
Marlon Henry Schweigert

*Análise de arquiteturas de microsserviços empregados a jogos
MMORPG voltada a otimização do uso de recursos
computacionais*

Joinville

2019

UNIVERSIDADE DO ESTADO DE SANTA CATARINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Marlon Henry Schweigert

**ANÁLISE DE ARQUITETURAS DE MICROSERVIÇOS
EMPREGADOS A JOGOS MMORPG VOLTADA A
OTIMIZAÇÃO DO USO DE RECURSOS
COMPUTACIONAIS**

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Charles Christian Miers
Orientador

Joinville, Novembro de 2019

**ANÁLISE DE ARQUITETURAS DE MICROSSERVIÇOS
EMPREGADOS A JOGOS MMORPG VOLTADA A
OTIMIZAÇÃO DO USO DE RECURSOS
COMPUTACIONAIS**

Marlon Henry Schweigert

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Bacharelado Ciência da Computação em turno Integral do CCT/UDESC.

Banca Examinadora

Charles Christian Miers - Doutor (orientador)

Débora Cabral Nazário - Doutora

Guilherme Piêgas Koslovski - Doutor

Agradecimentos

Aos meus pais, pelo amor, incentivo e apoio incondicional durante toda a minha jornada.

Ao professor Doutor Charles Christian Miers, pela orientação e apoio para a elaboração deste trabalho.

Agradeço a todos os professores por me proporcionar o conhecimento não apenas racional, mas a manifestação do caráter e afetividade da educação do processo de formação profissional. Portanto, que se dedicaram a mim, não somente por terem me ensinado, mas por terem me feito aprender. A palavra mestre, nunca fará justiça aos professores dedicados aos quais, sem nominar, terão os meus eternos agradecimentos.

A todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

*“É um erro acreditar que é possível
resolver qualquer problema importante
usando batatas.”*
(Douglas Adams)

Resumo

A crescente popularização de jogos *Massively Multiplayer Online Role-Playing Game* (MMORPG) demanda por novas abordagens tecnológicas, a fim de suprir as necessidades dos usuários com menor custo de recursos computacionais. Projetar essas arquiteturas, do ponto de vista da rede, é pertinente e impactante para o sucesso desses jogos. O objetivo deste trabalho é realizar uma análise voltada a identificar os recursos computacionais consumidos pelas arquiteturas de microsserviços Rudy, Salz e Willson, a qual são arquiteturas de microsserviços elaboradas para jogos MMORPG. Esse objetivo será atingido após realizar uma pesquisa referenciada, seguida de uma análise das principais arquiteturas e, preferencialmente, a execução de testes utilizando simulações de clientes sobre as arquiteturas implantadas em uma nuvem computacional para auxiliar na identificação de gargalos de recursos. Espera-se que os resultados obtidos possam auxiliar provedores de serviços MMORPG a reduzir gastos de manutenção e melhorar a qualidade de tais serviços.

Palavras-chaves: Arquitetura de Microsserviços, Desenvolvimento de Jogos, Rede de Jogos, Jogos Massivos, Otimização de Recursos, Nuvem de Computadores.

Abstract

The increasing popularization of *Massively Multiplayer Online Role-Playing Game* (MMORPG) demands new technological approaches in order to supply the requirements of users with lower cost of computational resources. Designing these architectures, from the network point of view, is relevant and impacting to the success of these games. The objective of this work is to propose an analysis aimed at identifying approaches to optimize the computational resources consumed by the Rudy, Salz and Willson micro-services architectures. This objective will be achieved after conducting a referenced search, followed by an analysis of the main architectures and, respectively, the execution of clients simulations using MMORPG architectures in a computational cloud to help identifying resources bottlenecks. The results obtained will help MMORPG service providers to reduce maintenance costs and improve the quality of such services.

Keywords: Microservice Architecture, Game Development, Game Network, Massive Games, Resource Optimization, Cloud Computing.

Sumário

Lista de Figuras	8
Lista de Tabelas	12
Lista de Abreviaturas	15
1 Introdução	17
2 Fundamentação Teórica	21
2.1 Jogos Eletrônicos	22
2.1.1 Árvore de gêneros de jogos eletrônicos	23
2.2 MMORPG	27
2.3 Jogabilidade de jogos MMORPG	28
2.4 Problemas em jogos MMORPG	32
2.4.1 Arquitetura de Clientes MMORPG	34
2.4.2 Arquitetura de Microsserviços	37
2.4.3 Microsserviços para jogos MMORPG	39
2.5 Arquiteturas MMORPG identificadas	43
2.5.1 Arquitetura elaborada por Rudy	43
2.5.2 Arquitetura elaborada por Salz	47
2.5.3 Arquitetura elaborada por Willson	50
2.6 Definição do Problema	52
2.7 Considerações parciais	54
3 Trabalhos Relacionados	55

3.1	Huang et al. (2004)	55
3.2	Villamizar et al. (2016)	57
3.3	Suznjevic e Matijasevic (2012)	59
3.4	Análise dos trabalhos relacionados	62
3.5	Considerações parciais	63
4	Proposta para análise de arquiteturas MMORPG	65
4.1	Proposta	66
4.2	Critérios de análise	67
4.3	Plano de testes	69
4.3.1	Cenário	71
4.3.2	Simulação de Clientes	74
4.3.3	Testes	78
4.4	Considerações parciais	79
5	Implementação	80
5.1	Tecnologias Utilizadas	80
5.2	Interconexão entre os microsserviços	82
5.2.1	Rudy	83
5.2.2	Salz	85
5.2.3	Willson	87
5.3	Ambiente de Testes	88
5.3.1	Rede do Servidor	90
5.3.2	Rede do Cliente	93
5.4	Considerações Parciais	95
6	Experimentos & Análises	97
6.1	Experimentos	97

6.1.1	Tempo de Resposta	98
6.1.2	Consumo de <i>Central Processing Unit</i> (CPU)	111
6.1.3	Consumo de Memória	121
6.1.4	Entrada de Rede	128
6.1.5	Saída de Rede	134
7	Considerações & Trabalhos futuros	141
	Referências	144

Listas de Figuras

2.1	Árvore de gêneros de jogos eletrônicos simplificada.	23
2.2	Sistema de autenticação para jogos MMORPG.	29
2.3	Área de interesse com base na proximidade de um jogador.	30
2.4	<i>Chat</i> baseado em contexto de posicionamento, utilizando Distância Euclidiana.	31
2.5	Personagens e os seus pontos de destino.	32
2.6	Personagens, objetos e <i>Non-Playable Characters</i> (NPCs) no ambiente.	32
2.7	Exemplo de Cliente MMORPG (Sandbox-Interactive Albion).	35
2.8	<i>Scene tree view</i> no motor gráfico Godot.	35
2.9	Modelo de um cliente genérico.	36
2.10	Microsserviços podem ter diferentes tecnologias.	38
2.11	Microsserviços são escaláveis.	39
2.12	Cliente pode realizar requisições <i>Create Read Update Delete</i> (CRUD) ao serviço.	40
2.13	Diagrama de requisições entre serviço e cliente com operações CRUD e <i>Remote Procedure Call</i> (RPC) em uma arquitetura monolítica.	41
2.14	Diagrama de requisições entre serviço e cliente com operações CRUD e RPC em uma arquitetura de microsserviços.	42
2.15	Diagrama de integração entre Cliente e Serviço, considerando a <i>engine</i> Unity3D.	42
2.16	Arquitetura Rudy completa.	44
2.17	Arquitetura Rudy completa com <i>firewall</i> .	45
2.18	Modelo de processos <i>Thread Pool</i> .	46
2.19	Arquitetura Salz.	48

2.20	Modelo de paralelismo do serviço de jogo na arquitetura Salz.	49
2.21	Arquitetura Willson.	51
2.22	Arquitetura de um jogo MMORPG genérico.	53
3.1	Arquitetura distribuída utilizando proxy.	56
3.2	Número de conexões no serviço pelo tempo decorrido.	56
3.3	Regressão linear comparado ao consumo de banda real do servidor.	57
3.4	Arquitetura monolítica web implementada na <i>Amazon Web Services</i> (AWS).	58
3.5	Arquitetura de microserviços web implementada na AWS.	59
3.6	Arquitetura de microserviços web implementada na AWS utilizando a tecnologia <i>lambda</i> .	60
3.7	Custo por um milhão de requisições em dólares utilizando diferentes arquiteturas sobre a AWS.	60
3.8	Regressão levando em conta a complexidade das ações e contexto dos personagens.	61
4.1	Ambiente de testes definido para a coleta de dados.	70
4.2	Rede de execução dos testes.	72
4.3	Área de interesse da simulação com raio de quatro células.	76
4.4	Autômato de movimentação dos personagens simulados.	76
5.1	Interconexões da arquitetura Rudy.	84
5.2	Interconexões da arquitetura Salz.	86
5.3	Interconexões da arquitetura Willson.	88
5.4	Cenário Cliente Servidor Genérico.	89
5.5	Cenário Cliente Servidor Genérico.	90
5.6	Sub-redes do servidor.	91
5.7	Arquitetura abstruída do OpenStack	91
5.8	Sub-redes do servidor com as máquinas virtuais	93

5.9	Quantidade de jogadores simultâneos cresce linearmente.	95
6.1	Tempo de resposta para criar contas	99
6.2	Tempo de resposta para criar personagens	101
6.3	Tempo de resposta para iniciar sessões	103
6.4	Tempo de resposta para instanciar personagens	105
6.5	Tempo de resposta para movimentar personagens	107
6.6	Tempo médio de resposta para movimentar personagem comparado ao número de jogadores	107
6.7	Tempo de resposta para enviar mensagens	108
6.8	Tempo médio de resposta para enviar mensagens comparado ao número de jogadores	109
6.9	Tempo de resposta para receber mensagens	110
6.10	Tempo médio de resposta para receber mensagens comparado ao número de jogadores	111
6.11	Consumo de CPU dos bancos de dados	112
6.12	Consumo de CPU pelo PostgreSQL comparado ao número de jogadores simultâneos.	114
6.13	Consumo de CPU pelo PostgreSQL comparado ao número de jogadores simultâneos.	115
6.14	Consumo de CPU dos microsserviços	117
6.15	Média do consumo de CPU dos microsserviços por jogador simultâneo	118
6.16	Comparação de consumo total de CPU pelas arquiteturas.	120
6.17	Consumo de memória dos bancos de dados	122
6.18	Consumo de memória média do PostgreSQL comparado ao número de jogadores simultâneos.	125
6.19	Consumo de memória dos microsserviços	126
6.20	Entrada de dados da rede dos bancos de dados	129

6.21	Vazão da entrada de dados para a rede <i>databases</i> .	130
6.22	Entrada de dados da rede das arquiteturas	131
6.23	Vazão da entrada de dados para a rede <i>gameservers</i> .	133
6.24	Saída de dados da rede dos bancos de dados	135
6.25	Vazão da saída de dados para a rede <i>gameservers</i> .	136
6.26	Saída de dados da rede dos microsserviços	138
6.27	Vazão da saída de dados para a rede <i>gameservers</i> .	140

Lista de Tabelas

2.1	Tipos de comunicação e quantidade de jogadores impactados por ocorrências conforme o seu gênero.	27
3.1	Complexidade da interação com o ambiente, por contexto da interação.	61
3.2	Trabalhos relacionados por categoria.	62
3.3	Trabalhos relacionados por recurso analisado.	63
3.4	Arquiteturas analisadas.	63
4.1	Possíveis conjuntos para a análise.	68
4.2	Tabela de interdependência das sub-redes.	73
4.3	Limite de recursos por instância de cada rede.	74
4.4	Requisitos das funcionalidades e respectivo impacto de implementação.	75
4.5	Requisitos mínimos funcionais para a implementação da simulação descrita.	75
5.1	Microsserviços da arquitetura Rudy.	83
5.2	Microsserviços da arquitetura Salz.	85
5.3	Microsserviços da arquitetura Willson.	87
5.4	Subredes da rede do servidor.	92
5.5	Configurações das máquinas virtuais do servidor.	93
6.1	Média, Variância, Máximo e Mínimo da operação Criar Conta	100
6.2	Média, Variância, Máximo e Mínimo da operação Criar Personagem	102
6.3	Tempo de resposta médio dos quadrantes.	104
6.4	Variância dos quadrantes na operação Iniciar Sessão.	104
6.5	Tempo de resposta médio e variância dos quadrantes para instanciar personagem.	106

6.6	Consumo de CPU por quadrante pelo PostgreSQL.	113
6.7	Consumo de CPU por quadrante pelo Redis.	114
6.8	Consumo de CPU por quadrante dos microsserviços.	118
6.9	Consumo total de CPU por quadrante dos microsserviços.	119
6.10	Consumo médio de memória por quadrante do PostgreSQL.	124
6.11	Consumo médio de memória pelos microsserviços.	127
6.12	Consumo médio da entrada da rede por quadrante do PostgreSQL.	130
6.13	Consumo médio da entrada da rede por quadrante do servidor de jogo.	133
6.14	Consumo médio da saída da rede por quadrante do servidor de jogo.	136
6.15	Consumo médio da saída da rede por quadrante do servidor de jogo.	140

Listagem

6.1 Informações do Bloco	122
------------------------------------	-----

Listas de Abreviaturas

ACID Atomicidade, Consistência, Isolamento e Durabilidade

API *Application Programming Interface*

AWS *Amazon Web Services*

C/S Cliente/Servidor

CPU *Central Processing Unit*

CRUD *Create Read Update Delete*

DCC Departamento de Ciência da Computação

FIFO *First In First out*

FPS *First-Person Shooter*

GOB *Golang Object Encoding*

HTTP *Hypertext Transfer Protocol*

IDE *Integrated Development Environment*

IP *Internet Protocol*

JSON *JavaScript Object Notation*

JWT *JSON Web Token*

LAN *Local Area Network*

MMO *Massively Multiplayer Online*

MMORPG *Massively Multiplayer Online Role-Playing Game*

MOBA *Multiplayer Online Battle Arena*

MVC *Model-View-Controller*

NoSQL *Not Only SQL*

NPC *Non-Playable Character*

P2P *Peer-to-Peer*

PaaS *Platform as a Service*

POF *Point of View*

PvNPCs *Player vs NPCs*

PvP *Player vs Player*

REST *Representational State Transfer*

RPC *Remote Procedure Call*

RPG *Role-Playing Game*

RTS *Real-Time Strategy*

SQL *Structured Query Language*

TCP *Transmission Control Protocol*

TPS *Third-person Shooter*

UDP *User Datagram Protocol*

VM *Virtual Machine*

WAN *Wide Area Network*

XDR *External Data Representation*

XML *Extensible Markup Language*

YAML *YAML Ain't Markup Language*

1 Introdução

Jogos *Massively Multiplayer Online Role-Playing Game* (MMORPG) são utilizados como negócio viável e lucrativo, sendo que, a experiência de jogabilidade na qual o usuário final será submetido é um fator crítico para o sucesso destes jogos. Este gênero, focado na interpretação de papéis de forma massiva em um ambiente compartilhado, tem como sua principal característica a comunicação e representação virtual de personagens em um mundo fantasia compartilhado. Neste mundo compartilhado cada jogador pode interagir com objetos ou tomar ações sobre outros jogadores em tempo real, tendo como principal objetivo a resolução de problemas dentro de uma comunidade virtual, na qual é resolvido específico a cada projeto (HANNA, 2015).

A maioria dos jogos MMORPG disponíveis no mercado estão implementados sobre uma arquitetura que executa o serviço sobre diversos servidores (CLARKE-WILLSON, 2017), nos quais o desempenho dos servidores e serviços influenciam tanto na experiência de jogabilidade do usuário final, quanto no custo de manutenção destes serviços (HUANG; YE; CHENG, 2004). Modelar um sistema de alto desempenho torna-se um trabalho essencial para a satisfação do usuário final neste cenário (HUANG; YE; CHENG, 2004). As ocorrências geradas por um sistema de baixo desempenho podem acarretar em frustração do usuário com o serviço e/ou aumento dos gastos com recurso computacional para manter tais serviços. Uma ocorrência é qualquer tipo de mal funcionamento em uma aplicação, não necessariamente aparente ao usuário final (HUANG; YE; CHENG, 2004). Evitar ou eliminar as ocorrências durante o projeto de desenvolvimento das arquiteturas do serviço é um processo crítico para o sucesso desses jogos.

Os avanços tecnológicos de sistemas distribuídos estão permitindo que as pessoas utilizem serviços com volumes massivos de dados para aplicações sensíveis a latência. Essa situação é propícia à área de jogos massivos, e tem formentado pesquisas deste ramo (KIM; KIM; PARK, 2008; HUANG; YE; CHENG, 2004; SALDANA et al., 2012; BARRI; GINE; ROIG, 2011). O principal objetivo destas pesquisas é reduzir a carga e o impacto da latência para o usuário final nesses serviços, aumentando a quantidade de jogadores simultâneos em um único serviço. Reduzir a carga e impacto da latência em serviços para jogos massivos resulta em uma melhor experiência de jogabilidade aos

usuários finais, sendo este um dos fatores críticos para o sucesso destes serviços (HUANG; YE; CHENG, 2004).

Entende-se por arquitetura de microsserviços uma arquitetura com diversas aplicações menores na qual utilizam troca de mensagens pela rede para implementar uma regra de negócio complexa, não exibindo necessariamente se este serviço é implementado por um ou mais microsserviços. Em específico, este paradigma de desenvolvimento de arquiteturas herda características da filosofia UNIX, descrito pelo matemático Malcolm Doug McIlroy em 2003 pela seguinte passagem:

Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

(RAYMOND, 2003)

Alguns estudos discorrem sobre a escalabilidade dos serviços baseados em microsserviços, sendo que tais estudos usam o desenvolvimento *web* como sua principal aplicação (Exit Games, 2017; JON, 2010; FRANCESCO, 2017). Entretanto, a estrutura de um serviço MMORPG baseado em microsserviços é mais abrangente, necessitando de maior desempenho comparado a serviços *web* (Exit Games, 2017; JON, 2010). Dessa forma, torna-se interessante a análise de serviços MMORPG baseados em microsserviços.

Deste modo o problema identificado é a falta de uma análise sobre o consumo de recursos para a manutenção de arquiteturas de microsserviços específicos a jogos MMORPG, utilizando como base as arquiteturas encontradas na literatura. Adicionalmente a este problema, será necessário o desenvolvimento de tais arquiteturas para submissão a experimentos e posterior análise.

Este trabalho tem como objetivo analisar o consumo de recursos das principais arquiteturas disponíveis na literatura. Nesse sentido, os objetivos específicos do atual trabalho são:

- Identificar arquiteturas empregadas na categoria de jogos MMORPG.
- Identificar os protocolos utilizados nessas arquiteturas.
- Identificar os microsserviços dessas arquiteturas.

- Identificar e avaliar ferramentas de análise de métricas para armazenar valores dos testes.
- Analisar o comportamento das arquiteturas aplicadas, levantando questões de desempenho e recursos utilizados.
- Propor alternativas de otimização para os problemas relacionados a consumo de recursos encontrados nas devidas arquiteturas identificadas.

Para dar suporte a proposta deste trabalho, uma revisão da literatura é apresentada a fim de esclarecer os conceitos principais referentes a temática de serviços MMORPG. Ao analisar os trabalhos relacionados identificados, houve uma dificuldade para encontrar os tópicos MMORPG e sistemas distribuídos baseado em microsserviços. Os resultados obtidos deste trabalho podem contribuir com futuros trabalhos científicos em diversas áreas de desenvolvimento e engenharia de software, computação distribuída ou consumo de recursos, específicos ou não a área de desenvolvimento de jogos.

A análise é inicialmente realizada através de uma pesquisa referenciada sobre arquiteturas de microsserviços e arquitetura de serviços MMORPG. Depois, um estudo sobre a intersecção de ambos os temas. Também são identificados trabalhos relacionados na presente literatura que foquem tanto em arquiteturas de microsserviços genéricas quanto em arquiteturas de serviços MMORPG. Em seguida é definida a proposta, junto a testes e cenários a fim de coletar informações e realizar a análise das arquiteturas de microsserviços específicas a jogos MMORPG.

Este trabalho de conclusão de curso possui natureza aplicada pois será necessário a implementação das arquiteturas descritas na literatura, utilizando as mesmas regras de negócio, viabilizando uma análise igualitária entre as arquiteturas propostas. A análise será efetuada de maneira qualitativa, pois será realizado um estudo a partir de valores gerados pelos experimentos, considerando as características individuais de cada arquitetura identificada.

Este trabalho está organizado em três capítulos, que dão suporte a análise das arquiteturas específicas a jogos MMORPG proposta. No Capítulo 2 são apresentados os conceitos necessários para o entendimento desse trabalho, com a finalidade de apresentar o funcionamento básico de um cliente e serviço MMORPG, arquitetura de um serviço baseado em microsserviços e por fim em específico algumas arquiteturas de

microsserviços MMORPG encontradas na literatura. O Capítulo 3 aborda trabalhos relacionados encontrados na literatura, tendo como objetivo principal destacar e comparar suas características, a fim de prover fundamentos para a análise dos serviços descritos na referenciamento teórica. A proposta é definida no Capítulo 4, a qual define o plano de extração de valores, a interpretação de tais valores e cenário a qual as arquiteturas propostas serão implantados.

2 Fundamentação Teórica

O termo *jogos eletrônicos* é amplamente difundido, entretanto as especificações, características e histórico deste termo não são de conhecimento popular. A Seção 2.1 trata a definição de jogo eletrônico, um levante histórico e o impacto da evolução do hardware no desenvolvimento dos jogos. Este termo é tomado como introdução para o conceito de gênero de jogo, abordado na Subseção 2.1.1, a qual referencia os principais gêneros, características e tecnologias (do ponto de vista de rede de computadores) que são comuns em cada gênero. Esta introdução acerca dos gêneros e suas tecnologias de comunicação busca trazer a importância do desempenho das arquiteturas dos jogos MMORPG e proporção da comunidade impactada caso hajam falhas de funcionamento nessas arquiteturas.

Após definir a categoria de jogo abordado, a Seção 2.2 apresenta uma introdução ao impacto de mercado desse gênero, uma definição simplista do gênero e a divisão das camadas de aplicação que permeiam uma arquitetura para um jogo MMORPG. Antes de abordar sobre as camadas da infraestrutura de uma arquitetura de jogo MMORPG e seus problemas recorrentes relativos a rede (Seção 2.4), faz-se obrigatório o entendimento sobre a jogabilidade deste gênero (Seção 2.3).

Os conceitos de cliente (Seção 2.4.1) e serviço (Seção 2.4.2) são abordados a fim de introduzir conceitos básicos de arquiteturas para jogos MMORPG. O objetivo destas seções é referenciar diversas tecnologias e técnicas utilizadas nesses sistemas a fim de permitir o desenvolvimento de uma arquitetura de microsserviços específica a jogos MMORPG. Por fim, torna-se obrigatório a apresentação de trabalhos relacionados (Seção 3) a arquitetura de jogos MMORPG desenvolvidos de forma distribuída ou sobre uma arquitetura de microsserviços. Esta seção em específico aborda exemplos de métodos e métricas utilizadas para mensurar o desempenho de tais arquiteturas, realizando por fim uma análise destes trabalhos (Subseção 3.4).

2.1 Jogos Eletrônicos

O primeiro sistema de entretenimento interativo foi construído em 1947, utilizando como base de exibição um tubo de raios catódicos. Essa criação foi patenteada em janeiro de 1948, datando então o início dos jogos eletrônicos (ADAMS, 2014; GOLDSMITH, 1947).

O jogo eletrônico, ou entretenimento interativo, é uma atividade intelectual que integra um sistema de regras, na qual utiliza tal sistema a fim de definir seus objetivos ou pontuação por meio de um computador, com o objetivo de despertar alguma emoção ao jogador (HANNA, 2015). Os jogos eletrônicos são aplicações convencionais, que executam sobre algum sistema operacional ou hardware apropriado a este fim. O sistema operacional, hardware ou base de execução da aplicação gráfica define a sua plataforma (*e.g.*, GNU/Linux, MS-Windows, Sony PS4, MS-XBox, web, etc.) (ADAMS, 2006).

Inicialmente, os jogos eram implementados de forma simples por conta da limitação de hardware das plataformas da década de 80. As implementações de jogos para *videogames* eram projetadas diretamente para algum hardware proprietário, sem sistema operacional, por muitas vezes sem utilizar comunicação por rede ou armazenamento em memória secundária (ROLLINGS; ADAMS, 2003). Além de diversas plataformas não terem acesso a rede, os serviços para jogos eram inviabilizados pelo custo de manutenção e pela ausência de demanda na qual teriam os requisitos mínimos para jogar (ADAMS, 2006). Na década de 80, o *videogame* Atari foi uma plataforma popular, vendendo 30.000 unidades em seu lançamento contra apenas 2.000 unidades do seu concorrente Intellivision (YARUSSO, 2006).

A crescente de recursos computacionais disponíveis em computadores pessoais e *videogames* após os anos 90, permitiu que desenvolvedores criassem novos estilos de jogos que utilizavam de hardware mais específico (ADAMS, 2006). Dentre esses recursos, iniciou-se o uso da rede de computadores para proporcionar a interação entre jogadores em equipamentos distintos (STATISTA, 2018a). Jogos como EA Habitat¹, CipSoft Tibia² e Jajex Runescape³ começaram a utilizar, como requisito obrigatório do jogo, a conexão com a Internet para interagir em um mundo compartilhado com outros jogadores. Tais jogos popularizaram um novo gênero, trazendo inovação tecnológica como complemento a sua jogabilidade, propondo novos desafios aos jogadores ao jogar com centenas ou milhares

¹EA Habitat: <http://www.mobygames.com/game/c64/habitat/credits>

²CipSoft Tibia: <http://www.tibia.com/>

³Jajex Runescape: <https://www.runescape.com>

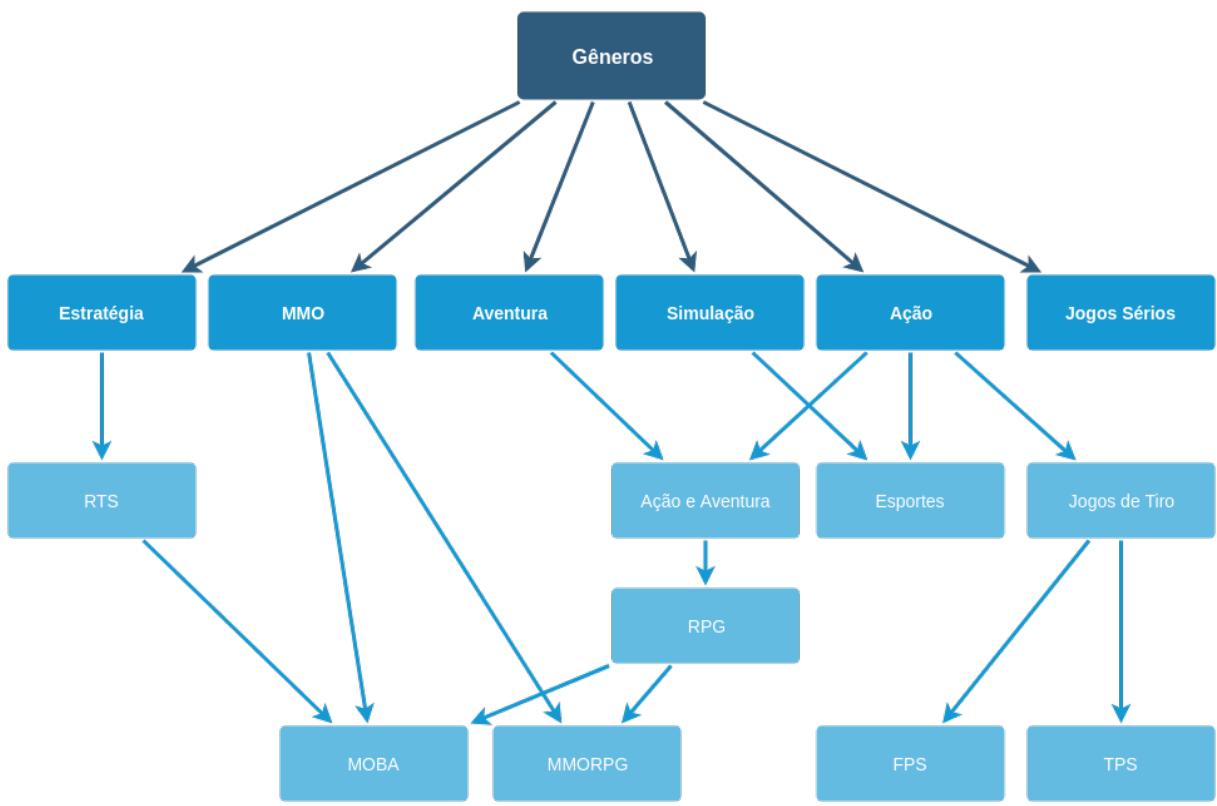
de jogadores simultâneos (GUINNESS, 2013; HUANG; YE; CHENG, 2004), criando o gênero de jogos *Massively Multiplayer Online* (MMO).

Muitos outros jogos do gênero foram criados devido ao aumento da quantidade de público alvo e viabilidade de novas tecnologias, na qual fomentaram a evolução dos jogos MMORPG. Nesse sentido, as redes de computadores realizaram papéis de impulsionadores para várias categorias de jogos, que antes não eram possíveis por conta da limitação de comunicação entre computadores. Sendo assim, torna-se necessário ter uma visão geral das principais categorias de jogos eletrônicos com relação as tecnologias empregadas do ponto de vista de redes de computadores.

2.1.1 Árvore de gêneros de jogos eletrônicos

A classificação por gênero é uma ferramenta tradicional para auxiliar a fácil identificação de características de alguma literatura, arte e outras mídias. Dentro de jogos eletrônicos, o gênero permite que jogadores busquem jogos com características conforme o seu interesse (CLARKE; LEE; CLARK, 2015). A árvore de gêneros para jogos eletrônicos pode ser visualizada na Figura 2.1.

Figura 2.1: Árvore de gêneros de jogos eletrônicos simplificada.



Adaptado de: (ADAMS, 2006)

Um gênero de jogo eletrônico é uma categoria específica para agrupar estilos de jogabilidade parecidos. Porém, um gênero não define de forma explícita o conteúdo expresso em algum jogo eletrônico, mas sim um desafio comum presente no jogo analisado (ADAMS, 2006; HANNA, 2015). Na Figura 2.1 foram apresentados os principais gêneros, na qual podem ser brevemente descritos como:

- Estratégia: São focados em uma jogabilidade que exija habilidades de raciocínio e/ou gerenciamento de recurso. Neste gênero, o jogador tem uma boa visualização do mundo, controlando indiretamente as suas tropas disponíveis (ROLLINGS; ADAMS, 2003). É comum encontrar jogos que disponibilizam algum modo de competição entre jogadores usando *Local Area Network* (LAN), *Wide Area Network* (WAN) ou *Peer-to-Peer* (P2P) (ADAMS, 2006).
 - *Real-Time Strategy* (RTS): Utiliza as características de um jogo de estratégia, porém esse subgênero indica que as ações dos jogadores são concorrentes. É comum encontrar modos de jogo competitivo utilizando LAN neste gênero (ADAMS, 2006).
- MMO: Preza pela interação com outros jogadores em um mundo compartilhado (ADAMS, 2006). SecondLife⁴ é um jogo focado na interação social, com artifícios de comércio e relacionamentos em um mundo fictício criado pela comunidade (KLEINA, 2018). Em grande parte, esses jogos utilizam tecnologia WAN e Cliente/Servidor (C/S).
 - *Multiplayer Online Battle Arena* (MOBA): Coloca um número fixo de jogadores separados em dois times, no qual o time com melhor estratégia de posicionamento e gerenciamento de recursos em equipe ganha a partida. Jogos MOBA perdem algumas características breves do gênero *Role-Playing Game* (RPG), deixando de lado a interpretação e contextualização de um mundo, fixando-se somente em um combate estratégico e momentâneo (distribuído em partidas atômicas) entre as equipes, carregando consigo somente as características de comércio e comunidade dos jogos MMO (ADAMS, 2006). Tal subgênero é popularmente conhecido pelos títulos Blizzard Dota 2⁵ e Riot League of Legends⁶. O jogo League of Legends obteve 100 milhões de usuários ativos em

⁴SecondLife: <https://www.secondlife.com/>

⁵Blizzard Dota 2: <http://br.dota2.com/>

⁶Riot League of Legends: <https://br.leagueoflegends.com/pt/>

2016 sendo o jogo mais jogado do mundo neste ano (STATISTA, 2018c), tendo torneios nacionais e internacionais (SPORTV, 2018). É popular nesse subgênero utilizar tecnologias como LAN, P2P e WAN.

