, da reserva remunerada ou reformado. Os dependentes que, após o decesso do militar, venham a receber algum benefício remuneratório previsto na lei, são considerados pensionistas. A Diretoria de Civis, Inativos e Pensionistas (DCIPAS) é o Órgão de Direção Geral responsável por controlar os efetivos após o término do serviço ativo, bem como os dependentes nas situações descritas até então.

Conforme regulado pelo Estatuto dos Militares [7], e sintetizado nas normas regulatórias de concessão de benefícios de reserva remunerada, reforma e pensões [24], muitos documentos são necessários para a concessão de direitos e benefícios previstos em lei. Pode-se citar toda a documentação de um cidadão comum, como registro de identidade, CPF (Cadastro de Pessoas Físicas), documentação comprobatória de vínculo familiar (certidão de nascimento ou casamento), e comprovante de endereço residencial, como exemplos dos principais documentos. Contudo, a vida militar também utiliza outros comprovantes que gerarão direitos, e entre eles pode-se citar documentos de comprovação de cursos como paraquedismo, que gera direitos remuneratórios, e diversos outros de similar natureza.

Todos esses documentos, ao iniciar qualquer um dos processos de concessão de direitos citados - pensão, reserva ou reforma - são fundamentais para acesso aos benefícios. Por este motivo, a DCIPAS armazena em seu arquivo de registros funcionais as pastas com documentos de militares inativos, de modo que possa comprovar a validade do benefício percebido por cada militar ou dependente.

A geração das pastas de documentos dos militares não é tarefa da DCIPAS, pois é delegada a cada organização em que estão alocados os militares. Em caso de início de algum dos processos de benefício, a Organização Militar (OM) envia a pasta à DCIPAS, que confecciona o processo, e concede os benefícios, com a maior brevidade possível, tão logo tenha todos os documentos comprobatórios organizados. Toda a documentação gerada tramita via ofício, até os Comandos Militares, que enviam via serviço de correio para o Quartel General do Exército em Brasília-DF. Em algumas capitais também é possível utilizar serviços da Força Aérea para o transporte de cargas das Forças Armadas, mas em todos os casos a tramitação de documentos é relativamente lenta, e pode levar alguns dias entre a geração de documentos e sua entrega ao destinatário final.

O processo de concessão de benefícios possui alguns outros óbices, além da demora no processamento. Os casos mais comuns, para reserva e reforma, envolvem a ausência de documentos comprobatórios na entrada do processo, ou que um documento apresentado

no processo possua vícios, como erros de digitação, assinatura inválida, entre outros aspectos. Isso faz com que o beneficiário tenha que solicitar uma nova versão do documento, seja na Organização Militar em que está servindo, ou seja em outra OM em que serviu anteriormente e que registrou o direito adquirido.

Para a concessão de pensão, quando o militar está ausente por falecimento ou outras situações, o problema pode se agravar, pois é comum haver mais de um beneficiário com direito. Assim sendo, é necessário consolidar todos os documentos de todos os beneficiários logo no início do processo, como regra para a geração da remuneração a que têm direito. Ocorre frequentemente que, ao iniciar o processo, muitos beneficiários vivem em região diversa da OM que o prepara, e isto leva a um atraso significativo em sua solução. Há outras situações em que ocorre o ingresso de mais beneficiários no processo após seu início, levando a atrasos significativos e ao sofrimento dos beneficiários, que na maioria dos casos precisam da remuneração para seu próprio sustento.

A DCIPAS trata os processos tanto na execução quanto no planejamento, e mantém uma equipe que revisa e melhora, quando possível, a forma de cumprir seu papel. Para apoiar o trâmite documental, foi verificado que um sistema de registro de documentos poderia ser usado para encaminhar eletronicamente os documentos oficiais, e que, além de reduzir o volume de pastas arquivadas na Diretoria, permitiria também que as Organizações Militares entregassem e revisassem documentos muito rapidamente, favorecendo assim os usuários do sistema previdenciário do Exército. Sob esta ótica, foi projetado e desenvolvido o Sistema de Controle de Documentação Funcional, o Papiro, que permite a colaboração de cadastros documentais, e que será detalhado na próxima seção.

6.2 Papiro: Sistema de Controle de Documentação Funcional

O Papiro foi desenvolvido a partir de iniciativa da DCIPAS, e tem como objetivo consolidar e catalogar todos os documentos relativos ao ingresso de militares na inatividade, e de mesmo modo apoiar os dependentes que porventura recebam remuneração sob a mesma perspectiva de direitos a que fazem jus. O sistema permite a digitalização de documentos físicos, mas também pode receber documentos já digitalizados, e padroniza o formato PDF na entrada e no armazenamento.

Os documentos são catalogados conforme atributos do militar, por exemplo o número de registro de identidade, e armazena dados fundamentais em um banco de dados relacional, para facilitar a consulta e a pesquisa posterior. O sistema foi codificado em Java, usa um banco de dados PostgreSQL [27] versão 8.4, e o servidor de aplicação está na plataforma Tomcat 6.0 [21]. O sistema foi, inicialmente, desenvolvido para executar em

plataforma Linux, com Kernel 2.6 ou superior, tanto em versão 32 quanto em versão 64 bits. Os registros de pastas são realizados no banco de dados PostgreSQL, mas o armazenamento dos documentos é diretamente no sistema de arquivos, em uma estrutura de pastas organizada usando como nomes os *hashes* dos arquivos. Dessa forma, os dados do negócio do aplicativo estão divididos entre o banco de dados e o sistema de arquivos.

A aplicação está disponível na rede corporativa do Exército, a EBNet. Esta rede é privativa da Força Terrestre e alcança todas as Organizações Militares do território nacional, facilitando assim o acesso ao sistema. Contudo, algumas regiões tem flutuações de conexão à rede corporativa, e por esse motivo a experiência do usuário fica prejudicada.

O sistema Papiro foi desenvolvido entre os anos 2009 e 2012, e encontra-se hospedado na rede corporativa do Exército, em computadores do DGP. Não há planejamento previsto para revisão do sistema, quer seja por mudança de plataforma operacional, seja por revisão do código para evolução do sistema. Seu uso está limitado à hospedagem no DGP, principalmente, porque sua arquitetura interna não permite hospedar múltiplas bases de arquivos.

Os principais usuários do sistema estão espalhados nos oito Comandos Militares da Força Terrestre: Amazônia, Leste, Nordeste, Norte, Oeste, Planalto, Sudeste e Sul. A responsabilidade desses usuários fica por conta de consolidar os documentos necessários nos processos de concessão de benefícios. Embora fosse possível criar uma base de arquivos para cada Comando Militar, isso prejudicaria a posterior consulta e consolidação de documentos, pois como já foi explicado, alguns processos ocorrem em mais de uma região do país, com beneficiários de um mesmo processo residindo em locais diferentes. Na próxima seção será apresentada uma proposta de infraestrutura para o sistema Papiro, de modo que o armazenamento de seus arquivos possa ser transparentemente distribuído entre os Comandos Militares, e que as atualizações de clientes reflitam em todas as réplicas com a maior brevidade possível.

Pode-se atribuir grande importância do sistema Papiro para os processos do DGP e da DCIPAS, por manipular dados pessoais dos militares e de seus familiares para a concessão de benefícios e direitos previstos em lei. O arquivo documental da DCIPAS possui atualmente mais de 122 mil documentos digitalizados no Papiro, e os documentos em processo de digitalização passam de dois milhões, sendo 400 mil armazenados na DCIPAS. Para permitir o cadastro dos documentos no sistema, é fundamental dividir esforços com as Organizações Militares onde os demais documentos estão armazenados. Assim sendo, é de primordial importância permitir que o Papiro opere de forma distribuída e, sobretudo, em uma plataforma tolerante a falhas, para oferecer suporte ao sistema de pagamento de pensões do Exército. Nesse cenário, o sistema Papiro foi escolhido para testar a proposta de tolerância a falhas apresentadas neste trabalho, pois o resultado

beneficiará diretamente o Exército Brasileiro e os usuários de seu sistema previdenciário.

6.3 Considerações Finais

Neste capítulo foi explicado o funcionamento geral do sistema Papiro, sua importância para o Exército Brasileiro e os processos em que está envolvido. Foi esclarecido que há limitações em sua arquitetura que reduzem a eficiência de processos executados pela DCIPAS, sobretudo porque a grande extensão territorial muitas vezes exige a ação em diversos Comandos Militares. O sistema não possui equipe destinada à sua manutenção corretiva e evolutiva, o que por sua vez limita as opções de adaptação ao cenário atual.

No próximo capítulo é apresentada uma proposta de infraestrutura tolerante a falhas para o sistema Papiro, de modo que o sistema possa operar com cópias em todos os Comandos Militares. A plataforma StackSync se encarrega de, transparentemente, coordenar a cópia de arquivos entre os servidores do sistema Papiro, e assim permitir sua execução em múltiplos servidores. Esta abordagem visa oferecer um ambiente de execução do Papiro de forma distribuída, sem necessidade de adaptar o código da aplicação ao novo cenário.

Capítulo 7

Proposta de Tolerância a Falhas para Sistemas Legados

Neste capítulo são discutidas e apresentadas propostas de aprimorar o funcionamento do sistema Papiro, apresentado no Capítulo 6. Será apresentada uma infraestrutura de hospedagem que permitirá o funcionamento distribuído do sistema e tolerante a falhas, sem a modificação ou adaptação do seu código. Esta infraestrutura se dará por meio do *middleware* StackSync, para a sincronização da área de armazenamento dos documentos catalogados entre os servidores correspodentes a cada Comando Militar.

Para validar a proposta de infraestrutura, foi construído um ambiente de simulação, em que foi possível testar o resultado da aplicação do *middleware* StackSync na sincronização de arquivos do sistema proposto. Foram realizados seis testes no ambiente sobre as principais operações, com a carga de arquivos, e em seguida a inserção, modificação e remoção de arquivos A análise dos resultados dos testes será apresentada na Seção 7.2.

7.1 Proposta de Infraestrutura Tolerante a Falhas

Esta pesquisa foca nas características relacionadas ao armazenamento de arquivos, enquanto que os aspectos necessários para a replicação da aplicação e do banco de dados não serão aprofundados. Muito pode ser feito para tornar transparente a hospedagem distribuída da aplicação, por exemplo como apresentado nos trabalhos de Metz [39] e de Sarat et al. [42], enquanto que a replicação de banco de dados pode ser alcançada com uso das técnicas apresentadas por Thomson et al. [47] e por Chairunnanda et al. [9].

Para hospedar o sistema Papiro entre as regiões brasileiras, sugere-se usar o ambiente de sistemas corporativos presentes nos Comandos Militares, em uma infraestrutura que permita sincronizar as áreas de armazenamento de arquivos catalogados. A sincronização deve ocorrer o mais breve possível, tão logo alguma das réplicas receba atualizações de

clientes. Também foi verificado que o DGP deve manter uma réplica, considerando que parte dos processos ocorre no Departamento, e outra cópia deve ficar no CITEx, que é o órgão responsável pela infraestrutura do Exército. Assim sendo, a proposta de armazenamento ficou composta por dez réplicas, conforme exibido na Figura 7.1. O *middleware* StackSync fica responsável por coordenar a sincronização dos arquivos entre as réplicas, tanto recebendo atualizações em arquivos, quanto enviando as atualizações aos demais clientes do conjunto.

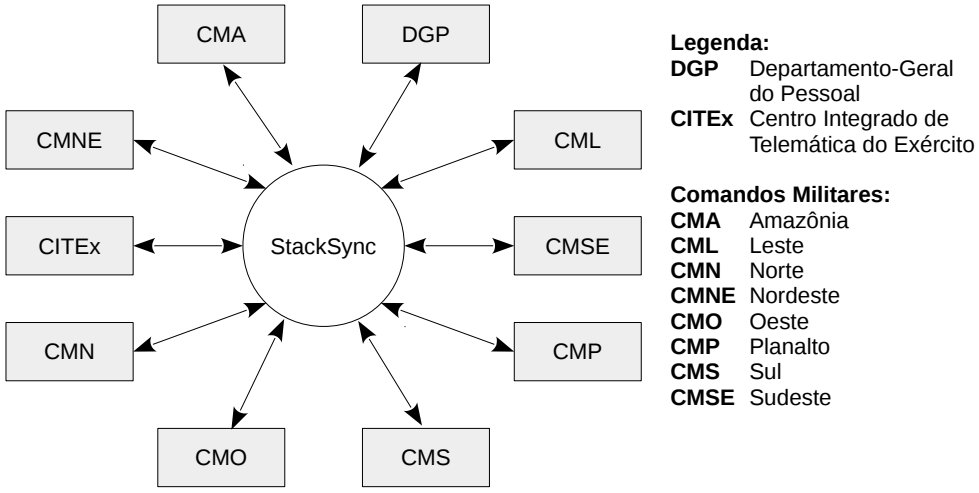


Figura 7.1: Arquitetura de Armazenamento Proposta para o Sistema Papiro.

Para implementar a tolerância a falhas utilizando a plataforma StackSync, propõe-se uma arquitetura de hospedagem inerentemente distribuída, ilustrada na Figura 7.2. Para atingir os objetivos desejados, a camada de infraestrutura compõe-se do *Middleware* StackSync, de um *cluster* da plataforma Swift e de uma plataforma de Metadados.

A arquitetura proposta indica o uso de dez servidores de aplicação, para atender os Comandos Militares e o DGP, com a presença de um servidor adicional de backup. O elemento Balanceador de Carga no topo da arquitetura tem como responsabilidade coordenar as requisições e sessões de clientes da aplicação hospedada, e monitorar a ocorrência de falhas e defeitos nos servidores de aplicação.

A plataforma StackSync permite a colaboração entre os clientes, e isso viabiliza a criação de mais de um servidor de aplicação com dados sincronizados pelo *middleware*. Cada servidor de aplicação precisa ter o cliente StackSync e uma área de dados sincronizada. O *middleware* oferece, nativamente, os meios para conectar aos *back-ends* de Armazenamento e de Metadados, que entregam os serviços conforme descritos no Capítulo 5.

A plataforma de *cluster* Swift tem como finalidade o armazenamento de objetos do StackSync, e garantir a disponibilidade desses dados mesmo sob a ocorrência de defeitos no ambiente. Para isso, distribui réplicas dos dados em diversos espaços de armazenamento,

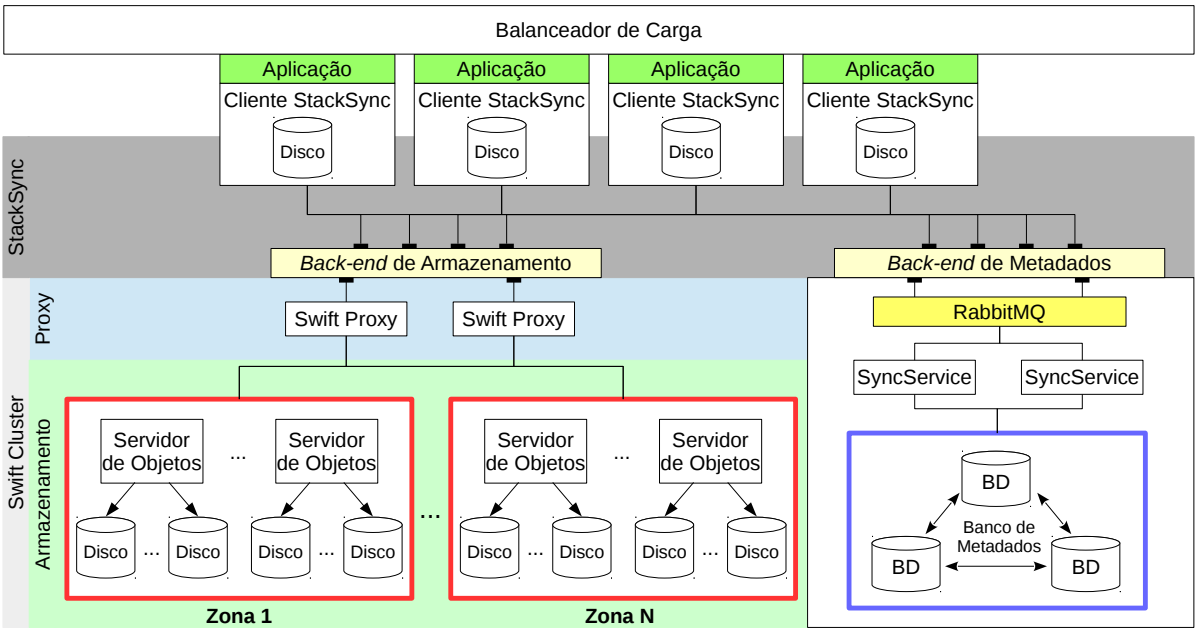


Figura 7.2: Arquitetura de Middleware com Tolerância a Falhas.

e coordena a recuperação desses dados por meio de um catálogo de objetos e permissões de acesso. Os elementos *proxy* utilizam os catálogos para avaliar se o cliente requisitante tem autorização para acessar os objetos, e podem operar em conjunto para dividir o processamento de requisições e tolerar falhas na camada de acesso. Os *proxies* comunicam-se com os servidores de objetos, que por sua vez conectam-se aos discos de armazenamento de objetos.

Como apresentado no Capítulo 5, a organização interna do OpenStack Swift permite o uso do conceito de zonas, o que proporciona a hospedagem de espaços de armazenamento geograficamente distantes, e de forma transparente. Os manuais de implantação da plataforma Swift [20] sugerem um mínimo de cinco zonas, e, face ao cenário em questão, propõe-se usar uma zona em cada *datacenter* destinado ao sistema Papiro. Cada zona da Figura 7.2 apresenta dois servidores de objetos, aos quais estão conectados dois ou mais discos, o que depende da demanda e da disponibilidade de hardware.

O *back-end* de Metadados da plataforma StackSync necessita do MOM RabbitMQ para permitir a comunicação assíncrona entre os clientes StackSync e o SyncService, que gerencia a sincronização entre os diversos clientes. O banco de dados que armazena os metadados da estrutura deve ser replicado em pelo menos duas cópias, para garantir a tolerância a falhas também neste componente. A arquitetura do SyncService permite a sua operação em conjuntos de dois ou mais serviços em execução, tornando-o também tolerante a falhas.

Com a arquitetura apresentada, é possível hospedar sistemas legados em uma forma de

operação distribuída, mas sem a necessidade de alterar o código fonte da aplicação legada. Este objetivo é particularmente importante para o sistema Papiro, conforme mencionado no Capítulo 6.

Para avaliar a proposta de uso do StackSync, foi criado um ambiente de simulação, permitindo assim a análise do consumo de recursos computacionais, o tempo de sincronização entre as réplicas do ambiente e o consumo do canal de comunicação, estas identificadas como as principais características de interesse para a implantação dessa infraestrutura. A seguir, são apresentadas as características físicas do ambiente de simulação.

7.1.1 Ambiente de Simulação

O ambiente de simulação foi hospedado em três servidores Dell Poweredge R900, com o seguinte perfil de hardware:

- 4 processadores Intel Xeon E7430 com quatro núcleos de 2.13GHz, totalizando 16 núcleos físicos;
- 128GB de memória principal DDR3 a 1333MHz;
- 8 portas de rede Gigabit Ethernet operando em conjunto;
- 4 portas de fibra ótica *Host Bus Adapter* (HBA) de 4GB.

As unidades de disco alocadas para os servidores foram disponibilizadas em um EMC Clariion CX4-240, um *storage* corporativo, com interfaces de comunicação por fibra ótica de 4GB. Nesse cenário, foram utilizados discos do *storage* do tipo FC de 15.000 RPM (rotações por minuto). Os servidores conectaram-se ao *storage* através de um *switch Fiber Channel* (FC), apropriado para redes de armazenamento em *storage* (SAN - *Storage Area Network*). Assim, todos os servidores Dell R900 puderam acessar os discos com o mesmo desempenho, sem prejudicar os testes.

Para utilizar adequadamente o hardware disponível, optou-se por operar em máquinas virtuais gerenciadas por um *Hypervisor* VMWare 4.1 [48]. O ambiente já é utilizado pelo DGP para seus sistemas corporativos, entretanto, durante o período de testes, o hardware ficou dedicado a este fim, sem concorrência com outros sistemas.

A plataforma StackSync, conforme citado no Capítulo 5, constitui-se do *back-end* de Armazenamento e do *back-end* de Metadados, que suportam o funcionamento da aplicação cliente. Para implementar a infraestrutura e sintetizar os resultados dos testes, foi criada uma máquina virtual com os serviços necessários, identificada por *StackSyncServer*.

O StackSyncServer encarrega-se de receber e processar requisições para a plataforma de armazenamento Swift, que abstrai a gravação de blocos de dados dos clientes StackSync,

conforme as atribuições do *back-end* de Armazenamento. A plataforma Swift foi montada sobre quatro discos distintos, de acordo com as recomendações do desenvolvedor do recurso [20], e assegura pelo menos três réplicas de todos os dados armazenados. As cópias são distribuídas pela plataforma, automaticamente, entre os discos "Node101" a "Node104". Para o *back-end* de Armazenamento, também é necessário o serviço de autenticação OpenStack Keystone, que utilizou um banco de dados relacional MySQL para o controle de acesso aos objetos armazenados.

O *back-end* de Metadados, também configurado na máquina virtual StackSyncServer, corresponde ao MOM RabbitMQ e ao serviço SyncService, o qual utiliza um banco de dados relacional PostgreSQL para armazenar os metadados de arquivos gerenciados pelo *middleware* StackSync. Toda a comunicação envolvendo metadados ocorre por meio de troca de mensagens usando o MOM RabbitMQ, no protocolo de comunicação descrito no Capítulo 5.

Os servidores de aplicação, que armazenam os dados do sistema Papiro, foram resumidos nos testes de forma a implementar somente as características de armazenamento de dados. Essa simplificação foi importante para que o processamento do negócio e das operações de usuário não criassem deturpações nas leituras de consumo de recursos computacionais. Assim sendo, os servidores de aplicação apresentam somente o serviço de cliente StackSync e pacotes necessários para a comunicação de dados, e o funcionamento do sistema operacional básico.

Cada máquina virtual citada constituiu-se de oito núcleos de processamento, e 12GB de memória principal, bem como interface Gigabit Ethernet de comunicação e 100GB em disco rígido, armazenados no *storage*. O hardware virtual foi alocado de modo a não representar gargalos para a execução dos testes.

7.1.2 Dados Usados na Simulação

Na primeira análise dos dados, realizada em 22 de outubro de 2014, o sistema possuía 72.400 arquivos catalogados. Destes arquivos, o volume de dados correspondia a aproximadamente 7,8GB, com arquivos variando do tamanho de 88KB a até 1544KB. O tamanho médio dos arquivos era de 108KB.

Em uma segunda análise, realizada em 18 de março de 2015, o sistema possuía 122.427 arquivos catalogados. Destes arquivos, o volume de dados correspondia a aproximadamente 16,3GB de dados, com arquivos variando de 82KB a até 1620KB de tamanho. O tamanho médio dos arquivos era de 133KB.

As análises trouxeram informações acerca do perfil de arquivos armazenados, o que delimitou alguns parâmetros utilizados na pesquisa. Com base nos arquivos do ambiente de produção do sistema Papiro, foi criado um conjunto de dez mil arquivos, com tamanho

médio de 90KB, totalizando 903MB de dados. Para os testes envolvendo cinco mil arquivos, foram usados 452MB de dados. Esses dados foram utilizados nos testes que serão descritos e analisados na próxima seção.

7.2 Testes no Ambiente

Para testar a eficiência da plataforma StackSync com relação ao funcionamento do sistema Papiro, foram realizados testes que mediram o consumo de CPU, o uso de memória principal, a entrada e a saída dos discos, e o consumo do canal de comunicação de rede. As medições foram realizadas durante a execução das operações em arquivos até o StackSync confirmar a completa sincronização, por meio de seus registros de *log*.

As operações analisadas foram de adicionar, modificar e remover arquivos. Estas operações sobrepõem quase todas as possibilidades de movimentação de dados. Não foram realizados testes de carga sobre operações em atributos de arquivos, nem de movimentação de arquivos entre diretórios, pois são ações que ge rariam somente desdobramentos nos metadados. Durante os testes, não foram identificados cenários em que ocorresse outro tipo de modificação nos arquivos do sistema Papiro, além das operações já descritas. Nas próximas subseções serão detalhados os testes realizados no ambiente, bem como os resultados obtidos.

7.2.1 Teste 1: Carga de 5.000 Arquivos

O primeiro teste baseou-se na carga inicial de arquivos, com os dados disponíveis no cliente 01 - CITEx, que foram replicados aos outros clientes, um de cada vez. O total de dados em arquivos correspondeu a 452MB, distribuídos em cinco mil arquivos, com tamanho médio de 90KB por arquivo. O intervalo de acionamento de cada cliente foi de seis minutos, embora o tempo de sincronização tenha sido mais breve, conforme apresentado na Tabela 7.1. O tempo total da simulação foi de 60 minutos. Cada cliente foi identificado pelo Comando Militar correspondente, além do DGP e do CITEx, totalizando dez clientes. O tempo de sincronização exibido corresponde ao tempo total, desde a inicialização do aplicativo cliente, até a mensagem do StackSync afirmando que todos os dados estão sincronizados.

O consumo de recursos computacionais pode ser observado nas Figuras 7.3 até 7.14. Os recursos analisados compreendem o consumo de CPU, separado entre a aplicação StackSync e os serviços do sistema operacional, o consumo de memória, o total de dados gravados em disco, o total de dados enviados e o total de dados recebidos pela interface de rede. Todos esses dados foram observados nos clientes StackSync e na máquina virtual que hospedava o *middleware* descrito na Subseção 7.1.1.

Tabela 7.1: Tempo de Execução e Sincronização do Teste 1.

Cliente	Início	Tempo de Sincronização
CITEx	00:00	03:55
CMA	06:00	03:34
CML	12:00	03:16
CMN	18:00	03:13
CMNE	24:00	03:20
CMO	30:00	03:22
CMP	36:00	03:18
CMS	42:00	03:16
CMSE	48:00	03:20
DGP	54:00	03:26

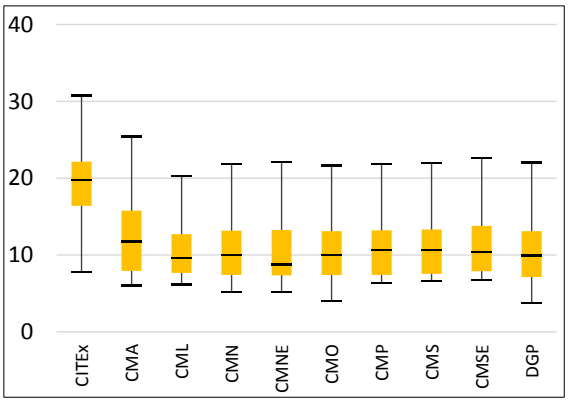


Figura 7.3: Consumo de CPU (%) pelos Clientes no Teste 1.

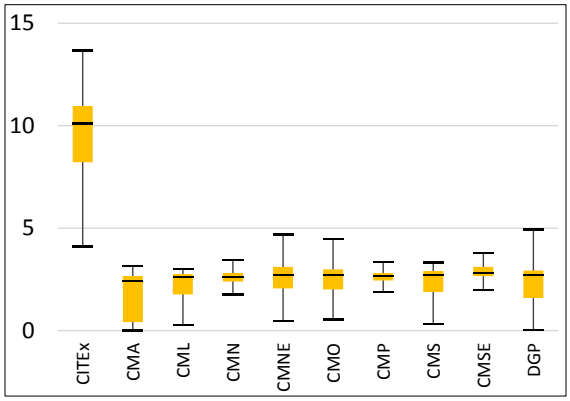


Figura 7.4: Consumo de CPU (%) pela Plataforma StackSync no Teste 1.

Conforme a Figura 7.3, pode-se visualizar o consumo de CPU em porcentagem, na escala de 0 a 40% da capacidade de processamento disponível, e cada coluna do gráfico indica um cliente, identificado pela sigla correspondente. Cada coluna do gráfico indica o valor mínimo, que é a menor leitura de consumo de processamento, representada pela barra mais abaixo na coluna; o valor de mediana, que divide as leituras em duas metades e apresenta a mediana como valor do meio, representada pela barra do meio da coluna; e o valor máximo, que é a maior leitura de consumo de CPU para o teste, por cliente, que é a barra superior de cada coluna.

A faixa amarela determina o segundo e o terceiro quartis de leituras, ou seja, ao se dividir o conjunto de leituras em quatro, o primeiro quartil fica entre o valor mínimo e o limite inferior da faixa amarela. O segundo quartil inicia onde termina o primeiro, e encerra na mediana. O terceiro quartil começa na mediana e estende-se até o limite superior da faixa amarela. Por fim, o quarto quartil começa no limite superior da faixa amarela e termina no valor máximo, representado pela barra superior.

Durante o Teste 1, ainda conforme o que está apresentado na Figura 7.3, os Clientes StackSync consumiram, no máximo, 30% de CPU. O primeiro cliente, denominado CITEx, apresentou mediana de consumo de CPU em 20%. Os demais clientes utilizaram menos processamento, com mediana em aproximadamente 10%, e picos de até 20%.

Como pode ser visto na Figura 7.4, o eixo vertical representa o consumo de CPU pela plataforma StackSync durante o Teste 1, variando de 0 a 15% da capacidade de processamento. O consumo de CPU pela plataforma StackSync é maior durante a inicialização do primeiro cliente, que cataloga os metadados. A partir do segundo cliente, o consumo de CPU pelo servidor StackSync cai significativamente. A mediana durante a inicialização do primeiro cliente é de 10% da capacidade de processamento, com picos de até 15%, e para os demais a mediana ficou em cerca de 3%, com picos abaixo de 5% de consumo.

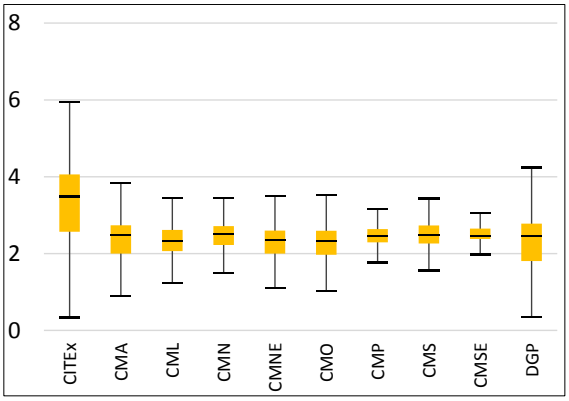


Figura 7.5: Consumo de CPU (%) pelo S. O. dos Clientes no Teste 1.