- MMORPG: Herda características dos gêneros ação e aventura, RPG, e MMO diretamente. Nesse gênero é permitido interações em um mundo compartilhado aos jogadores, na qual a interação entre outros jogadores (herdado dos jogos MMO), com o mundo (herdado dos jogos de ação e aventura) e com objetivos guiados por NPCs (herdados de jogos RPG) se faz como desafio e objetivo do jogo (ADAMS, 2006). Um título popular para esse gênero é o jogo Blizzard Word of WarCraft, o qual tem o título de maior comunidade de jogo em um único serviço do mundo⁷. A grande parte dos jogos MMORPG utilizam tecnologia WAN e C/S.
- Aventura: Caracterizado por desafios envolvendo ações com diversos NPCs ou com o ambiente a fim de solucionar desafios (ADAMS, 2006). A grande parte desses jogos utilizam arquiteturas WAN, P2P ou LAN.
 - Ação e Aventura: Herda características da categoria de Aventura. O jogador é imerso em um mundo para interagir com o ambiente e com NPCs, além de se preocupar com a movimentação no cenário (ADAMS, 2006). Um título popularmente conhecido desse gênero é a série de jogos nomeada Nintendo The Legend of Zelda⁸. É comum nesses jogos encontrar tecnologia LAN ou P2P para modo de jogo cooperativo.
- Simulação: Caracterizados por abordar temas da realidade. São comuns jogos de construção e gerenciamento, animais de estimação, vida social e simulação de veículos (ADAMS, 2006). A grande parte desses jogos não permite a interação entre os demais jogadores. É popular encontrar serviços como *ranking*, loja e janela de notícias utilizando C/S.
 - Esportes: Trata somente da simulação de esportes, nos quais os times podem ser controlados tanto por uma inteligência artificial quanto por jogadores *online* (ADAMS, 2006). O jogo FIFA⁹ é um título popular nesse segmento. É comum encontrar tecnologias P2P e LAN.

⁷Blizzard Word of WarCraft: <https://worldofwarcraft.com/pt-br/>

⁸Nintendo The Legend of Zelda: <https://www.zelda.com/>

⁹FIFA: <https://www.easports.com/br/fifa>

- Ação: Preza pela habilidade de coordenação motora e reflexos do jogador, para tomar uma ação a fim de superar seus desafios no cenário alcançando algum objetivo (ADAMS, 2006). É comum encontrar tecnologias LAN, P2P, WAN e C/S.
 - Jogos de Tiro: Usa um número finito de armas para executar ações a distância. O posicionamento, movimentação estratégica e mira são fatores de desafio ao jogador nesse gênero (ADAMS, 2006). É comum encontrar tecnologias LAN, P2P ou WAN.
 - * *First-Person Shooter* (FPS): Utiliza o método de gravação conhecido como *Point of View* (POV). Nesse método, o modo de exibição do mundo é dado como a visão de um personagem do jogo, na qual o jogador tem visão pelo próprio personagem (HANNA, 2015; ADAMS, 2006). É comum encontrar tecnologias LAN, P2P ou WAN.
 - * *Third-person Shooter* (TPS): Diferente dos jogos FPS, os jogos TPS utilizam câmeras soltas no cenário no qual o jogador é visível na cena exibida (HANNA, 2015; ADAMS, 2006). É comum encontrar tecnologias LAN, P2P ou WAN.
- Jogos sérios: Tem como objetivo transmitir um conteúdo educacional (HANNA, 2015). O jogo Sherlock Dengue 8 (BUCHINGER, 2014) é um título desenvolvido com o objetivo de conscientizar os problemas e a prevenção da Dengue no Brasil. É comum encontrar tecnologias LAN, P2P, WAN e C/S.

A árvore de gêneros guia tanto os usuários finais para classificar jogos que lhe agradem, quanto desenvolvedores a fim de seguir tendências de mercado pelas características do gênero. Dessa forma, encontra-se um padrão nos jogos, o qual inclusive orienta o desenvolvimento das arquiteturas de tais jogos (HANNA, 2015).

Dentre vários gêneros, alguns utilizam popularmente algumas tecnologias de rede. A Tabela 2.1 indica a correlação de tecnologias de rede comuns nos gêneros, além de trazer a correlação de número de jogadores por gênero de jogo em seus serviços. Essa correlação é importante para identificar as características referentes a jogabilidade com multijogadores (HANNA, 2015).

Dentre todos os jogos, o gênero MMORPG é o mais impactado pela quantidade de jogadores(KIM; KIM; PARK, 2008), visível na Tabela 2.1. Nesse sentido, as

Tabela 2.1: Tipos de comunicação e quantidade de jogadores impactados por ocorrências conforme o seu gênero.

	LAN	WAN	P2P	C/S	Jogadores Impactados por falhas
ESTRATÉGIA	✓	✓	✓		até 10 (MICROSOFT, 2005)
RTS	✓	✓		✓	até 10 (BLIZZARD, 2010)
MOBA	✓	✓	✓	✓	até 10 (RIOT, 2009)
MMO		✓		✓	mais que 1000 (JAJEX, 2018)
MMORPG		✓		✓	mais que 1000 (JAJEX, 2018)
AVENTURA	✓	✓	✓	✓	até 100 (MOJANG, 2009)
AÇÃO	✓	✓	✓	✓	até 10 (MDHR, 2017)
AÇÃO E AVENTURA	✓	✓	✓	✓	até 10 (MDHR, 2017)
SIMULAÇÃO	✓	✓	✓	✓	até 10 (SCS, 2016)
ESPORTES	✓	✓	✓	✓	até 10 (EA, 2018)
FPS	✓	✓	✓	✓	até 100 (DICE, 2013)
TPS	✓	✓	✓	✓	até 100 (DICE, 2013)
JOGOS SÉRIOS	✓	✓	✓	✓	até 10 (BUCHINGER, 2014)

Fonte: O próprio autor.

arquiteturas do serviço e cliente se tornam um ponto crítico no desenvolvimento a fim de suportar a carga necessária ao desenho do jogo. Por esse motivo, a escolha por abordar o gênero MMORPG se torna interessante do ponto de vista computacional, visto que o impacto de falhas em tais serviços impactam um vasto número de jogadores, ao comparar com outros gêneros.

2.2 MMORPG

Jogos MMORPG são utilizados como negócio viável e lucrativo, sendo que a experiência de jogabilidade na qual o usuário final será submetido é um fator crítico para o sucesso. O mercado de jogos MMORPG vem crescendo desde 2012 (BILTON, 2011), sendo no ano de 2017 um dos mais lucrativos (STATISTA, 2018b). A projeção deste mercado para 2018 era de mais de 30 bilhões de dólares americanos em circulação sobre esta categoria de jogos (STATISTA, 2017), porém foi ultrapassado no ano de 2017 com 30,7 bilhões de dólares (STATISTA, 2018b).

MMORPG são jogos de interpretação de papéis massivos, originados dos gêneros RPG. A principal característica desse estilo de jogo é a comunicação e representação virtual de um mundo fantasia, no qual cada jogador pode interagir com objetos virtuais compartilhados ou tomar ações sobre outros jogadores em tempo real, tendo como principais objetivos a resolução de problemas conforme a sua regra de *design*, o desenvolvimento

do personagem e a interação entre os jogadores(HANNA, 2015).

Um jogo MMORPG é arquitetado em três partes (KIM; KIM; PARK, 2008):

- **Cliente:** Aplicação que realiza as requisições com a interface do serviço, exibindo o estado de jogo de forma imersiva ao jogador. Este tema é melhor abordado na Subseção 2.4.1.
- **Servidor:** O computador, ou o conjunto de computadores, que recebe as requisições do cliente a fim de serem processadas pelo Serviço.
- **Serviço:** Implementa as regras de negócio e requisitos do jogo. O serviço disponibiliza uma interface com ações possíveis ao cliente sobre algum protocolo de rede. Este tema é abordado na Subseção 2.4.2.

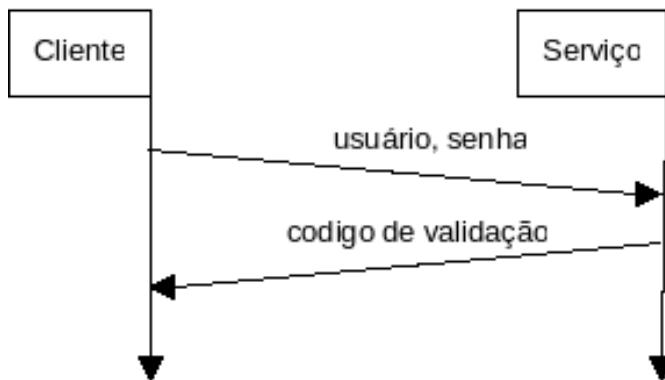
Jogos MMORPG trabalham como serviço. Por este motivo, o modelo de negócios de tais jogos, do ponto de vista de redes de computador, é suscetível a perda financeira em casos de negação de serviço (HUANG; YE; CHENG, 2004). A maioria dos jogos MMORPG disponíveis no mercado estão implementados sobre uma arquitetura que executa sobre diversos servidores (CLARKE-WILLSON, 2017), nos quais o desempenho destes servidores influencia tanto na experiência de jogabilidade do usuário final, quanto no custo de manutenção destes serviços (HUANG; YE; CHENG, 2004). Por sua vez, o Cliente é implementado em algum ambiente convencional a jogos, como motores gráficos, bibliotecas gráficas ou sobre alguma outra plataforma, como web. Nesse sentido, torna-se necessário descrever as características de jogabilidade de jogos MMORPG a fim de melhor compreender o funcionamento da arquitetura de um cliente e de um serviço para jogos MMORPG.

2.3 Jogabilidade de jogos MMORPG

É comum serviços MMORPG terem regras de negócio parecidas, visto que pertencem ao mesmo gênero. Dessa forma, explicar regras de negócio recorrentes neste gênero facilita a compreensão básica de forma genérica de um jogo MMORPG e auxilia a compreensão do modelo computacional implementado nestes serviços. Deste modo, torna-se necessário definir algumas funcionalidades básicas que estão dentro do contexto de jogabilidade de jogos MMORPG para melhor compreensão de sua arquitetura em seções futuras.

O sistema de autenticação é o sistema que geralmente inicia o cliente de algum jogo MMORPG (SALZ, 2016a; RUDDY, 2011). Este sistema é implementado via protocolo *Hypertext Transfer Protocol* (HTTP) ou RPC, a fim de disponibilizar um código para validar todas as futuras ações da seção do usuário. Este sistema pode ser visualizado de forma macro na Figura 2.2.

Figura 2.2: Sistema de autenticação para jogos MMORPG.



Fonte: Adaptado de (THOMPSON, 2008)

O sistema de autenticação visualizado na Figura 2.2 é popularmente conhecido, porém não o único. O jogo *Realm of the Mad God*¹⁰ é um exemplo na qual não exige autenticação do usuário, entretanto ele não armazena o progresso do jogador caso o jogador não efetue a autenticação.

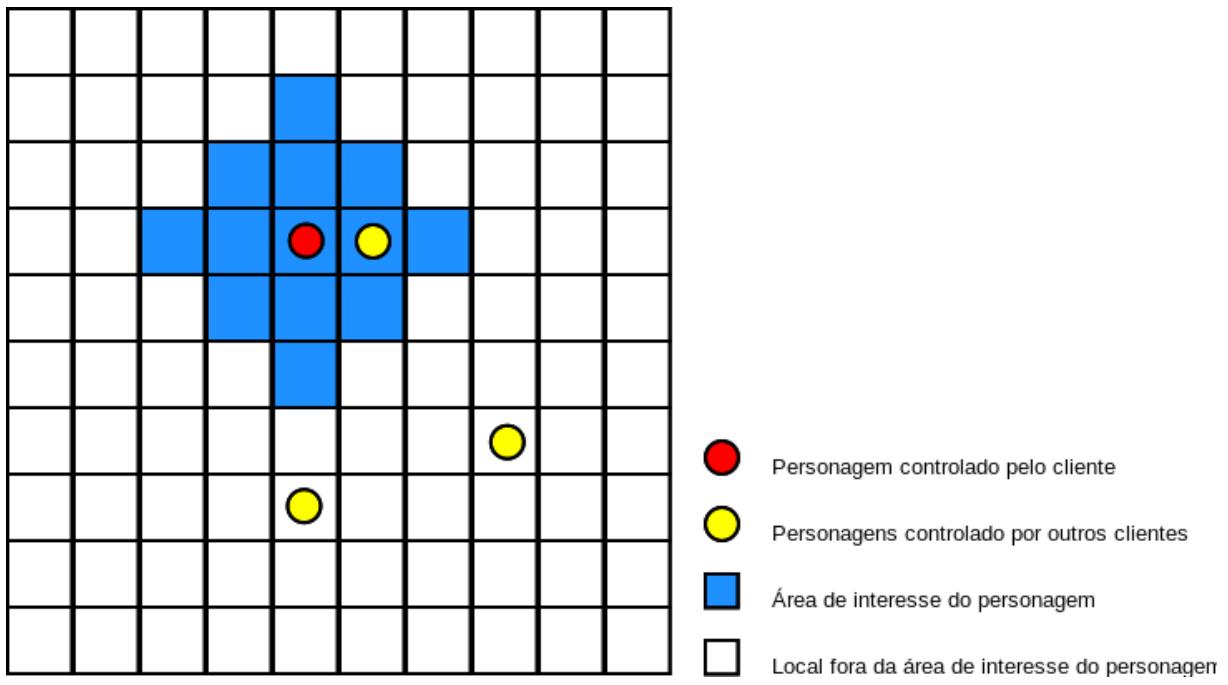
Esse método de autenticação é definido pela RFC7519 (JONES et al., 2015), com a tecnologia *JSON Web Token* (JWT). O código de validação repassado é auditado em qualquer serviço pertencente ao jogo, visto que ele foi assinado pelo sistema de autenticação do serviço (IKEM, 2018).

Após a autenticação, é comum existir um sistema para seleção de personagem, caso o jogo seja desenhado com este objetivo. Efetuada a seleção ou criação de um personagem, este será imerso no mundo compartilhado do jogo com os demais jogadores (RUDDY, 2011). Nos jogos MMORPG é comum a restrição da visão do personagem (Figura 2.3), ora pelas características de jogabilidade do gênero MMORPG ora por motivos de desempenho e otimização. Como o jogador não precisa obter dados de regiões que não estão em sua área de interesse, não há necessidade da transmissão de informações dos objetos que estão fora desse contexto (SALZ, 2016a).

Um exemplo de área de interesse pode ser visualizado na Figura 2.3, na qual

¹⁰Realm of the Mad God: <https://www.realmofthemadgod.com/>

Figura 2.3: Área de interesse com base na proximidade de um jogador.



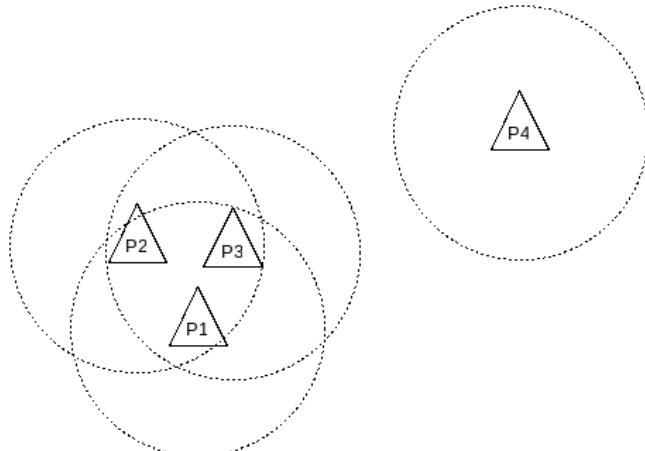
o personagem selecionado (destacado em vermelho) tem uma área de interesse de baixa distância (SALZ, 2016a), sendo que o jogador não tem informações dos demais objetos e jogadores fora de sua área de interesse a nível de rede. Esta característica impede trapaças (visto que o cliente tem informações que só estão contidas em sua área de interesse) e reduz a carga de cada atualização a cada cliente (SALZ, 2016a).

Após o jogador estar com o controle do personagem no ambiente, é comum em tais jogos que ele possa realizar determinadas ações. Nesse sentido, algumas ações comuns dentro do ambiente de um jogo MMORPG (JON, 2010) são:

- Enviar e receber mensagem no *chat*;
- Mover-se pelo ambiente;
- Interagir com outros jogadores, NPCs ou objetos fixos do ambiente; e
- Obter itens do ambiente.

O envio e recepção de mensagens do *chat* é dado com o contexto do posicionamento do personagem (SALZ, 2016a), visível na Figura 2.4. Somente outros personagens dentro de um raio podem receber alguma mensagem emitida pelo jogador P_n .

Figura 2.4: *Chat* baseado em contexto de posicionamento, utilizando Distância Euclidiana.



Fonte: Adaptado de (SALZ, 2016a)

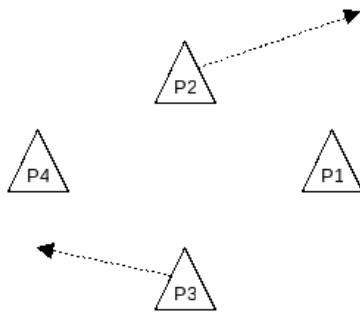
Essa distância (Figura 2.4) pode ser calculada utilizando Distância Euclidiana (DEZA, 2009), na qual a distância entre dois personagens pode ser calculada pela equação $d(p, q) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2}$. Para diminuir a complexidade das comparações, a fim de decidir quais personagens P_n devem receber a mensagem, é comum utilizar técnicas de divisão de área utilizando algoritmos como *Quadtree* ou *Octree* (LENGYEL, 2011), subdividindo os quadrantes de uma região do ambiente do jogo a fim de facilitar a consulta de quais personagens estão em determinada área deste ambiente.

Nesse sentido, a Figura 2.4 mostra a interseção entre o raio de quatro personagens. Nesse exemplo, mostra-se visível que as mensagens de P_1 devem ser visíveis a P_2 e P_3 , mas não a P_4 , caso seja utilizado a Distância Euclidiana como regra de distância.

O sistema de movimento pelo ambiente do jogo possibilita que cada jogador movimente seu personagem pelo ambiente a fim de explorá-lo. Dessa maneira, este é um sistema crítico para um jogo MMORPG, visto que o posicionamento de objetos é utilizado para diversas consultas de proximidade, além de necessitar uma frequência de atualização constante do posicionamento dos jogadores a fim de manter a integridade de funcionamento do serviço de ambiente do jogo (SALZ, 2016a). Um exemplo de movimentação no ambiente pode ser visualizado na Figura 2.5.

Com a movimentação descrita na Figura 2.5, o personagem torna-se livre a explorar o ambiente seguindo as regras de negócio do jogo, permitindo a interação com o ambiente, objetos posicionados no cenário ou outros jogadores. Dessa forma, a interação com estes elementos também é afetada pela área de interesse, na qual o personagem terá

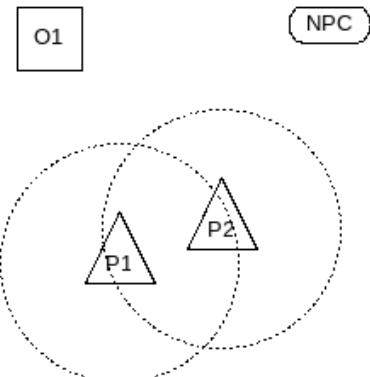
Figura 2.5: Personagens e os seus pontos de destino.



Fonte: O próprio autor.

um raio limitante para cada tipo de interação no ambiente. Um exemplo de ambiente com personagens (P_1 e P_2), objetos (O_1) e NPCs (NPC somente) pode ser visualizado na Figura 2.6 (SALZ, 2016a).

Figura 2.6: Personagens, objetos e NPCs no ambiente.



Fonte: O próprio autor.

Na Figura 2.6 fica visível que a área de interesse pode ser um fator limitante conforme a operação com jogadores, objetos e NPCs (SALZ, 2016a). Dessa forma, o desempenho necessário para a execução dessas tarefas está tanto atrelado ao conjunto de algoritmos utilizado, quanto aos recursos computacionais gastos para tais operações.

Essas operações precisam de desempenho para não causar frustração ao jogador final (HOWARD et al., 2014). Nesse sentido, torna-se necessário conhecer os problemas computacionais recorrentes com relação aos serviços de jogos MMORPG.

2.4 Problemas em jogos MMORPG

Uma métrica popular para mensurar o desempenho de um serviço MMORPG é o número de conexões (HUANG; YE; CHENG, 2004) simultâneas suportadas. Em geral, caso o

serviço ultrapasse o limite para o qual foi projetado, diversas falhas de conexão, problemas de lentidão ou dessincronização com o cliente podem ocorrer. Neste contexto, as ocorrências comuns são (HUANG; YE; CHENG, 2004):

- **Longo tempo de resposta aos clientes:** implica em uma qualidade insatisfatória de jogabilidade ao usuário ou até mesmo impossibilitando o uso do serviço.
- **Dessincronização com os clientes:** realiza reversão na aplicação. Reversão é definida pela situação na qual uma requisição é solicitada ao servidor, um pré-processamento aparente é executado e essa requisição é negada, sendo necessário desfazer o pré-processamento aparente realizado ao cliente.
- **Problemas internos ao serviço:** podem estar relacionados a diversos outros erros internos de implementação ou a capacidade de recurso computacional (*e.g.*, sobrecarga no banco de dados, gerenciamento lento do espaço ou inconsistências dentro do jogo perante a regra de negócios).
- **Falha de conexão entre o cliente e o serviço:** causa a negação de serviço ao usuário final.

Existem algumas causas comuns para as ocorrências descritas (HUANG; YE; CHENG, 2004):

- **Baixo poder computacional do servidor:** poder computacional abaixo do necessário para a qualidade de experiência de jogabilidade do usuário final desejada.
- **Complexidade de algoritmos:** o serviço usa algoritmos de alta complexidade ou regras de negócios que demandam por um algoritmo complexo.
- **Limitado pela própria arquitetura:** está limitado diretamente pelo número de conexões, não suportando a carga recebida.
- **Limitado pela rede:** a quantidade de requisições não é suportada pelo meio físico na qual a arquitetura está implantada.

Tais ocorrências estão diretamente correlacionadas a carga na qual tais serviços estão submetidos. Esta carga pode ser amenizada utilizando técnicas de provisionamento

de recursos e balanceamento de carga (HUANG; YE; CHENG, 2004), entretanto o serviço pode estar limitado pela sua arquitetura.

A área de desenvolvimento web compartilha várias ocorrências comuns geradas por sobrecarga do serviço (KHAZAEI et al., 2016). Em desenvolvimento web é comum utilizar a abordagem de microsserviços para resolver o problema de sobrecarga, modularizando o funcionamento em módulos menores. Da mesma forma, faz sentido modularizar um serviço MMORPG em microsserviços para suportar cargas maiores e diminuir o custo de manutenção (VILLAMIZAR et al., 2016).

Do ponto de vista da arquitetura de computadores, as operações existentes em um jogo MMORPG seguem um padrão de interação com o mundo, criar, excluir ou manipular objetos deste mundo. Para suprir o desenvolvimento de tais sistemas, se faz necessário compreender os padrões de desenvolvimento de tais arquiteturas, os quais implementam as operações básicas de interação com o mundo.

2.4.1 Arquitetura de Clientes MMORPG

A arquitetura de um cliente MMORPG é um aspecto fundamental, mas não único, para o sucesso de um jogo deste gênero. O seu funcionamento é totalmente visível ao usuário final e tem o principal objetivo de exibir o estado do mundo de forma gráfica ao usuário (SALZ, 2016a). Um exemplo de cliente MMORPG é o jogo Sandbox-Interactive Albion¹¹, que pode ser visualizado na Figura 2.7.

A Figura 2.7 representa na prática, como será exibido ao usuário final o ambiente do jogo. O modelo teórico apresentado no cliente leva como base o campo de visão do jogador, no qual pode ser visualizado na Figura 2.3.

Do ponto de vista de computação gráfica, um cenário 3D pode ser visualizado como uma árvore. Utilizar árvores para descrever um cenário ajuda tanto no formato de armazenamento em disco para leitura facilitada (*e.g.*, *JavaScript Object Notation* (JSON), *Extensible Markup Language* (XML), etc.), operações de inserção e exclusão, organização do projeto e redução da complexidade utilizando transformações lineares em sistemas gráficos como *OpenGL* e *DirectX* (LENGYEL, 2011). Esse modelo é amplamente utilizado em motores gráficos, na qual pode ser encontrado em motores gráficos populares como

¹¹Sandbox-Interactive Albion: <https://albiononline.com/en/home>

Figura 2.7: Exemplo de Cliente MMORPG (Sandbox-Interactive Albion).



Fonte: (SALZ, 2016a)

Godot, *Unity3D* e *Unreal 3*. A Figura 2.8 ilustra um exemplo, na qual exibe a árvore de um cenário na *Integrated Development Environment* (IDE) do motor gráfico *Godot*.

Figura 2.8: *Scene tree view* no motor gráfico Godot.

Fonte: O próprio autor.

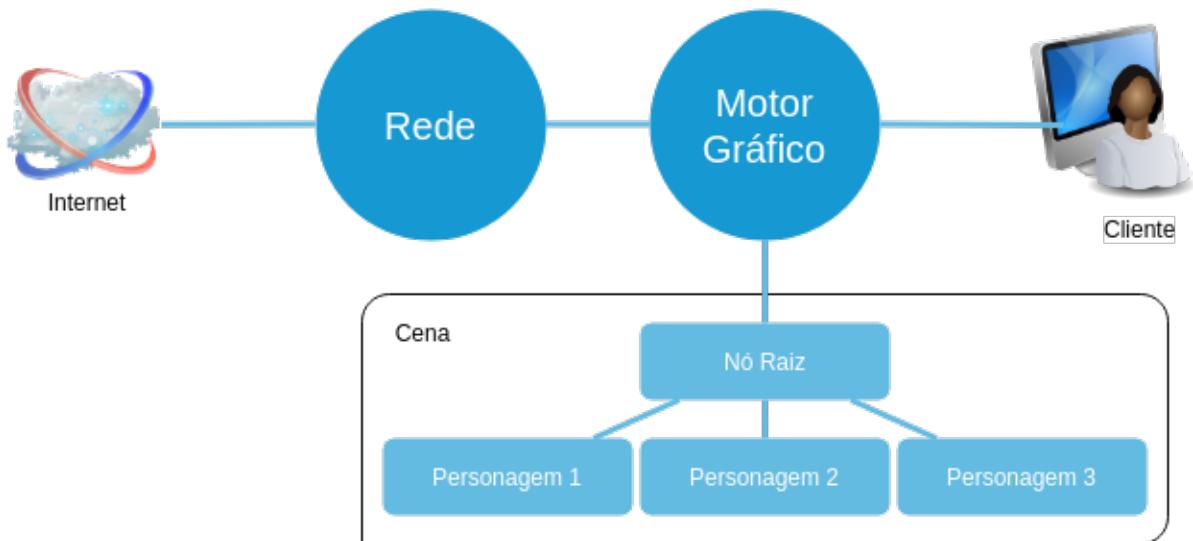
Dentro dessa árvore, cada nodo tem uma funcionalidade específica. Essas funcionalidades variam a cada motor gráfico, podendo existir nodos focados à parte física, renderização ou controles (LINIETSKY, 2018). Em um jogo MMORPG, o seu serviço

será responsável por enviar atualizações dos parâmetros aos nodos frequentemente e o cliente será responsável por realizar chamadas remotas a fim de descrever as ações a qual o jogador aplicou sobre seu personagem (Exit Games, 2017).

Do ponto de vista da rede de computadores, a arquitetura de um cliente de jogo MMORPG deve suportar consultas e chamadas de métodos remotos em um serviço (SALZ, 2016a). Um cliente, para um jogo MMORPG, pode seguir o estilo de arquitetura *Representational State Transfer* (REST), porém não obrigatoriamente sobre o protocolo HTTP, mas sim usando algum protocolo RPC sobre o protocolo *Transmission Control Protocol* (TCP) ou *User Datagram Protocol* (UDP) (SALZ, 2016a; CLARKE-WILLSON, 2017). Essa comunicação é realizada pelo módulo *Network*, presente em um cliente MMORPG.

O módulo de *Network* implementado em um cliente de jogo MMORPG é responsável por realizar as requisições RPC conforme as ações realizadas pelo jogador. Este também é responsável por aplicar os parâmetros na *Scene Tree* ou chamar métodos remotos no cliente por ordens do serviço. Além disso, o módulo *Network* possui uma *Thread* dedicada ao gerenciamento de entrada e saída em relação ao serviço (SALZ, 2016a). Os principais módulos podem ser observados na Figura 2.9.

Figura 2.9: Modelo de um cliente genérico.



Adaptado de: (ZELESKO; CHERITON, 1996; FARBER, 2002)

A Figura 2.9 refere-se a uma visão macro de um cliente MMORPG, na qual é possível ver o ator *jogador* o qual pode executar ações sobre seu personagem por meio do motor gráfico. Por sua vez, existe uma entrada de dados a mais comparado a esquemas de jogos *offline*. Neste caso, o módulo *Network* será igualmente uma entrada de dados,

a qual poderá manipular a cena do motor gráfico (FARBER, 2002). Para facilitar o desenvolvimento, a aplicação de cliente é dividida em diversos módulos, entretanto são relevantes ao atual trabalho (SALZ, 2016a):

- **Engine:** É o conjunto que aplicará regras sobre os objetos na *Scene Tree*, receberá entradas do usuário e exibirá a *Scene Tree* de forma imersiva. Unity3D¹² e GodotEngine¹³ são exemplos de *engines*.
- **Network:** É o módulo responsável pela comunicação entre o serviço e o cliente, a fim de requisitar chamadas de métodos ou obter informações do servidor para sincronizar os estados de jogo.

Utilizando esses dois módulos é possível sincronizar os estados de jogo e exibi-los ao jogador. Entretanto, faz-se necessário compreender o funcionamento do serviço a fim de escolher um protocolo padrão para essa sincronização.

2.4.2 Arquitetura de Microsserviços

Entende-se por microsserviço, aplicações que executam operações menores de um macrosserviço, da melhor forma possível (CLARKE-WILLSON, 2017; NEWMAN, 2015). O objetivo de uma arquitetura de microsserviços é funcionar separadamente de forma autônoma, contendo baixo acoplamento (NEWMAN, 2015). Seu funcionamento deve ser desenhado para permitir alinhamentos de alta coesão e baixo acoplamento entre os demais microsserviços existentes em um macrosserviço (ACEVEDO; JORGE; PATIÑO, 2017).

Arquiteturas de microsserviços iniciam uma nova linha de desenvolvimento de aplicações preparadas para executar sobre nuvens computacionais, promovendo maior flexibilidade, escalabilidade, gerenciamento e desempenho, sendo a principal escolha de arquitetura de grandes empresas como Amazon, Netflix e LinkedIn (KHAZAEI et al., 2016; VILLAMIZAR et al., 2016). Um microsserviço é definido pelas seguintes características (ACEVEDO; JORGE; PATIÑO, 2017):

- Deve possibilitar a implementação como uma peça individual do macrosserviço.
- Deve funcionar individualmente.

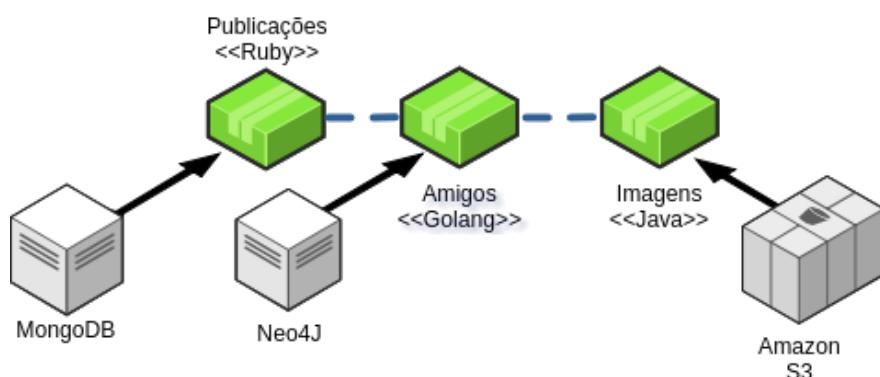
¹²Unity3D: <https://www.unity3d.com>

¹³GodotEngine: <https://www.godotengine.org>

- Cada microsserviço deve ter uma interface. Essa interface deve ser o suficiente para utilizar o microsserviço.
- A interface deve estar disponível na rede para chamada de processamento remoto ou consulta de dados.
- O microsserviço pode ser utilizado por qualquer linguagem de programação e/ou plataforma.
- O microsserviço deve executar com as dependências mínimas.
- Ao agregar vários microsserviços, o macrosserviço resultante poderá prover funcionalidades complexas.

O microsserviço deverá ser uma entidade separada. A entidade deve ser implantada sobre um sistema isolado (*e.g.*, Docker¹⁴, *Virtual Machines* (VMs), *etc.*). Toda a comunicação entre os microsserviços de um macrosserviço será executada sobre a rede, a fim de reforçar a separação entre cada serviço. As chamadas pela rede com o cliente ou entre os microsserviços será executada através de uma *Application Programming Interface* (API), permitindo a liberdade de tecnologia em que cada microsserviço será implementado (NEWMAN, 2015). Isso permite que o sistema suporte tecnologias distintas que melhor resolvam os problemas relacionados ao contexto deste microsserviço. Isso pode ser visualizado na Figura 2.10.

Figura 2.10: Microsserviços podem ter diferentes tecnologias.