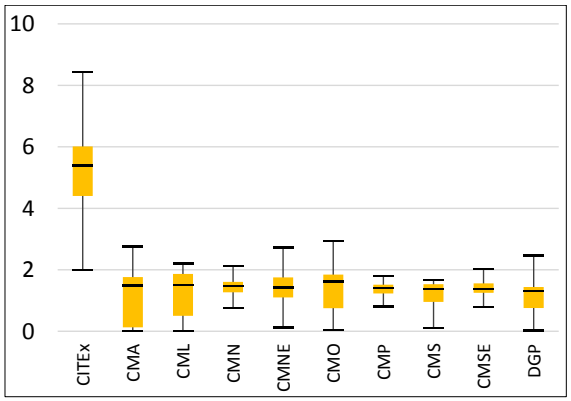


Figura 7.6: Consumo de CPU (%) pelo S. O. da Plataforma StackSync no Teste 1.

Na Figura 7.5, o eixo vertical representa o consumo de CPU em porcentagem, e cada coluna representa um cliente do conjunto. O consumo de CPU pelo sistema operacional durante a inicialização do primeiro cliente (coluna CITEx) teve mediana em 4%, com picos de processamento em até 6%, e para os demais clientes foi de 2%, com picos de 4%. Em relação ao consumo de processamento pelo sistema operacional da plataforma StackSync, apresentado na Figura 7.6, a variação foi de 0 a 10% da capacidade de processamento. Durante a inicialização do primeiro cliente o consumo foi maior, com mediana em quase 6% e picos de 8% de consumo, e durante o ingresso dos demais clientes ficou abaixo de 2%, e máximo de consumo abaixo de 3%.

O consumo de memória principal foi estável durante o Teste 1, como exibido nas Figuras 7.7. O eixo vertical representa o total de memória consumida em megabytes, variando de 0 a 4GB, enquanto que cada coluna representa o consumo de um cliente do conjunto. O cliente CITEx utilizou mais memória, em virtude de inicializar o processo com todos os arquivos e enviá-los à plataforma StackSync. Para esta tarefa, utilizou a

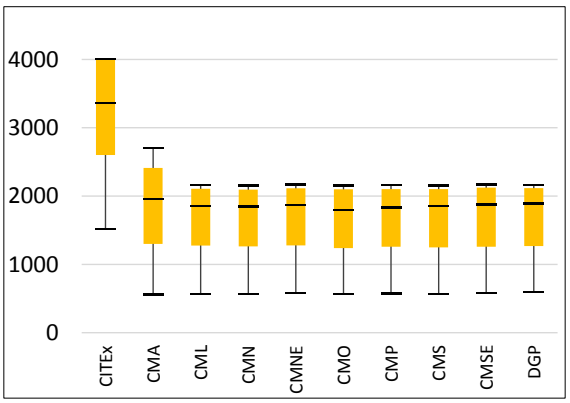


Figura 7.7: Consumo de Memória (MB) pelos Clientes no Teste 1.

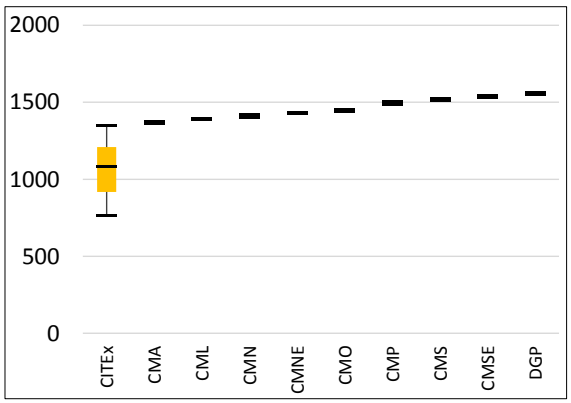


Figura 7.8: Consumo de Memória (MB) pela Plataforma StackSync no Teste 1.

mediana de 3,2GB de memória, com máximo de consumo de 4GB. O consumo de memória pelos outros clientes teve mediana de 2GB, com valor máximo abaixo de 2,5GB.

Com relação à plataforma StackSync, conforme exibido na Figura 7.8, o eixo vertical representa o consumo de memória pela plataforma StackSync em megabytes, variando de 0 a 2GB, e as colunas indicam o consumo de memória durante a inicialização de cada cliente. O consumo de memória foi aumentando conforme os clientes entraram no conjunto. O primeiro cliente apresentou mediana de 1GB de memória utilizada, com máximo em 1,3GB. Após o ingresso dos outros nove clientes, o consumo de memória chegou a 1,5GB, ou seja, 15% mais memória do que ao incluir apenas 1 cliente. Os valores de mínimo, mediana e máximo, durante a inicialização dos clientes 2 a 10, variou menos de 1% para cada um deles.

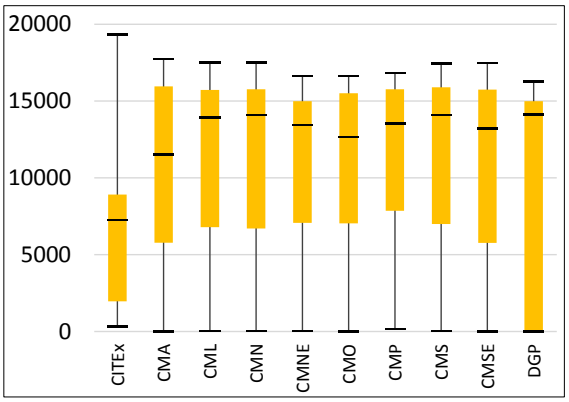


Figura 7.9: Gravação de Blocos em Disco pelos Clientes no Teste 1.

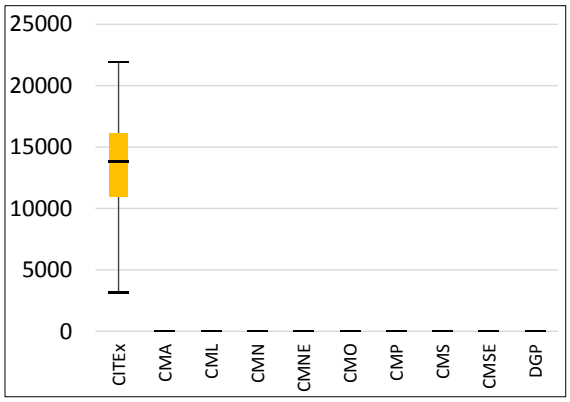


Figura 7.10: Gravação de Blocos em Disco pela Plataforma StackSync no Teste 1.

A gravação em disco foi estável nos clientes durante a inicialização do ambiente StackSync, como exibido na Figura 7.9, no qual o eixo vertical indica o total de blo-

cos de 512 bytes gravados por segundo durante a inicialização dos clientes, variando de 0 a 20.000 blocos, e as colunas apresentam o total de blocos gravados por cliente na inicialização. O primeiro cliente apresentou picos de até 20.000 blocos de gravação por segundo, e mediana em 7.000 blocos. Os clientes restantes tiveram a gravação com picos de até 17.000 blocos, e mediana de 13.000 blocos. Deve-se ressaltar que a gravação em disco correspondeu não só às operações em arquivos no armazenamento compartilhado, mas também em registros de controle dos arquivos locais, como parte do funcionamento do aplicativo cliente.

Na Figura 7.10, o eixo vertical indica o total de blocos de 512 bytes gravados por segundo, variando de 0 a 25.000 blocos, e cada coluna representa a quantidade de blocos gravados por cliente. A gravação concentrou-se durante a inicialização do cliente CITEx, em particular porque foi o cliente que inseriu dados no armazenamento compartilhado, e que gerou picos de até 22.000 blocos gravados por segundo, com mediana em 14.000 blocos. Os demais clientes somente leram tais dados, então, ao ingressarem no conjunto, a plataforma StackSync não precisou gravar dados em disco, mantendo a gravação abaixo de 50 blocos por segundo. O volume de gravações na plataforma foi compatível com o volume de operações nos clientes. Também é importante citar que a gravação em disco na plataforma englobou todos os serviços hospedados, desde o armazenamento, como também o banco de dados do *back-end* de Metadados, e por esse motivo era previsto que o volume de gravações da plataforma fosse maior do que no lado cliente.

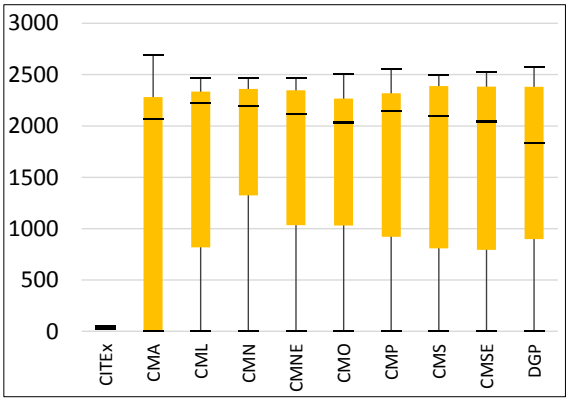


Figura 7.11: Dados Recebidos (KB) pelos Clientes StackSync no Teste 1.

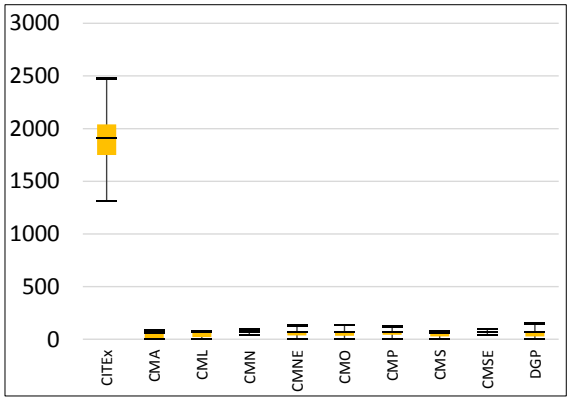


Figura 7.12: Dados Recebidos (KB) pela Plataforma StackSync no Teste 1.

Ao passo que o primeiro cliente, o qual inicializou o ambiente de armazenamento enviando arquivos, não deveria receber dados durante esse processo, os outros clientes precisariam receber os dados. Isso está apresentado na Figura 7.11, na qual o eixo vertical representa o total em kilobytes de dados recebidos, e as colunas representam os dados recebidos por cada cliente. O primeiro cliente recebeu menos do que 1% do total de dados

recebidos pelos demais clientes, relacionados ao controle dos metadados e de gerência da plataforma. Os clientes restantes receberam todos os arquivos por rede, com a mediana da velocidade de transmissão em cerca de 2MB por segundo, e picos de até 2,6MB por segundo. O consumo do canal de comunicação foi estável, pois os picos de transmissão não alcançaram 3MB por segundo, principalmente em virtude da taxa de transferência (*throughput*) oferecida pela plataforma.

A Figura 7.12 mostra a recepção de dados por rede pela plataforma, no qual o eixo vertical representa o total de dados recebidos em kilobytes, e cada coluna indica os dados recebidos pela plataforma durante a inicialização dos clientes. A inicialização do cliente CITEx gerou um tráfego com mediana de 1,9MB por segundo, com picos de até 1,5MB por segundo. A recepção ficou compatível com o *throughput* da plataforma e com a velocidade apresentada pelos clientes. Durante a inicialização do cliente CITEx, a plataforma recebeu todos os dados dos arquivos, e nos demais clientes os dados foram enviados a eles. A recepção de dados, nos outros casos, ficou por conta de comandos relacionados aos metadados e a comunicação da própria plataforma, consumindo menos do que 250KB do canal de comunicação para isso.

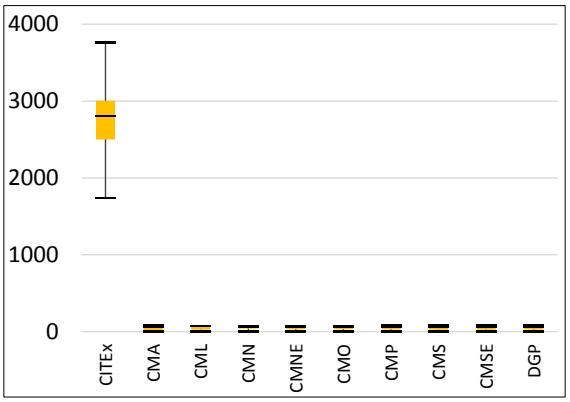


Figura 7.13: Dados Transmitidos (KB) pelos Clientes no Teste 1.

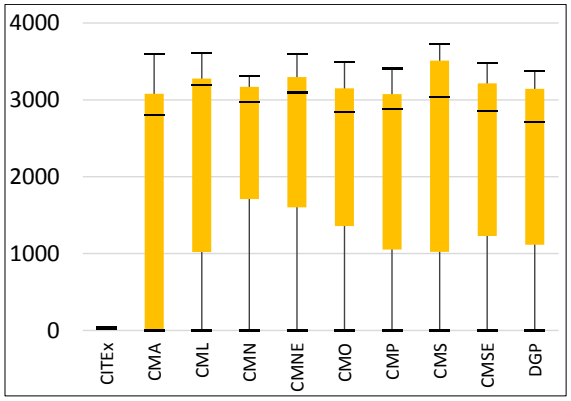


Figura 7.14: Dados Transmitidos (KB) pela Plataforma StackSync no Teste 1.

Durante a inicialização do ambiente pelo cliente CITEx, foi necessário transferir os arquivos para a plataforma StackSync, como apresentado na Figura 7.13. O eixo vertical indica o total de dados transmitidos em rede, e cada coluna demonstra os dados transmitidos pelos clientes. O cliente CITEx transmitiu dados com picos de até 3,8MB por segundo, e mediana em 2,8MB por segundo. Os clientes restantes receberam dados, por isso para eles a taxa de transmissão ficou baixa, representando somente a comunicação de controle da plataforma e de metadados, com picos abaixo de 100KB por segundo.

Na Figura 7.14, o eixo vertical indica o total de dados transmitidos pela plataforma StackSync em kilobytes, e as colunas apresentam os dados transmitidos durante a ini-

Tabela 7.2: Tempo de Execução e Sincronização do Teste 2.

Cliente	**Início**	**Tempo de Sincronização (min)**
CITEx	00:00	08:47
CMA	00:10:00	09:08
CML	00:20:00	08:57
CMN	00:30:00	09:03
CMNE	00:40:00	09:28
CMO	00:50:00	09:07
CMP	01:00:00	09:22
CMS	01:10:00	09:33
CMSE	01:20:00	09:37
DGP	01:30:00	09:12

cialização de cada cliente. A plataforma transmitiu dados principalmente a partir da inicialização do segundo cliente, já que no primeiro foi necessário receber os arquivos, ou seja, na inicialização do primeiro cliente a transmissão de dados foi mínima. A partir do segundo cliente, a transmissão de dados teve mediana em cerca de 3MB de dados por segundo, mas de forma estável, com picos de transmissão abaixo de 4MB por segundo.

7.2.2 Teste 2: Carga de 10.000 Arquivos

O segundo teste foi realizado com a carga inicial de arquivos no cliente 01 - CITEx, que foram depois replicados aos demais clientes, um de cada vez. O total de dados em arquivos correspondeu a 903MB, distribuídos em dez mil arquivos, com tamanho médio de 92KB por arquivo. O intervalo de acionamento de cada cliente foi de dez minutos, embora o tempo de sincronização tenha sido mais breve, conforme apresentado na Tabela 7.2, da mesma forma como analisado no Teste 1. O tempo de sincronização dos clientes foi sinalizado pelo *log* da aplicação cliente StackSync. O tempo total da simulação foi de 100 minutos.

O consumo de recursos computacionais pode ser observado nas Figuras 7.15 até 7.26. As medições seguiram a mesma metodologia adotada para o Teste 1, descrito na Subseção 7.2.1.

O consumo de CPU pelos clientes StackSync no Teste 2 foi estável, apresentado na Figura 7.15. O gráfico apresenta, no eixo vertical, o consumo de processamento em porcentagem, variando de 0 a 40%, e cada coluna indica o processamento consumido por cliente, em sua respectiva inicialização. O cliente CITEx apresentou picos de processamento de até 33% de CPU, com mediana em 23%. Os outros clientes tiveram picos de processamento de até 20% de CPU, enquanto que a mediana de consumo foi de 10%. O primeiro

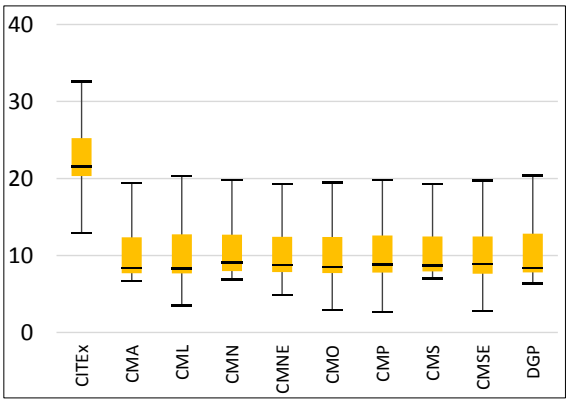


Figura 7.15: Consumo de CPU (%) pelos Clientes o Teste 2.

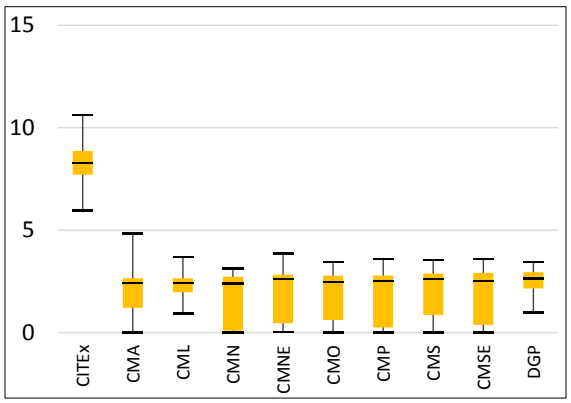


Figura 7.16: Consumo de CPU (%) pela Plataforma StackSync o Teste 2.

cliente inicializou o ambiente de armazenamento, enviando arquivos à plataforma, e por este motivo consumiu mais recursos computacionais.

A plataforma StackSync utilizou menos CPU para do que os clientes seus processos, visualizado na Figura 7.16. O gráfico indica, no eixo vertical, o consumo de CPU em porcentagem, variando de 0 a 15%, e cada coluna representa o consumo durante a inicialização dos clientes no teste. A inicialização do cliente CITEx exigiu da plataforma StackSync picos de até 11% de CPU, com mediana em 8%. A inicialização dos demais clientes exigiu menos recursos, com picos de no máximo 5% de processamento e mediana de consumo em 2%. Assim sendo, fica claro que a inicialização de dados no ambiente exige mais processamento do que a sincronia nos outros clientes do conjunto.

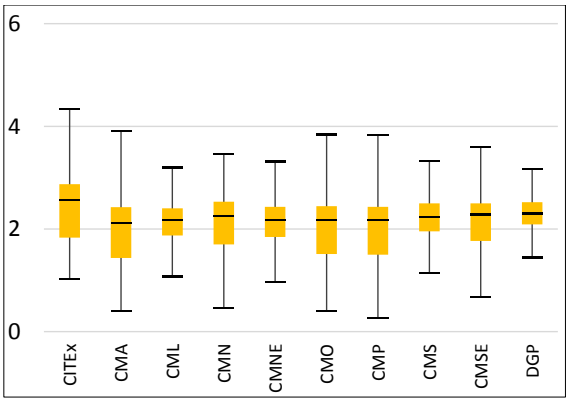


Figura 7.17: Consumo de CPU (%) pelo S. O. dos Clientes no Teste 2.

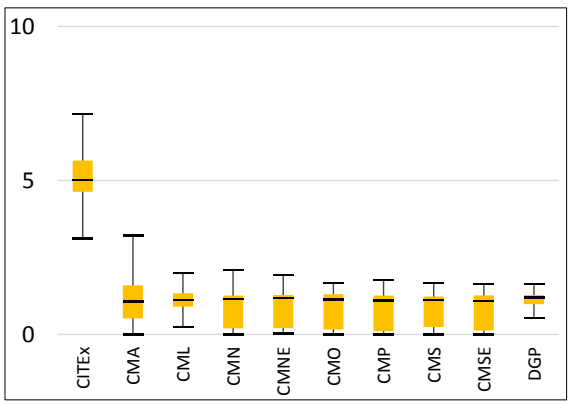


Figura 7.18: Consumo de CPU (%) pelo S. O. da Plataforma StackSync no Teste 2.

Na Figura 7.17 esclarece-se o consumo de CPU pelo sistema operacional dos clientes StackSync, na qual o eixo vertical indica o consumo de processamento em porcentagem, variando de 0 a 6%, e as colunas indicam o total consumido por cada cliente durante sua

inicialização. O consumo máximo de CPU pelos sistemas operacionais dos clientes não passou de 5%, e a mediana ficou abaixo de 3%. O comportamento foi similar em todos os clientes, com a variação entre valor mínimo e valor máximo abaixo de 3%.

A Figura 7.18 apresenta o consumo de CPU do sistema operacional da plataforma StackSync, na qual o eixo vertical indica o consumo de CPU em porcentagem, e as colunas representam o total consumido durante a inicialização de cada cliente do conjunto. A inicialização do cliente CITEx apresentou consumo maior de CPU pelo sistema operacional da plataforma, em particular porque está contido neste processamento as operações em bancos de dados. O consumo máximo para o primeiro cliente foi de 7%, com mediana em 5%, e durante a inicialização dos demais clientes o consumo máximo de CPU foi de 3%, com mediana de 1%.

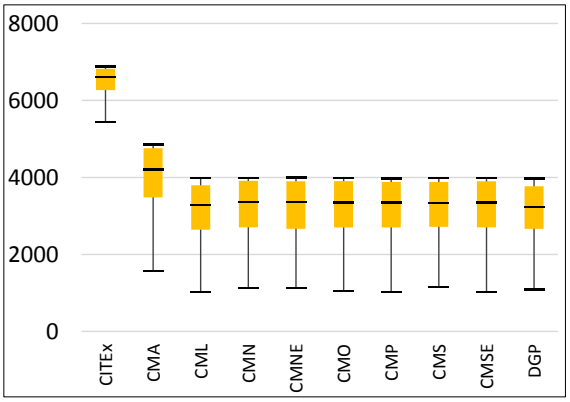


Figura 7.19: Consumo de Memória (MB) pelos Clientes no Teste 2.

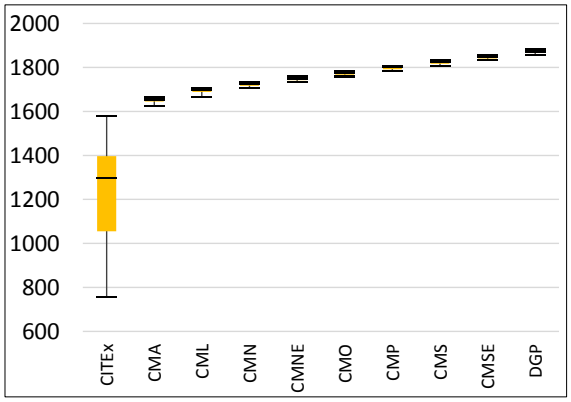


Figura 7.20: Consumo de Memória (MB) pela Plataforma StackSync no Teste 2.

De modo similar ao ocorrido no Teste 1, a Figura 7.19 mostra o consumo de memória, com o eixo vertical em megabytes, variando de 0 a 8GB, e cada coluna indicando o consumo de memória dos clientes na respectiva inicialização. O cliente CITEx inicializou o ambiente com um consumo de até o máximo de 7GB, com mediana em 6,8GB. Os outros clientes iniciaram o consumo em cerca de 2GB de memória, e durante a execução do teste o limite de consumo foi de 5GB no segundo cliente, e de 4GB nos demais. Essa diferença ocorreu porque o sistema de sincronização exigiu mais recursos do primeiro cliente, que enviou os arquivos para a plataforma, e precisou de menos memória para os clientes seguintes, os quais só receberam os arquivos para sincronização.

O consumo de memória pela plataforma StackSync pode ser observado na Figura 7.20. O eixo vertical indica o total de memória, em megabytes, variando de 0 a 2GB, e cada coluna representa o consumo de memória pela plataforma durante a inicialização dos respectivos clientes. A inicialização do primeiro cliente começou com a necessidade de 800MB de memória, e progrediu até o máximo de 1,6GB de memória. Para o ingresso

dos clientes seguintes, o consumo foi mais estável, variando menos do que 50MB por cliente entre consumo mínimo e máximo, até atingir um valor de 1,9GB no término da inicialização do último cliente.

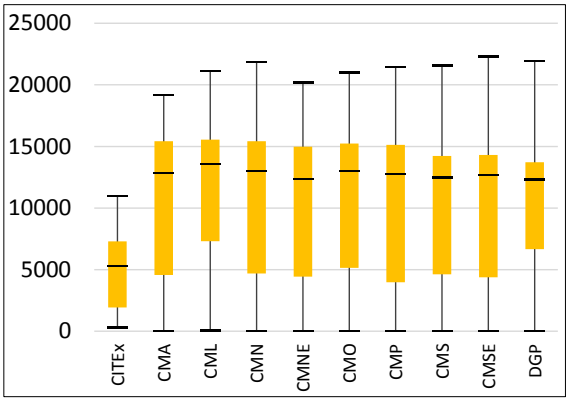


Figura 7.21: Gravação de Blocos em Disco pelos Clientes no Teste 2.

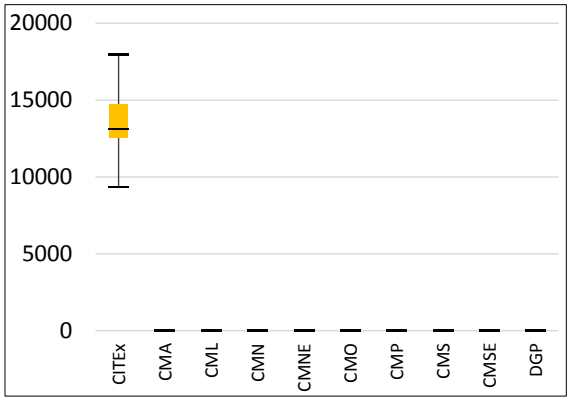


Figura 7.22: Gravação de Blocos em Disco pela Plataforma StackSync no Teste 2.

De modo similar ao visualizado no Teste 1, a gravação em disco apresentada na Figura 7.21 indica os blocos de 512 bytes gravados em disco a cada segundo, variando de 0 a 25 mil blocos, e cada coluna representa o total de blocos gravados pelos clientes durante o Teste 2. A inicialização do primeiro cliente exigiu a gravação de até 11.000 blocos em disco por segundo, com mediana em 5.000 blocos por segundo. Os demais clientes gravaram picos de até 23.000 blocos por segundo, com mediana em 14.000 blocos gravados por segundo. O primeiro cliente apresentou gravação inferior aos outros, sobretudo porque seu papel foi de carregar os arquivos na plataforma, e os outros clientes precisaram recuperar os dados da plataforma e gravar localmente.

A Figura 7.22 apresenta a gravação em disco na plataforma StackSync durante o Teste 2, na qual o eixo vertical indica o total de blocos de 512 bytes gravados por segundo, variando de 0 a 20 mil blocos, e cada coluna representa o total gravado durante a inicialização dos respectivos clientes. A gravação durante a inicialização do cliente CITEx, em que se inseriu dados na plataforma, demonstra picos de até 18.000 blocos gravados por segundo, com mediana em 13.000 blocos por segundo. A inicialização dos clientes seguintes não apresentou gravações significativas, ficando abaixo de 50 blocos gravados por segundo, em especial porque os dados foram recebidos no início e não houveram modificações de dados durante o Teste 2.

A Figura 7.23 apresenta o volume de dados recebidos através da rede pelos clientes StackSync, sendo que o eixo vertical apresenta o volume de dados trafegados por segundo em kilobytes, variando de 0 a 3MB por segundo, e as colunas informam o total trafegado por cliente durante o Teste 2. Como o cliente CITEx enviou dados para a plataforma

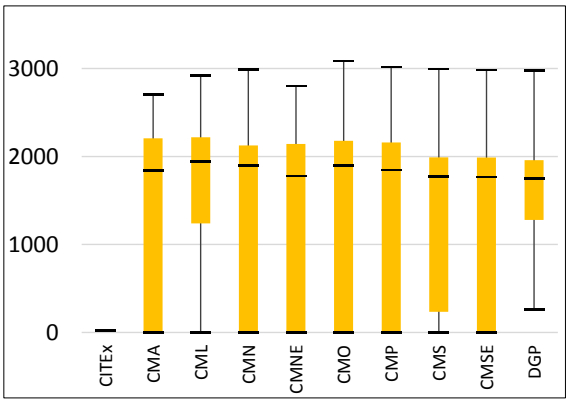


Figura 7.23: Dados Recebidos (KB) pelos Clientes no Teste 2.

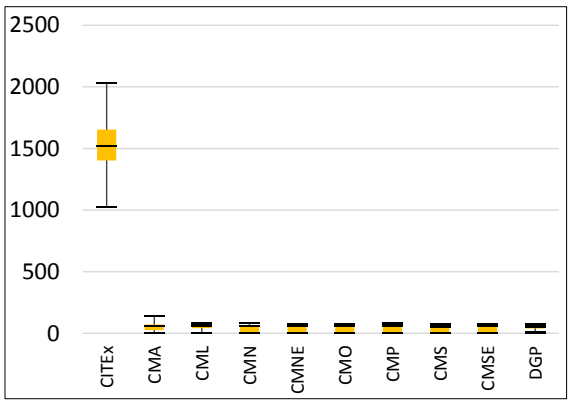


Figura 7.24: Dados Recebidos (KB) pela Plataforma StackSync no Teste 2.

StackSync neste teste, o recebimento de dados foi reduzido, com máximo em 30KB por segundo. Os outros clientes receberam todos os dados de sincronização, e na taxa máxima de 3MB por segundo, com mediana em 2MB por segundo de transferência.

A Figura 7.24 apresenta a recepção de dados através da rede pela plataforma StackSync, na qual o eixo vertical apresenta o total de dados recebidos em kilobytes, variando de 0 a 2,5MB, e cada coluna apresenta os dados recebidos pela plataforma durante a inicialização dos respectivos clientes. Ao inicializar o cliente CITEx, a plataforma recebeu dados com picos de transferência de até 2MB por segundo, e mediana em 1,5MB por segundo. Os demais clientes, ao ingressarem no conjunto, trocaram apenas informações de controle de metadados com a plataforma, o que gerou tráfego de entrada de no máximo 100KB por segundo na plataforma StackSync.