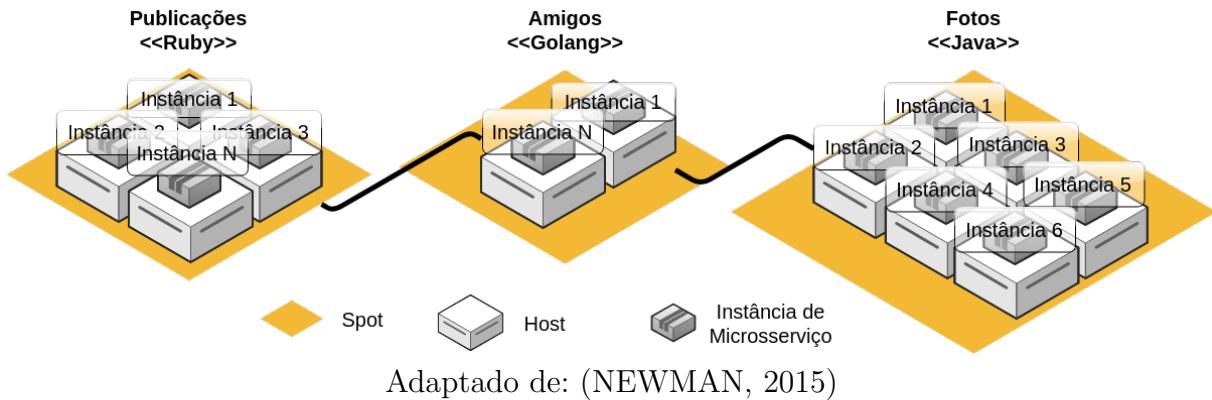
Adaptado de: (NEWMAN, 2015)

Uma arquitetura de microsserviços é escalável, como visível na Figura 2.11. A arquitetura permite o aumento do número de microsserviços sob demanda para suprir

¹⁴Docker: <https://www.docker.com/>

a necessidade de escalabilidade. Este modelo computacional obtém maior desempenho, principalmente se executar sobre plataformas de computação elástica, na qual o orquestrador do macrosserviço pode aumentar o número de instâncias conforme a necessidade de requisições (NADAREISHVILI et al., 2016).

Figura 2.11: Microsserviços são escaláveis.



Microsserviços desenvolvidos para web utilizam arquitetura REST baseado sobre o protocolo HTTP. É uma boa prática utilizar o corpo com conteúdo da requisição e resposta no formato JSON nas chamadas a uma API de microsserviço web (NADAREISHVILI et al., 2016). Entretanto, não é uma prática comum para um serviço MMORPG utilizar o protocolo HTTP pela sua elevada carga administrativa na requisição (HUANG; YE; CHENG, 2004). Por esse motivo, torna-se relevante compreender a composição de uma arquitetura com microsserviços para MMORPG, comparando-a a microsserviços web.

2.4.3 Microsserviços para jogos MMORPG

A fim de otimizar o custo operacional das arquiteturas de microsserviços de jogos MMORPG, é incomum a utilização de protocolos *Web* em tais arquiteturas. Por esse motivo, a seção atual mostrará o funcionamento básico do protocolo RPC e a sua utilização para atualização dos parâmetros na *Scene Tree* e o modelo REST (SALZ, 2016a).

Em engenharia de software, é comum a utilização de arquiteturas *Model-View-Controller* (MVC) a fim de organizar o código fonte e prover agilidade de desenvolvimento (CHADWICK; SNYDER; PANDA, 2012; THOMPSON, 2008). A separação de um serviço MMORPG pode ser dada seguindo este padrão de projeto, dividindo-se em três camadas (HUANG; CHEN, 2010):

1. *Model*: Representa qualquer dado presente no jogo (*e.g.*, itens, personagens, NPCs,

objetivos, etc.).

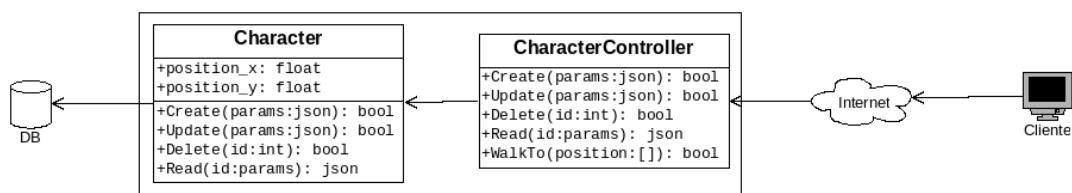
2. *View*: Representa o modo a qual estes dados serão exibidos, do ponto de vista de redes (*e.g.*, O mapa será exibido somente na área de interesse do jogador, no formato JSON).
3. *Controller*: Representa as operações sobre modelos que serão requeridos pelos jogadores (*e.g.*, andar, pegar item, interagir com NPCs, etc.).

Dentro de um *Controller* é implementado operações a qual o cliente pode requirir através de chamadas RPC a fim de manipular ou obter o estado de *Models* da aplicação. Esses métodos padrões seguem o protocolo CRUD, contendo quatro métodos principais para complementar as consultas sobre os *Models* (CHADWICK; SNYDER; PANDA, 2012; THOMPSON, 2008):

1. *Create*: Representa a criação de um novo objeto no banco.
2. *Update*: Representa a atualização de um objeto no banco.
3. *Delete*: Representa a exclusão de um objeto no banco.
4. *Read*: Representa a consulta sobre este objeto no banco.

Para o padrão CRUD, faz-se necessário que os métodos *Read*, *Update* e *Delete* repassem o parâmetro de identificação do objeto a ser consultado (THOMPSON, 2008). Outros argumentos necessários nos métodos *Create* e *Update* são os atributos do objeto, além do seu retorno ser um valor booleano representando se a operação foi bem sucedida (CHADWICK; SNYDER; PANDA, 2012; THOMPSON, 2008). Entretanto, outros métodos mais apropriados ao funcionamento de um *Controller* podem existir. Como exemplo, pode-se visualizar uma interface CRUD na Figura 2.12.

Figura 2.12: Cliente pode realizar requisições CRUD ao serviço.



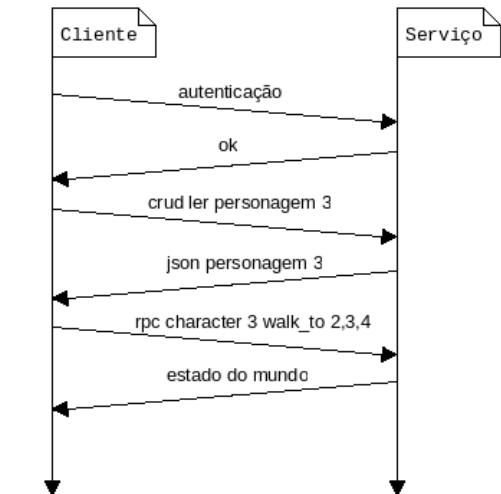
Fonte: Adaptado de (SALZ, 2016a).

Essas operações são executadas utilizando protocolo RPC (SALZ, 2016a). As requisições de métodos remotos são realizadas entre dois processos distintos, a fim de gerar uma computação distribuída (XEROX, 1976). Entretanto, a base do protocolo não é legível como em servidores web que utilizam JSON para transmissão de estrutura de dados, mas sim uma codificação binária nomeado *External Data Representation* (XDR) (Internet Society, 2006).

Uma técnica comum em jogos é a compressão de pacotes utilizando mapeamento *hash* de bytes (THOMPSON, 2008). Tanto o cliente quanto o serviço precisam ter a mesma estrutura de dados. Dessa forma, é possível trocar o nome das funções solicitadas em RPC por poucos bytes para transitar na rede. Já para operações CRUD, pode-se utilizar tanto requisições sobre o protocolo HTTP ou sobre um protocolo otimizado sobre TCP dependendo da necessidade de desempenho (THOMPSON, 2008).

Dado um serviço que não é implementado sobre uma arquitetura de microserviços, a utilização de dois protocolos irá complicar o gerenciamento de threads para responder de duas formas diferentes, entretanto o seu esquema de estados ficará simplificado. Esta simplificação pode ser visualizada na figura 2.13.

Figura 2.13: Diagrama de requisições entre serviço e cliente com operações CRUD e RPC em uma arquitetura monolítica.

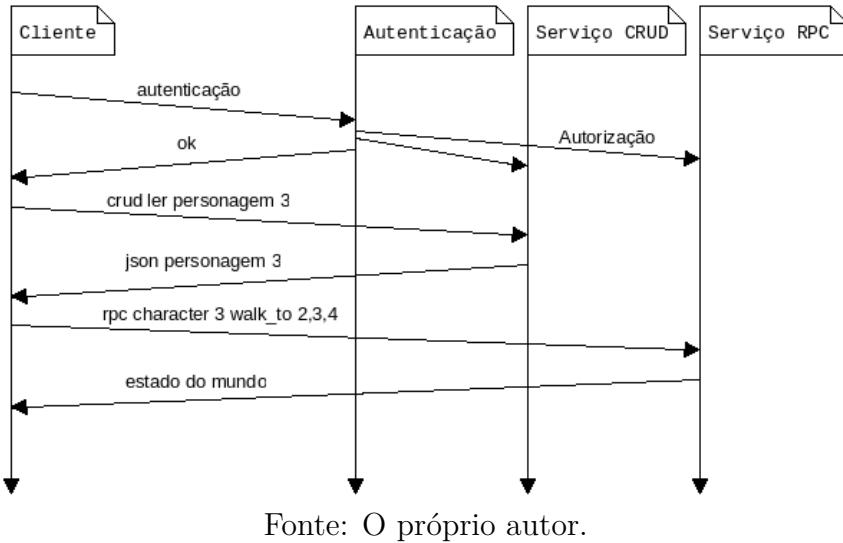


Fonte: Adaptado de (THOMPSON, 2008)

Entretanto, diferente da Figura 2.13, a fim de simplificar a complexidade da implementação e gerenciamento de concorrência no serviço, pode-se implementar utilizando o paradigma de microserviços. Como relatado na Seção 2.4.2, uma arquitetura de microserviços permite múltiplas tecnologias, pois a comunicação entre todos os elementos de um microserviço será pela rede. Por esse motivo, é possível utilizar um serviço web

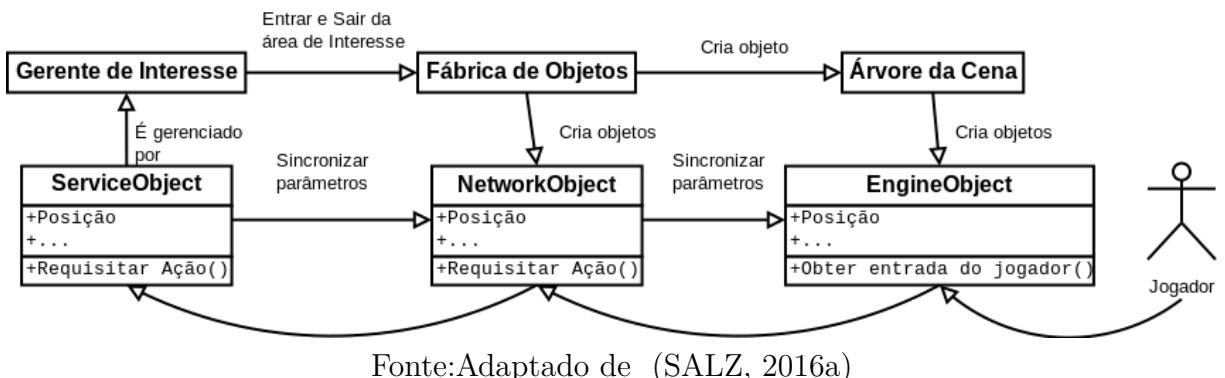
para realizar operações CRUD e um serviço dedicado para realizar operações RPC. Essa arquitetura pode ser melhor visualizada na Figura 2.14.

Figura 2.14: Diagrama de requisições entre serviço e cliente com operações CRUD e RPC em uma arquitetura de microserviços.



Como resultado da integração entre Cliente, Serviço e Motor Gráfico, o resultado final obtido é descrito pelo diagrama presente na Figura 2.15. Tal integração está presente no jogo Sandbox-Interactive Albion¹⁵ (SALZ, 2016a). Percebe-se, neste contexto, que o jogo deverá funcionar sem o motor gráfico, do ponto de vista de redes e estrutura de dados (SALZ, 2016a).

Figura 2.15: Diagrama de integração entre Cliente e Serviço, considerando a *engine* Unity3D.



A Figura 2.15 ilustra a separação da camada de renderização de objetos da árvore da cena, a camada de integração com o cliente e serviço (descrito anteriormente como módulo *Network*) e o serviço. Nesse sentido, a alteração entre clientes e serviços facilita

¹⁵Sandbox-Interactive Albion: <https://albiononline.com/en/home>

um sistema de teste de carga e busca de erros automatizado, facilitando a manutenção e desenvolvimento incremental do serviço (SALZ, 2016a).

Utilizando estes padrões de projeto, algumas arquiteturas tornam-se populares no desenvolvimento de jogos MMORPG. Para este fim, torna-se de interesse a este trabalho realizar um levantamento de algumas arquiteturas comuns em jogos massivos.

2.5 Arquiteturas MMORPG identificadas

Dentre as arquiteturas identificadas, as qualificadas dentro do paradigma de arquiteturas de microsserviços são as arquiteturas Rudy (Subseção 2.5.1), Salz (Subseção 2.5.2) e Willson (Subseção 2.5.3).

Em geral, as arquiteturas de forma genérica contém microsserviços *web* para *E Commerce*, operações CRUD através da web e distribuição de atualizações. Os microsserviços de gerenciamento de jogo, por sua vez, respondem através do protocolo TCP, podendo ser sobre RPC ou protocolos proprietários. A gerência de jogo é a principal mudança entre as arquiteturas, na qual contém abordagens de paralelismo diferentes:

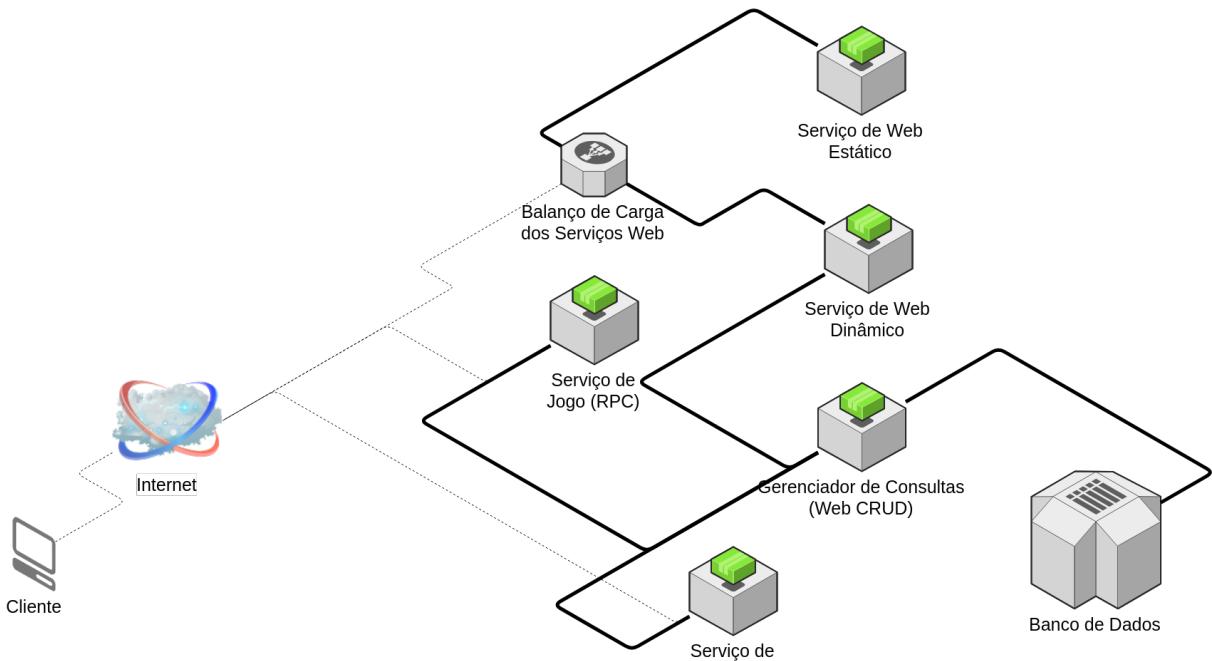
- A arquitetura Rudy (Subseção 2.5.1) utiliza uma abordagem com um número menor de microsserviços, focado em manter serviços para consulta de dados de forma eficiente entre os serviços *web* e o serviços de gerenciamento de jogo.
- A arquitetura Salz (Subseção 2.5.2) aborda um modelo de paralelismo mais complexo comparado a arquitetura Rudy, utilizando diversos microsserviços para funções específicas das funcionalidades do jogo. Dessa forma, o ambiente do jogo torna-se escalável ao número de jogadores, porém a latência tende a aumentar.
- A arquitetura Willson (Subseção 2.5.3) utiliza um modelo intermediário, evitando a divisão em múltiplos serviços para o gerente de jogo e utilizando um modelo de paralelismo próximo a arquitetura Salz.

2.5.1 Arquitetura elaborada por Rudy

A arquitetura Rudy (RUDDY, 2011) tem como objetivo criar múltiplos mundos isolados, na qual cada microsserviço será responsável por um ambiente a qual não compartilha da-

dos com os demais ambientes. Esta é uma característica importante para esta arquitetura, visto que o processamento de ações pelo serviço de jogo não precisa lidar com múltiplos processos (RUDDY, 2011). Esta arquitetura pode ser visualizada na Figura 2.16.

Figura 2.16: Arquitetura Rudy completa.



Adaptado de: (RUDDY, 2011).

No total, seis microsserviços distintos constam na arquitetura Rudy (Figura 2.16) para o seu funcionamento, não sendo necessário o microsserviço de pagamento (utilizado somente para regra de negócios). Os microsserviços que compõem a arquitetura são definidos pelas suas seguintes responsabilidades (RUDDY, 2011):

1. **Serviço de web estático:** Armazena documentos estáticos para o serviço web (*e.g.*, imagens, executáveis do jogo, páginas web fixas, etc). Responde sobre o protocolo HTTP.
2. **Serviço de web dinâmico:** Sistema web para cadastro de contas, guias, informações sobre atualizações, compras e demais demanda de páginas dinâmicas. Responde sobre o protocolo HTTP.
3. **Balanço de carga web:** Realiza a distribuição de carga sobre o *Serviço web estático* e o *Serviço web dinâmico*. Responde sobre o protocolo HTTP.
4. **Serviço de Jogo:** Gerencia um mundo inteiro, em um único serviço. Esta abordagem segregava os jogadores em diversos canais, contendo um número máximo de

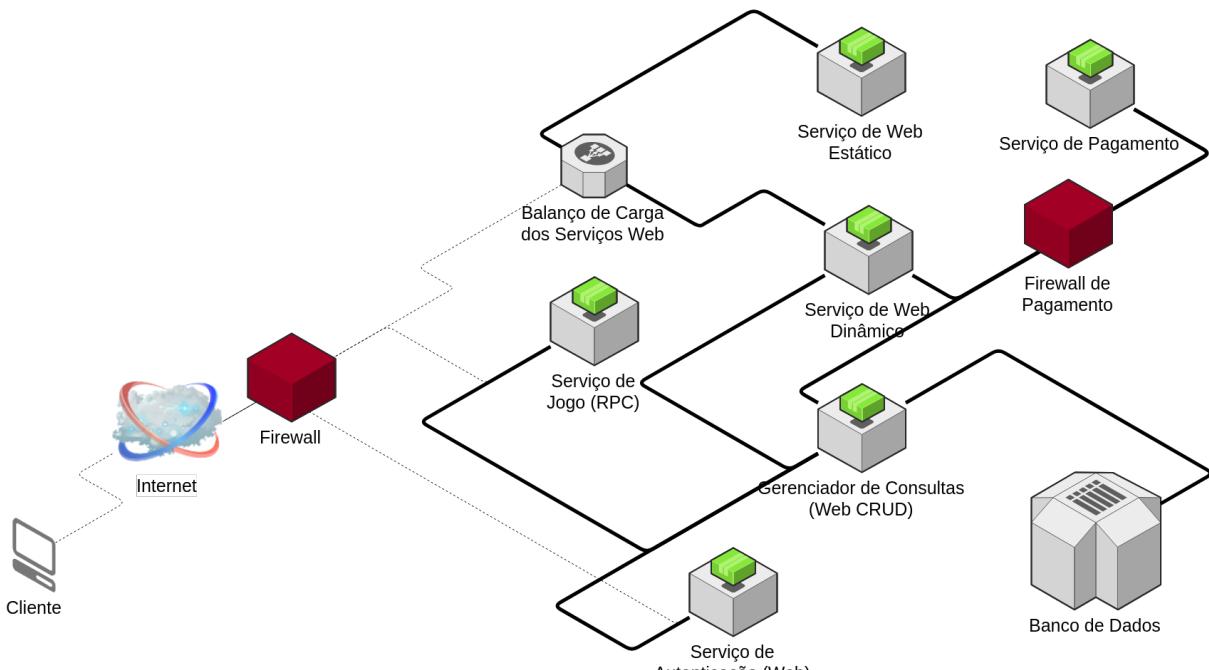
conexões por canal. Cada canal opera sobre uma instância deste microsserviço. Este serviço opera sobre o protocolo RPC.

5. **Serviço de Autenticação:** Gerencia a autenticação das conexões ao *Serviço de Jogo*. Este serviço opera sobre o protocolo RPC.

6. **Gerenciador de Consultas:** Realiza consultas em memória e disco, utilizando vários bancos de dados diferentes, simulando o uso de banco de dados distribuídos, algo complexo de ser implementado utilizando banco de dados SQL (*e.g.*, PostgreSQL¹⁶, MySQL¹⁷, etc). Este serviço opera sobre o protocolo HTTP.

No contexto do atual trabalho, o serviço de pagamento será ignorado, visto que ele não serve para o funcionamento básico do serviço. Dentre todos os microsserviços, o usuário só tem acesso ao serviço de balanço de carga pelo protocolo HTTP e o serviço de jogo sobre o protocolo RPC, tendo os demais serviços protegidos por um *firewall* (RUDDY, 2011). A proteção do *firewall* é aplicada no serviço de pagamento e no ponto de acesso ao serviço, podendo ser visualizada na Figura 2.17.

Figura 2.17: Arquitetura Rudy completa com *firewall*.



Adaptado de: (RUDDY, 2011).

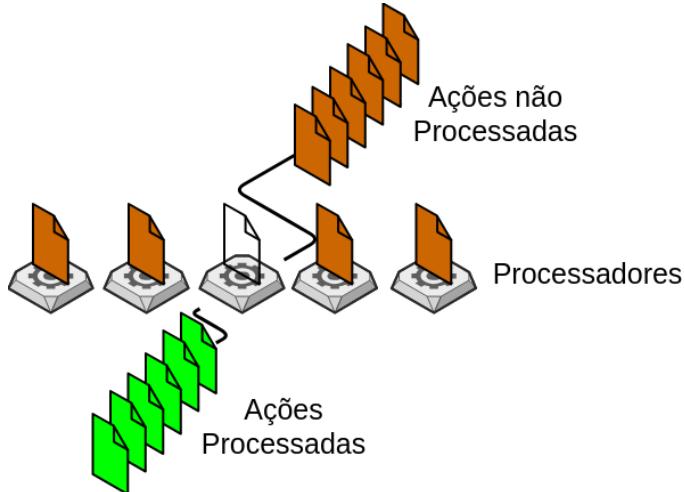
O funcionamento interno do serviço de jogo trabalha em rodadas, visando não penalizar usuários com baixa transferência de dados entre o cliente e o serviço. Cada cli-

¹⁶PostgreSQL: <https://www.PostgreSQL.org>

¹⁷MySQL: <https://www.mysql.com/>

ente produz requisições para serem consumidas pelo ciclo de processamento do gerente de jogo. Entretanto, o serviço irá consumir de forma igualitária uma requisição de um jogador diferente, como uma fila (SALZ, 2016a; RUDDY, 2011). O modelo de processamento do gerente de ambiente pode ser visualizado na Figura 2.18.

Figura 2.18: Modelo de processos *Thread Pool*.



Adaptado de: (RUDDY, 2011; RINGLER, 2014).

O modelo de processamento do gerente de mundo, visualizado na Figura 2.18 trabalha com um padrão *Thread Pool* (RINGLER, 2014; RUDDY, 2011), executando a chamada de método remoto de cada jogador em uma fila, o qual prioriza executar as chamadas sem repetir a mesma conexão. Dessa forma, cada jogador pode executar somente um método concorrente, sem competir com os demais. Todas as requisições são enfileiradas no *buffer* de rede do serviço. Caso o cliente entre em sua vez de processamento, e nenhuma chamada remota esteja na fila, ele é pulado.

Um serviço que demanda atenção na arquitetura Rudy é o Gerenciador de Consultas, um serviço web que implementa uma camada sobre diversos bancos de dados a fim de prover variedade entre vários bancos de forma facilitada por requisições web, utilizando operações CRUD. Implementar esta camada garante uma padronização de acesso ao banco de dados, porém adiciona um possível gargalo a arquitetura (RUDDY, 2011). Os pontos positivos de utilizar esta camada de consultas na arquitetura são:

1. Não permite acesso direto ao banco de dados do serviço web e do serviço de jogo.
2. Permite maior manejo a migrações em tabelas e troca de tecnologias.
3. Define uma sintaxe estrita para consulta, via CRUD.

4. Permite acesso do banco a diversos serviços, sem gerenciar o banco.
5. Permite contar número de requisições e tempo das requisições.

Contudo, adicionar uma camada sobre os bancos de dados para gerenciamento tem pontos negativos (RUDDY, 2011).

1. Aumenta a complexidade de implementação, teste, administração e ponto de falha.
2. Adiciona limites como número de conexões, número de requisições, etc.

Entretanto, esta arquitetura não permite escalar um único ambiente para um número de jogadores simultâneos maior ao designado pelo hardware que hospeda o serviço. Por este motivo, as arquiteturas Salz e Willson tomam abordagens para subdividir o ambiente do jogo em mais serviços, podendo assim escalar um único ambiente para mais jogadores simultâneos.

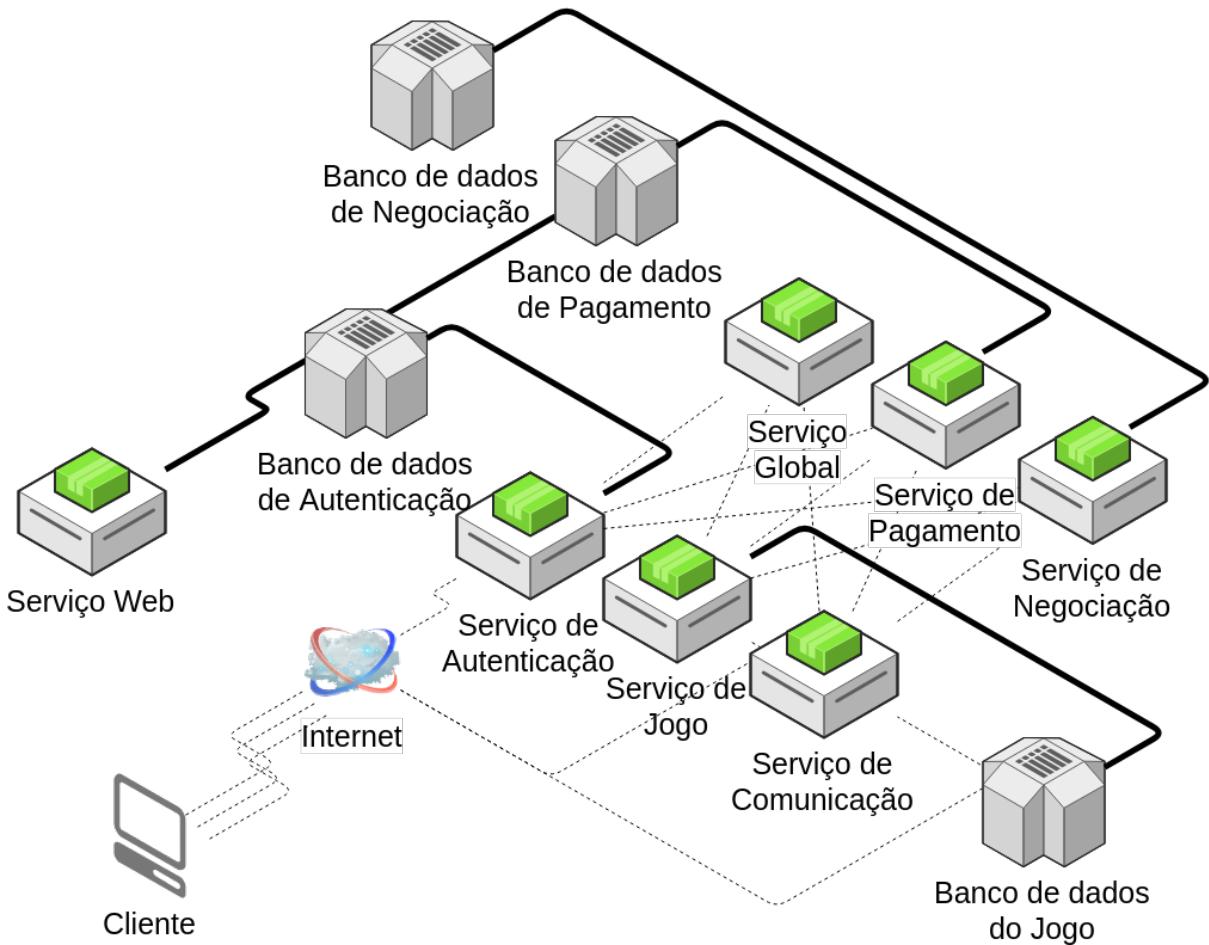
2.5.2 Arquitetura elaborada por Salz

A arquitetura elaborada por Salz (SALZ, 2016a), a qual pode ser visualizada na Figura 2.19 contém sete microsserviços especializados para seu funcionamento. Para o funcionamento adequado, o cliente necessita manter três conexões abertas com o servidor, sendo a conexão de jogo a com maior demanda de banda e a qual necessita o menor tempo de latência (SALZ, 2016a).

A Figura 2.19 exibe a arquitetura Salz de forma completa, na qual encontram-se quatro bancos de dados distintos, sendo eles o banco de *Pagamento*, *Negociação*, *Autenticação* e *Jogo*, sendo os bancos de Autenticação e Jogo com tecnologias *Not Only SQL* (NoSQL). Os demais bancos utilizam tecnologia *Structured Query Language* (SQL). Referente aos microsserviços que compõem a arquitetura, tem-se os seguintes elementos (SALZ, 2016b):

1. **Serviço de Comunicação:** Gerencia a troca de mensagens entre os jogadores. Opera sobre o protocolo RPC.
2. **Serviço de Autenticação:** Recepiona e gerencia as conexões dos clientes entre os demais microsserviços. Opera sobre o protocolo HTTP.

Figura 2.19: Arquitetura Salz.



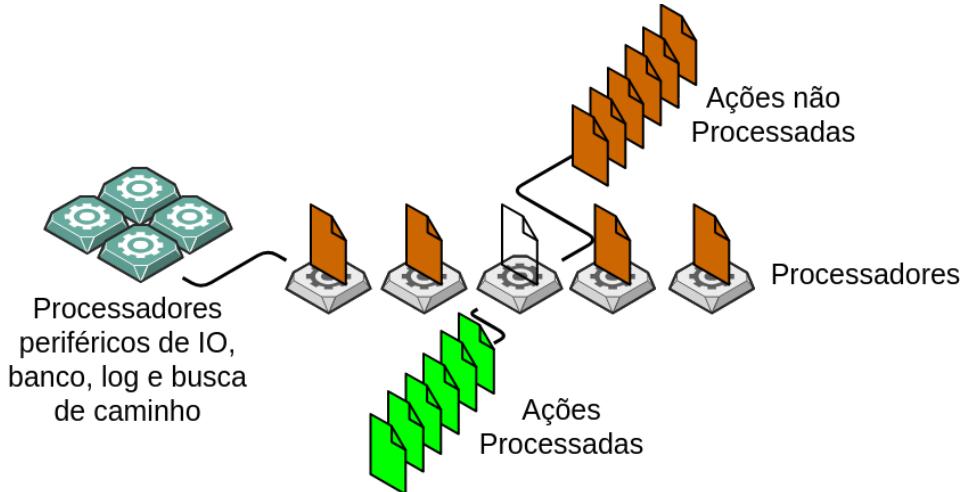
Adaptado de: (SALZ, 2016a).

3. **Serviço de Jogo:** Gerencia o estado do mundo, referente a um *chunk* do ambiente. Opera sobre o protocolo RPC.
4. **Serviço Global:** Gerencia operações globais (*e.g.*, interações entre grupos, procedimentos recorrentes globais, *etc.*). Opera sobre HTTP.
5. **Serviço de Pagamento:** Efetua operações bancárias com serviços externos de pagamento e gerencia o estado de pagamento das contas. Opera sobre o protocolo HTTP.
6. **Serviço de Negociação:** Opera como um serviço de leilão para itens do jogo. Opera sobre o protocolo HTTP.

Os microsserviços que compõem a arquitetura Salz utilizam em grande parte serviços *web*, utilizando somente o protocolo RPC para o serviço de jogo, autenticação e comunicação. No serviço de jogo, tanto o cliente quanto o serviço podem invocar métodos

remotos através do protocolo RPC, tendo como padrão métodos com retorno nulo (SALZ, 2016b; Exit Games, 2018). Esta abordagem garante que o usuário não fique esperando pelo retorno do serviço para continuar a lógica do jogo (FARBER, 2002). Para este funcionamento, o modelo de processamento paralelo deste serviço possui mais regras, a qual não são abordadas pela arquitetura Rudy (Figura 2.16). O modelo de paralelismo do serviço de jogo pode ser visualizado na Figura 2.20.

Figura 2.20: Modelo de paralelismo do serviço de jogo na arquitetura Salz.



Adaptado de: (SALZ, 2016b; CLARKE-WILLSON, 2013).

O microsserviço de jogo executa a lógica do jogo, visível na Figura 2.20, para uma única área em um único processo. Esta decisão é dada por conta da interação entre objetos ser complicada de executar em paralelo, visto o gerenciamento do custo de gerenciamento de semáforos necessários, caso execute estas ações em paralelo (SALZ, 2016b).

Outra facilidade de implementar um modelo com um único processo para as interações entre objetos é facilitar o não manejo de objetos a qual não estão no campo de visão de todos os jogadores do serviço, realizando estas operações somente quando necessário (SALZ, 2016a; SALZ, 2016b).

As entradas e saídas de mensagens(*e.g.*, rede, Banco de dados, *etc.*) e busca de caminho é executado por processos de trabalho separado do principal, utilizando a técnica de *Thread Pool* (SALZ, 2016b; SALZ, 2016a; RINGLER, 2014). Todos os demais microsserviços operam sobre múltiplos processos, a partir do processo de conexão TCP ao serviço (SALZ, 2016b). Diferente da arquitetura Rudy(Seção 2.5.1) a qual prioriza a resolução de requisições, evitando a executar métodos consecutivos da mesma conexão, na

arquitetura Salz (Figura 2.19) as chamadas de processo remoto são executadas em ordem *First In First out* (FIFO) (SALZ, 2016b). Desta forma, quanto melhor a conexão com o serviço, mais requisições um cliente poderá executar. Nesse sentido, uma conexão que transfere mais requisições terá mais vantagem sobre uma conexão a qual não transfere tantas requisições, executando mais operações por segundo.

Na arquitetura Salz, são necessárias três conexões TCP com o servidor, de forma contínua (SALZ, 2016a). O custo desta operação é alto, e por isso nem sempre é viável utilizar esta abordagem, principalmente para jogos nos quais a demanda de banda seja menor. Para evitar esta decisão, da mesma forma é notória a abordagem utilizada pela arquitetura Willson.

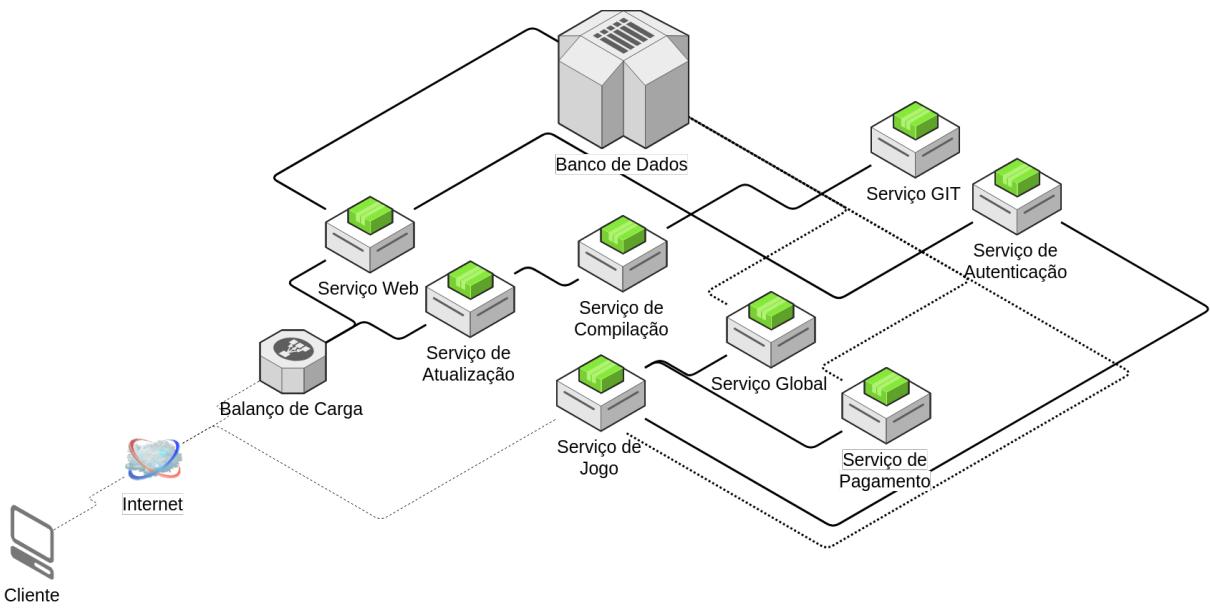
2.5.3 Arquitetura elaborada por Willson

A arquitetura elaborada por Clarke-Willson leva como principal característica a preocupação da disponibilização de atualizações aos clientes (CLARKE-WILLSON, 2013). Essas atualizações são versionadas a todos os microsserviços utilizando sistemas de versionamento de código fonte (CLARKE-WILLSON, 2017; CLARKE-WILLSON, 2013). Por este motivo, a sua arquitetura inclusive compreende microsserviços de armazenamento de arquivos sobre protocolo HTTP para atualização dos clientes (CLARKE-WILLSON, 2017). Uma outra característica do serviço de jogo é a utilização de uma única conexão TCP por cliente. Essa arquitetura pode ser visualizada na Figura 2.21.

A arquitetura Willson, exibido na Figura 2.21, mostra um gradual entre a arquitetura Rudy (Figura 2.16) e a arquitetura Salz (Figura 2.19), propondo uma arquitetura híbrida, na qual divide funcionalidades em outros microsserviços, porém ainda mantém diversas funcionalidade junto ao serviço de jogo evitando consumo de rede (SALZ, 2016a; CLARKE-WILLSON, 2013). Essa abordagem garante menor latência de resposta, porém terá maior consumo de recursos da mesma máquina hospedeira do serviço (CLARKE-WILLSON, 2013). Os microsserviços que compõem a arquitetura Willson são (CLARKE-WILLSON, 2013; CLARKE-WILLSON, 2017):

1. **Serviço web:** Exibe informações do jogo, oferece operações CRUD para cadastro de usuários e o sistema de pagamento. Opera sobre o protocolo HTTP.
2. **Serviço de Atualização:** Integrado junto ao processo de desenvolvimento, for-

Figura 2.21: Arquitetura Willson.



Fonte: (CLARKE-WILLSON, 2017).

necendo versionamento do cliente por um sistema web. Este microsserviço utiliza serviços internos ao desenvolvimento da aplicação. Opera sobre o protocolo HTTP.

3. **Serviço de Autenticação:** Gerencia a autenticação dos jogadores através de um serviço web. Também gerencia ações sociais (*e.g.* sistema de amigos, grupos, nações, comércio, *etc*). Opera sobre o protocolo HTTP.
 4. **Serviço de Balanceamento de carga:** Gerencia a carga entre os serviços web da arquitetura. Opera sobre o protocolo HTTP.
 5. **Serviço de Jogo:** Processa a lógica de jogo. Cada instância deste microsserviço gerencia um *chunk* do ambiente do jogo. Opera sobre o protocolo RPC.
 6. **Serviços Globais:** Processa rotinas globais do jogo, a qual não dependem do posicionamento do jogador. Opera sobre o serviço RPC.

O serviço de jogo da arquitetura Willson é comparada ao serviço similar na arquitetura Salz, definido no Subcapítulo 2.5.2, entretanto utiliza a técnica de *thread pool* com valor de processos da fila de processadores fixo conforme o *hardware* hospedeiro do serviço (CLARKE-WILLSON, 2017). Para modelos de paralelismo, pode-se utilizar os seguintes números de processos paralelos (CLARKE-WILLSON, 2013):

1. O dobro de número de núcleos: exige maior carga do serviço por troca de contexto entre os processos.

2. O número exato de núcleos mais n: O serviço perderá tempo de processamento, trocando de contexto para outro processo, porém de forma mais sutil ao dobro do número de núcleos de processamento.
3. O número de núcleos: exigindo um número menor de carga de contexto, dividindo somente com o sistema operacional.
4. O número de núcleos menos um: liberando um núcleo para o sistema operacional.

O número padrão de processos paralelos para processamento de requisições é o número de núcleos menos um, visando evitar a troca de contexto com o sistema operacional (CLARKE-WILLSON, 2013). Essa troca de contexto trás variação na latência da resposta do serviço, a qual não tem controle pelo próprio serviço. Nesse sentido, a melhor escolha é o número de núcleos menos um, escolhido após um teste de *stress*, definido no Subcapítulo 2.5.3.

Entretanto, não foi encontrado análises públicas sobre as arquiteturas Rudy, Salz e Willson, com relação ao uso de recursos dessas arquiteturas. Nesse sentido, o atual trabalho define o seu problema sobre o consumo de recursos em buscar uma análise sobre o comportamento das arquiteturas referenciadas.

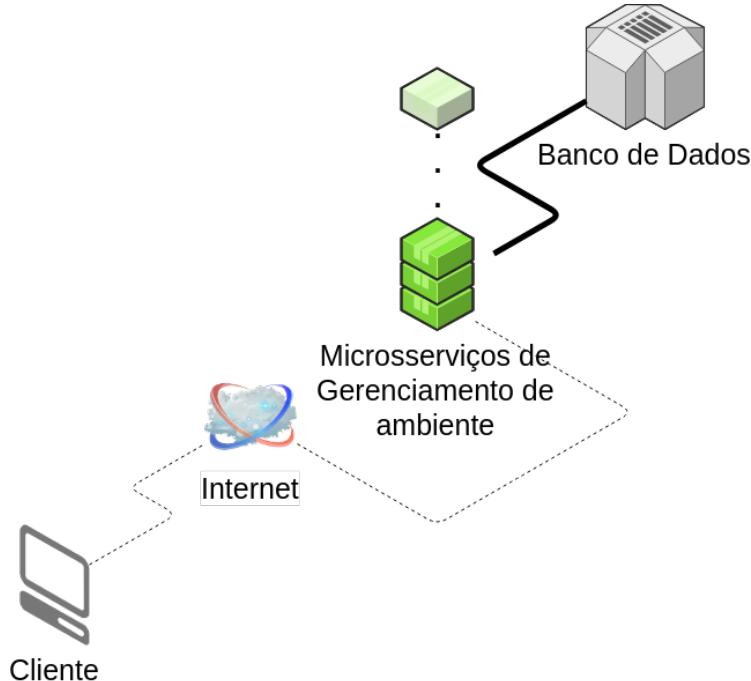
2.6 Definição do Problema

A escolha de uma arquitetura para serviços MMORPG que não suportem uma elevada carga de trabalho podem ser um eventual problema na escalabilidade do negócio de empresas que forneçam tais serviços. Em alguns serviços, a divisão da carga é dada pela área de interesse, dividindo o ambiente em pedaços a fim de diminuir a carga no serviço. Porém, locais populares no ambiente do jogo ainda são vulneráveis a disponibilidade de serviço (HUANG; YE; CHENG, 2004).

Nesse sentido, arquiteturas de microsserviços surgiram com o objetivo de aumentar a disponibilidade e viabilizar produtos de demanda massiva. Tais arquiteturas dividem o seu funcionamento em módulos menores a fim de proporcionar maior demanda utilizando uma abordagem distribuída. Assim, o custo de atender uma alta demanda de conexões pode consumir uma quantia de recursos computacionais ou uma qualidade insatisfatória de serviço, inviabilizando a sua utilização no mercado.

Um exemplo de implementação de arquitetura de um jogo MMORPG pode ser visualizado na Figura 2.22, no qual cada cliente deve conectar em um microsserviço de gerenciamento de mundo para receber as atualizações da região e submeter seus comandos. Nessa arquitetura, é necessário levar em conta o funcionamento, método de compartilhamento dos dados e abordagem para processamento do ambiente do jogo a fim de descrever a sua escalabilidade e qualidade de serviço.

Figura 2.22: Arquitetura de um jogo MMORPG genérico.



Fonte: O próprio autor

A arquitetura genérica descrita na Figura 2.22 também precisa corresponder as demandas necessárias do *GameDesign* do próprio jogo. Neste caso, só um tipo de microsserviço é visível nesta rede, porém poderiam existir microsserviços responsáveis pela parte social, comercial e estatísticas, aumentando o grau de complexidade da arquitetura. Em relação aos recursos utilizados por uma arquitetura, este trabalho foca na análise das arquiteturas de microsserviços descritas na literatura a fim de analisar o seu comportamento e desempenho. Porém, não foi encontrado nenhuma análise das arquiteturas de microsserviços propostas por Rudy, Salz e Willson, tornando a escolha de um modelo de arquitetura uma tarefa complexa a qual demande testes de tentativa e erro ou análises particulares para viabilizar o funcionamento destes sistemas. Dessa forma, este trabalho tem como principal objetivo guiar, utilizando análises, futuras escolhas de modelos a qual se adequem melhor a realidade de futuros projetos de jogos MMORPG baseados em microsserviços.

2.7 Considerações parciais

Este capítulo conceituou jogos eletrônicos, gênero de jogo e especificou as características de um jogo MMORPG. Após apresentar sobre o gênero de jogo abordado, detalha-se a sua jogabilidade, problemas relevantes a este gênero do ponto de vista de rede de computadores e por fim sobre técnicas e abordagens populares acerca do desenvolvimento destes serviços, em específico sobre o paradigma de arquitetura de microsserviços.

O desenvolvimento de arquiteturas de microsserviços para jogos MMORPG é uma tarefa multidisciplinar a qual pequenas variações de protocolo, algoritmos ou microsserviços dentre os modelos impacta diretamente no consumo de recurso destas arquiteturas. Nesse sentido, o atual capítulo trouxe inicialmente um levantamento teórico sobre as principais características deste gênero de jogo, características das principais camadas de cliente, servidor e serviço e por sua vez mostrou abordagens viáveis de desenvolvimento para ambas as camadas. Por fim trouxe uma introdução ao paradigma de arquiteturas de microsserviços, abordando em seguida a sua aplicação para jogos MMORPG.

Após a introdução técnica necessária para entendimento das arquiteturas, foi realizado um levantamento de arquiteturas de microsserviços da literatura, descrevendo sua funcionalidade, topologia e por fim protocolos utilizados. Entretanto, não foi encontrado trabalhos correlacionados as arquiteturas encontradas, dessa forma encontrando um dos pontos de apoio científico deste trabalho.

A literatura aborda previsibilidade de carga, análise de disponibilidade e uso de recurso de tais arquiteturas (a fim de guiar escolhas na etapa de *Game Design* do produto e viabilizar o comércio do mesmo), relatados na Seção 3. Torna-se de interesse para projetistas de desenvolvimento de jogos MMORPG analisar o impacto e uso de recursos computacionais ao implantar uma arquitetura de microsserviços MMORPG visando melhorar a disponibilidade de tais jogos. Nesse sentido, se faz necessário realizar um levantamento bibliográfico a cerca dos temas de jogos MMORPG e arquiteturas de microsserviços com a finalidade de comprovar a utilidade da proposta do atual trabalho.

3 Trabalhos Relacionados

Para nortear o desenvolvimento da análise de microsserviços utilizados em jogos MMORPG proposto no atual trabalho, essa seção apresenta outros trabalhos que têm o escopo ou objetivo similar, no qual monitoraram e analisaram serviços de jogos MMORPG ou arquiteturas de microsserviços. Ao apresentar estes trabalhos, busca-se apresentar o contexto e objetivo, e então aprofundar em características dos trabalhos, métricas utilizadas e ferramentas que auxiliaram nas análises.

Para encontrar os trabalhos relacionados, foi efetuada uma busca pelos termos MMORPG ou *microservices*. Entretanto, os dois termos dificilmente se correlacionam nos meios de busca disponíveis para a elaboração deste TCC. Nesse sentido, os trabalhos relacionados abordam arquiteturas de jogos MMORPG ou arquiteturas de microsserviços.

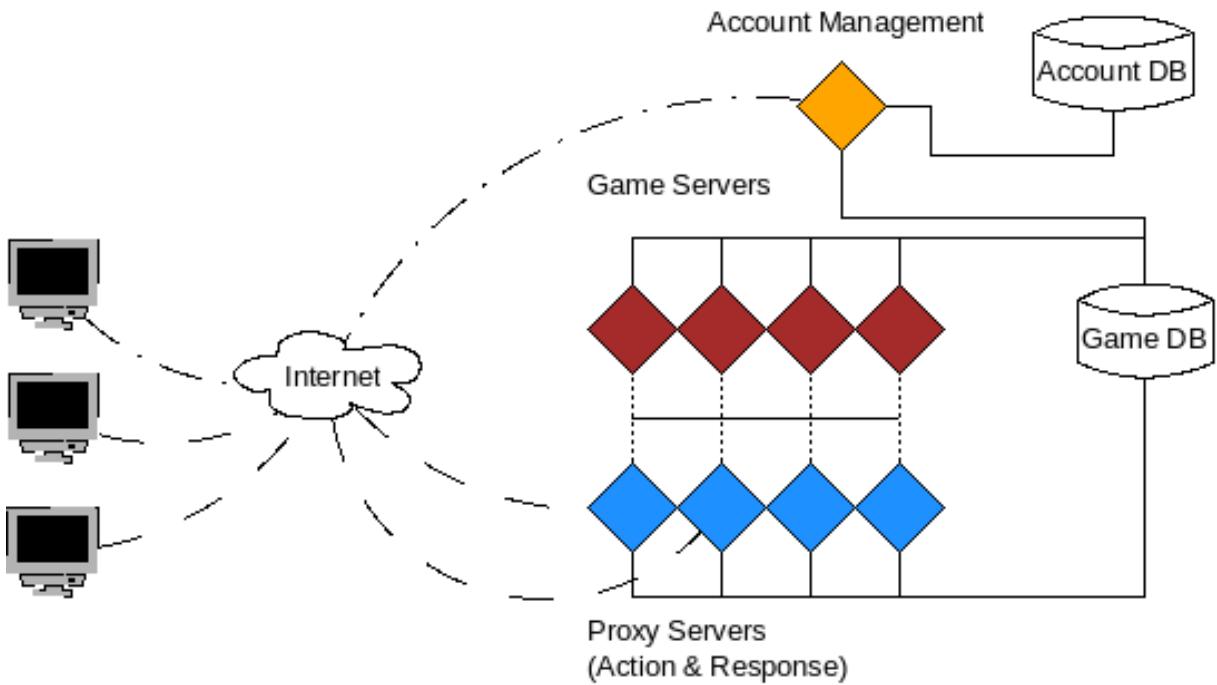
Como critério de análise, foi observado em questão qual a classificação em que o trabalho encontra-se, entre previsão de carga ou análise de arquitetura, e quais métricas são utilizadas na análise.

3.1 Huang et al. (2004)

O trabalho de (HUANG; YE; CHENG, 2004) investiga a relação entre os recursos utilizados e o número de conexões presentes em um serviço MMORPG distribuído. Neste trabalho é relatado que a infraestrutura utiliza três serviços: Um *Game Server* sobre protocolo TCP, um *Proxy Server* também sobre protocolo TCP, e um servidor web para autenticação que executa sobre uma interface HTTP. O foco de análise é o *Proxy Server*, um serviço especificado em receber requisições e repassar atualizações da área de interesse destes jogadores, e o *Game Server*, um serviço especificado para consumir as requisições realizadas pelo jogador.

A infraestrutura do servidor de jogo contém um *Proxy Server Farm* utilizando o algoritmo *Round Robin* com pesos para balanceamento de carga entre cada cliente. Cada *Proxy Server* é responsável por comunicar com os demais microsserviços privados ao servidor, baseado com a área de interesse de sua conexão. O protocolo de comunicação

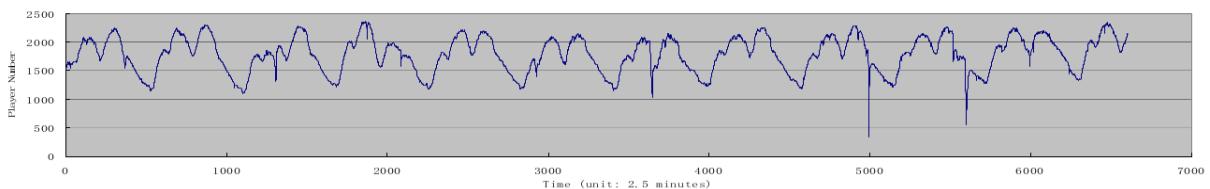
Figura 3.1: Arquitetura distribuída utilizando proxy.



Adaptado de: (HUANG; YE; CHENG, 2004).

utilizado entre o Cliente e *Proxy Server* é baseado em RPC (FARBER, 2002; BORELLA, 2000), porém não é relatado sobre o protocolo de comunicação utilizado entre o *Proxy Server* e o *Game Server*. A sua arquitetura pode ser observada na Figura 3.1, na qual obteve seus dados durante 100 dias para realizar as análises. A Figura 3.2 demonstra uma amostra do número de conexões pelo tempo no serviço obtido.

Figura 3.2: Número de conexões no serviço pelo tempo decorrido.

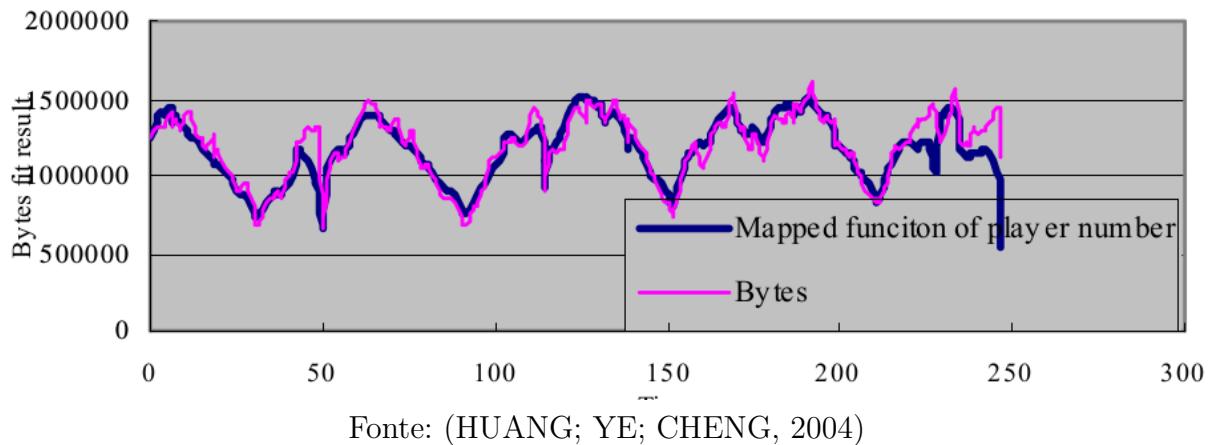


Fonte: (HUANG; YE; CHENG, 2004).

Como análise, o autor correlacionou o número de conexões com número de pacotes e banda, consumidos, utilizando uma função estatística linear. Esta função pode ser utilizada com regressão linear para prever consumo de recursos futuros e por fim realocar mais recursos ao serviço, contribuindo com escalabilidade vertical autônoma. Um exemplo de aplicação dessa regressão linear pode ser visualizada na Figura 3.3, no qual o autor compara o consumo de banda real comparado a regressão linear.

Entretanto, a escalabilidade horizontal não pode ser prevista, visto que não é

Figura 3.3: Regressão linear comparado ao consumo de banda real do servidor.



Fonte: (HUANG; YE; CHENG, 2004)

analisado o posicionamento de cada personagem a fim de dividir os ambientes em pedaços menores com outros serviços. Como trabalhos futuros é relatado a análise do posicionamento de personagens para escalabilidade horizontal, a análise de outras arquiteturas e a análise de outros gêneros de jogos, além do impacto de utilizar balanço de carga e provisionamento de recursos de forma dinâmica.

3.2 Villamizar et al. (2016)

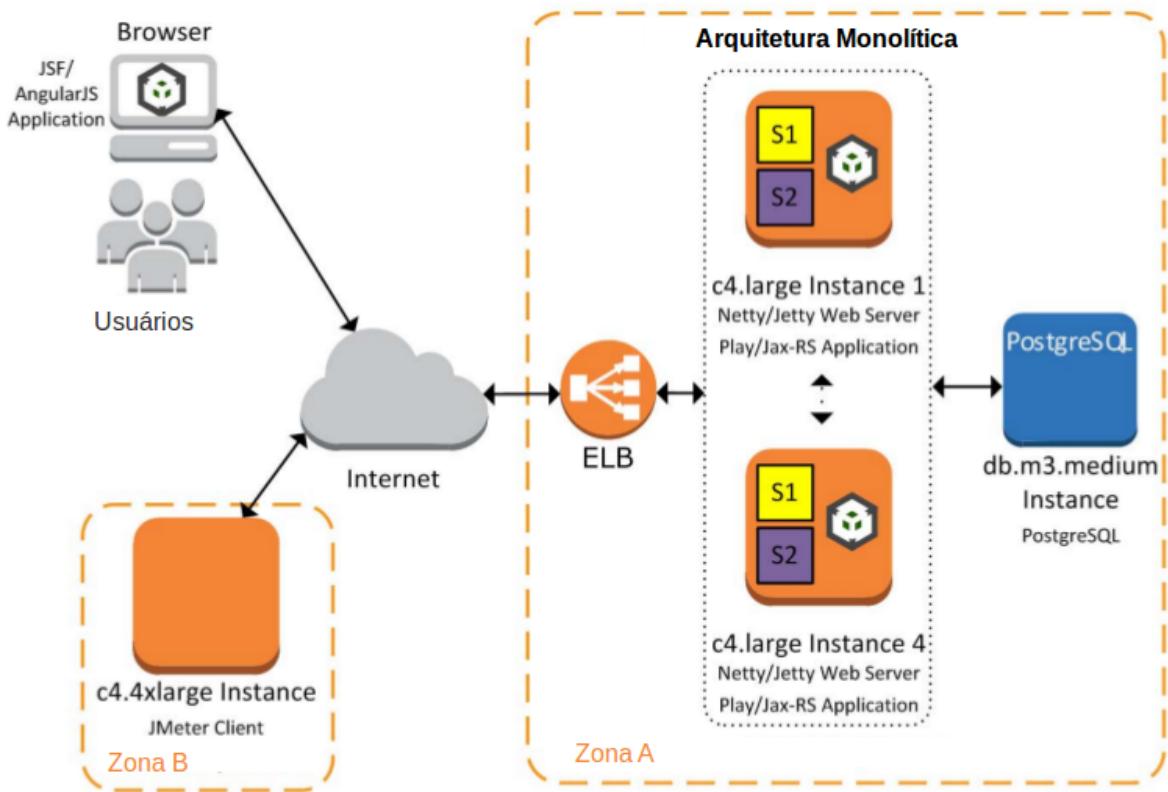
O trabalho de (VILLAMIZAR et al., 2016) investiga o custo de arquiteturas de microsserviços, arquiteturas *Platform as a Service* (PaaS) orientadas a eventos e aplicações monolíticas para aplicações web. A sua principal motivação é a comparação de custos para a tradução de sistemas legados para arquiteturas distribuídas. Para isso, o autor preparou três instâncias de testes com suas configurações desenhadas a fim de ter o maior número de requisições por minuto com o mesmo custo financeiro.

Instância I. Utilizando quatro instâncias AWS *c4.large*, uma instância AWS *c4.xlarge* e uma instância AWS *db.m3.medium*. A Figura 3.4 exibe a implantação de uma aplicação web monolítica. Essa arquitetura foi implementada utilizando *Jax-RS* e *Play Framework*.

Instância II. Utilizando três instâncias AWS *c4.xlarge*, uma instância AWS *t2.small* e uma instância AWS *db.m3.medium*. A Figura 3.5 exibe a implantação de uma aplicação de microsserviços web. Essa arquitetura foi implementada utilizando *Play Framework*, framework para desenvolvimento web para a linguagem Java e Scala¹.

¹Play Framework: <https://www.playframework.com/>

Figura 3.4: Arquitetura monolítica web implementada na AWS.



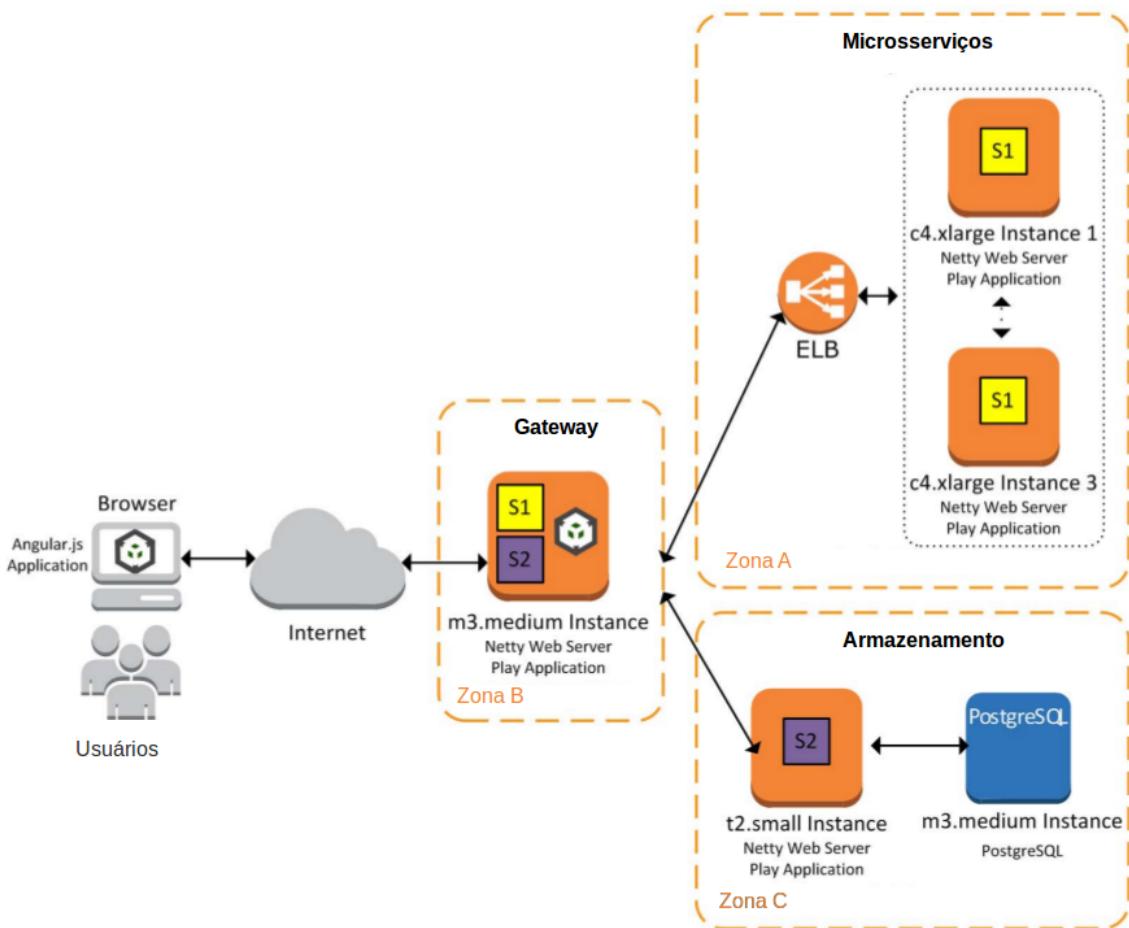
Fonte: (VILLAMIZAR et al., 2016)

Instância III. Utilizando duas instâncias AWS *lambda* `S1`, duas instâncias AWS *lambda* `S2`, uma instância AWS *S3 Bucket* e uma instância AWS `db.m3.medium`. A Figura 3.6 exibe a implantação de uma aplicação de microsserviços web utilizando a tecnologia AWS *lambda*. Essa arquitetura foi implementada utilizando o *framework* `Node.js`², nas quais as funções de *gateway* são implementadas em quatro funções independentes do tipo *microservice-http-endpoint*.

Foi concluído que a arquitetura de microsserviços, nas condições desta aplicação, podem reduzir até 13.42% em gastos com a infraestrutura. Essa redução pode ser observada na Figura 3.7. O autor alerta sobre tolerância a falhas, transações distribuídas, distribuição de dados e versionamento de serviço.

²Node.js: <https://nodejs.org/en/>

Figura 3.5: Arquitetura de microsserviços web implementada na AWS.