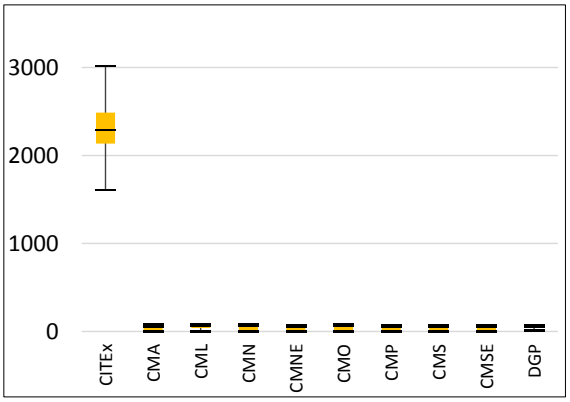


Figura 7.25: Dados Transmitidos (KB) pelos Clientes no Teste 2.

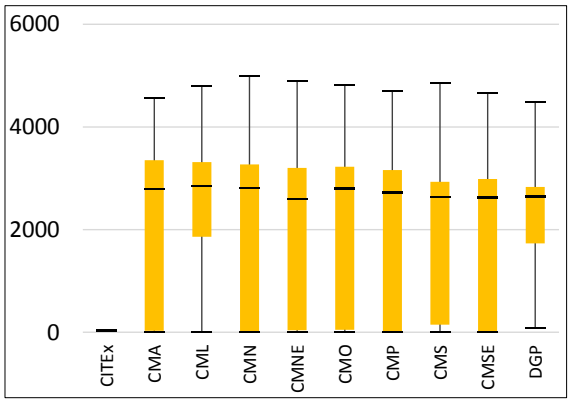


Figura 7.26: Dados Transmitidos (KB) pela Plataforma StackSync no Teste 2.

A Figura 7.25 mostra os dados transmitidos pelos clientes StackSync durante o Teste 2, na qual o eixo vertical apresenta o total de dados transmitidos em kilobytes, variando

Tabela 7.3: Tarefas Executadas no Teste 3.

Tarefa	Ação	Bloco	Cliente
1	Adicionar	b11	CITEx
2	Modificar	b1	CMA
3	Remover	b2	CML
4	Adicionar	b12	CMN
5	Modificar	b3	CMNE
6	Remover	b4	CMO
7	Adicionar	b13	CMP
8	Modificar	b5	CMS
9	Remover	b6	CMSE
10	Adicionar	b14	DGP
11	Modificar	b7	CITEx
12	Remover	b8	CMA
13	Adicionar	b15	CML
14	Modificar	b9	CMN
15	Remover	b10	CMNE

de 0 a 3MB por segundo, e as colunas indicam o envio de dados através da rede durante a inicialização dos clientes. O cliente CITEx enviou os dados dos arquivos à plataforma, e para isso enviou à uma taxa de até 3MB por segundo de dados, com mediana de 2,3MB por segundo de transmissão. Os outros clientes somente trocaram informações de controle e de metadados com a plataforma, a uma taxa máxima de 100KB por segundo de transmissão.

A Figura 7.26 exibe os dados transmitidos pela plataforma StackSync durante o Teste 2, sendo que o eixo vertical representa o total de dados transmitidos, em kilobytes, variando de 0 a 6MB por segundo, e cada coluna representa o total de dados transmitidos pela plataforma durante a inicialização dos respectivos clientes. A inicialização do cliente CITEx apresentou somente a transmissão de dados de controle, que ficou abaixo de 30KB por segundo, e para os demais clientes a transmissão atingiu picos de 5MB por segundo, com mediana em 2,5MB por segundo.

7.2.3 Teste 3: Simulação do Ambiente de Produção

A simulação do ambiente de produção se deu por meio de tarefas executadas consecutivamente, de modo que fosse possível verificar a estabilidade e o desempenho da plataforma. Para isto, o ambiente contou com todos os dez clientes sincronizados, em decorrência do término da execução do Teste 2. Cada cliente possuía, no início do teste, dez mil arquivos, totalizando 903MB de dados. Cada operação manipulou mil arquivos por vez, e os arquivos foram sistematicamente separados em blocos, de modo que as operações não se sobrepusessem antes do término da tarefa anterior nos mesmos arquivos.

Tabela 7.4: Tempo de Execução e Sincronização do Teste 3.

Cliente	**Tempo de Sincronização (min)**
CITEx	19:31
CMA	19:37
CML	19:36
CMN	19:52
CMNE	19:37
CMO	19:41
CMP	19:41
CMS	19:46
CMSE	19:26
DGP	19:42

Ao todo, foram executadas quinze tarefas de acréscimo de arquivos, modificação de conteúdo e remoção de arquivos, em intervalos de um minuto, conforme a sequência descrita na Tabela 7.3. Os blocos de arquivos foram numerados de b1 a b15, dos quais de b1 a b10 foram os arquivos que iniciaram sincronizados no teste, e de b11 a b15 foram blocos de arquivos novos. A coluna Cliente indica em qual dos clientes a tarefa foi executada.

Ao término da execução de todas as tarefas, os clientes possuíam dez mil arquivos cada. Foi medido o tempo que cada cliente StackSync levou para completar a sincronização dos arquivos, apresentados na Tabela 7.4, considerando que as tarefas foram executadas nos primeiros quinze minutos. As tarefas não foram avaliadas individualmente, pois a sincronização ocorre como um todo, e a plataforma não permite discriminar o sucesso de cada operação, em parte pelo mecanismo proporcionado pelo MOM e as filas de mensagens. A simulação ocorreu no tempo total de 30 minutos, embora as leituras tenham sido baseadas no tempo de sincronização de cada cliente, em que o último ocorreu ao tempo de 19:42 minutos.

O consumo de recursos no Teste 3 pode ser visualizado nas Figuras 7.27 até 7.32. A metodologia de avaliação segue a mesma adotada nos Testes 1 e 2.

Durante o Teste 3, o consumo de CPU foi estável, como apresentado na Figura 7.27, em que o eixo vertical indica o consumo de CPU em porcentagem, e as colunas apresentam o consumo por cliente. Os picos de processamento em clientes ficaram em 30% do processamento disponível, com mediana em 15%. A plataforma consumiu picos de menos de 20% e mediana em 7%.

O consumo de processador pelos sistemas operacionais está apresentado na Figura 7.28. O eixo vertical indica o percentual de CPU utilizado durante o Teste 3, e cada coluna representa o consumo dos clientes, sendo que a coluna denominada Server representa a plataforma StackSync. O total de processamento consumido ficou abaixo de 5%

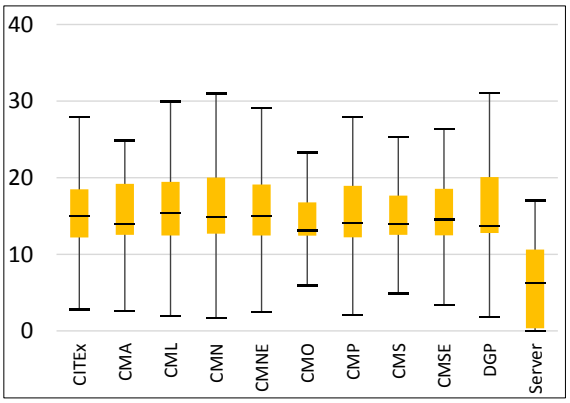


Figura 7.27: Consumo de CPU (%) durante o Teste 3.

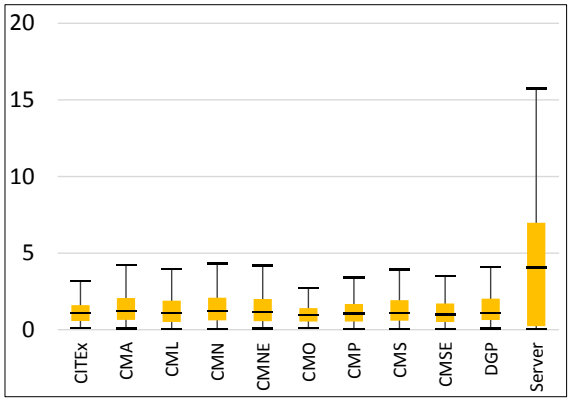


Figura 7.28: Consumo de CPU (%) pelos Sistemas Operacionais durante o Teste 3.

nos clientes StackSync, enquanto que a plataforma consumiu picos de até 16%, com mediana em 5%. Isso ocorreu porque os serviços de operações em bancos de dados foram contabilizados junto ao sistema operacional, devido a restrições no mecanismo de medição.

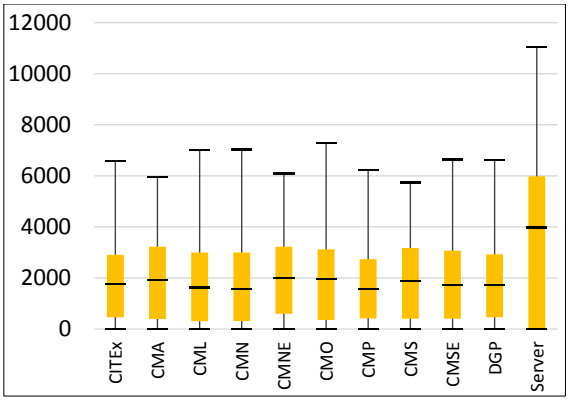


Figura 7.29: Gravação em Disco durante o Teste 3.

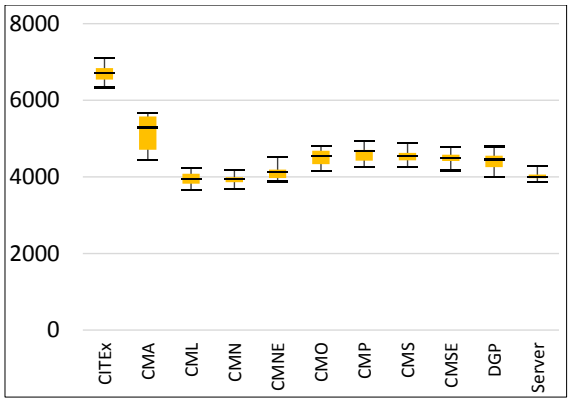


Figura 7.30: Consumo de Memória Principal (MB) durante o Teste 3.

A gravação em disco durante o Teste 3 apresentou resultados similares aos Testes 1 e 2, conforme apresentado na Figura 7.29, na qual o eixo vertical representa o total de blocos de 512 bytes gravados por segundo, variando de 0 a 12 mil blocos, e as colunas representam o total de blocos gravados por cliente, sendo que a coluna denominada Server indica o total gravado pela plataforma StackSync. Ocorreram picos de gravação, da ordem de até 7.500 blocos por segundo nos clientes, com mediana em 2.000 blocos gravados por segundo. Na plataforma StackSync, o volume de gravações apresenta picos de cerca de 11.000 blocos por segundo, e mediana em 4.000 blocos gravados por segundo. A diferença entre a plataforma e os clientes deve-se principalmente ao fato de que foram realizadas

ações de gravação de dados, particularmente ao inserir e modificar arquivos, que geraram mais operações de gravação na plataforma.

O consumo de memória principal seguiu as tendências observadas nos Testes 1 e 2, em que não houve variação significativa durante a execução. A Figura 7.30 apresenta o consumo de memória, na qual o eixo vertical indica o total de memória utilizada em megabytes, variando de 0 a 8GB, e cada coluna representa o total consumido pelos clientes, sendo apresentado na coluna denominada Server o consumo de memória da plataforma StackSync. O cliente CITEx utilizou mais memória ao iniciar o ambiente e, por isso, utilizou até de 6,2 a até 7GB de memória, enquanto que os demais clientes utilizaram de 4 a 4,6GB de memória. A plataforma precisou também de 4GB a 4,2GB de memória para gerenciar o ambiente durante o Teste 3.

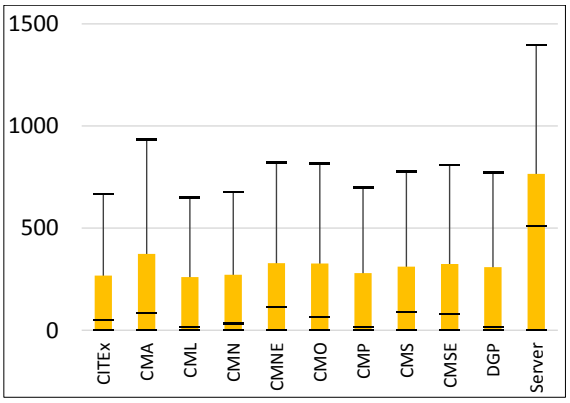


Figura 7.31: Dados Recebidos (KB) durante o Teste 3.

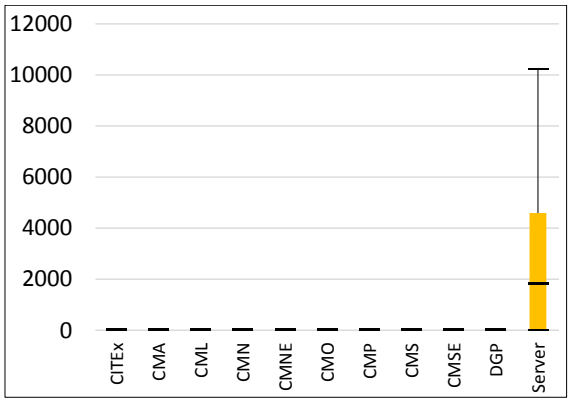


Figura 7.32: Dados Transmitidos (KB) durante o Teste 3.

Na Figura 7.31 está apresentada a recepção de dados através da rede pelos clientes e pela plataforma StackSync, na qual o eixo vertical representa o volume de dados recebidos em kilobytes, variando de 0 a 1,5MB por segundo, e as colunas indicam os dados recebidos pelos clientes, sendo que a coluna denominada Server indica os dados recebidos pela plataforma. O recebimento de dados pelos clientes ocorreu em picos de até 1MB por segundo, com mediana em 100KB por segundo, tendo em vista que o teste apresentou diversos momentos de inatividade. A plataforma StackSync recebeu dados à uma taxa máxima de 1,5MB por segundo, com mediana em 500KB por segundo.

A transmissão de dados pode ser visualizada na Figura 7.32, em que o eixo vertical representa o total de dados transmitidos em kilobytes, variando e 0 a 12MB por segundo, e as colunas representam os dados transmitidos pelos clientes, sendo que a coluna denominada Server indica os dados transmitidos pela plataforma StackSync. Durante o Teste 3, os clientes receberam dados em praticamente todo o teste, sendo que os dados transmitidos resumiram-se a informações de controle. Por isso, a plataforma StackSync apresentou

grande volume de transmissão, e os clientes, não. A plataforma chegou ao pico de 10MB por segundo de transmissão, com mediana em 2MB por segundo.

7.2.4 Teste 4: Operações Consecutivas de Gravação

Os Testes 4, 5 e 6 foram realizados em sequência, e manipularam os mesmos arquivos utilizados no Teste 1, compostos por 452MB de dados, distribuídos em cinco mil arquivos. A proposta do Teste 4 foi de verificar a estabilidade e o desempenho da plataforma StackSync ao receber várias tarefas consecutivas, praticamente ao mesmo tempo, de sincronização de arquivos, em particular de arquivos adicionados. Foram adicionados cinco blocos de mil arquivos, cada bloco adicionado a um cliente StackSync. A Tabela 7.5 apresenta o tempo utilizado para completar a sincronização em cada cliente.