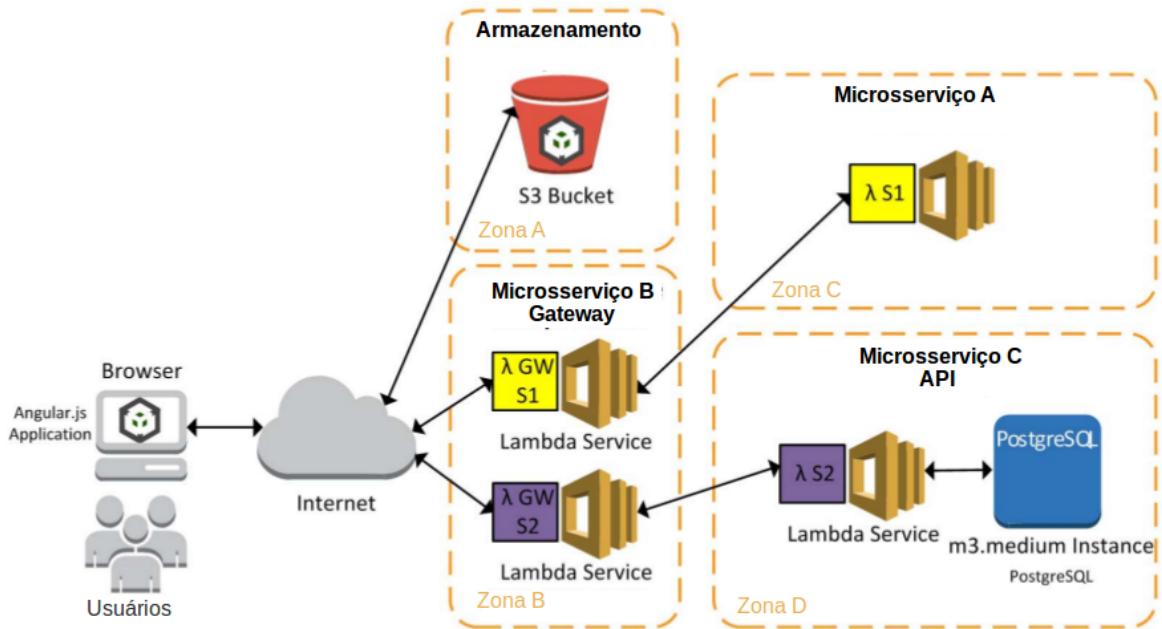
Fonte: (VILLAMIZAR et al., 2016)

3.3 Suznjevic e Matijasevic (2012)

O trabalho de (SUZNJEVIC; MATIJASEVIC, 2012) tem seu objetivo a fim de prever a carga a qual um serviço MMORPG pode receber utilizando a complexidade das operações nos contextos de *Player vs Player* (PvP) e *Player vs NPCs* (PvNPCs) a qual um personagem pode realizar em um ambiente. Este trabalho usa com base o modelo descrito na Seção 3.1, um modelo distribuído em serviços na qual efetuam o processamento de uma região do ambiente virtual.

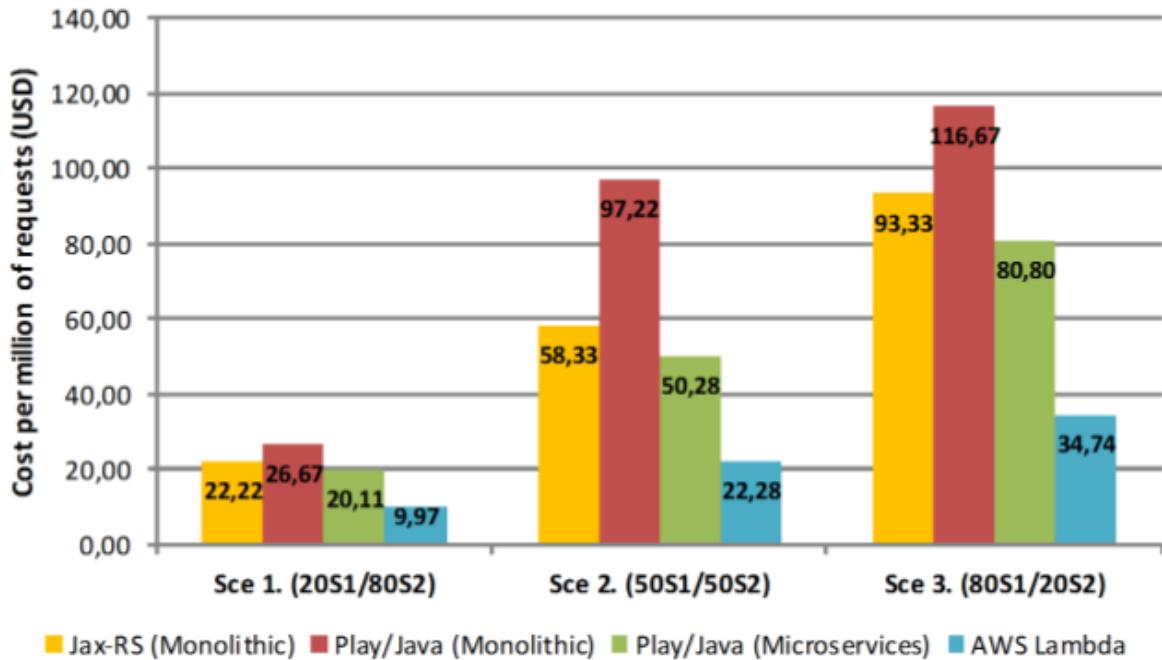
A análise realizada leva em conta a complexidade das ações no ambiente, a qual pode ser descrita na Tabela 3.1. Essa tabela exibe as ações que podem ser executadas para interagir com o ambiente, seja essa interação com PvP (jogador com outro jogador) ou PvNPCs (jogador com um personagem ou objeto gerenciado pelo serviço). Os contextos analisados nessa tabela são:

Figura 3.6: Arquitetura de microsserviços web implementada na AWS utilizando a tecnologia *lambda*.



Fonte: (VILLAMIZAR et al., 2016)

Figura 3.7: Custo por um milhão de requisições em dólares utilizando diferentes arquiteturas sobre a AWS.



Fonte: (VILLAMIZAR et al., 2016)

- *Questing*: Contexto de missão, na qual um grupo de jogadores ou um grupo de NPCs podem ser afetados nas ações.
- *Trading*: Contexto de negociação, na qual a complexidade leva em conta somente o

Tabela 3.1: Complexidade da interação com o ambiente, por contexto da interação.

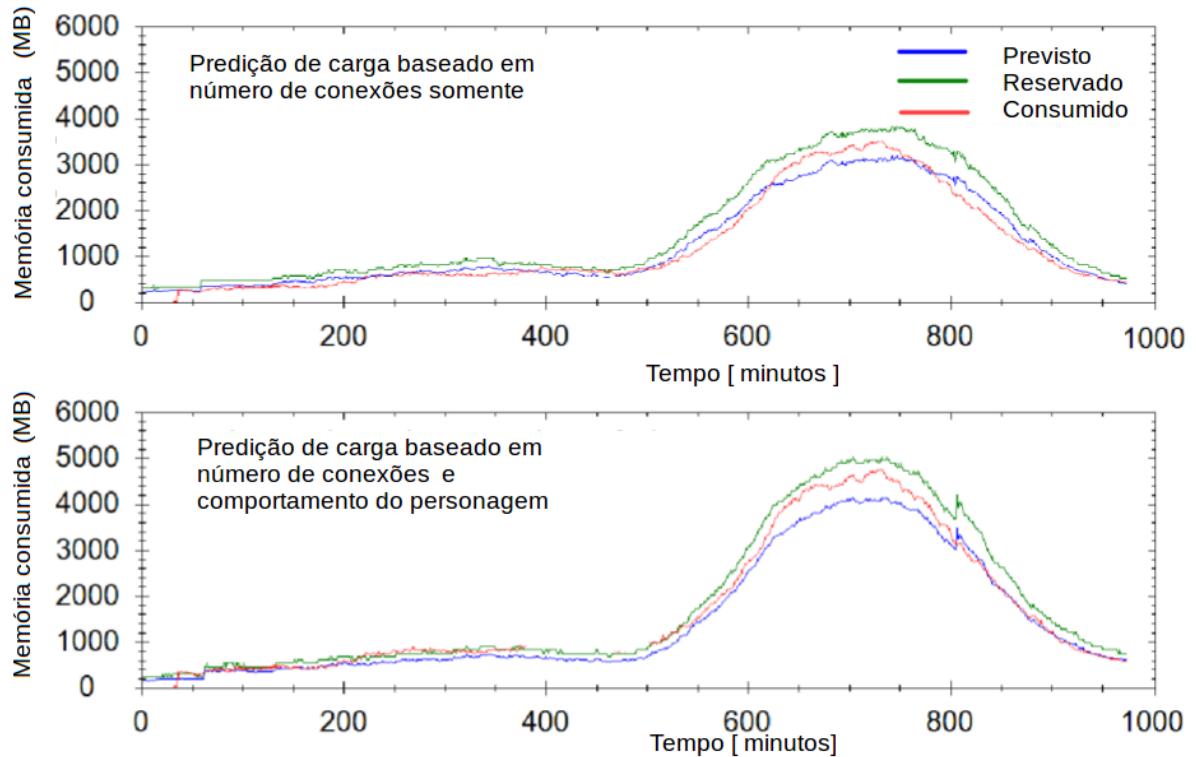
Contexto da ação	PvP	PvNPCs	Número de NPCs	Network [kbits/s]
Questing	$O(n)$	$O(n \log(n))$	$N \leq 6$	11.4
Trading	$O(n)$	$O(n)$	$N \leq 20$	8.1
Dungeons	$O(n^2)$	$O(n^2)$	$N \leq 20$	18.3
PvP combat	$O(n^3)$	$O(n)$	$N = 0$	24.1
Raiding	$O(n^2 \log(n))$	$O(n^3)$	$N \leq 40$	32.0

Fonte: (SUZNJEVIC; MATIJASEVIC, 2012)

número de itens negociados.

- *Dungeons*: Contexto de exploração, na qual o ambiente pode ser modificado conforme as ações dos personagens em um ambiente isolado para este grupo.
- *PvP combat*: Contexto de batalha entre jogadores, na qual as ações entre os jogadores influenciam diretamente o estado do personagem oponente.
- *Raiding*: Representa um contexto específico de exploração, na qual múltiplos jogadores unem forças a fim de combater outro grupo de jogadores ou NPCs.

Figura 3.8: Regressão levando em conta a complexidade das ações e contexto dos personagens.



Fonte: (SUZNJEVIC; MATIJASEVIC, 2012)

A abordagem utilizou as complexidades das ações, o número de conexões e o contexto de cada personagem no ambiente para predizer a banda utilizada. Pode-se visualizar uma regressão na Figura 3.8.

O autor conclui que o contexto de interação com o ambiente de cada personagem tem relevância com o consumo de CPU e banda, a qual pode ser calculada com sua complexidade a fim de desenvolver uma ferramenta para predição de carga sobre serviços MMORPG. Entretanto, essa predição não é feita em tempo real, não contribuindo para a automação da escalabilidade vertical e horizontal da arquitetura de microsserviços.

3.4 Análise dos trabalhos relacionados

Para os trabalhos relacionados, são analisados o tipo de pesquisa realizado sobre Microsserviços e serviços MMORPG. Também são levantados quais os recursos computacionais relacionados com estas análises.

Nos trabalhos relacionados existem duas abordagens utilizadas pelos autores, na qual estão relacionados a Previsão de Carga ou Comparação entre Arquiteturas de microsserviços (VILLAMIZAR et al., 2016; SUZNJEVIC; MATIJASEVIC, 2012):

- **Comparação de arquiteturas:** O autor utiliza alguma métrica a fim de decidir qual a melhor alternativa de arquitetura para determinada aplicação.
- **Previsão de carga:** Projetar a carga futura sobre algum recurso a fim de escalar automaticamente a sua arquitetura na nuvem, a fim de reduzir gastos de recursos ociosos e diminuir o impacto de ocorrências aos usuários finais.

Dessa forma, pode-se dividir os trabalhos relacionados com a sua categoria. Esta relação pode ser visualizada na Tabela 3.2.

Tabela 3.2: Trabalhos relacionados por categoria.

Autor	Categoria
(SUZNJEVIC; MATIJASEVIC, 2012)	Previsão de carga
(VILLAMIZAR et al., 2016)	Comparação de arquiteturas
(HUANG; YE; CHENG, 2004)	Previsão de carga

Fonte: O próprio autor.

Para o trabalho referente a comparação de arquiteturas (VILLAMIZAR et al., 2016), o recurso utilizado foi o custo por um determinado número de requisições

web. Já para os trabalhos referentes a previsão de carga (SUZNJEVIC; MATIJASEVIC, 2012; HUANG; YE; CHENG, 2004) foi levantado o consumo da banda, CPU e memória, entretanto não levantou-se o número de conexões e latência aos usuários, na qual podem ser observados na Tabela 3.3. Nenhum trabalho relatou limite de conexões ou latência do sistema.

Tabela 3.3: Trabalhos relacionados por recurso analisado.

Autor	CPU	Memória	Banda	Custo	Latência	Limite de conexões	Complexidade de Algoritmos
(HUANG; YE; CHENG, 2004)	✗	✗	✓	✗	✗	✗	✗
(VILLAMIZAR et al., 2016)	✗	✗	✗	✓	✗	✗	✗
(SUZNJEVIC; MATIJASEVIC, 2012)	✓	✓	✓	✗	✗	✗	✓

Fonte: O próprio autor.

Outro ponto a ser analisado são as arquiteturas utilizadas. Os trabalhos referentes a previsão de carga não utilizaram uma arquitetura de microsserviços, mesmo sendo um sistema distribuído. Nesses serviços, os sistemas eram dependentes entre si e precisavam um dos outros para o seu funcionamento. A relação de arquiteturas abordadas pode ser visualizado na Tabela 3.4, na qual mostra quais trabalhos abordaram os temas sobre arquiteturas de microsserviços e jogos MMORPG. Além disso, não foi descrito um método de replicação automática dos serviços a fim de prover alta disponibilidade de forma automatizada, sendo abordado como um trabalho futuro.

Tabela 3.4: Arquiteturas analisadas.

Autor	Arquitetura de Microsserviços	Arquitetura Distribuída	MMORPG
(HUANG; YE; CHENG, 2004)	✗	✓	✓
(VILLAMIZAR et al., 2016)	✓	✓	✗
(SUZNJEVIC; MATIJASEVIC, 2012)	✗	✓	✓

Fonte: O próprio autor.

Este Trabalho de Conclusão de Curso pretende analisar uma arquitetura de microsserviços especificada a jogos MMORPG, visando métricas de desempenho e custo de operação. Os recursos analisados serão CPU, Memória, Banda, Custo, Latência, Limite de Conexões e Complexidade dos Algoritmos empregados.

3.5 Considerações parciais

Após a busca de trabalhos relacionados, encontrou-se três principais trabalhos a qual analisam ora arquiteturas de microsserviços ora arquiteturas de jogos MMORPG. Nesse

sentido, tais trabalhos relacionados não realizam uma intercessão entre ambos os temas, a qual o atual trabalho propõe a análise. Também foi identificado as métricas a qual estes trabalhos analisaram. Nesse sentido, o atual trabalho irá propor a análise sobre as mesmas métricas, entretanto completando com métricas que são de interesse para analisar pontos de falha em tais arquiteturas.

Por fim, foi apresentado trabalhos relacionados a qual guiaram os moldes do plano de testes e a proposta deste trabalho, porém com utilizando ambos os temas de arquiteturas de microsserviços e jogos MMORPG, realizando uma análise com mais características sobre o serviço do jogo.

4 Proposta para análise de arquiteturas MMORPG

As arquiteturas de serviços MMORPG são desenvolvidas visando suprir as necessidades do projeto do jogo desenvolvido, de forma a viabilizar a utilização deste serviço. Nesse sentido, jogos com mecânicas de projeto similares possuem implementações parecidas para os clientes. Entretanto, a arquitetura escolhida impacta no custo de operação e qualidade do serviço aos jogadores. Por este motivo, diferentes arquiteturas com o mesmo *design* não são comparáveis entre si, visto que dependem das regras de negócio do jogo.

Ao desenvolver um serviço MMORPG é necessário decidir uma arquitetura que possibilite reduzir custos, consumo de recursos e minimize ocorrências para os jogadores a fim de viabilizar a sua implantação como produto. Porém, a impossibilidade de comparação direta entre as arquiteturas de serviço MMORPG instiga a análise das características básicas destas arquiteturas que possam influenciar o *game design*, tais como consumo de recursos, tempo de resposta, latência e número máximo de clientes simultâneos conectados nos microsserviços. Sendo assim, uma análise do consumo de recursos computacionais das arquiteturas levantadas previamente na literatura tem valor científico no auxílio da escolha de implementações de arquiteturas de microsserviços, em específico para serviços MMORPG.

Neste capítulo é descrita a proposta para análise de consumo de recursos computacionais em arquiteturas MMORPG. Inicialmente, é descrita a proposta em alto nível (Seção 4.1), trazendo os objetivos desta análise, quais recursos e métricas serão analisadas (TCC-II). Os Critérios de Análise (Seção 4.2) exibem como os dados obtidos devem ser interpretados, baseando-se nos objetivos da análise das arquiteturas. O Plano de Testes (Seção 4.3) exibe como será realizada a coleta dos dados, descrevendo o ambiente, cenário, critérios e os testes que serão realizados.

4.1 Proposta

Tendo analisado os trabalhos relacionados (Seção 3) e as arquiteturas específicas para jogos MMORPG, o presente trabalho tem como objetivo analisar as arquiteturas MMORPG visando complementar a análise de arquitetura e consumo de recursos computacionais não analisados nos trabalhos relacionados. Em específico, serão obtidos os valores referentes aos uso dos seguintes recursos computacionais nas arquiteturas Rudy (Subseção 2.5.1), Salz (Subseção 2.5.2) e Willson (Subseção 2.5.3):

1. **CPU:** o uso de CPU, com sua representação sendo em relação a porcentagem de processamento nos núcleos utilizados;
2. **Memória:** Quantidade de memória utilizada pelos processos do serviço/arquitetura. A sua representação será como dado absoluto;
3. **Rede:** Banda de rede utilizada nas operações de entrada e saída para cada microserviço, utilizando valores absolutos. Juntamente será obtido o valor de latência do cliente ao microsserviço, verificando se o congestionamento da rede afeta a latência do microsserviço.

Além dos recursos computacionais, esta análise levará em conta valores referentes a outras métricas. As métricas, cujos os valores serão obtidos são:

1. **Número máximo de jogadores simultâneos:** Descobrir o limite de conexões para as arquiteturas propostas a análise. Será representado como valor absoluto.
2. **Tempo de resposta das requisições:** Descobrir o tempo de resposta por categoria de requisição, conforme o número de jogadores no serviço. Será representado como tempo decorrido, em milissegundos.

Todos estes valores serão obtidos a partir de simulações, e por este motivo faz-se necessário descrever o comportamento dos jogadores simulados (Seção 4.3.2). Espera-se, em situações adversas, caracterizar os comportamentos das arquiteturas bem como gargalos e os custos de recursos computacionais para manutenção das arquiteturas de microsserviços. Para este fim, faz-se necessário a descrição dos critérios que serão utilizados durante a análise dos valores obtidos nos experimentos.

4.2 Critérios de análise

A fim de padronizar a análise dos dados obtidos, estes serão estabelecidos usando como base o comportamento dos valores obtidos em referenciais. Neste sentido, a análise dos dados obtidos será guiada pelo esperado dos valores obtidos em um serviço padrão:

1. **Consumo CPU:** Espera-se estressar com um elevado número de requisições.
2. **Consumo Memória:** Espera-se estressar com requisições nas quais exija armazenamento em memória.
3. **Vazão Rede - Entrada:** Espera-se estressar com requisições nas quais tenham uma carga de dados elevada.
4. **Vazão Rede - Saída:** Espera-se estressar com respostas nas quais tenham uma carga de dados elevada.
5. **Número de Conexões Simultâneas:** Servirá como guia de comparação com os demais valores e desempenho da arquitetura;
6. **Tempo de resposta das requisições:** Servirá como guia de desempenho da arquitetura; e

Em um caso de uso ideal, todos os recursos não são estressáveis, com um número de conexões simultâneas elevado. Porém, espera-se para este trabalho um possível conjunto de ocorrências, na qual podem ou não ocorrer. Este conjunto servirá de guia / exemplo de problemas relevantes retirado dos valores obtidos. Logo, a simulação e cenários foram elaborados para forçar tais ocorrências.

A partir dos valores obtidos, e seguindo o esperado de uma arquitetura totalmente relacionada ao número de conexões, espera-se encontrar um conjunto de eventuais problemas nas arquiteturas. Um conjunto exemplo destes problemas estão listados na Tabela 4.1.

Tabela 4.1: Possíveis conjuntos para a análise.

Recursos							Descrição
CPU	Memória	Rede Entrada	Rede Saída	Conexões Simultâneas	Tempo de Resposta	Latência	
↑				↖			Rotina de processamento de requisições está ocupando muita CPU
	↑			↓			O microsserviço está armazenando informações as quais poderiam estar alocadas em outros microsserviços
		↑		↓			Uma entrada de dados elevada pode indicar o uso de um protocolo inapropriado para o serviço
			↑	↓			Caso a saída esteja muito elevada pode indicar uma configuração inapropriada de elementos que são transitados na rede ou uso inadequado de protocolos
				↓	↑		Pode estar relacionado ao desempenho de processamento, modelo de paralelismo ou congestionamento de rede
				↓	↑	↑	Está relacionado com congestionamento da rede
↓		↑	↑				Possível gargalo na rede ou protocolo ineficiente
↑	↑	↓	↓				Possível gargalo nos algoritmos utilizados no serviço
				↓			Bloqueio de novas conexões pelo sistema operacional ou modelo de paralelismo
↑	↑	↑	↑	↑	↓	↑	Límite de processamento da arquitetura
↓	↓	↓	↓	↑	↓	↓	Teste ideal

Fonte: O próprio autor.

Não foram encontrados trabalhos para guiar a caracterização dos dados, sendo assim a caracterização foi definida genericamente. Dessa forma, a linha de base para a caracterização será encontrada ao decorrer da análise, utilizando dos valores de testes com poucas conexões (nenhum cliente e um cliente), esperando que o serviço escale linearmente conforme o inicio do processo. A linha de base definida será uma das contribuições para trabalhos futuros.

A Tabela 4.1 relaciona os recursos conforme dois padrões:

- Valores acima da média (\uparrow).
- Valores baixo da média (\downarrow).

Espera-se encontrar problemas mais detalhados, além de problemas padronizados de forma genérica na Tabela 4.1. Pode ser possível identificar eventuais problemas conforme o tipo de requisição e projeto da arquitetura. Estes problemas servirão como guias na análise final das arquiteturas.

Segundo estes critérios de análise, torna-se necessário definir um plano de testes a fim de obter os dados conforme os cenários e casos de uso definidos no atual trabalho. Tais testes servem para ocasionar situações nos serviços a fim de obter dados para posterior análise.

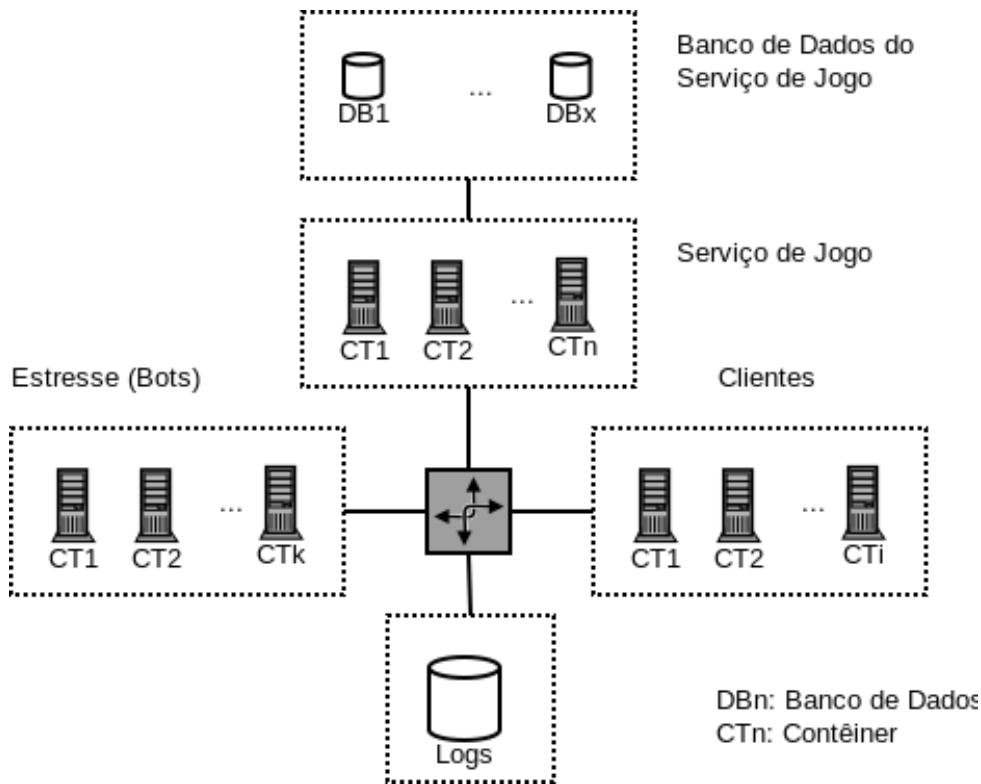
4.3 Plano de testes

O plano de testes define os cenários que serão aplicados sobre as arquiteturas de microserviços para jogos MMORPG selecionadas. Esta seção serve para descrever formas de estressar as arquiteturas, a fim de obter valores para análise. Entretanto, antes de relatar os cenários de teste, é importante descrever o ambiente no qual serão realizados os experimentos. A Figura 4.1 descreve a infraestrutura utilizada para execução das camadas de aplicação utilizadas nos testes.

Como visível na Figura 4.1, o ambiente de testes planejado está organizado em cinco regiões. Essas regiões foram isoladas com o objetivo de diminuir o impacto de desempenho e consumo de recursos por outras ferramentas durante a coleta de dados. Por este motivo, as regiões da infraestrutura planejada são:

1. **Serviço de Jogo:** A camada de serviço da infraestrutura dos testes concentrará a arquitetura dos microsserviços referente às arquiteturas de microsserviços analisadas.
2. **Banco de dados do serviço de jogo:** A camada de banco de dados do serviço de jogo conterá os serviços de dados e web a fim de manter um padrão de banco de dados para ambos os serviços utilizados e auxiliar na inicialização dos testes.

Figura 4.1: Ambiente de testes definido para a coleta de dados.



Fonte: O próprio autor.

3. **Estresse:** A camada de estresse será responsável por realizar requisições ao serviço a fim de estressá-lo, simulando padrões de requisição de um padrão de um jogador.
4. **Cliente:** A camada de cliente será composta pelos mesmos elementos da camada de estresse, porém em um ambiente controlado para que a alta demanda dos clientes neste ambiente não interfira nas métricas obtidas.
5. **Dados:** A camada de dados será composta por um banco de dados de *log* a fim de armazenar os dados obtidos da camada Cliente e Serviço, posteriormente utilizado exclusivamente na coleta de dados.

Tais regiões da infraestrutura utilizada no ambiente de testes devem manter um padrão ao qual seu propósito é garantir a inexistência de interferência entre os testes, focando em obter métricas válidas para posterior análise. Dessa forma, espera-se dividir as aplicações conforme a sua rede, facilitando o seu monitoramento. Entretanto, por se tratar de um sistema baseado em serviço, espera-se utilizar uma considerável parte do mesmo sistema de cliente para as três arquiteturas, excluindo-se os casos no quais a arquitetura necessite de alterações.

Para os casos de uso, serão utilizadas as arquiteturas de microsserviços específicos a jogos MMORPG obtidos da literatura. São essas elas:

1. Arquitetura Rudy (Subseção 2.5.1), na qual baseia-se na segregação de jogadores por canais;
2. Arquitetura Salz (Subseção 2.5.2), na qual baseia-se em gerar muitos serviços escaláveis; e
3. Arquitetura Willson (Subseção 2.5.3), na qual baseia-se em extrair microsserviços de regras de negócio mais custosas.

Tais arquiteturas vão impactar o serviço de jogo, banco de dados e as requisições as quais os clientes deverão realizar. Espera-se obter os valores referente a diferença de consumo de recursos computacionais dentro de cenários controlados utilizando o ambiente de testes.

Com o objetivo de obter dados, torna-se claro a necessidade de estresse das arquiteturas em múltiplos casos diversos, garantindo assim a confiabilidade dos dados obtidos. Dessa forma, foi desenvolvido um cenário que comporte ambas as arquiteturas de microsserviços propostas na análise. Este cenário possibilita a execução do experimento junto a simulação de clientes.

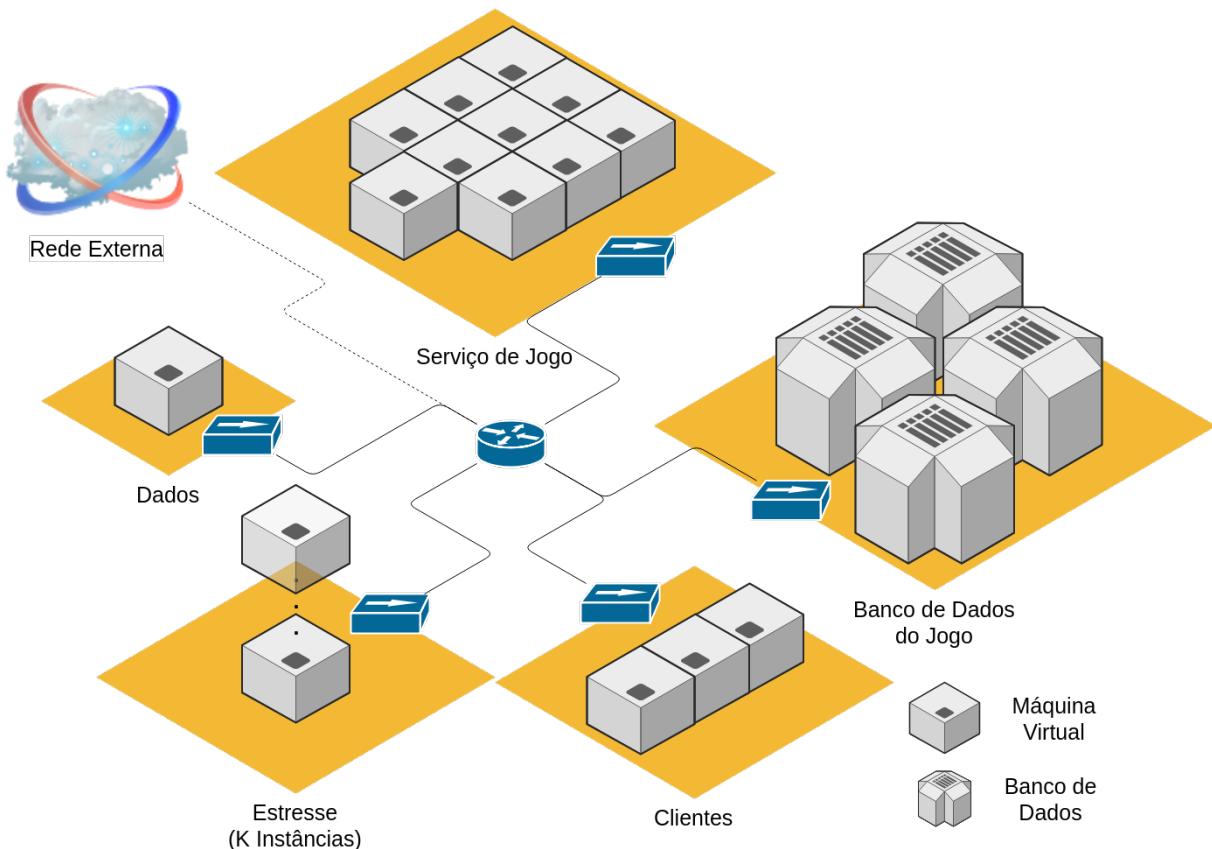
4.3.1 Cenário

O cenário reflete diretamente sobre o ambiente proposto para esta análise. Este será executado sobre cinco camadas, nas quais cada camada estará isolada em sub-redes em uma nuvem de computadores.

O cenário será composto por cinco sub-redes, as quais cada uma será responsável pela operação de uma região do ambiente planejado (Figura 4.2). Esta divisão física em redes facilitará a orquestração dos serviços na rede, seguindo boas práticas de implantação de microsserviços. Além disso, pode-se garantir que a interface do serviço público para o cliente segue o padrão de um serviço MMORPG real.

O cenário, exibido na Figura 4.2 descreve cinco sub-redes. Estas redes são descritas da seguinte forma:

Figura 4.2: Rede de execução dos testes.



Fonte: O próprio autor.

1. **Dados:** Armazena e processa os dados obtidos da arquitetura. É usado pelas redes *Cliente* e *Serviço de Jogo*.
2. **Banco de dados do serviço de jogo:** Armazena dados do jogo. É usado pela rede *Serviço de Jogo*.
3. **Serviço de Jogo:** Executa sistemas *web* e *RPC*, em específico da camada de serviço de jogo. Gera métricas para os serviços na rede *Dados* e manipula os bancos de dados na rede *Banco de dados do serviço de jogo*.
4. **Estresse:** Executa múltiplos clientes, a fim de estressar o serviço de Jogo.
5. **Clientes:** Executa um número controlado de clientes para obter métricas de tempo de resposta e latência entre as redes *Cliente* e *Serviço de Jogo*.

Conforme a descrição de cada rede, existe uma interdependência dentre as redes. Tal interdependência pode ser visualizada na Tabela 4.2.

A Tabela 4.2 refere-se a dependência das redes, conforme os serviços necessários

Tabela 4.2: Tabela de interdependência das sub-redes.

<i>Linha depende de Coluna</i>	Dados	Banco de dados do serviço de jogo	Serviço de Jogo	Estresse	Clientes
Dados	–	Não	Não	Não	Não
Banco de dados do serviço de jogo	Não	–	Não	Não	Não
Serviço de Jogo	Sim	Sim	–	Não	Não
Estresse	Não	Não	Sim	–	Não
Clientes	Sim	Não	Sim	Não	–

Fonte: O próprio autor.

para o seu funcionamento. Portanto, espera-se bloquear o tráfego de pacotes entre redes que são independentes.

A implantação do serviço de jogo utilizará métodos de implantação de micro-serviços, com ferramentas para gerenciamento de nós em uma rede de contêineres. Por este motivo, todas as máquinas virtuais da rede *Serviço de Jogo* terão os mesmos recursos, facilitando o comportamento do gerenciador ao escalar os serviços.

A implantação dos bancos de dados do serviço de jogo devem ser implantadas diretamente no sistema operacional da máquina virtual. Seguindo assim uma boa prática de não armazenar dados dentro de contêineres. Por sua vez, estas máquinas virtuais serão focadas em armazenamento.

A implantação dos clientes sobre a rede *Clientes* executará um número limitado de contêineres fixo sobre as máquinas virtuais, para que não estresse as instâncias desta rede. Nesse sentido, os recursos computacionais requeridos por estas instâncias são menores em relação aos demais.

A estrutura de implantação da rede *Estresse* será dada por um gerenciador de nós em uma rede de contêineres. Esta rede terá mais instâncias sob demanda conforme o caso de teste.

O limite de recursos do cenário é definido na Tabela 4.3, sendo definida a

faixa de endereços de cada rede a qual foi projetada. Estes valores podem ser alterados conforme a necessidade dos testes, tendo seus recursos reduzidos ou incrementados.

Tabela 4.3: Limite de recursos por instância de cada rede.

Nome da Rede	Rede	Instâncias	Armazenamento / Ins.	N. Núcleos	Memória
Dados	10.0.*.* / 255.255.0.0	1	250GB	4	4GB
Banco de dados do serviço de jogo	10.51.*.* / 255.255.0.0	4	100GB	4	2GB
Serviço de Jogo	10.52.*.* / 255.255.0.0	10	25GB	4	4GB
Estresse	10.100.*.* / 255.255.0.0	N	25GB	4	4GB
Clientes	10.101.*.* / 255.255.0.0	3	10GB	2	1GB

Fonte: O próprio autor.

A partir da Tabela 4.3, espera-se definir um limite de recursos máximos utilizados pelo Serviço de Jogo. A única rede que pode variar conforme o teste será a rede Estresse, visto que a demanda para estressar uma rede pode mudar conforme as características do teste.

A partir deste cenário é definido qual o comportamento dos clientes na execução dos testes. Nesse sentido, uma definição das características mínimas de regra de negócio, padrão de comportamentos e interface esperada para o serviço e cliente é necessária.

4.3.2 Simulação de Clientes

Utilizando a simulação de clientes objetiva-se estressar as arquiteturas utilizando um ataque com *bots*. Neste cenário, para padronizar a coleta de dados, todos os *bots* terão a mesma rotina evitando assim um comportamento aleatório, a qual pode descharacterizar os dados obtidos para análise.

Porém, como requisito para estipular as requisições, faz-se obrigatório realizar um levantamento de requisitos no qual tanto o serviço quanto o cliente devem implementar. Nesse sentido, a Tabela 4.4 relaciona a funcionalidade com o impacto de implementação de tal funcionalidade.

A Tabela 4.4 relata uma lista de funcionalidades mínimas que serão executadas na simulação. A partir destas funcionalidades, pode-se definir quais requisições estarão disponíveis na API para o cliente requisitar ao serviço. A lista de comandos públicos é exibida na Tabela 4.5.

A Tabela 4.5 descreve todos os comandos que estarão disponíveis na rede. Além destes, serão implementados outros comandos para API privada do serviço. Porém, os

Tabela 4.4: Requisitos das funcionalidades e respectivo impacto de implementação.

Requisito	Descrição	Implementação
Identificação	Gera uma numeração única (token) com base em uma tupla de dados.	Será implementado utilizando algoritmo de <i>hash</i> , de forma a garantir que este token seja único e diferente a cada implementação.
Autenticação	Recebe o token e garante que não existe nenhuma conexão utilizando o mesmo token.	Será implementado usando um serviço de chave valor, como o Redis.
Seleção de Personagem	Uma conexão deve requerer o controle de um personagem.	Será implementado utilizando uma árvore de cena interna ao serviço, onde o tipo do nó será Personagem e o seu nome será o nome do personagem.
Envio de Mensagem	Será possível enviar mensagens e receber. Elas serão baseadas na região. Será mantido uma distância fixa para todos os casos de uso.	Deve existir uma estrutura de busca interna ao serviço para consultar personagens de uma região em relação a um usuário.
Movimentação	Será possível movimentar o personagem para as células adjacentes. Isso indica que o posicionamento do personagem será baseado em uma matriz.	Esta ação deve comunicar a atualização para todos os demais jogadores da região de interesse.
Ataque	Ao atacar, o jogador causará um dano aleatório baseado em seu nível em todos os inimigos das células adjacentes.	Esta ação irá manipular a árvore de objetos da cena do serviço e deverá notificar todos os jogadores desta área de interesse.
Consumo de Itens	Ao consumir um item, os atributos do personagem serão alterados, influenciando na regra de negócio utilizada nas arquiteturas de teste.	Implicará na utilização de um banco de dados em memória e manipulação de dados não visíveis pela árvore de cena.

Fonte: O próprio autor.

Tabela 4.5: Requisitos mínimos funcionais para a implementação da simulação descrita.

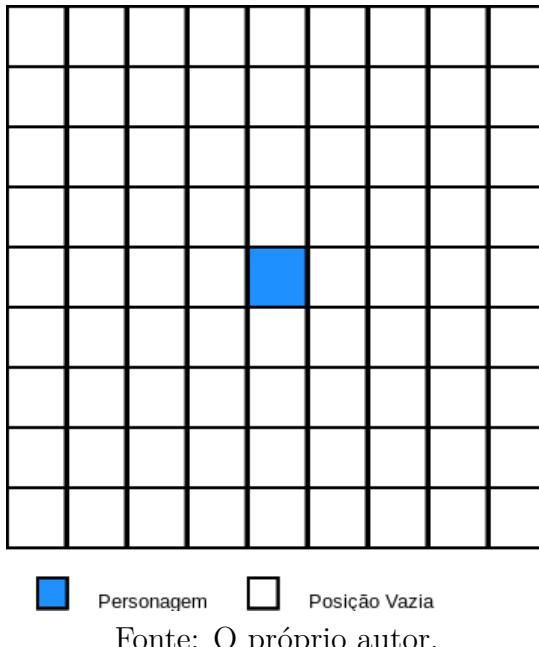
Nome	Argumentos	Retorno	Protocolo	Direção
<i>Auth</i>	<i>Username, Password</i>	JSON	Web	Cliente para Serviço
<i>CreateAccount</i>	<i>Username, Password</i>	JSON	Web	Cliente para Serviço
<i>UpdateAccount</i>	<i>Username, Password</i>	JSON	Web	Cliente para Serviço
<i>CreateCharacter</i>	<i>Token, Character Name</i>	JSON	Web	Cliente para Serviço
<i>DeleteCharacter</i>	<i>Token, Character ID</i>	JSON	Web	Cliente para Serviço
<i>SelectCharacter</i>	<i>Token, Character ID</i>	JSON	RPC	Cliente para Serviço
<i>WalkTo</i>	<i>Token, PosX, PosY</i>		RPC	Cliente para Serviço
<i>ConsumeItem</i>	<i>Token, Item</i>		RPC	Cliente para Serviço
<i>AttackHere</i>	<i>Token</i>		RPC	Cliente para Serviço
<i>SendMessage</i>	<i>Token, Message</i>		RPC	Cliente para Serviço
<i>UpdateMapEstate</i>	<i>NPC, Action, MoreData</i>		RPC	Serviço para Cliente
<i>UpdateCharacterEstate</i>	<i>NPC, Action, MoreData</i>		RPC	Serviço para Cliente
<i>ReceiveMessage</i>	<i>NPC, Message</i>		RPC	Serviço para Cliente
<i>ReBind</i>	<i>IP, Port</i>		RPC	Serviço para Cliente

demais comandos da API privada não serão utilizados pelo cliente, sendo necessariamente um requisito para o funcionamento do serviço com determinada arquitetura.

O ambiente da simulação será baseado em matrizes. Cada mapa pode ser visualizado como uma matriz de 100x100 unidades. Dessa forma, temos um ambiente com valor exato, o qual facilita cálculos internos do serviço e cliente, assim promovendo um ambiente vasto porém sem utilizar recursos de forma exagerada para os testes. Os personagens podem locomover-se para as células adjacentes a sua localização. O raio de interesse é de quatro células. Um exemplo de estado do ambiente do jogo pode ser visualizado na Figura 4.3.

A partir da área de interesse do jogo, como o da Figura 4.3, o *bot* poderá decidir

Figura 4.3: Área de interesse da simulação com raio de quatro células.

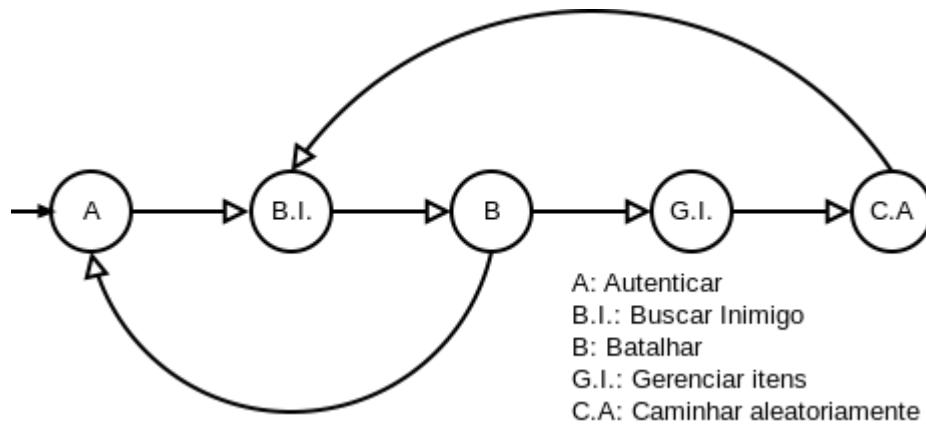


Fonte: O próprio autor.

suas ações baseado em um autômato. Caso ele alcance os extremos do mapa, ele será movimentado para outro mapa. Nos casos de arquiteturas com múltiplos gerenciadores de mundo, será utilizado o comando *ReBind* para realizar a conexão com o microsserviço correto após o translado do personagem.

Todas as ações do *bot* no mapa são baseadas em um autômato. Sendo assim, espera-se obter um padrão de movimentação a fim de evitar ciclos que um jogador comum dificilmente realizará (*e.g.*, Andar para frente e para trás, ficar equipando itens em ciclo, ficar consumindo itens até acabar, *etc.*). A movimentação do personagem seguirá o autômato descrito na Figura 4.4.

Figura 4.4: Autômato de movimentação dos personagens simulados.



Fonte: O próprio autor.

A Figura 4.4 descreve o comportamento de um *bot* no ambiente do jogo simu-

lado. Ele seguirá um padrão de procurar batalhas, batalhar, gerenciar itens do personagem e buscar novas batalhas. Este conjunto de ações simula o comportamento de busca de itens dentro de um jogo MMORPG. As características específicas de cada estado são definidas da seguinte forma:

- Autenticar: Realizará autenticação com o serviço *web* ou *rpc* apropriado a arquitetura. Neste passo o *bot* receberá as informações do seu personagem.
- Buscar Inimigo: Caminhará de forma aleatória a fim de buscar um inimigo, caso não exista em sua área de interesse. Caso encontre em sua área de interesse, irá aproximar deste inimigo.
- Batalhar: Irá atacar um inimigo, aplicando dano a este NPC. O personagem poderá receber dano. Estes valores serão fixos baseados em seu equipamento. O personagem pode perder todos os seus pontos de vida, sendo desconectado do serviço.
- Gerenciar Itens: O *bot* irá consumir itens para recuperar pontos de vida e usar equipamentos melhores aos já utilizados.
- Caminhar Aleatoriamente: O personagem caminhará aleatoriamente por um número de passos máximo. Após isso, voltará a buscar uma batalha.

Esta sequência de ações visa forçar aos *bots* a exploração do cenário. A cada mudança de estado, o jogador irá anunciar no chat a sua troca de ação. Isso contribuirá com o monitoramento do comportamento dos personagens tanto quanto usará a funcionalidade de chat. Para simular a ação de resposta de mensagens de um jogador, cada *bot* que receber uma mensagem terá uma porcentagem fixa de 25% de probabilidade de responder a sua ação no atual momento. Um *bot* não poderá responder a uma mensagem na qual ele mesmo emitiu na região.

Ao realizar um *ReBind* ou transitar de um mapa para o outro, o estado atual do autômato volta para o estado **Autenticar**. Caso já esteja autenticado, continuará a sua busca por inimigos na nova região.

O ambiente final da simulação possui três mapas no eixo horizontal e no eixo vertical, visto que esta é a combinação mínima para que o serviço tenha um mapa central com bordas para efetuar transições de novos personagens.

Após definido as características dos clientes simulados, pode-se definir os testes que serão executados sobre as arquiteturas de microsserviços específicos para jogos MMORPG. Tais testes servem de guia para obter as métricas para a análise das arquiteturas de microsserviços especificadas.

4.3.3 Testes

Os testes que serão executados no atual trabalho esperam analisar a quantia de recursos mínimos para executar os microsserviços e o crescimento de consumo de recursos conforme o crescimento de clientes simultâneos. Nesse sentido, foram definidos os seguintes testes:

1. Executar 3 vezes as arquiteturas com zero jogadores simultâneos, por 5 minutos;
2. Executar 3 vezes as arquiteturas com um jogador apenas; e
3. Executar 3 vezes as arquiteturas com zero jogadores simultâneos, aumentando a cada 30 segundos o número jogadores (um por vez) ao serviço, até o serviço obter algum erro interno de microsserviço e terminar a sua execução por erro interno ou chegar a 100 jogadores simultâneos.

Os dados são capturados ao decorrer do tempo, podendo assim relacionar os recursos consumidos com o número de conexões simultâneas no serviço e com os demais recursos. Esta estrutura auxiliará a visualização e interpretação dos dados para posterior análise.

Todos os testes serão executados utilizando a arquitetura Rudy, Salz e Willson. Eles serão executados sequencialmente, aplicando sobre o cenário cada arquitetura e seus devidos clientes sobre o cenário, a fim de obter as métricas para posterior análise. A partir desta sequência de testes, são obtidos valores suficientes para a análise das arquiteturas de microsserviços específicas a jogos MMORPG. Estes serão utilizados para análise conforme os critérios definidos na Seção 4.2, buscando possíveis gargalos e problemas de escalabilidade do sistema.

4.4 Considerações parciais

Este capítulo definiu os objetivos da análise de microsserviços específicos a jogos MMORPG. Para alcançar tais objetivos, se fez necessário definir quais métricas serão obtidas e como estes dados serão analisados. Por fim, definiu-se qual o ambiente, cenário e testes que serão executados para obter tais dados.

Dessa forma, foi definido que será obtido os valores de uso de CPU, memória, vazão de rede (tanto para entrada quanto saída), número de conexões simultâneas, tempo de resposta das requisições e latência entre cliente e serviço. Estes valores serão analisados conforme possíveis problemas conhecidos, definidos na seção de critérios (Seção 4.2).

A fim de obter tais valores para análise, o plano de testes definiu um ambiente baseado em camadas conforme a demanda das arquiteturas submetidas a análise. Foram projetadas cinco sub-redes no cenário de testes baseado nas regiões definidas no ambiente de testes. A partir destas sub-redes, foi estipulado valores de recursos computacionais limites e números de instâncias máximas. Entretanto, não foi possível definir o limite de instâncias para a sub-rede de estresse, visto que não foi encontrado estudos anteriores para ter uma linha base de consumo de recursos. Dessa forma, o atual trabalho contribuirá conjuntamente com futuros trabalhos guiando o consumo de recursos para as arquiteturas Rudy, Salz e Willson. Por fim, foram definidas as regras de negócio básicas para os clientes simulados, baseados em um autômato.

Os testes que serão executados sobre as redes estão baseados na finalidade de obtenção de valores de recursos mínimos para execução dos serviços e crescimento de valores de recursos conforme um número de clientes simultâneos.

5 Implementação

Este capítulo descreve a implementação dos microsserviços que compõe o jogo em cada arquitetura. Isto é necessário pois as escolhas técnicas, envolvendo as linguagens adotadas, as tecnologias utilizadas e as boas práticas aplicadas no desenvolvimento das aplicações, interferem na qualidade (consumo de recursos e o desempenho das aplicações) dos microsserviços.

Devido às diversas possibilidades de linguagens de programação e bibliotecas disponíveis, a Seção 5.1 tem o objetivo de descrever as tecnologias utilizadas para o desenvolvimento das aplicações e justificar tais escolhas. Além disso, a Seção 5.1 apresenta o conjunto de serviços externos utilizados para facilitar o desenvolvimento dos microsserviços. Os serviços externos, apesar de não estarem diretamente implementados na arquitetura das soluções, foram utilizados durante o processo e por tal motivo devem ser evidenciados.

Diante da complexidade das arquiteturas de microsserviços desenvolvidos, torna-se necessário descrever quais microsserviços foram implementados, e como estão dispostos na rede. Para este fim, a Seção 5.2 contextualiza a interconexão dos microsserviços desenvolvidos.

5.1 Tecnologias Utilizadas

A seleção das tecnologias e bibliotecas que são executadas nas aplicações é importante, visto que estas implicam diretamente no desempenho e consumo de recursos das arquiteturas selecionadas. Entretanto, essa seleção precisa estar de acordo com as regras de negócio impostas para os testes.

Inicialmente, existe uma preocupação com a linguagem de programação utilizada, visto que esta precisa ter um bom desempenho, um suporte a programação paralela e, ao mesmo tempo, conter bibliotecas que auxiliem no desenvolvimento ágil dos serviços. Neste sentido, foi levantado um conjunto de linguagens de programação, o qual viabiliza o projeto de acordo com os seguintes critérios:

- Alto desempenho para programas paralelos;
- Biblioteca para conexão com banco de dados PostgreSQL;
- Biblioteca para uso de cache (Redis);
- Biblioteca para escrita de serviços RPC, sobre o protocolo TCP;
- Biblioteca para escrita de serviços *web*, com API no formato JSON;
- Linguagem compilada;
- Linguagem com tipagem estática; e
- Simplicidade de escrita de código, preferencialmente.

A partir destes critérios, a linguagem selecionada foi a linguagem GoLang, por se tratar de uma linguagem a qual o autor possui domínio e satisfaz os critérios aqui listados. Outro critério, o qual é implícito para o desenvolvimento, é a homogeneidade da linguagem de programação em todos os microsserviços. Tal necessidade gera o benefício da escrita de código reutilizável, como o núcleo de regras de negócio que pode ser aplicado a todas as arquiteturas desenvolvidas.

Assim, foi selecionada uma única linguagem de programação compatível com a viabilização do projeto. Em função da diversidade de bibliotecas disponíveis, também é necessário definir quais bibliotecas foram utilizadas para o desenvolvimento das arquiteturas dos microsserviços. Dentre as bibliotecas utilizadas, destacam-se:

- gin: Servidor Web para serviços de alto desempenho;
- gorm: Biblioteca de objetos relacionais, a qual suporta conexão direta ao banco PostgreSQL;
- go-redis: Biblioteca para conexão ao serviço Redis;
- net/rpc: Biblioteca nativa da linguagem GoLang para escrita de serviços RPC sobre o protocolo TCP;
- graphite: Biblioteca de conexão ao banco de métricas; e
- testify: Biblioteca auxiliar a suíte nativa de testes da linguagem.

Estas bibliotecas foram utilizadas na camada de infraestrutura, sendo seu uso aplicado a todas as arquiteturas. A camada de infraestrutura é responsável pela realização da interação entre a rede e as regras de negócio dos microsserviços.

Destaca-se, dentre as bibliotecas citadas, a biblioteca `testify`. Tal biblioteca foi utilizada na garantia de integridade das aplicações desenvolvidas. Sua finalidade é realizar a inspeção automatizada do funcionamento da aplicação. Entretanto, a biblioteca não é utilizada durante a execução da aplicação, denotando-se como uma biblioteca auxiliar.

As bibliotecas auxiliares foram utilizadas no processo de integração contínua. O processo de integração contínua foi implantado utilizando os serviços externos Github e TravisCI. Este processo é descrito como processo de construção na arquitetura Willson, sendo responsável pelo teste e envio de imagens de contêineres a um serviço de registro na nuvem.

Após descrever as tecnologias utilizadas, faz-se necessário descrever o ambiente distribuído desenvolvido. Esta descrição é necessária a fim de garantir uma melhor visibilidade dos microsserviços implementados, na qual são citados na análise.

5.2 Interconexão entre os microsserviços

A interconexão entre os microsserviços refere-se ao projeto de rede de uma arquitetura qualquer, disponibilizando visualmente a camada de rede. Ao todo foram implementados onze microsserviços, os quais possuem a comunicação através de troca de mensagens. Tais conexões entre si, com os bancos de dados e com os seus respectivos clientes, foram definidas pelos autores das arquiteturas. Consequentemente, torna-se relevante descrever a interconexão entre os microsserviços, exibindo assim seus protocolos de comunicação de rede:

- A arquitetura Rudy contém um microsserviço intermediário para conexão com o banco de dados. Esta disposição de microsserviços está exposta na Subseção 5.2.1.
- A arquitetura Salz remove a responsabilidade de mensageria do jogo do microsserviço de jogo. Outra alteração é a exposição do microsserviço de autenticação como uma interface pública na rede, permitindo a conexão a partir dos clientes. Esta disposição de microsserviços da arquitetura Salz é exposta na Subseção 5.2.2.

- A arquitetura Willson tem o funcionamento semelhante a arquitetura Rudy, porém não utiliza um microsserviço intermediário para organização de consultas ao banco de dados. A disposição dos microsserviços nesta arquitetura é visível na Subseção 5.2.3.

As arquiteturas foram desenvolvidas com escopo limitado. Esta limitação é descrita na Subseção 4.3.2, no qual evidenciam-se quais ações os clientes realizam no serviço. Tal redução de escopo visa analisar as funcionalidades básicas de um serviço MMORPG evitando regras de negócio específicas.

5.2.1 Rudy

A arquitetura Rudy foi implementada em versão reduzida para o escopo dos testes. A arquitetura Rudy implementada contém quatro microsserviços e dois banco de dados, sendo um para dados permanentes (PostgreSQL) e um para dados temporários (Redis). Os seus microsserviços estão listados na Tabela 5.1.

Tabela 5.1: Microsserviços da arquitetura Rudy.

Nome	Protocolo	Público na Rede
rgame	RPC/TCP	✓
rweb	HTTP/JSON	✓
rauth	RPC/TCP	
rcrud	RPC/TCP	

Fonte: O próprio autor.

A Tabela 5.1 relaciona os microsserviços aos protocolos na qual respondem e a sua visibilidade, do ponto de vista do cliente. Isto significa que, microsserviços públicos a rede podem e são acessados pelos clientes. Os microsserviços da arquitetura Rudy, descritos na Tabela 5.1, possuem funcionalidades próprias. As funcionalidades de cada microsserviço são:

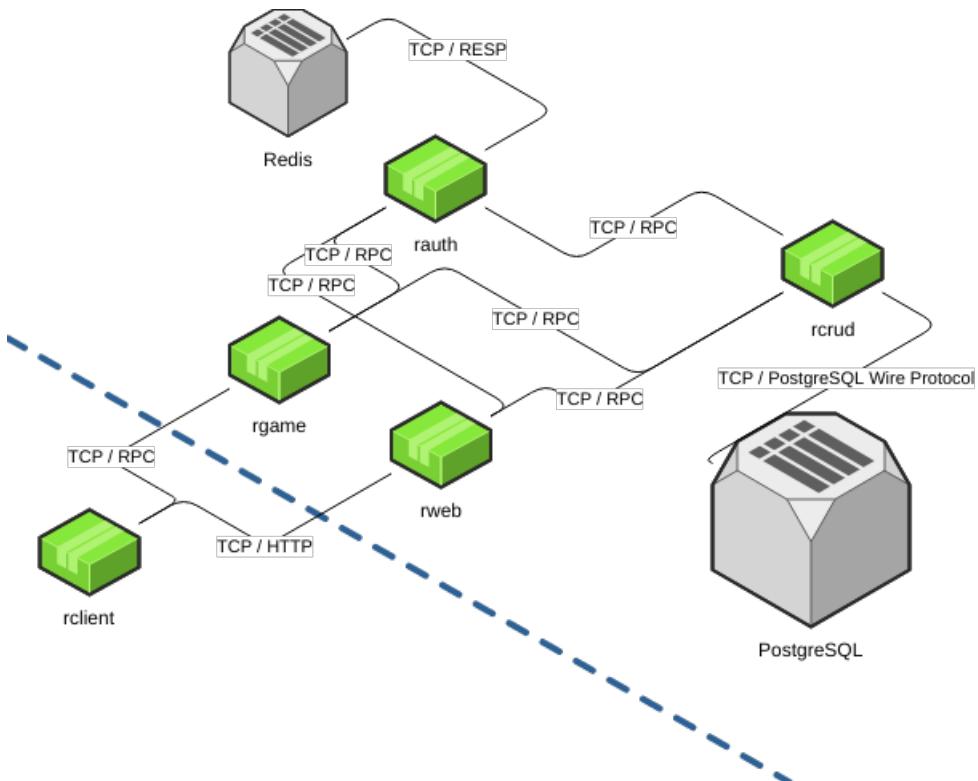
- *rgame*: Gerencia o posicionamento dos personagens, a execução das operações do mundo e a troca de mensagens do *chat* do jogo. Ele também é responsável por autenticar contas ao mundo do jogo.
- *rweb*: Recebe requisições web como uma API, essas requisições podem ser referentes a criação de novas contas ou personagens para o jogo.

- *rauth*: Autentica e garante uma única conexão por conta em determinada arquitetura.
- *rcrud*: Realiza o acesso ao banco de dados, retornando os dados da consulta a qual é requisitada.

Dado tais características, temos um ponto de atenção no microsserviço *rcrud*, o qual cria uma camada de acesso ao banco de dados. Este padrão de projeto tende a facilitar a manutenção do banco de dados, centralizando todas as operações a dados temporários e permanentes.

O objetivo do microsserviço *rcrud* é mover o escopo de consulta a dados em um ponto centralizado, evitando assim falhas ao armazenar e obter tais dados. Este processo é escalável, entretanto adiciona uma camada maior ao obter os dados, na qual tende a aumentar o tempo de resposta para as operações ao banco. A disposição da arquitetura Rudy está visível na Figura 5.1.

Figura 5.1: Interconexões da arquitetura Rudy.



Fonte: O próprio autor.

Outro ponto de atenção é o acesso ao microsserviço de autenticação *rauth*, o qual pode ser visualizado na Figura 5.1. Tal microsserviço é privado, dessa forma o microsserviço de mundo fica responsável por ser um ponto intermediário de comunicação

entre o cliente e o serviço que realiza a autenticação. Este padrão de projeto é comum em aplicações Web, entretanto não recomendado pelo padrão sugerido JWT.

Mesmo não sendo um padrão recomendado, tal característica não deve ter um impacto significativo para arquiteturas de jogos MMORPG, visto que esta operação é realizada poucas vezes no serviço. O padrão JWT recomenda tornar o microsserviço de autenticação público, na qual o cliente realiza a autenticação diretamente com o microsserviço de autenticação, reduzindo o tráfego interno de rede.

5.2.2 Salz

A arquitetura Salz, em versão reduzida para o escopo dos testes, contém quatro microsserviços e dois bancos de dados, sendo um para dados permanentes (PostgreSQL) e um para dados temporários (Redis). Os seus microsserviços estão listados na Tabela 5.2.

Tabela 5.2: Microsserviços da arquitetura Salz.

Nome	Protocolo	Público na Rede
sgame	RPC/TCP	✓
schat	RPC/TCP	✓
sweb	HTTP/JSON	✓
sauth	RPC/TCP	✓

Fonte: O próprio autor.

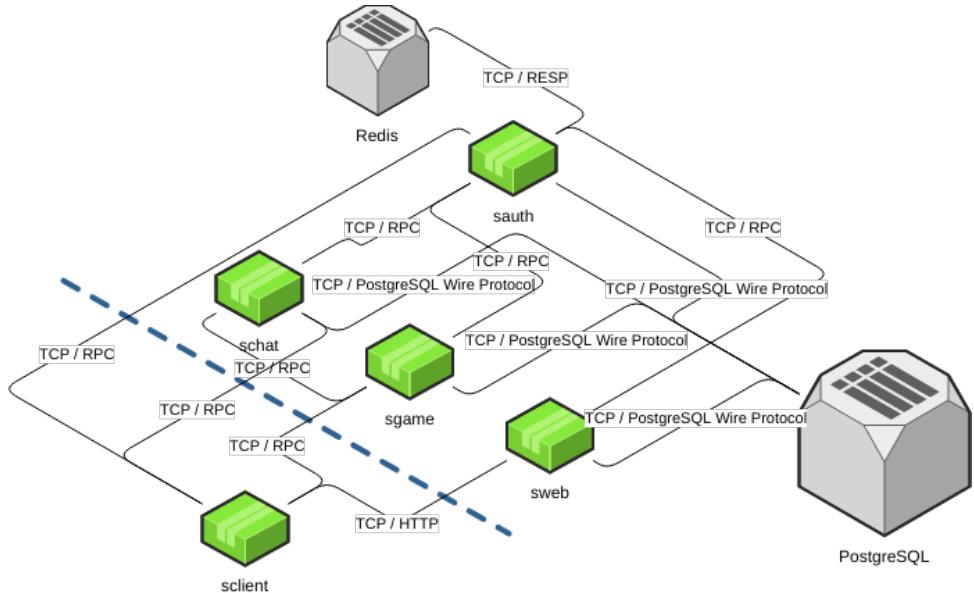
Os microsserviços da arquitetura Salz, descritos na Tabela 5.2, possuem funcionalidades próprias. A funcionalidade de cada microsserviço é:

- *sgame*: Gerencia o posicionamento dos personagens, execução das operações do mundo e troca de mensagens do chat do jogo.
- *schat*: Gerencia mensagens entre jogadores e possui um forte acoplamento com o microsserviço *sgame*.
- *sweb*: Recebe requisições web como uma API a fim de criar novas contas e novos personagens para o jogo.
- *sauth*: Autentica e garante uma única conexão por conta em determinada arquitetura.

Ao definir as funcionalidades dos microsserviços da arquitetura Salz, nota-se que o microsserviço *schat* possui algumas particularidades. Tal microsserviço é responsá-

vel por gerenciar todo o serviço de chat da arquitetura, seja baseado em região de interesse ou global. Entretanto, a informação de posicionamento não está armazenada neste microsserviço. Nesse sentido, esta característica necessita de uma sincronização de dados, a qual implica diretamente em custo de recurso ou tempo de resposta para o cliente. A topologia de rede da arquitetura Salz pode ser visualizada na Figura 5.2.

Figura 5.2: Interconexões da arquitetura Salz.



Fonte: O próprio autor.

Outra característica desta arquitetura é a comunicação direta entre o microsserviço de autenticação *sauth* e o cliente. Isto evita uma conexão intermediária, a qual utilizaria o microsserviço *sgame*. A comunicação direta realizada permite que exista uma maior vazão de dados sobre autenticação e sessão. Outro ponto a ser ressaltado é a possibilidade de utilizar JWT para autenticação, sem a necessidade de verificação pelo próprio microsserviço de autenticação.

Com a interconexão da arquitetura Salz sendo feita de forma direta, o sistema de autenticação torna-se independente, permitindo assim adicionar novos serviços de forma flexível. Também nota-se uma possível utilização de verificação de sessão por assinatura digital. Essa assinatura digital é utilizada no microsserviço de autenticação, diminuindo o consumo da rede para mensagens de verificação pelo serviço de autenticação. Entretanto, não foi utilizado tal método de validação de sessão.

O método de autenticação atual consiste na validação de assinatura utilizando o banco de dados de dados temporários (*cache*). Esse método utiliza uma comunicação entre os microsserviços *schat* e *sgame* com o microsserviço *sauth*. Esta escolha foi realizada

para manter os mesmos métodos de autenticação em todas as arquiteturas, evitando variação de algoritmo ou tecnologia.

Definidas as características da arquitetura Salz, a sincronização de posições entre o microsserviço *schat* e *sgame* pode ser um ponto de atenção, logo deve-se atentar a frequência da sincronização necessária pela regra de negócio do jogo implementado.