Tabela 7.5: Tempo de Sincronização do Teste 4.

Cliente	**Tempo de Sincronização (min)**
CITEx	06:40
CMA	06:28
CML	06:46
CMN	06:32
CMNE	06:31
CMO	06:41
CMP	06:41
CMS	06:46
CMSE	06:26
DGP	06:32

As medições realizadas no Teste 4 podem ser visualizadas nas Figuras 7.33 até 7.38. A metodologia de avaliação segue a mesma adotada nos outros testes.

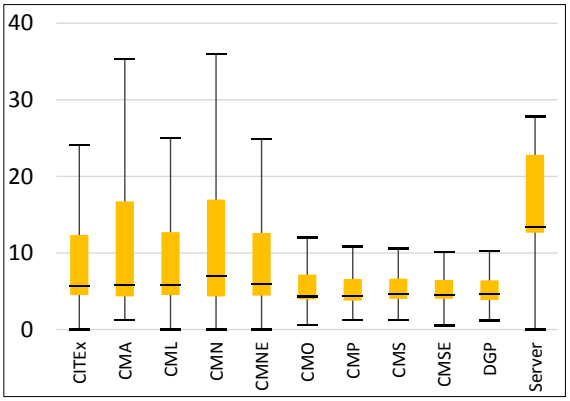


Figura 7.33: Consumo de CPU (%) durante o Teste 4.

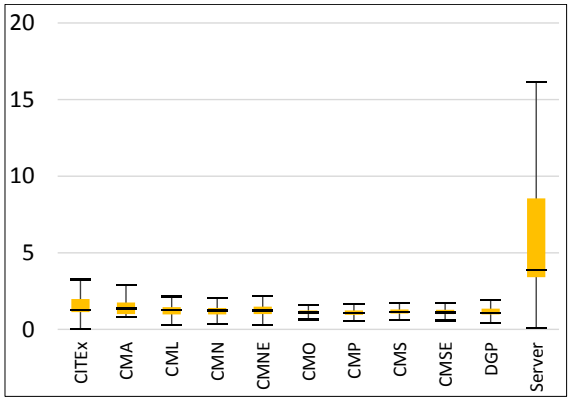


Figura 7.34: Consumo de CPU (%) pelos Sistemas Operacionais durante o Teste 4.

O Teste 4 iniciou acrescentando os arquivos nos primeiros clientes, CITEx, CMA, CML, CMN e CMNE. Na Figura 7.33 observa-se o consumo de CPU relacionando cada coluna a um cliente, e para a plataforma StackSync, identificada pela coluna Server. O eixo vertical indica o percentual de processamento utilizado durante o teste, variando de 0 a 40%. Os clientes que adicionaram os arquivos utilizaram mais CPU, em virtude do processamento de inclusão dos arquivos no armazenamento compartilhado, chegando a 35% de processamento em momentos de pico, com mediana de consumo em 5%. Os demais clientes tiveram um consumo de CPU menor, com picos de até 11%, e mediana 5%, principalmente porque somente receberam os dados pela plataforma StackSync. A plataforma apresentou maior consumo de CPU, sobretudo porque a articulação entre as diversas cópias exige mais processamento. Ainda assim, a mediana de consumo de CPU na plataforma ficou em 15%, com o máximo consumido em 27%.

O consumo de processamento pelos sistemas operacionais durante o Teste 4 foi equivalente ao observado nos testes anteriores, com pequenas variações nos primeiros clientes que participaram da inclusão dos arquivos no armazenamento compartilhado. A Figura 7.34 apresenta este consumo de CPU, na qual o eixo vertical indica o total de processamento em porcentagem, variando de 0 a 20%, e as colunas indicam o consumo dos clientes, sendo a coluna denominada Server a que representa o processamento da plataforma. Os clientes utilizaram no máximo 3% de CPU, com mediana em 2%, enquanto que a plataforma StackSync utilizou até 17% de processamento, com mediana em 5%. O processamento de banco de dados da plataforma ficou destacado como serviço do sistema operacional, aumentando a percepção de uso durante o teste.

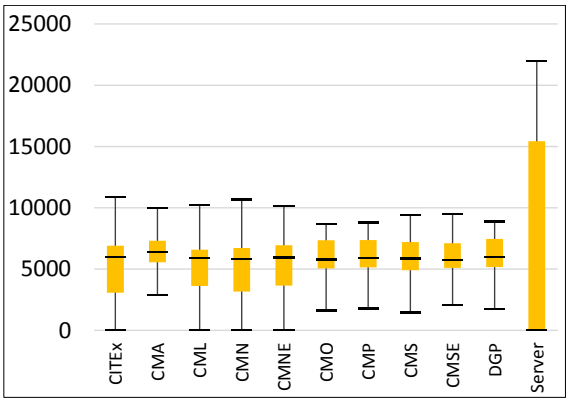


Figura 7.35: Gravação de Blocos em Disco durante o Teste 4.

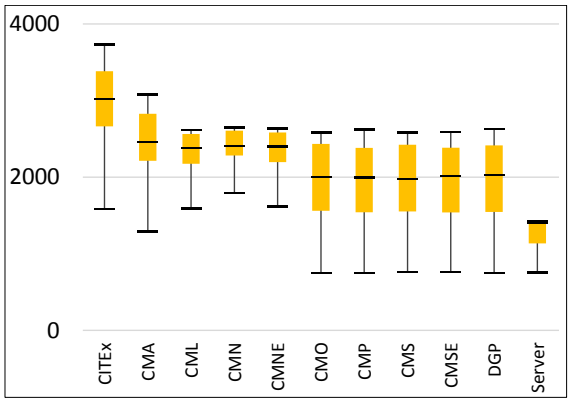


Figura 7.36: Consumo de Memória (MB) durante o Teste 4.

A gravação em disco ocorreu de maneira uniforme entre os clientes StackSync, conforme observado na Figura 7.35, em que o eixo vertical representa o total de blocos de 512 bytes gravados por segundo, e as colunas indicam o total gravado pelos clientes, sendo

a coluna denominada Server a que representa a gravação da plataforma StackSync. Os clientes gravaram dados em uma taxa máxima de 11.000 blocos por segundo, com mediana em 6.000 blocos por segundo. A plataforma, por sua vez, gravou até 22.000 blocos (11MB) por segundo para sincronizar os dados adicionados neste cenário, porém, devido aos longos períodos de inatividade durante o Teste 4, a mediana ficou abaixo de 100 blocos por segundo.

A Figura 7.36 apresenta o uso de memória pelos clientes e pela plataforma StackSync, identificada pela coluna Server, e onde o eixo vertical representa o total de memória utilizada em megabytes, variando de 0 a 4GB. O consumo de memória foi maior nos clientes que inseriram arquivos no ambiente, e uniforme entre os outros clientes. O consumo não chegou a 4GB de memória para nenhum dos clientes, e estabilizou-se em menos de 2GB para a plataforma StackSync.

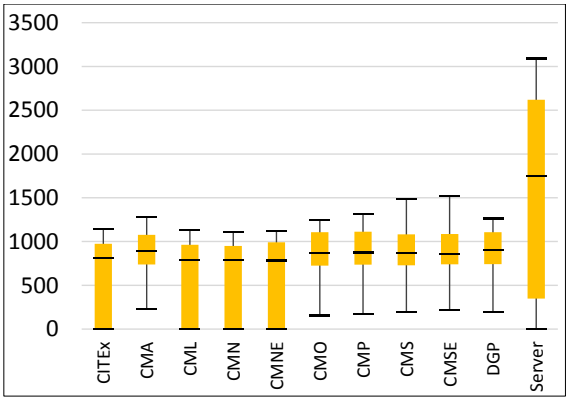


Figura 7.37: Dados Recebidos (KB) durante o Teste 4.

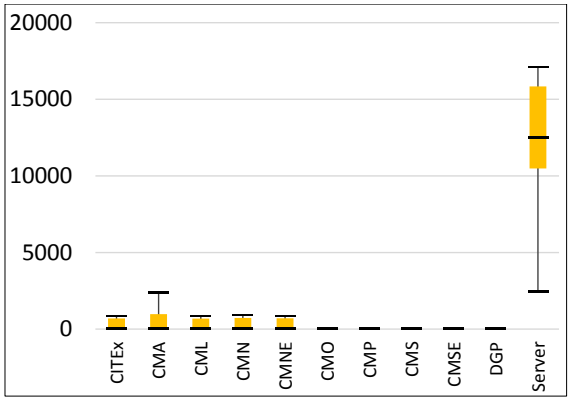


Figura 7.38: Dados Transmitidos (KB) durante o Teste 4.

A recepção de dados por rede ocorreu de forma similar aos outros testes realizados, conforme visualizado na Figura 7.37, em que o eixo vertical representa o volume de dados trafegados por segundo, em kilobytes, e as colunas representam os dados recebidos por cada cliente, sendo a coluna denominada Server a que indica os dados recebidos pela plataforma. Os clientes receberam dados em taxas de, no máximo, 1,5MB por segundo, com a mediana em 900KB por segundo. A plataforma transferiu mais dados, recebendo até a taxa de 3MB por segundo, com mediana de 1,8MB por segundo.

A transmissão de dados foi mais expressiva, conforme observado na Figura 7.38. O eixo vertical representa o total de dados enviados, em kilobytes, e as colunas indicam a transmissão por cliente, sendo a coluna de nome Server a que indica o envio de dados através da rede pela plataforma StackSync. Os clientes que inicializaram os dados no ambiente compartilhado transmitiram a taxas abaixo de 2MB por segundo, com mediana de 100KB por segundo, enquanto que os clientes restantes enviaram somente dados de

Tabela 7.6: Tempo de Sincronização do Teste 5.

Cliente	Tempo de Sincronização (min)
CITEx	09:25
CMA	09:18
CML	09:26
CMN	09:32
CMNE	09:31
CMO	09:21
CMP	09:21
CMS	09:26
CMSE	09:26
DGP	09:32

controle e metadados para a plataforma. O serviço de sincronização enviou dados em taxas mais altas, com mediana a 12MB de dados por segundo, e picos de até 17MB por segundo de envio de dados.

7.2.5 Teste 5: Operações Consecutivas de Modificação

Como citado na descrição do Teste 4, o Teste 5 foi realizado concomitantemente ao anterior. O procedimento foi o de alterar cinco mil arquivos, praticamente ao mesmo tempo, em blocos de mil arquivos e em cinco clientes StackSync diferentes, para testar o desempenho e a estabilidade da plataforma. O tempo de sincronização entre os clientes ocorreu como descrito na Tabela 7.6. Cada cliente levou o tempo citado na tabela para sincronizar as mudanças executadas no Teste 5.

As medições realizadas no Teste 5 podem ser visualizadas nas Figuras 7.39 até 7.44. A metodologia de avaliação segue a mesma adotada nos outros testes.

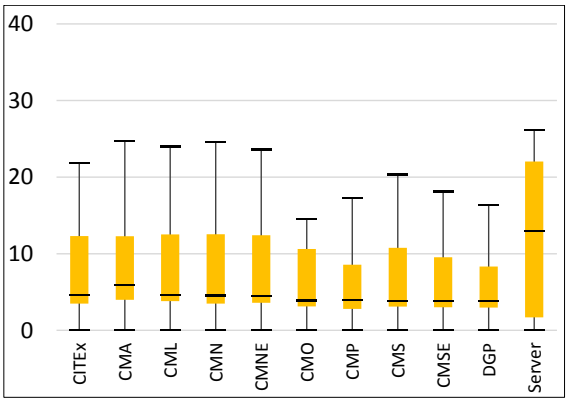


Figura 7.39: Consumo de CPU (%) durante o Teste 5.

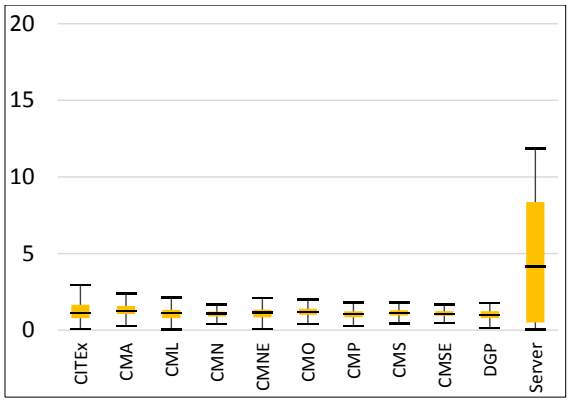


Figura 7.40: Consumo de CPU (%) pelos Sistemas Operacionais durante o Teste 5.

A Figura 7.39 apresenta o consumo de CPU em porcentagem de cada cliente e da plataforma StackSync, representada pela coluna Server, e na qual o eixo vertical indica o consumo de CPU, variando de 0 a 40%. Para as operações realizadas de alteração de arquivos, o consumo de CPU chegou a picos de 25% nos clientes e na plataforma StackSync. A mediana para os clientes ficou em cerca de 5%, e para a plataforma em 13%. Os clientes em que ocorreu o processamento de modificação dos arquivos foi diferente dos outros clientes, embora que a uma taxa menor do que 10% de processamento.

O consumo de CPU pelos sistemas operacionais durante o Teste 5 seguiu de forma semelhante aos outros testes, como observado na Figura 7.40, em que o eixo vertical indica o consumo de CPU, variando de 0 a 20%, e as colunas informam o consumo por cliente, sendo que a coluna denominada Server apresenta informações da plataforma StackSync. Os sistemas operacionais dos clientes consumiram menos do que 3% de CPU, com mediana em 1%, e a plataforma consumiu até 12% de processamento, com mediana em 4%. Mais uma vez, o processamento dos metadados influenciou a comparação, pois não foi possível separar o processamento de bancos de dados dos outros serviços de sistema operacional.

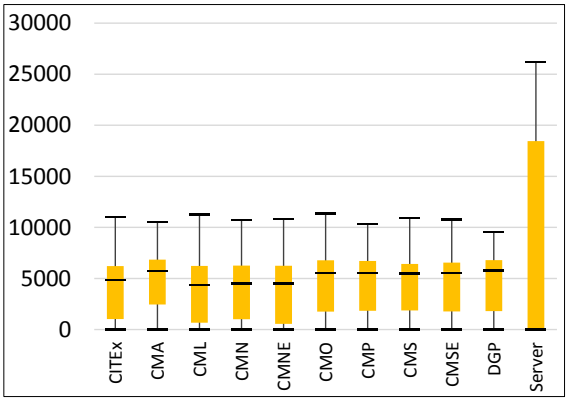


Figura 7.41: Gravação em Disco durante o Teste 5.

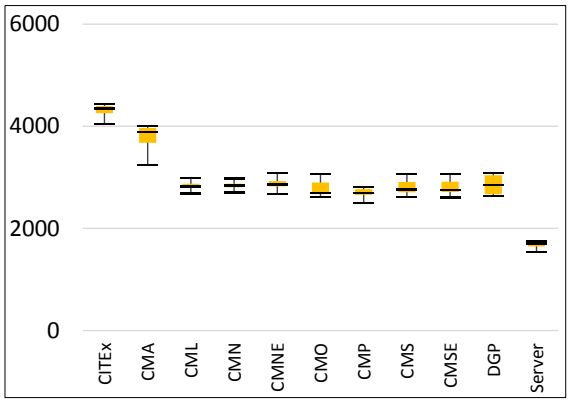


Figura 7.42: Consumo de Memória Principal (MB) durante o Teste 5.