5.2.3 Willson

A arquitetura Willson, em versão reduzida para o escopo dos testes, contém três microsserviços e dois bancos de dados. Os microsserviços de banco de dados são um para dados permanentes (PostgreSQL) e um para dados temporários (Redis). Os microsserviços da arquitetura Willson estão listados na Tabela 5.3.

Tabela 5.3: Microsserviços da arquitetura Willson.

Nome	Protocolo	Público na Rede
wweb	HTTP/JSON	✓
wgame	RPC/TCP	✓
wauth	RPC/TCP	

Fonte: O próprio autor.

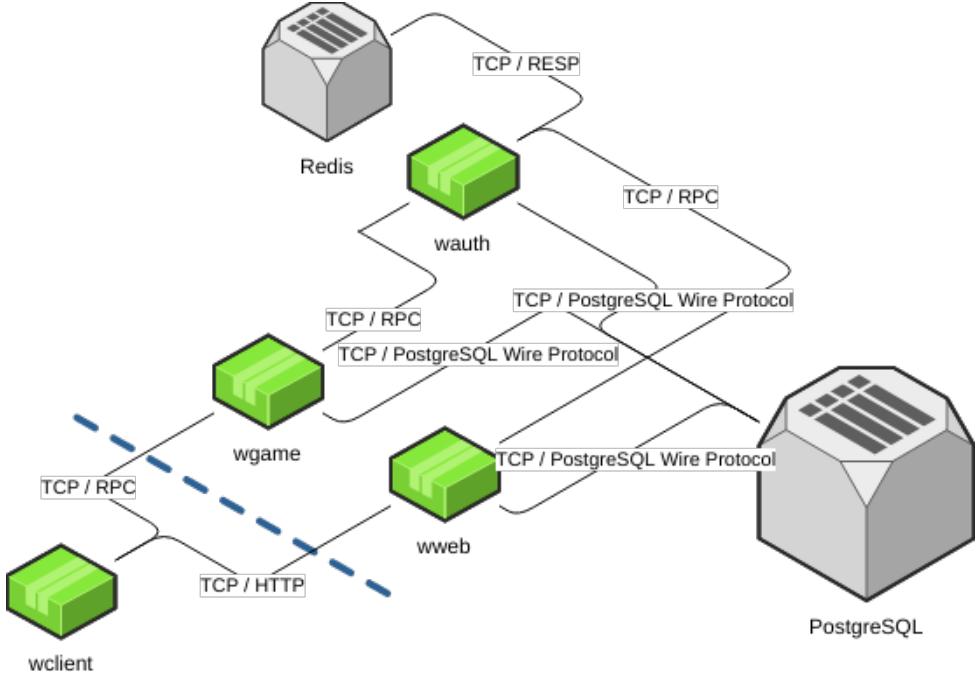
Os microsserviços da arquitetura Willson, descritos na Tabela 5.3, possuem funcionalidades próprias. As funcionalidades de cada microsserviço são:

- *wgame*: Gerencia o posicionamento dos personagens, a execução das operações do mundo e a troca de mensagens do chat do jogo e também gerencia a sessão do jogador, que têm um forte acoplamento com o microsserviço *wauth*.
- *wweb*: Recebe requisições web como uma API a fim de criar novas contas e novos personagens para o jogo.
- *wauth*: Autentica e garante uma única conexão por conta em determinada arquitetura.

Definidos os microsserviços da arquitetura Willson, na Tabela 5.3, uma característica interessante é a ausência de um microsserviço de gerência de mensagens de *chat* e um intermediário para gestão de dados do banco. Estas características tornam a arquitetura mais simples, dividindo os microsserviços em domínios específicos, os quais

têm foco em escalabilidade. A interconexão da arquitetura Willson pode ser visualizada na Figura 5.3.

Figura 5.3: Interconexões da arquitetura Willson.



Fonte: O próprio autor.

A Arquitetura Willson, conforme visualizado na Figura 5.3, simplifica a interconexão dos microsserviços e reduz um microsserviço de sua topologia. Dessa forma, espera-se que ocorra uma diminuição no consumo de rede e no tempo de resposta quando comparado as demais arquiteturas.

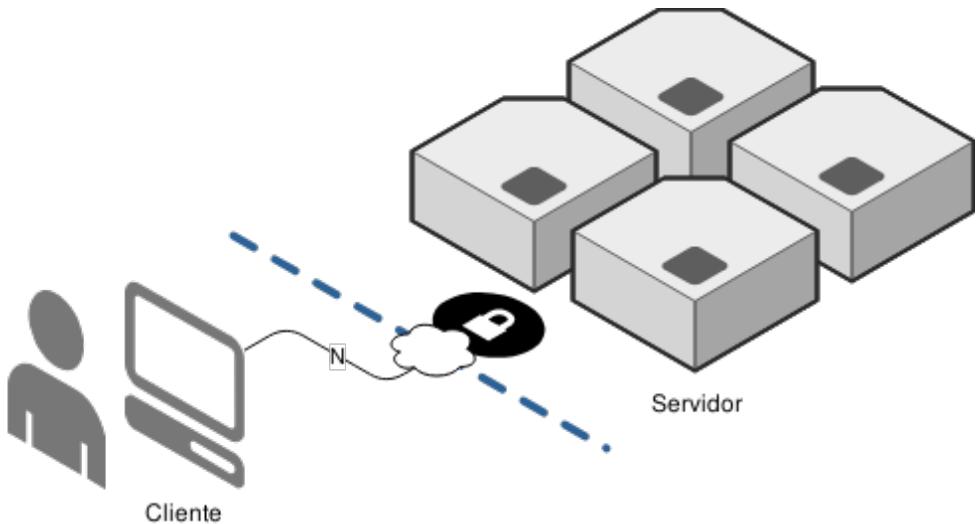
5.3 Ambiente de Testes

O ambiente de testes refere-se ao ambiente físico ou virtual a qual as arquiteturas e clientes foram implantados. Este ponto é importante para permitir a reprodução dos experimentos executados.

Em um cenário de jogos massivos, encontram-se duas regiões distintas. Em uma visão superficial, estas regiões provém do paradigma Cliente-Servidor, como visível na Figura 5.4.

A Figura 5.4 exibe um usuário que utiliza um cliente genérico para conectar em um servidor genérico. Este servidor pode ser uma ou mais máquinas físicas ou virtuais, utilizando múltiplos processos. Em específico, para jogos MMORPG, o cliente estará

Figura 5.4: Cenário Cliente Servidor Genérico.



Fonte: O próprio autor.

em uma rede diferente do servidor, visto que o objetivo do jogo é prover um mundo compartilhado.

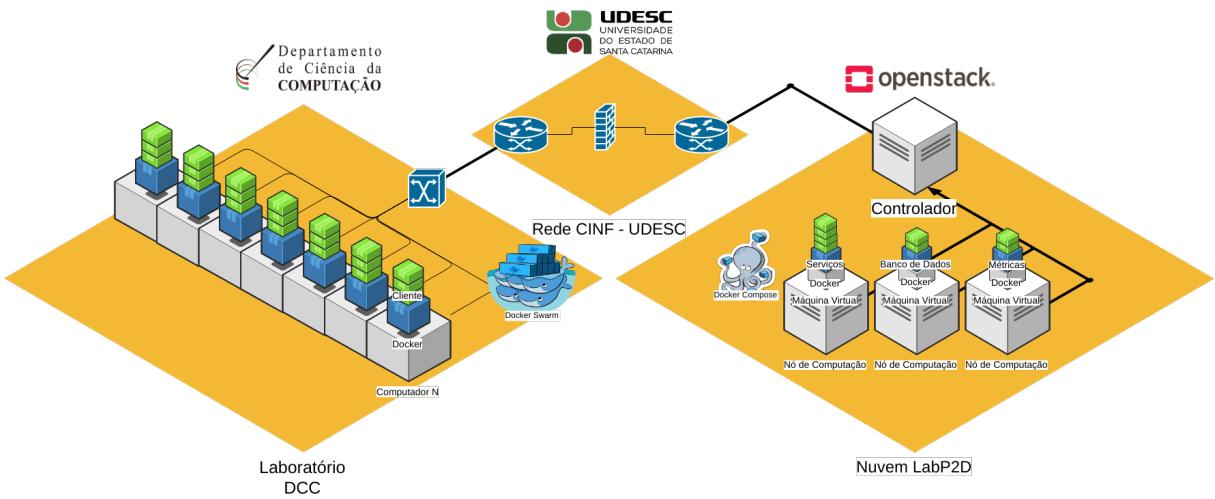
A rede utilizada pelo cliente usualmente é uma rede LAN, a qual conecta-se pela Internet na rede do servidor. Para os testes, tal rede possui características específicas para gerenciar a implantação dos clientes de forma automatizada, a qual será abordado na Subseção 5.3.2.

Neste contexto, há duas redes que separam os ambientes de microserviços e serviços satélites destas aplicações. Esta separação física é necessária nos testes para adicionar um limitador de transporte de dados e latência, tal qual ocorre em aplicações que executam em produção para a categoria de jogos MMORPG. Esta separação é visível na Figura 5.5.

As redes do cliente e servidor, visíveis na Figura 5.5, são implantadas em plataformas diferentes. Os dados técnicos de cada rede são descritos nas Subseções 5.3.1 e 5.3.2.

Nota-se que existe uma rede intermediária desempenhando o papel da separação física entre servidor e cliente. No atual trabalho esta rede pertence ao Departamento de Ciência da Computação (DCC).

Figura 5.5: Cenário Cliente Servidor Genérico.



Fonte: O próprio autor.

5.3.1 Rede do Servidor

A rede do servidor é implantada sobre uma infraestrutura virtualizada, gerenciada pelo OpenStack na versão *Rocky*. Tal estrutura garante um ambiente próximo ao utilizado em produção para serviços MMORPG.

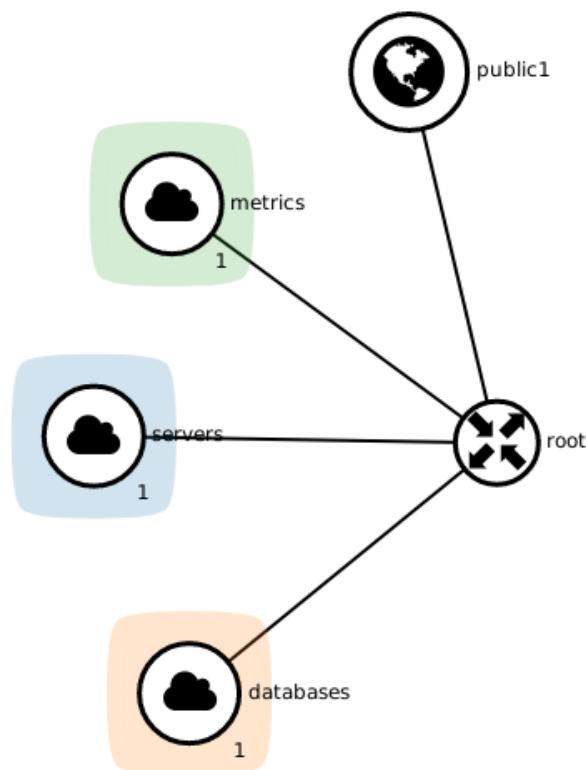
Entretanto, como exibido na Seção 5.2, os serviços para jogos MMORPG são compostos por múltiplos processos implementando uma arquitetura de microsserviços. Por este motivo, torna-se interessante dividir o ambiente do servidor em sub-redes. Esta divisão pode ser visualizada na Figura 5.6

Como exibido na Figura 5.6, para o contexto deste trabalho, faz sentido dividir em sub-redes para os seguintes contextos:

- Microsserviços: Contém os processos das arquiteturas Rudy, Salz e Willson;
- Banco de Dados: Contém os processos de banco de dados permanente e temporário; e
- Métricas: Contém serviços para armazenamento de dados em ordem temporal.

Esta divisão garante o isolamento apropriado para gerenciamento dos projetos na rede, facilitando a implantação dos microsserviços. Sua realização ocorreu utilizando sub redes no sistema OpenStack, no qual é definido como um sistema de gerenciamento de unidades de computação, armazenamento e comunicação sobre um aglomerado com-

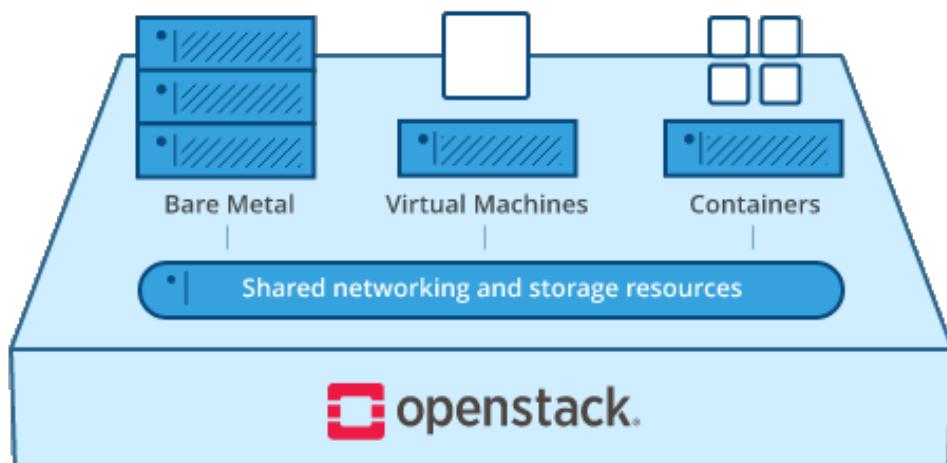
Figura 5.6: Sub-redes do servidor



Fonte: O próprio autor.

putacional (OpenStack Team, 2019). O escopo geral do OpenStack é visualizado na Figura 5.7.

Figura 5.7: Arquitetura abstráida do OpenStack



Fonte: (OpenStack Team, 2019).

Como visualizado na Figura 5.7, o ambiente virtual é gerado sobre máquinas servidoras, nas quais são gerenciadas pelo OpenStack. O ambiente virtual pode conter serviços, como por exemplo bancos de dados, armazenamento de arquivos, máquinas virtuais ou contêineres (OpenStack Team, 2019).

Entretanto, no contexto dos testes, faz sentido utilizar máquinas virtuais de tal forma que pode-se executar *scripts* para acessar dados da máquina virtual e métricas do gestor de contêineres. Desta forma, torna-se independente do ambiente físico real que está executando as aplicações.

Conforme a escolha de execução do servidor sobre máquinas virtuais, surge a necessidade da configuração de sub-redes no ambiente do servidor. Certamente é possível construir um ambiente com uma única sub-rede realizando esta divisão por redes do gestor de contêineres, porém tal escolha não é uma boa prática para organização do projeto, visto que existem partes do serviço que possuem contextos diferentes. Deste modo, foram criadas sub-redes de acordo com os contextos levantados na Figura 5.6. A Tabela 5.4 mostra a divisão de tais subredes, conforme a faixa de *Internet Protocol (IP)* aplicada.

Tabela 5.4: Subredes da rede do servidor.

Nome da subrede	Faixa
metrics	192.168.0.0/24
databases	192.168.1.0/24
servers	192.168.2.0/24

Fonte: O próprio autor.

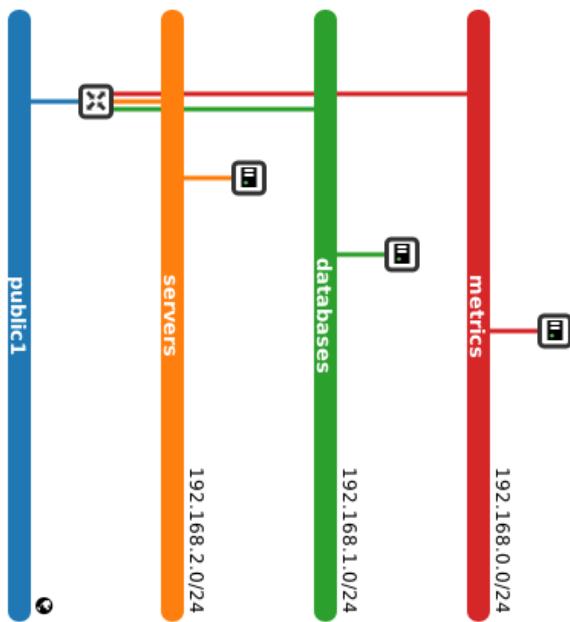
As subredes, descritas na Tabela 5.4, foram criadas seguindo um padrão de faixa de IP conforme a configuração exigida pela nuvem Tchê. As faixas de IP dependem das redes externas a nuvem computacional gerenciada pelo OpenStack. Entretanto, tal configuração não implica na performance da aplicação, somente em sua organização.

Cada subrede possui uma máquina virtual para execução dos serviços. O esquema das máquinas virtuais está visível na Figura 5.8.

Conforme visível na Figura 5.8, cada sub-rede possui uma única máquina virtual, sendo que esta executará um gerenciador de contêineres com os serviços necessários para a execução do servidor. As máquinas virtuais por sua vez possuem configurações distintas e essas configurações são exibidas na Tabela 5.5.

Conforme exibido na Tabela 5.5, a única variação entre as configurações das máquinas virtuais do servidor é o armazenamento da máquina virtual *metric_machine*.

Figura 5.8: Sub-redes do servidor com as máquinas virtuais



Fonte: O próprio autor.

Tabela 5.5: Configurações das máquinas virtuais do servidor.

Nome	Sistema Operacional	IP	vCPU	Memória	Armazenamento
server_machine	Ubuntu LTS 18.04 Bionic Beaver	192.168.2.*	4 Cores	8 GB	40GB
database_machine	Ubuntu LTS 18.04 Bionic Beaver	192.168.1.*	4 Cores	8 GB	40GB
metric_machine	Ubuntu LTS 18.04 Bionic Beaver	192.168.0.*	4 Cores	8 GB	120GB

Fonte: O próprio autor.

Tal característica é dada pela necessidade de armazenamento de métricas, nas quais precisam de volumes de dados dedicados a este serviço.

Em todas as sub-redes, por padrão, foi utilizado o gestor de contêineres Docker Compose. Este gestor de contêineres permite executar aplicações de multi-contêineres usando como entrada um arquivo *YAML Ain't Markup Language* (YAML). Este formato é específico para notação de dados legíveis a humanos.

Entretanto, a rede do cliente interfere diretamente na execução dos experimentos, visto que possui características próprias. Nesse sentido, torna-se necessário a descrição da rede do cliente.

5.3.2 Rede do Cliente

Os clientes executaram sobre uma rede LAN. Estes computadores formam um aglomerado de contêineres para realizar o acesso as arquiteturas na rede do servidor (Subseção 5.3.1). Os computadores utilizados possuem características próprias, as quais devem ser citadas

para melhor compreendimento das características da rede de clientes.

O aglomerado computacional é composto por nove computadores. Todos os computadores possuem 8 *cores*, 16 GB de memória e 1TB de armazenamento.

Em especial, percebe-se que este aglomerado é composto por características homogêneas. Este ambiente foi montado com tais características para remover complexidade, evitando possíveis problemas de diferença de sistema operacional ou hardware.

Para formar o aglomerado de contêineres foi utilizado a tecnologia *Docker Swarm*. Tal tecnologia permite o controle da infraestrutura pelo *Docker Engine* (Serviço de controle de contêineres) de forma remota, podendo-se definir os serviços em arquivos YAML, a partir das máquinas administradoras do aglomerado.

Outra característica importante do *Docker Swarm* é que, pelo fato de ser uma interface de gerenciamento do *Docker Engine*, podemos obter dados de consumo de recursos individuais por contêineres. Nesse sentido, permite-se obter os recursos consumidos por contêineres registrado no serviço Docker.

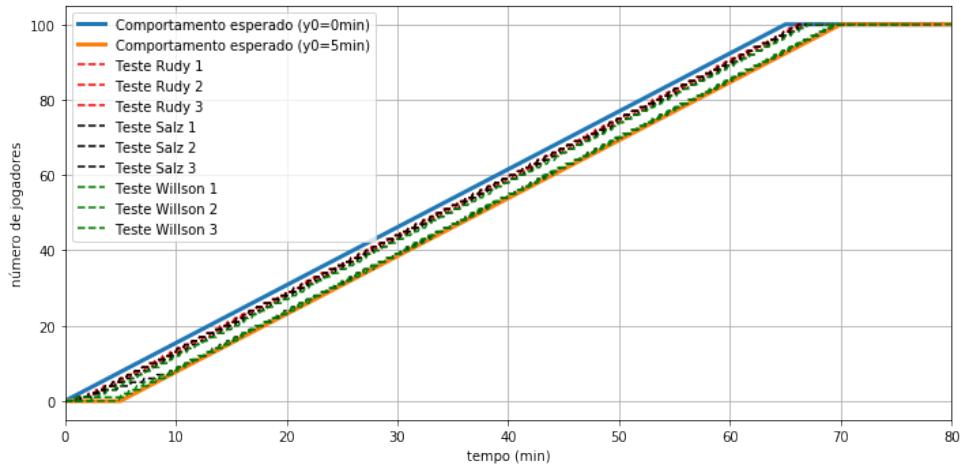
Seguindo o padrão de testes, definido na Subseção 4.3.3, foi executado o seguinte protocolo de testes:

1. Executar 3 vezes as arquiteturas com zero jogadores simultâneos, por 5 minutos;
2. Executar 3 vezes as arquiteturas com um jogador apenas; e
3. Executar 3 vezes as arquiteturas com zero jogadores simultâneos, aumentando a cada 30 segundos o número jogadores (um por vez) ao serviço, até o serviço obter algum erro interno de microsserviço e terminar a sua execução por erro interno ou chegar a 100 jogadores simultâneos.

A partir deste protocolo, podemos analisar a estabilidade, junto ao script de coleta de dados, as características do servidor sem sofrer cargas, um único jogador e N jogadores. Os dados coletados nos testes com zero e um jogadores mostraram que as arquiteturas estavam estáveis, porém não trouxeram dados suficientes para mostar diferenças entre as arquiteturas. Nesse sentido, a análise não exibirá tais gráficos, já que a única conclusão para todos é da estabilidade e baixo consumo de recursos, sem relevância estatística entre ambas.

Para realizar o teste com um número crescente de jogadores simultâneos foi utilizado um script na linguagem Ruby na qual escala automaticamente um novo contêiner do cliente a cada 30 segundos, na rede de clientes. Este comportamento produz uma progressão de carga linear, que é visível na Figura 5.9.

Figura 5.9: Quantidade de jogadores simultâneos cresce linearmente.



Fonte: O próprio autor.

A Figura 5.9 exibe o crescimento linear de jogadores simultâneos. Tal crescimento está diretamente relacionado ao número de contêineres escalados no aglomerado. Cada jogador realiza as operações descritas na Figura 4.4.

É notório que o número de jogadores simultâneos é exatamente igual ao número de contêineres na rede de clientes. Dessa forma, garantimos o isolamento entre contêineres, evitando multiplexação de conexões entre processos diferentes.

A partir do comportamento definido na Figura 5.9, espera-se que o consumo dos recursos e tempo de resposta corresponda ao seu crescimento, tendendo a uma complexidade linear mínima. Dessa forma, pode-se analisar o comportamento do consumo de recurso, dado o número de jogadores simultâneos.

5.4 Considerações Parciais

Este capítulo descreve a implementação e implantação das arquiteturas, levando em consideração o meio físico na qual foi implantado. Para realizar tal descrição, é necessário citar tecnologias utilizadas e como são conectados.

Durante o desenvolvimento da implantação, é utilizado a linguagem Golang,

a qual permitiu o desenvolvimento de forma rápida e eficiente nas aplicações para este trabalho.

Esta escolha implica diretamente nas tecnologias utilizadas. Dessa forma, faz sentido escolher tecnologias que combinem com os objetivos das arquiteturas.

Nesse sentido foram utilizados serviços de armazenamento de dados permanentes e temporários *OpenSource*. Redis e PostgreSQL são os bancos de dados utilizados. Esta escolha é dada pela fácil integração com a linguagem e bibliotecas selecionadas para o desenvolvimento.

Durante o processo de desenvolvimento foi utilizado o processo de integração contínua. Dessa forma, é utilizado o serviço TravisCI para realização de testes e construção de imagens Docker. Por sua vez, o TravisCI utiliza o DockerHub para armazenamento das imagens, as quais são utilizadas para implantar sobre a infraestrutura descrita como ambiente de testes.

Para a execução dos testes, foi utilizado um script para escalar contêineres de forma automática. Dessa forma temos um comportamento linear, podendo utilizar tal comportamento como gráfico, auxiliando na análise.

6 Experimentos & Análises

A partir dos dados coletados, foi realizada uma análise dos dados seguindo os critérios levantados durante a proposta do atual trabalho. Tal análise visa compreender o consumo dos recursos empregados pelas arquiteturas com base em suas características específicas.

Cada caso de análise é dividido em experimentos, no qual cada experimento tem o objetivo de analisar um conjunto de recursos dado algum cenário. Nesse sentido, faz sentido agrupar tais experimentos, agrupados na Seção 6.1, para posterior comparação.

6.1 Experimentos

Cada experimento utiliza as arquiteturas de microsserviços para jogos MMORPG Rudy, Salz e Willson. Cada arquitetura é executada no mesmo ambiente, devidamente isolado, em momentos diferentes. Para cada arquitetura, foram realizadas três execuções com os mesmos parâmetros para assegurar que um padrão de comportamento existe. Cada execução de um experimento, é seguido o protocolo:

1. Início dos serviços do banco de dados.
2. Início dos serviços de jogo.
3. Início dos clientes.
4. Clientes são adicionados de 1 em 1, a cada 30 segundos, até atingir a quantidade de 100 clientes simultâneos.
5. Após a finalização do experimento os serviços em todos os ambientes são removidos, mantendo somente os dados capturados.

A partir de tais execuções, os dados são capturados, processados e analisados. Durante as execuções, os dados sobre a latência da rede mostraram-se estáveis, variando entre 5ms e 15ms, dessa forma foram ignorados como um experimento a parte, porém servem para validar a estabilidade da rede.

6.1.1 Tempo de Resposta

Este experimento visa analisar o tempo de resposta, em relação ao número de jogadores simultâneos. Espera-se que o seu crescimento seja de tendência linear junto ao crescimento de jogadores concorrentes. Neste contexto existem os seguintes valores:

- Jogadores simultâneos: Variável capturada a partir do microsserviço de jogo.
- Tempo de Resposta: Variável capturada a partir dos clientes.

Tal experimento permitirá analisar a qualidade das arquiteturas de microserviços. As características observadas nestas análises servem também como critério de desempate nas análises de experimentos.

Nota-se que o número de jogadores simultâneos e o tempo de resposta são indexados pelo tempo a qual tais dados foram capturados. Nesse sentido, pode-se relacionar o tempo de resposta de todos os clientes ao número de jogadores simultâneos. Dado tal contexto, faz sentido realizar uma análise separando os ambientes baseando-se em algumas ações na qual operam sobre microsserviços e recursos diferentes.

Dentro deste contexto, existem dois tipos de requisições: as que não pertencem a estrutura de repetição infinita do autômato do cliente e as requisições que pertencem a estrutura de repetição infinita do autômato do cliente. As requisições que não pertencem a estrutura de repetição infinita são:

- Criar conta;
- Criar personagem;
- Iniciar sessão; e
- Instanciar personagem.

As requisições que pertencem a estrutura de repetição infinita são:

- Movimentar personagem;
- Enviar mensagem; e
- Receber mensagem.

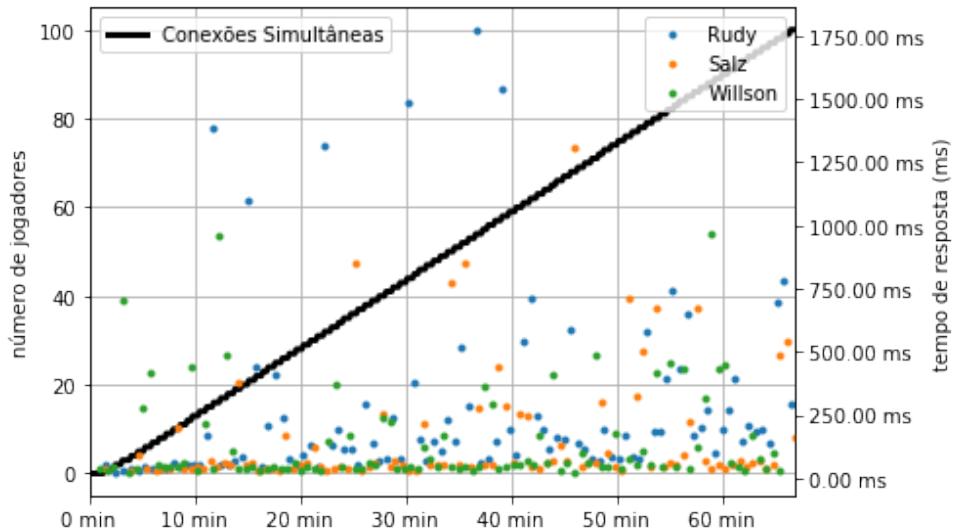
Criar conta

A operação de criar de conta é efetuada sobre o protocolo TCP/HTTP. Nesta operação, o cliente envia um formulário em formato JSON com informações básicas comuns em jogos MMORPG para uma instância de conta. Os dados são validados e inseridos no banco de dados, caso a validação esteja correta.

Uma característica importante para esta requisição é apresentada ao exibir a sua frequência no tempo de contato entre um usuário desta aplicação e o serviço. A tendência é que um usuário realize uma única requisição durante todo o seu tempo de vida ao entrar em contato com algum jogo MMORPG.

O microsserviço responsável por receber estas requisições não lida com concorrência conforme a demanda dos jogadores simultâneos. A Figura 6.1 torna visível que o comportamento do tempo de resposta não depende do número de jogadores simultâneos.

Figura 6.1: Tempo de resposta para criar contas



Fonte: O próprio autor.

Porém, existe, visivelmente, na Figura 6.1 uma disparidade entre dados em relação a arquitetura utilizada. Esta disparidade exibe poucos pontos demorando muito para executar a operação de criação de conta.

Dado o contexto, o único recurso compartilhado com microsserviços que recebem conexões concorrentes é o acesso aos dados da arquitetura. Tal acesso é dado diretamente aos bancos de dados Redis e PostgreSQL ou, no caso da arquitetura Rudy, pelo microsserviço *rcrud*. Nesse sentido, existe pela natureza da formação dos serviços uma diferença no tempo de resposta perceptível aos clientes.

Para quantificar a Figura 6.1, faz sentido buscar os valores de média, variância, máximo e mínimo dos dados obtidos. A Tabela 6.1 exibe estes valores estatísticos.

Tabela 6.1: Média, Variância, Máximo e Mínimo da operação Criar Conta

Arquitetura	Média	Variância	Máximo	Mínimo
Rudy	263,09 ms	115701,29	1774,0 ms	19,0 ms
Salz	152,53 ms	50575,35	1304,0 ms	28,0 ms
Willson	135,98 ms	30573,78	968,0 ms	20,0 ms

Como exibido na Tabela 6.1, pode-se perceber que existe uma diferença abrupta na média de tempo de resposta da API e na variância. Os valores de máximo e mínimo somente são exibidos como pontos de atenção, entretanto como visível na Figura 6.1, são poucas requisições que extrapolam a região de maior densidade de requisições, justificando uma grande variância.

No contexto do atual trabalho, pode-se assumir a variância como disparidade entre os dados. Quanto maior o valor, mais instável é o tempo de resposta de determinada operação ao serviço, entretanto esta mesmo sendo instável tenderá a responder em uma média de tempo.

A partir deste contexto, pode-se definir que a média de tempo de resposta do serviço, ao criar uma nova conta, segue a seguinte relação:

$$\overline{CriarConta_w} < \overline{CriarConta_s} < \overline{CriarConta_r}$$

Deste modo, entende-se:

- $\overline{CriarConta_r}$: Tempo de resposta médio da operação criar conta na arquitetura Rudy;
- $\overline{CriarConta_r}$: Tempo de resposta médio da operação criar conta na arquitetura Salz; e
- $\overline{CriarConta_r}$: Tempo de resposta médio da operação criar conta na arquitetura Willson.

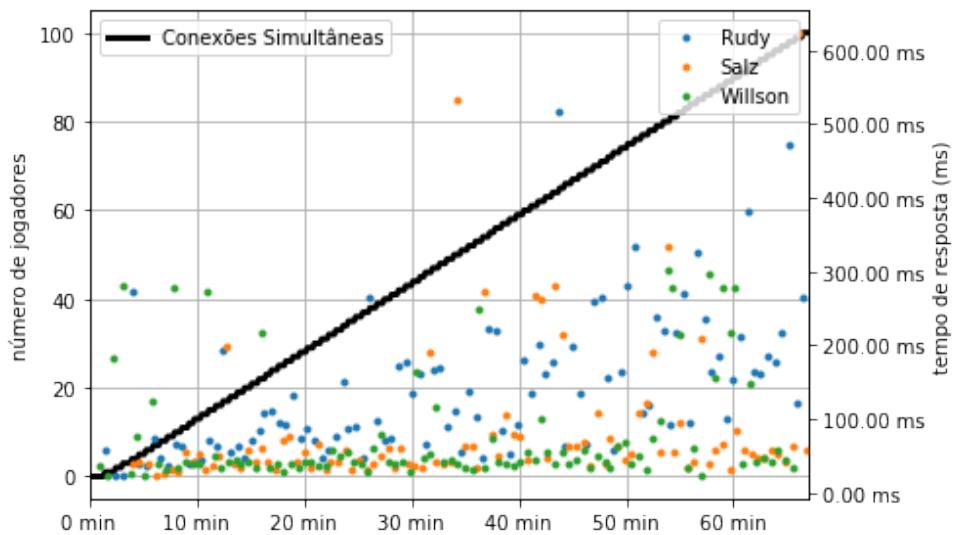
Dessa forma, pode-se dizer que para a operação de criar conta, em média, do ponto de vista de tempo de resposta, a arquitetura Willson é superior a arquitetura Salz

e Rudy, respectivamente. Também pode-se afirmar que a estabilidade desta operação, do ponto de vista de tempo de resposta, também segue a mesma sequência.