Tendo em vista que todos os clientes estavam sincronizados no início do Teste 5, a gravação em disco foi muito próxima em todos, como visualizado na Figura 7.41. O eixo vertical representa o total de blocos de 512 bytes gravados por segundo, sendo que as colunas indicam a gravação por cliente, com a coluna de nome Server representando a gravação da plataforma. Os clientes gravaram em uma taxa máxima de 11.000 blocos por segundo, com mediana em 5.000 blocos por segundo. A gravação na plataforma, por sua vez, foi maior, pois além de gravar as alterações dos dados, também há o controle dos metadados. Na plataforma o pico foi de até 26MB por segundo, mas com a mediana próxima a zero, em virtude dos longos períodos de inatividade.

O consumo de memória, durante o Teste 5, pode ser observado na Figura 7.42. O eixo vertical indica o total de memória utilizada em megabytes, e as colunas apresentam o consumo pelos clientes, sendo a coluna denominada Server a que representa a memória utilizada pela plataforma. A variação de consumo de memória para os clientes ficou abaixo de 200MB, e o consumo máximo foi de 4,6GB. A plataforma StackSync utilizou 1,8GB de memória para operar durante o Teste 5.

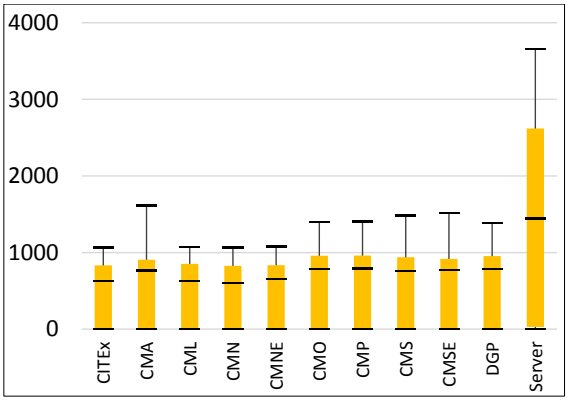


Figura 7.43: Dados Recebidos (KB) durante o Teste 5.

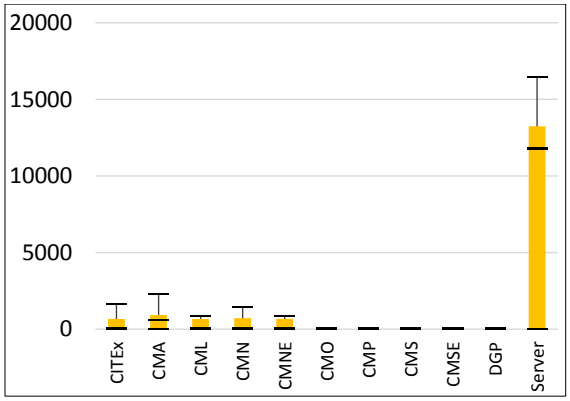


Figura 7.44: Dados Transmitidos (KB) durante o Teste 5.

Na execução do Teste 5, os clientes receberam dados em uma escala uniforme, com menores taxas nos clientes em que as mudanças ocorreram. Isso aconteceu porque os clientes já possuíam os dados alterados, não necessitando copiá-los novamente do armazenamento compartilhado. a Figura 7.43 apresenta a taxa de recebimento de dados em kilobytes, visualizada no eixo vertical, e as colunas indicam o total de dados recebidos pelos clientes e pela plataforma StackSync, identificada como Server. A taxa de recepção de arquivos ficou abaixo de 2MB por segundo nos clientes, com mediana abaixo de 1MB por segundo. A plataforma apresentou picos de recebimento de quase 4MB por segundo, com mediana pouco maior do que 1MB por segundo.

De acordo com a Figura 7.44, o eixo vertical indica o total de dados enviados através da rede pelos clientes e pela plataforma StackSync, sendo que as colunas representam os dados enviados pelos clientes, e a coluna denominada Server representa os dados enviados pela plataforma StackSync. Os clientes em que ocorreram as alterações nos arquivos, por ocasião do planejamento do Teste 5, enviaram dados à plataforma, mas em uma taxa que não chegou a 3MB por segundo nos picos de transmissão. A plataforma precisou enviar as alterações a todos os clientes, e por esse motivo sua taxa de transmissão chegou a picos de 16MB por segundo, com mediana em 13MB por segundo.

Tabela 7.7: Tempo de Sincronização do Teste 6.

Cliente	**Tempo de Sincronização (min)**
CITEx	00:25
CMA	00:28
CML	00:26
CMN	00:32
CMNE	00:31
CMO	00:21
CMP	00:21
CMS	00:26
CMSE	00:26
DGP	00:32

7.2.6 Teste 6: Operações Consecutivas de Remoção

O último teste realizado consistiu de remover cinco mil arquivos, organizados em blocos de mil, e as operações ocorreram em clientes distintos. Como o resultado da exclusão envolve basicamente o controle de metadados, ela ocorre com mais rapidez que as outras operações. Na Tabela 7.7 pode-se observar o tempo final de sincronização de todos os clientes, na coluna da direita, relacionada ao nome do cliente à esquerda. Os tempos foram medidos com base nos *logs* gerados pelos clientes StackSync.

As medições realizadas no Teste 6 podem ser visualizadas nas Figuras 7.45 até 7.50. A metodologia de avaliação segue a mesma adotada nos outros testes.

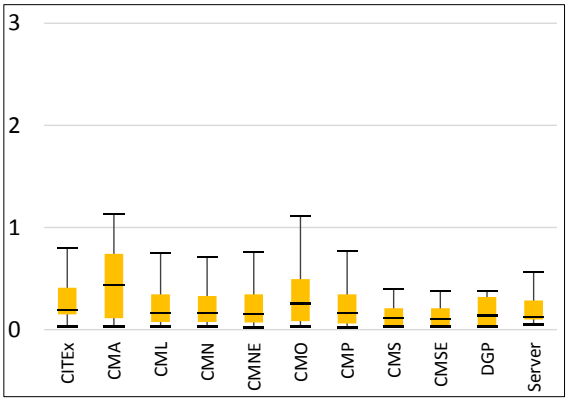


Figura 7.45: Consumo de CPU (%) durante o Teste 6.

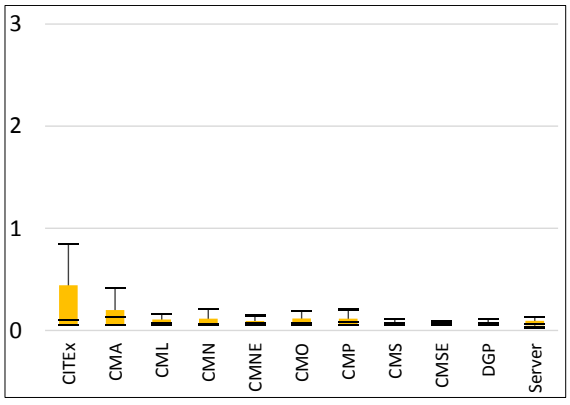


Figura 7.46: Consumo de CPU (%) pelos Sistemas Operacionais durante o Teste 6.

A execução do Teste 6 representou o mínimo consumo de recursos, como observado nas Figuras 7.45 e 7.46. O consumo de CPU ficou abaixo de 1%, indicando que a ação de excluir arquivos, mesmo que em uma quantidade significativa, representa pouco impacto na execução da plataforma StackSync.

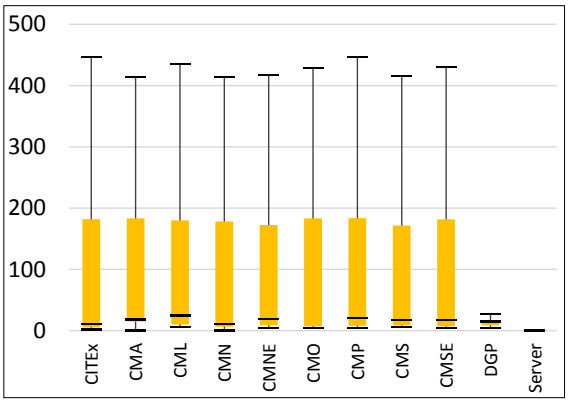


Figura 7.47: Gravação de Blocos em Disco durante o Teste 6.

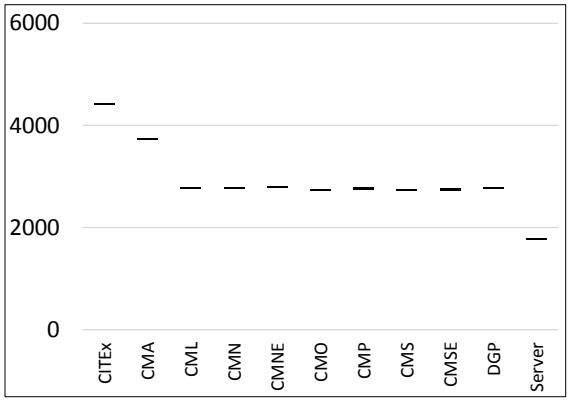


Figura 7.48: Consumo de Memória Principal (MB) durante o Teste 6.

As operações de exclusão geraram um volume de operações de gravação muito abaixo do que foi observado nos outros testes. Na Figura 7.47 observa-se que os clientes apresentaram picos de gravação em blocos de dados de até 500 blocos (250KB de dados) por segundo, e a plataforma gravou poucos blocos em comparação com os outros testes, tendo em vista a operação limitada aos metadados.

Devido à curta execução do Teste 6, a gravação em disco foi breve e a memória principal praticamente não variou, conforme visualizado na Figura 7.48, em que o eixo vertical indica o consumo de memória em megabytes, e as colunas indicam a memória utilizada pelos clientes e pela plataforma StackSync, representada pela coluna Server. O consumo de memória foi muito próximo ao observado nos Testes 4 e 5, em virtude da sua execução consecutiva, e os valores de mediana, máximo e mínimo mantiveram-se iguais.

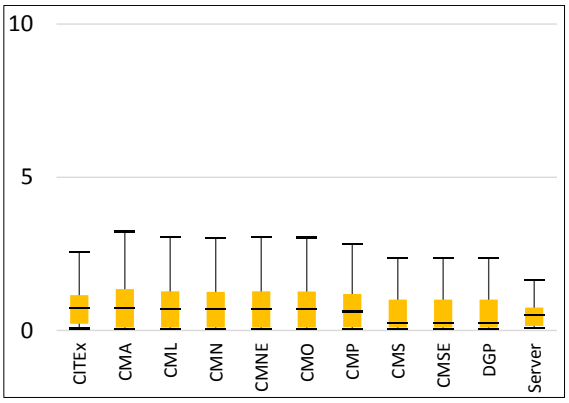


Figura 7.49: Dados Recebidos (KB) durante o Teste 6.

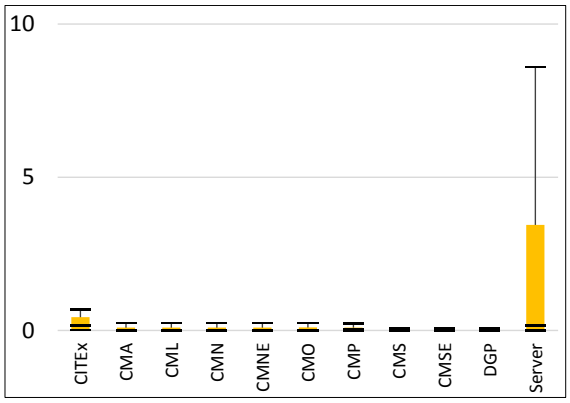


Figura 7.50: Dados Transmitidos (KB) durante o Teste 6.

O volume de mensagens trocadas entre os clientes e a plataforma StackSync foi bem reduzida, em especial porque somente metadados trafegaram entre clientes e a plataforma.

As Figuras 7.49 e 7.50 apresentam as taxas de transmissão, que não atingiram 10KB de tráfego por segundo em nenhum caso, mostrando a eficiência de sinalização da plataforma quando se opera somente os metadados do armazenamento compartilhado.

7.2.7 Análise dos Resultados

De posse dos resultados dos testes, foi possível esclarecer alguns aspectos importantes. A começar pelo consumo de processador, que foi possível observar que o funcionamento do *middleware* não impacta significativamente no sistema operacional e os serviços básicos por ele providos. Em todos os testes, o consumo de CPU pelo sistema operacional e serviços não ultrapassou 10%, e a mediana não passou de 5%. Por sua vez, a plataforma StackSync e seus aplicativos clientes participaram com picos de mais de 30% de consumo de CPU, apesar da mediana ficar quase sempre igual ou abaixo de 10%. Isso demonstra que a aplicação cliente StackSync consome processamento em rajadas, mas tem períodos de baixa atividade significativos.

Ainda sobre o processamento, ficou esclarecido que a plataforma consome mais CPU do que as estações clientes, o que indica que esta característica tem que ser levada em conta ao elaborar um ambiente de produção para usar a plataforma.

Com relação ao consumo de memória principal, em todos os testes realizados o consumo de memória foi muito estável. Isso se deve, particularmente, ao fato de que o gerenciamento de memória ficou a cargo do processo Java, em que é baseado o StackSync, e por este motivo não foi verificado o real consumo de objetos e dados dentro da máquina virtual Java. Apesar disso parecer um fator de dúvida, ainda fica esclarecido que o consumo de memória poderia ter variado, pois o processo Java da plataforma Stacksync estava configurado para utilizar mais memória dinamicamente, até um total de 6GB, mas não precisou de memória adicional, utilizando 4GB ou menos. Ficou claro que a plataforma utiliza muita memória para operar, mas que é estável no consumo, pois ao se observar os gráficos de consumo de memória, é perceptível que os valores da mediana, máximo e mínimo estão muito próximos, o que implica efetivamente na estabilidade desse consumo.

A gravação em disco foi apresentada como uma ação de alto custo para o desempenho das aplicações, e importante no cenário pesquisado. Foram apresentadas situações em que se esperava muita gravação em disco, e foi demonstrado pelos gráficos que alguns clientes não gravaram dados em certos casos. Do mesmo modo, a própria plataforma não gravou dados em alguns testes, mas que era algo esperado, pela natureza do teste realizado. Em alguns casos o o volume dos dados gravados não passou de centenas de kilobytes, contudo, nos principais eventos de carga, a transferência passou para a escala de megabytes, proporcional aos dados adicionados ou modificados.

A transmissão e a recepção de dados são características fundamentais para se avaliar a portabilidade de um sistema para uma arquitetura distribuída. Foram explanados em detalhes o consumo de rede de transmissão e recepção, para comprender e avaliar em que circunstâncias pode-se utilizar o *middleware* StackSync. Similar às outras métricas, pode-se observar que, em todos os testes, o tráfego de informações entre clientes e a plataforma ficou abaixo dos 20Mbps, o que indica que a plataforma pode obter sucesso ao operar em canais de comunicação mais lentos do que a rede em que foi testada.

Com relação aos Testes 1 e 2, foi observado que o acréscimo de clientes à plataforma não deteriora o funcionamento dos clientes já presentes, o que é um fator desejável e fundamental em sistemas distribuídos. Enquanto que o Teste 3 ensejava verificar o funcionamento do sistema como se estivesse sendo operado por usuários, em especial para verificar a estabilidade das funções do *middleware*, os Testes 4, 5 e 6 procuraram forçar a plataforma além do que já havia sido testado. O sistema se comportou muito bem, e conseguiu sincronizar todos os dados com um desempenho razoável, variando de 1,9 megabytes por segundo, a até 2,2MB/s de volume de dados processados e sincronizados.

7.3 Considerações Finais

A arquitetura proposta neste capítulo tem como objetivo prover tolerância a falhas para sistemas legados, em particular os que utilizam dados armazenados diretamente em arquivos. O estudo realizado sobre o sistema Papiro, do Exército Brasileiro, demonstrou que há aplicabilidade para esta abordagem, e a importância do sistema justificou esta pesquisa.

Com base nos resultados obtidos neste capítulo, foi esclarecido que é possível operar o sistema Papiro sobre a plataforma StackSync e, assim, atender todos os Comandos Militares do Exército. Com as garantias de operação oferecidas pela arquitetura proposta, será possível otimizar também os processos realizados pela Diretoria de Inativos, Civis e Pensionistas e Assistência Social, a DCIPAS. Tais processos são fundamentalmente voltados para os militares da reserva e seus dependentes, usuários do sistema previdenciário do Exército. Os processos da DCIPAS são marcados por entraves burocráticos de trâmite de documentos, e o sistema Papiro poderá vencer esses obstáculos, desde que se garanta o funcionamento do sistema, mesmo em condições adversas.

As características analisadas acerca da execução da plataforma e dos clientes do *middleware* foram centrais na determinação do desempenho e da estabilidade do StackSync. A aplicação da arquitetura para hospedagem de sistemas legados fornecerá meios de operação transparentes, tanto no consumo de recursos computacionais, quanto na distribuição

da aplicação hospedada, que não precisa se adaptar à infraestrutura para operar adequadamente.

Capítulo 8

Conclusão

Neste trabalho, foi proposta uma arquitetura capaz de prover tolerância a falhas e escalabilidade, de forma transparente, e sem adaptações no código fonte dos sistemas hospedados, embora a tolerância a falhas não fosse uma característica nativa destes sistemas. Esta infraestrutura se mostrou adequada para comportar uma categoria de sistemas legados do Departamento-Geral do Pessoal do Exército Brasileiro (DGP), de maneira que sua aplicação permitirá o uso desses sistemas legados em uma escala anteriormente indisponível.

O Exército Brasileiro pode, a partir da metodologia apresentada neste trabalho, hospedar seus sistemas legados sob uma nova perspectiva, em que o armazenamento distribuído entre as regiões do território nacional possam operar de forma mais eficiente seus processos de trabalho. O DGP, na condição de provedor de serviços na área de pessoal do Exército, pode fazer uso deste novo recurso como iniciativa de acelerar os processos de concessão de benefícios previdenciários, nos quais a burocracia e a morosidade no trâmite de informações prejudicam seriamente os usuários dos serviços.

Para fazer uso da solução proposta, o DGP deve iniciar as operações em seu próprio datacenter e utilizar recursos do CITEx para compartilhar o armazenamento do sistema Papiro. Em seguida, utilizando a cadeia de comando e o relacionamento entre Organizações Militares da Instituição, proceder com a expansão dos locais de armazenamento distribuído em outros Estados brasileiros, tendo como premissa a visão estratégica e o princípio da oportunidade.

O estudo realizado sobre sistemas distribuídos trouxe conceitos que permitiram delimitar a forma como um *middleware* poderia contribuir com o objetivo principal da pesquisa. A aplicação de MOM na arquitetura proposta foi determinante para alcançar a tolerância a falhas, que, quando associada à escalabilidade, torna-se inviável através de comunicação síncrona.

Com os resultados obtidos, foi possível verificar a real contribuição que a arquitetura proposta tem a oferecer para sistemas legados, que é de prover características, antes

indisponíveis, sem adaptar o código fonte de sistemas. Esta contribuição é particularmente útil para o Exército Brasileiro, tendo em vista a presença de sistemas dessa natureza nos importantes processos e serviços prestados pela Força Terrestre, tanto ao público interno, quanto ao público externo.

A Diretoria de Inativos, Civis e Pensionistas e de Assistência Social (DCIPAS) possui processos em que a burocracia limita a qualidade no atendimento a seus usuários, mas que podem ser modificados através da nova abordagem de infraestrutura. O sistema Papiro, utilizado pela DCIPAS, tem papel fundamental no cenário de concessão de benefícios e direitos previdenciários, por coordenar o armazenamento e acesso aos documentos relacionados aos militares e seus dependentes.

A proposta foi apresentada à equipe gestora dos dados cadastrados no sistema Papiro, e sua utilização fica condicionada ao provimento de hardware nos Comandos Militares destinados à hospedagem do sistema. A arquitetura apresentada terá utilidade na rede corporativa do Exército, sincronizando os ambientes de armazenamento do sistema Papiro, e assim prover os resultados esperados em sua adoção.

A arquitetura proposta nesta pesquisa trará melhorias para o processo de concessão de benefícios previdenciários, e o maior resultado será sentido pelos usuários dos serviços providos pelo DGP. A partir da maior disponibilidade do sistema oferecida na rede corporativa do Exército, em que os usuários poderão cadastrar documentos diretamente na origem, o trâmite de documentos de geração de direitos será muito mais ágil, favorecendo assim uma resposta efetiva para quem mais precisa, e sobretudo em momentos de fragilidade, como ocorre durante a perda de um familiar.

Esta pesquisa pode ser aprofundada para atender aplicações que necessitem de um desempenho global superior ao apresentado nos testes deste trabalho. A substituição de elementos como o MOM RabbitMQ, ou o *cluster* Swift podem incrementar a taxa de processamento e de transferência de dados do *middleware*. Os algoritmos implementados pelo cliente StackSync e pelo SyncService também podem ser otimizados, face ao desempenho obtido, e observadas as características dos testes realizados.

Durante os testes de carga inicial, é possível modificar o processo de sincronia, de maneira que a cópia de dados para um novo cliente StackSync possa ser feita por um mecanismo síncrono, sem o uso do *middleware* de troca de mensagens. Dessa forma, pode ser possível acelerar o processo de inclusão ou de recuperação de um cliente no conjunto, desde que a arquitetura ofereça meios para permitir a sincronização de dados já existentes.

A arquitetura proposta foi avaliada no cenário de uma aplicação legada, com características de funcionamento e formato de dados específicos. Em uma pesquisa futura, a arquitetura apresentada neste trabalho pode ser analisada em outros perfis de aplicações legadas, com arquivos de dados em tamanhos variados, e outras características de im-

plementação em que a infraestrutura proposta possa oferecer meios de recuperação e de tolerância a falhas.

Referências

[1] Habtamu Abie, I. Dattani, M. Novkovic, J. Bigham, S. Topham, e R. Savola. GEMOM - significant and measurable progress beyond the state of the art. In *Systems and Networks Communications, 2008. ICSNC '08. 3rd International Conference on*, pages 191–196, Oct 2008. 40, 46, 47

[2] Xiao-Fei An e Li-Ying Bian. Design of message-oriented middleware of distance teaching platform based on distributed message control. In *Computational Aspects of Social Networks (CASoN), 2010 International Conference on*, pages 141–143, Sept 2010. 40, 43

[3] Stephanos Androutsellis-Theotokis e Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, dezembro 2004. 23

[4] Algirdas Avizienis, Jean-Claude 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. xi, xiv, 25, 26, 27, 28, 29, 30, 31, 32

[5] Prosunjit Biswas, Farhan Patwa, e Ravi Sandhu. Content level access control for openstack swift storage. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 123–126, New York, NY, USA, 2015. ACM. 58

[6] Anita Borg, Jim Baumbach, e Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 90–99, New York, NY, USA, 1983. ACM. 28

[7] Exército Brasileiro. Estatuto dos Militares das Forças Armadas. Online, agosto 2015. http://www.planalto.gov.br/ccivil_03/Leis/l6880compilada.htm. 63, 64

[8] Exército Brasileiro. Estrutura Organizacional do Exército Brasileiro: Estrutura Administrativa e Força Terrestre. Online, julho 2015. http://www.eb.mil.br/web/guest/estrutura-organizacional. 63

[9] Prima Chairunnanda, Khuzaima Daudjee, e M. Tamer Özsu. ConfluxDB: Multimaster Replication for Partitioned Snapshot Isolation Databases. *Proc. VLDB Endow.*, 7(11):947–958, July 2014. 68

[10] G. Coulouris, J. Dollimore, e Tim Kindberg. *Sistemas Distribuídos - 4ed: Conceitos e Projeto*. Bookman Ed, 2007. xi, 8, 9, 10, 11, 12, 14, 15, 19, 20, 21, 22, 23

[11] Edward Curry e Qusay H. Mahmoud. *Message-Oriented Middleware*. John Wiley & Sons, 2004. xi, 1, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50

[12] Edemberg Rocha da Silva. *Manutenção de clusters semânticos em sistemas de integração de dados em ambientes P2P*. PhD thesis, Universidade Federal de Pernambuco, 2014. 24

[13] Harvey M Deitel, Paul J Deitel, e David R Choffnes. *Operating systems*. Pearson/Prentice Hall, 2004. 6, 8, 9, 21

[14] Chefe do DGP. Diretrizes do Chefe do Departamento-Geral do Pessoal. Online, janeiro 2015. http://www.dgp.eb.mil.br/index.php/diretriz. 63

[15] Jr. Enslow, P.H. What is a "distributed"data processing system? *Computer*, 11(1):13–21, Jan 1978. 39, 40

[16] Fábio Favarim. Componentes em um esquema de tolerância a faltas adaptativa. Master's thesis, Universidade Federal de Santa Catarina, Centro Tecnológico, Florianópolis, 2003. xi, 25, 31, 32, 33, 34, 35, 36, 37

[17] Sérgio Murilo Maciel Fernandes. Avaliação de dependabilidade de sistemas com mecanismos tolerantes a falha: desenvolvimento de um método híbrido baseado em edspn e diagrama de blocos. Master's thesis, Universidade Federal de Pernambuco. CIN. Ciência da Computação, 2007. 25, 29, 33, 34, 35

[18] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9):948–960, 1972. 6

[19] OpenStack Foundation. Projeto OpenStack. Online, julho 2015. https://www.openstack.org/. 52

[20] OpenStack Foundation. Projeto OpenStack Swift. Online, janeiro 2015. https://swiftstack.com/openstack-swift/. 58, 59, 70, 72

[21] The Apache Software Foundation. Servidor de Aplicação Apache Tomcat. Online, agosto 2015. http://tomcat.apache.org/. 65

[22] F. C. Gärtner. Fundamentals fo fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, mar 1999. 14, 26, 29, 32, 33

[23] Departamento Geral do Pessoal. Portal do Departamento-Geral do Pessoal do Exército Brasileiro. Online, julho 2014. http://www.dgp.eb.mil.br/. 1, 2

[24] Departamento Geral do Pessoal. Normas Técnicas de Reserva Remunerada, Reforma e Pensão no Exército Brasileiro. Online, junho 2015. http://dcipas.dgp.eb.mil.br/index.php/normas-tecnicas2. 64

[25] Eliza Helena Areias Gomes. Uma abordagem de reserva antecipada de recursos em ambientes oportunistas. Master's thesis, Universidade Federal de Santa Catarina, Centro Tecnológico, Florianópolis, 2013. xi, 7

[26] Sérgio Gorender, Roberto Freire Cunha, et al. Um modelo híbrido e adaptativo para sistemas distribuídos tolerantes a falhas. 2005. 26, 29, 30, 31

[27] The PostgreSQL Global Development Group. Sistema de Gerenciamento de Banco de Dados PostgreSQL. Online, agosto 2015. http://www.postgresql.org/. 59, 65

[28] Phillip C. Heckell. Projeto SyncAny. Online, agosto 2015. https://www.syncany.org/. 59

[29] Gregor Hohpe e Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 41, 46

[30] N.M. Ibrahim e M.F. Hassan. Agent-based message oriented middleware(MOM) for cross-platform communication in SOA systems. In *Computer Information Science (ICCIS), 2012 International Conference on*, volume 1, pages 547–552, June 2012. 42, 44

[31] Alex Jeukens. Tolerância a falhas em sistemas de agentes móveis. Master's thesis, Universidade Estadual de Campinas. Instituto de Computação, Campinas, SP, Fevereiro 2003. 31, 35

[32] Leslie Lamport e Nancy Lynch. Chapter on distributed computing. Technical report, DTIC Document, 1989. 9

[33] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985. 25, 26, 29, 32, 33

[34] David Liben-Nowell, Hari Balakrishnan, e David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 233–242, New York, NY, USA, 2002. ACM. 23

[35] Pedro Garcıa López, Marc Sánchez Artigas, Cristian Cotes, Guillermo Guerrero, Adrian Moreno, e Sergi Toda. Stacksync: Architecturing the personal cloud to be in sync, 2013. 51

[36] Pedro Garcia Lopez, Marc Sanchez-Artigas, Sergi Toda, Cristian Cotes, e John Lenton. Stacksync: Bringing Elasticity to Dropbox-like File Synchronization. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 49–60, New York, NY, USA, 2014. ACM. xi, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62

[37] Francis Berenger Machado e Luiz Paulo Maia. *Arquitetura de Sistemas Operacionais*, volume 4. LTC, 2004. xi, 8, 9

[38] Marcos Roberto Alves Martins. *Integração Sistêmica em Middleware.* PhD thesis, Universidade de São Paulo, 2010. 18, 19, 21

[39] C. Metz. IP Anycast Point-to-(any) Point Communication. *Internet Computing, IEEE*, 6(2):94–98, Mar 2002. 68

[40] V.P. Nelson. Fault-tolerant computing: Fundamental Concepts. *Computer*, 23(7):19–25, July 1990. 26, 29, 30, 32

[41] Jon Postel e Joyce Reynolds. RFC 959: File Transfer Protocol, 1985. 59

[42] S. Sarat, Vasileios Pappas, e A. Terzis. On the Use of Anycast in DNS. In *Computer Communications and Networks, 2006. ICCCN 2006. Proceedings.15th International Conference on*, pages 71–78, Oct 2006. 68

[43] Warren Smith. An information architecture based on publish/subscribe messaging. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, pages 27:1–27:2, New York, NY, USA, 2011. ACM. 57

[44] Pivotal Software. Projeto RabbitMQ. Online, julho 2015. http://www.rabbitmq.com/. 57

[45] Marcos Tork Souza. *Controle de acesso para sistemas distribuídos.* PhD thesis, Universidade de São Paulo, 2010. 10

[46] Andrew S. Tanenbaum e Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. xi, xiv, 1, 8, 9, 10, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 41, 42, 43, 44, 45, 46

[47] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, e Daniel J. Abadi. Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems. *ACM Trans. Database Syst.*, 39(2):11:1–11:39, May 2014. 68

[48] VMWare. VMWare Virtualization. Online, julho 2015. http://www.vmware.com/. 71

[49] Erlang Web. Linguagem de Programação Erlang. Online, junho 2015. http://www.erlang.org/. 57

[50] AMQP Working Group. AMQP: Advanced Message Queueing Protocol. Online, julho 2015. http://www.amqp.org/. 57