Criar personagem

A operação de criar de personagem é efetuada sobre o protocolo TCP/HTTP. Nesta operação, o cliente envia um formulário em formato JSON com informações básicas comuns em jogos MMORPG para uma instância de personagem e dados de autenticação da conta. Os dados são validados e inseridos no banco de dados, caso a validação esteja correta.

Figura 6.2: Tempo de resposta para criar personagens



Fonte: O próprio autor.

Como pode-se perceber através da Figura 6.2, o comportamento do tempo de resposta da perspectiva do cliente tem comportamento similar a operação criar conta (Subseção 6.1.1). Nesse sentido, faz sentido aplicar o mesmo método de análise para validar se existem comportamentos similares, haja vista que a natureza desta operação é próxima.

Segundo o método da média dos dados obtidos, pode-se obter os valores de média, variância, máximo e mínimo. É possível visualizar estes valores na Tabela 6.2.

A partir da Tabela 6.2, pode-se observar que a média dos valores seguem o comportamento encontrado na operação Criar Conta (Subseção 6.1.1). Entretanto, é notório a diferença com relação a estabilidade da operação de criação de personagem.

Para realizar esta operação, o serviço realiza diversas validações, geralmente realizando a troca de mensagens entre os microsserviços de autenticação e validações

Tabela 6.2: Média, Variância, Máximo e Mínimo da operação Criar Personagem

Arquitetura	Média	Variância	Máximo	Mínimo
Rudy	135,43 ms	8782,99	517,0 ms	23,0 ms
Salz	82,48 ms	9106,62	624,0 ms	24,0 ms
Willson	73,66 ms	4886,60	301,0 ms	22,0 ms

Fonte: O próprio autor.

de campos. Tal situação, diferente da operação Criar Conta (Subseção 6.1.1), realiza poucas validações lógicas de regra de negócio aplicando-as diretamente ao banco de dados PostgreSQL.

A partir desta característica, pode-se analisar a diferença na estabilidade entre as arquiteturas Rudy e Salz. Por mais que a arquitetura Salz seja melhor em média - do ponto de vista de tempo de resposta - comparada a arquitetura Rudy, a arquitetura Rudy possui uma maior estabilidade em seu tempo de resposta. Tal estabilidade é proveniente do enfileiramento das requisições no microsserviço *rcrud*.

Mesmo existindo uma flutuação maior na arquitetura Salz, nota-se que seus pontos máximos de tempo de resposta são inferiores aos da arquitetura Rudy. Nesse sentido, por mais que o tempo de resposta tenha uma maior flutuação na arquitetura Salz, a arquitetura Rudy entrega uma menor qualidade. Nesse sentido, deduz-se a seguinte relação:

$$\overline{\text{CriarPersonagem}_w} < \overline{\text{CriarPersonagem}_s} < \overline{\text{CriarPersonagem}_r}$$

Onde entende-se:

- $\overline{\text{CriarPersonagem}_r}$: Tempo de resposta médio da operação criar personagem na arquitetura Rudy;
- $\overline{\text{CriarPersonagem}_s}$: Tempo de resposta médio da operação criar personagem na arquitetura Salz; e
- $\overline{\text{CriarPersonagem}_w}$: Tempo de resposta médio da operação criar personagem na arquitetura Willson.

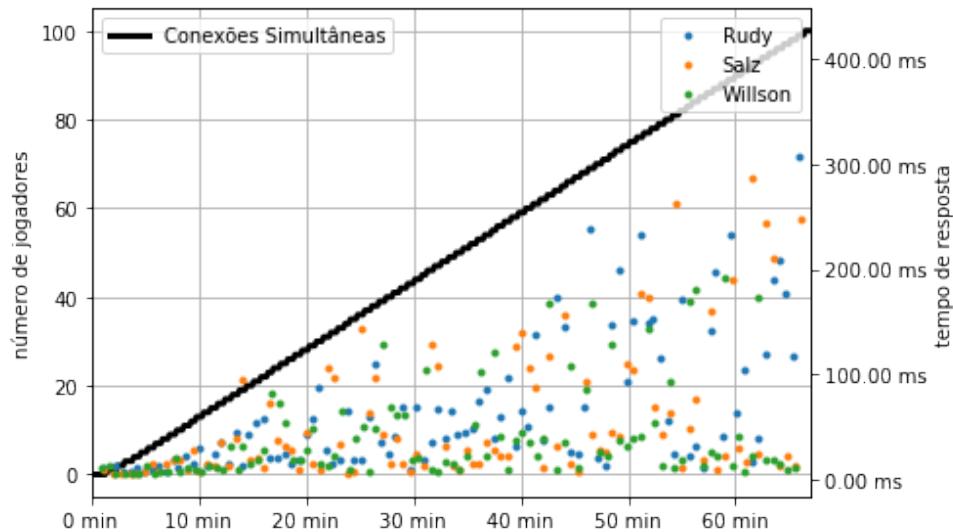
Esta relação também expressa a qualidade de serviço, do ponto de vista de

tempo de resposta, entregue ao usuário, haja vista que a variação dos valores entre as arquiteturas Salz e Rudy são relevantes estatisticamente. Porém, com uma magnitude baixa para utilizar como um argumento válido. Dessa forma, entende-se que a arquitetura que entrega a melhor qualidade de serviço é a Willson, seguida pelas arquiteturas Salz e Rudy, respectivamente.

Iniciar sessão

A operação de iniciar sessão é efetuada sobre o protocolo TCP/RPC. Nesta operação, o cliente envia um formulário em formato *Golang Object Encoding* (GOB) com informações básicas comuns em jogos MMORPG para uma instância de personagem e dados de autenticação da conta. Os dados são validados e o serviço retorna um dado assinado e armazenado para garantir a unicidade da sessão da conta.

Figura 6.3: Tempo de resposta para iniciar sessões



Fonte: O próprio autor.

Tal qual as análises realizadas sobre as operações Criar Conta (Subseção 6.1.1) e Criar Personagem (Subseção 6.1.1), a Figura 6.3 exibe tempo de requisições que foram realizadas somente uma vez por cliente. Entretanto, o tempo de resposta está crescendo linearmente conforme o crescimento dos jogadores, diferente das outras operações.

A partir dessa diferença de comportamento, faz sentido analisar se existe algum crescimento, baseado na diluição destes dados a valores mais significativos para comparação. Nesse contexto, faz sentido analisar o crescimento da média ao dividir as requisições em quadrantes. Estes valores são exibidos na Tabela 6.3.

Tabela 6.3: Tempo de resposta médio dos quadrantes.

Quadrante	Rudy	Salz	Willson
Primeiro	18,6 ms	20,88 ms	15,68 ms
Segundo	45,4 ms	52,38 ms	40,48 ms
Terceiro	86,92 ms	70,88 ms	55,04 ms
Quarto	113,24 ms	57,04 ms	55,48 ms

Fonte: O próprio autor.

A Tabela 6.3 exibe o comportamento de crescimento conforme o crescimento de número de usuários, entretanto a média no último quadrante não aumenta proporcionalmente como nos quadrantes anteriores. Não se pode, dessa forma, dizer que o crescimento é linear.

Outra informação exibida na Tabela 6.3 é que a arquitetura Salz diminui drasticamente a média do seu tempo de resposta no quarto quadrante. Também tem-se um incremento reduzido, se comparado aos demais quadrantes, quando visualiza-se o crescimento do quarto quadrante da arquitetura Willson. Nesse sentido, o pode-se adicionar a variável de estabilidade da API analisada.

Para realizar a análise de estabilidade, faz sentido buscar a variância dos dados de cada quadrante. Dessa forma pode-se analisar se existe uma flutuação maior comparado aos quadrantes anteriores, mesmo com uma média próxima. A Tabela 6.4 exibe a variância dos quadrantes.

Tabela 6.4: Variância dos quadrantes na operação Iniciar Sessão.

Quadrante	Rudy	Salz	Willson
Primeiro	205,92	865,47	247,58
Segundo	477,60	1138,57	960,09
Terceiro	2785,59	4091,28	2307,08
Quarto	4801,06	5192,71	3740,09

Fonte: O próprio autor.

A Tabela 6.4 exibe a informação referente a variação dos dados nos quadrantes. Dessa forma, pode-se afirmar que a variação aumenta conforme a carga do serviço aumenta. Nesse sentido, por mais que a média não apresente comportamento linear, este comportamento linear existe, onde o mesmo é expresso pelo comportamento da variância. Porém, a qualidade de serviço está expresso aos valores médios, e não a variação, a qual única e exclusivamente neste caso somente exibe o comportamento dos dados. Dessa forma, deduz-se a seguinte relação:

$$\overline{IniciarSessao} < \overline{IniciarSessao_w} < \overline{IniciarSessao_r}$$

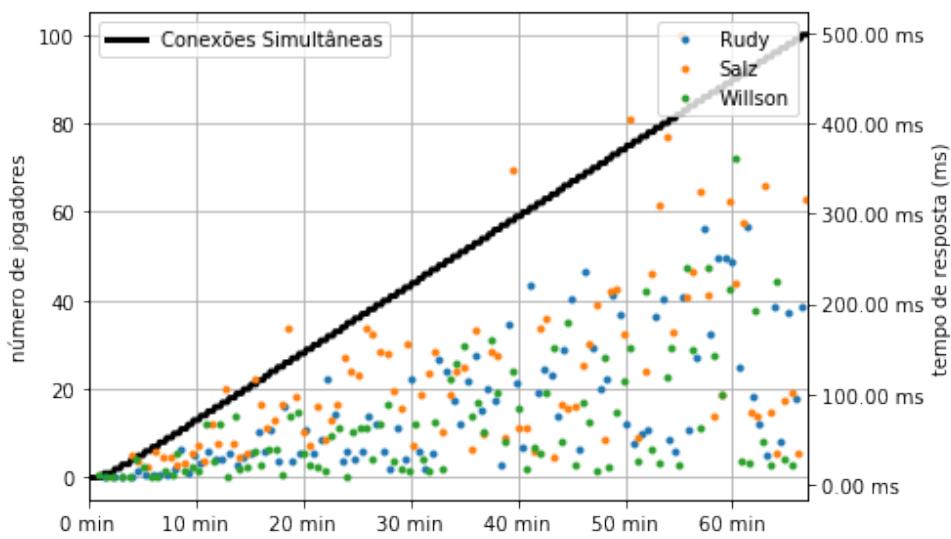
Na qual entende-se:

- $\overline{IniciarSessao_r}$: Tempo de resposta médio da operação iniciar sessão na arquitetura Rudy;
- $\overline{IniciarSessao_s}$: Tempo de resposta médio da operação iniciar sessão na arquitetura Salz; e
- $\overline{IniciarSessao_w}$: Tempo de resposta médio da operação iniciar sessão na arquitetura Willson;

Instanciar personagem

A operação de instanciar personagem é efetuada sobre o protocolo TCP/RPC. Nesta operação, o cliente envia um formulário em formato GOB com informações de autenticação assinadas pelo serviço junto a informação de qual personagem deve ser instanciado no mundo virtual. Os dados são validados e o personagem selecionado é instanciado no mundo.

Figura 6.4: Tempo de resposta para instanciar personagens



Fonte: O próprio autor.

A Figura 6.4 exibe o mesmo comportamento da operação iniciar sessão, abordada na Subseção 6.1.1. Faz sentido aplicar os mesmos métodos para obter informação

neste contexto, obtendo os valores de média e variância por quadrante. Tais informações são exibidas na Tabela 6.5.

Tabela 6.5: Tempo de resposta médio e variância dos quadrantes para instanciar personagem.

Quadrante	Rudy		Salz		Willson	
	Média	Variância	Média	Variância	Média	Variância
Primeiro	25,00 ms	209,04	56,92 ms	1350,23	24,64 ms	368,15
Segundo	59,12 ms	1244,59	107,00 ms	1617,25	42,24 ms	8616,81
Terceiro	123,00 ms	3250,48	138,29 ms	8661,54	86,72 ms	2331,48
Quarto	143,56 ms	6984,89	202,12 ms	14807,94	118,56 ms	8616,81

Fonte: O próprio autor.

Diferente do comportamento da operação Iniciar Sessão (Subseção 6.1.1), ambos os comportamentos de variação e média exibidos na Tabela 6.5 seguem um comportamento linear. Deduz dessa forma que:

$$\overline{InstanciarPersonagem}_w < \overline{InstanciarPersonagem}_r < \overline{InstanciarPersonagem}_s$$

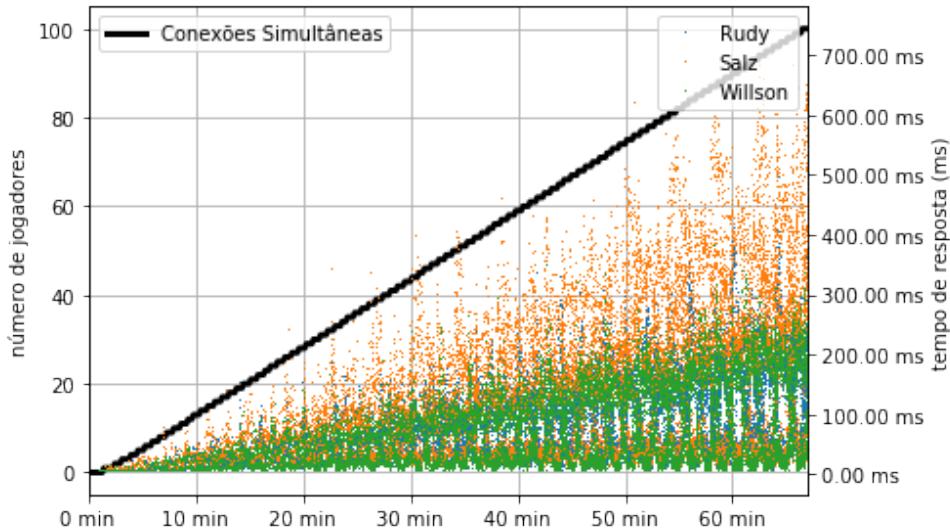
No qual entende-se:

- $\overline{InstanciarPersonagem}_r$: Tempo de resposta médio da operação instanciar personagem na arquitetura Rudy;
- $\overline{InstanciarPersonagem}_s$: Tempo de resposta médio da operação instanciar personagem na arquitetura Salz; e
- $\overline{InstanciarPersonagem}_w$: Tempo de resposta médio da operação instanciar personagem na arquitetura Willson.

Movimentar personagem

A operação de movimentar personagem é efetuada sobre o protocolo TCP/RPC. Nesta operação, o cliente envia um formulário em formato GOB com informações de autenticação assinadas pelo serviço junto a um vetor de direção para onde o personagem deve caminhar no mundo virtual. Os dados são validados e o personagem instanciado é movimentado.

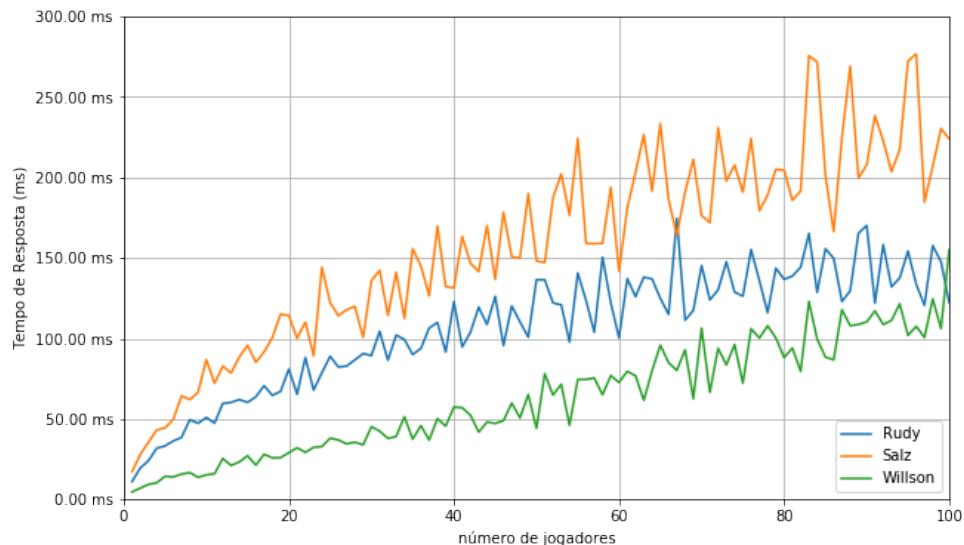
Figura 6.5: Tempo de resposta para movimentar personagens



Fonte: O próprio autor.

A Figura 6.5 exibe os dados obtidos de tempo de resposta ao movimentar o personagem instanciado no *chat* durante a variação do número de jogadores simultâneos. Percebe-se um gráfico denso, dada a quantidade de pontos obtidos nos testes. Dessa forma, faz sentido analisar o comportamento médio por jogador simultâneo. A Figura 6.6 exibe o comportamento médio por jogador.

Figura 6.6: Tempo médio de resposta para movimentar personagem comparado ao número de jogadores



Fonte: O próprio autor.

Visivelmente, a Figura 6.6 permite deduzir dessa forma que:

$$\overline{MoverPersonagem_w} < \overline{MoverPersonagem_r} < \overline{MoverPersonagem_s}$$

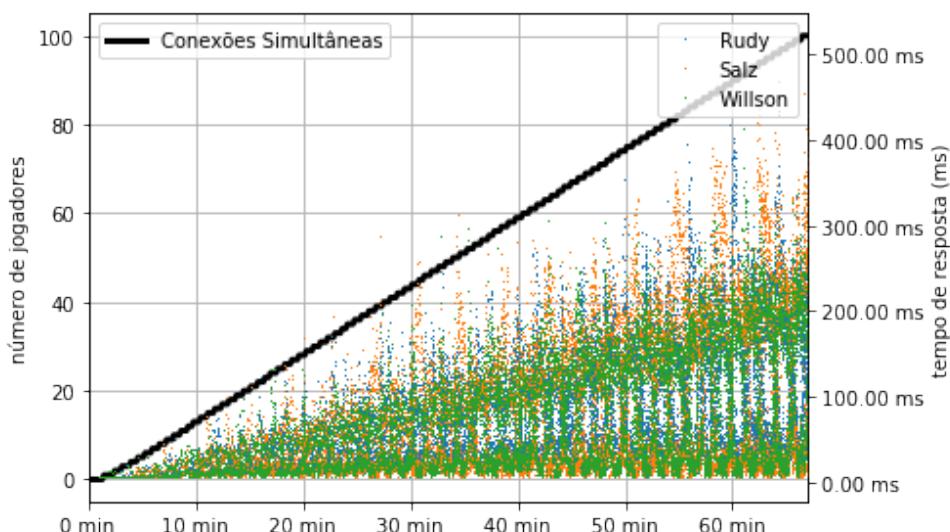
No qual entende-se:

- $\overline{MoverPersonagem_r}$: Tempo de resposta médio da operação mover personagem na arquitetura Rudy;
- $\overline{MoverPersonagem_s}$: Tempo de resposta médio da operação mover personagem na arquitetura Salz; e
- $\overline{MoverPersonagem_w}$: Tempo de resposta médio da operação mover personagem na arquitetura Willson.

Enviar mensagem

A operação de enviar mensagem é efetuada sobre o protocolo TCP/RPC. Nesta operação, o cliente envia um formulário em formato GOB com informações de autenticação assinadas pelo serviço junto a uma linha de texto. Os dados são validados e a linha de texto é distribuído aos personagens dentro do raio de visão do personagem emissor instanciado.

Figura 6.7: Tempo de resposta para enviar mensagens

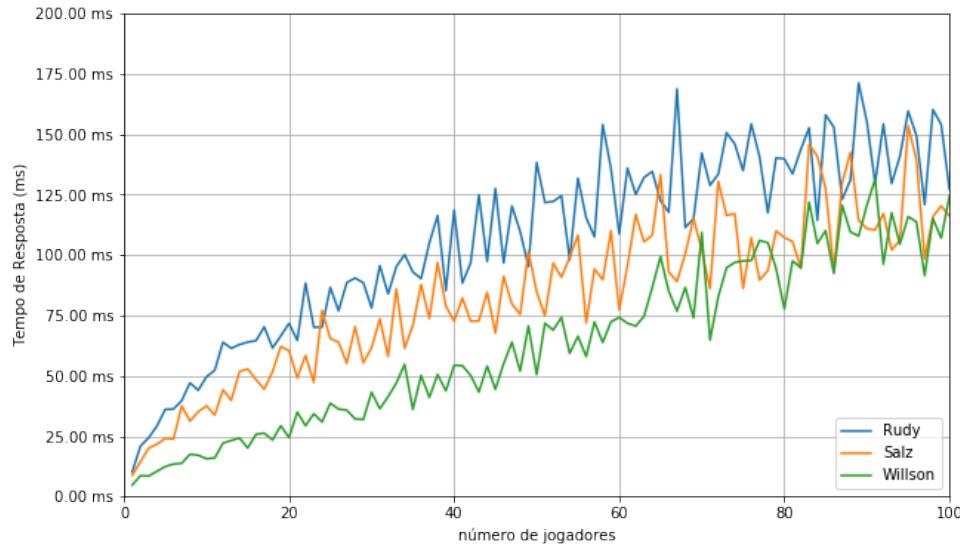


Fonte: O próprio autor.

A Figura 6.7 exibe os dados obtidos de tempo de resposta ao enviar uma mensagem no chat durante a variação do número de jogadores simultâneos. Percebe-se

um gráfico denso, dada a quantidade de pontos obtidos nos testes. Dessa forma, faz sentido analisar o comportamento médio por jogador simultâneo. A Figura 6.8 exibe o comportamento médio por jogador.

Figura 6.8: Tempo médio de resposta para enviar mensagens comparado ao número de jogadores



Fonte: O próprio autor.

Visivelmente, a Figura 6.8 permite deduzir dessa forma que:

$$\overline{\text{EnviarMensagem}_w} < \overline{\text{EnviarMensagem}_s} < \overline{\text{EnviarMensagem}_r}$$

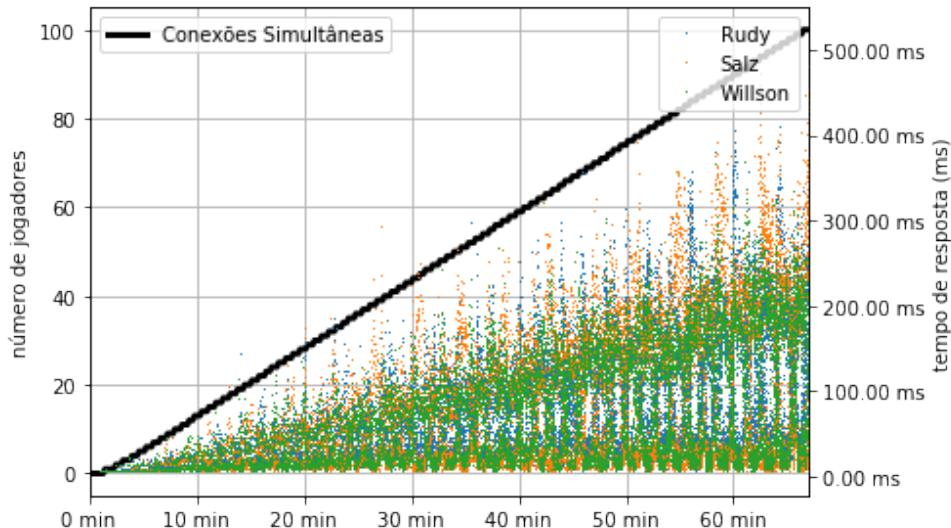
No qual entende-se:

- $\overline{\text{EnviarMensagem}_r}$: Tempo de resposta médio da operação enviar mensagem na arquitetura Rudy;
- $\overline{\text{EnviarMensagem}_s}$: Tempo de resposta médio da operação enviar mensagem na arquitetura Salz; e
- $\overline{\text{EnviarMensagem}_w}$: Tempo de resposta médio da operação enviar mensagem na arquitetura Willson.

Receber mensagem

A operação de receber mensagem é efetuada sobre o protocolo TCP/RPC. Nesta operação, o cliente envia um formulário em formato GOB com informações de autenticação assinadas pelo serviço. Os dados são validados e o serviço retorna todas as linhas de texto que foram encaminhadas ao personagem instanciado.

Figura 6.9: Tempo de resposta para receber mensagens



Fonte: O próprio autor.

A Figura 6.9 exibe os dados obtidos de tempo de resposta ao receber uma mensagem no chat durante a variação do número de jogadores simultâneos. Percebe-se um gráfico denso, dada a quantidade de pontos obtidos nos testes. Dessa forma, faz sentido analisar o comportamento médio por jogador simultâneo. A Figura 6.10 exibe o comportamento médio por jogador.

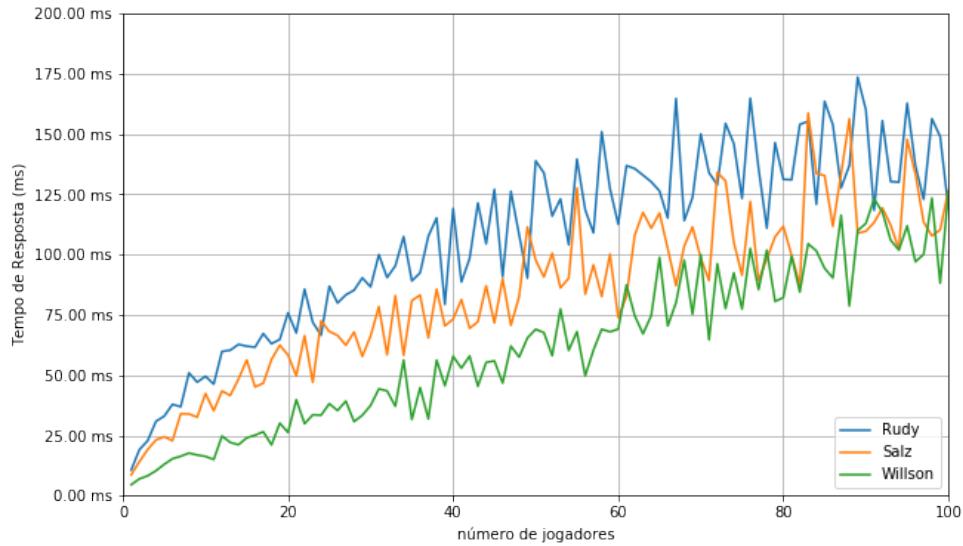
Visivelmente, a Figura 6.10 permite deduzir dessa forma que:

$$\overline{ReceberMensagem_w} < \overline{ReceberMensagem_s} < \overline{ReceberMensagem_r}$$

Onde entende-se:

- $\overline{ReceberMensagem_r}$: Tempo de resposta médio da operação receber mensagem na arquitetura Rudy;
- $\overline{ReceberMensagem_s}$: Tempo de resposta médio da operação receber mensagem na

Figura 6.10: Tempo médio de resposta para receber mensagens comparado ao número de jogadores



Fonte: O próprio autor.

arquitetura Salz; e

- $\overline{\text{ReceberMensagem}_w}$: Tempo de resposta médio da operação receber mensagem na arquitetura Willson.

6.1.2 Consumo de CPU

Este experimento visa analisar o consumo de CPU unitariamente, em relação ao número de jogadores simultâneos. Espera-se que o seu crescimento seja de tendência linear junto ao crescimento de jogadores concorrentes. Neste contexto existem os seguintes valores:

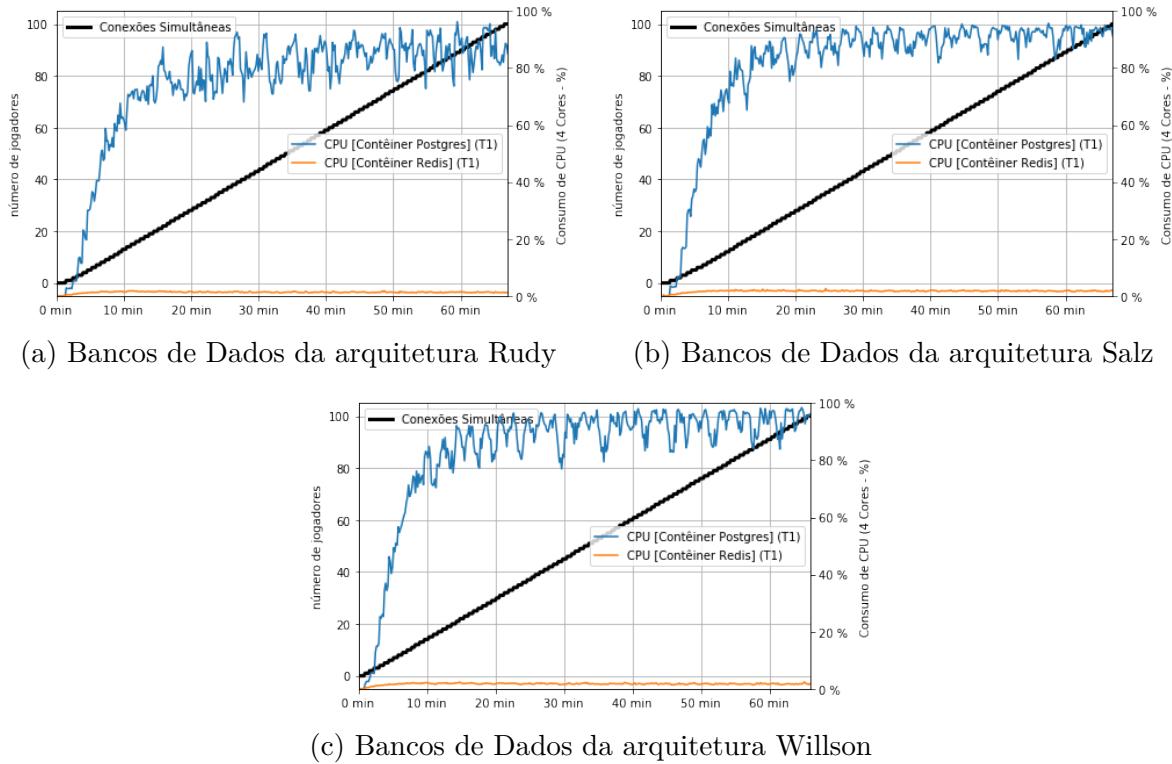
- Jogadores simultâneos: Variável capturada a partir do microsserviço de jogo; e
- Consumo de CPU: Variável capturada a partir do monitor de recursos do Docker.

Nota-se que o número de jogadores simultâneos e o consumo de CPU são indevidos pelo tempo a qual tais dados foram capturados. Nesse sentido, pode-se relacionar o número de jogadores simultâneos ao consumo de CPU dos microsserviços e do banco de dados. Dado tal contexto, faz sentido realizar uma análise separando os ambientes de Banco de Dados de Microsserviços.

Banco de Dados

Considerando o ambiente de banco de dados, pode-se realizar a associação de número de jogadores simultâneos e CPU consumida pelos contêineres de banco de dados. Essa associação é realizada pelo tempo de registro das métricas. O resultado desta associação pode ser visualizado na Figura 6.11.

Figura 6.11: Consumo de CPU dos bancos de dados



Fonte: O próprio autor.

A Figura 6.11 exibe o comportamento do consumo de CPU durante a execução dos testes para as arquiteturas Rudy (Subfigura 6.11a), Salz (Subfigura 6.11b) e Willson (Subfigura 6.11c). Em todas as figuras é notório o seguinte comportamento:

- Baixo consumo de CPU do servidor de dados temporários (Redis), com comportamento linear estável; e
- Alto consumo de CPU do servidor de dados permanentes (PostgreSQL), com comportamento multimodal.

Este comportamento era esperado, dado a natureza distinta de ambos os banco de dados, na qual PostgreSQL é um banco de dados Atomicidade, Consistência, Isolamento e Durabilidade (ACID) e o Redis é um banco de dados NoSQL. Entretanto, é

notório a discrepância entre ambos os bancos de dados, afirmando-se que o banco de dados temporário é estável em relação ao consumo de CPU.

Ao analisar o comportamento do banco de dados persistentes para as arquiteturas Rudy (Subfigura 6.11a), Salz (Subfigura 6.11b) e Willson (Subfigura 6.11c) observa-se um comportamento diferente na arquitetura Rudy. Dada esta percepção, faz sentido analisar a tendência destas curvas com base em sua média. A Tabela 6.6 exibe os valores das médias baseado em todo o decorrer do experimento e dividindo o experimento em quadrantes.

Tabela 6.6: Consumo de CPU por quadrante pelo PostgreSQL.

Quadrante	Rudy	Salz	Willson
Primeiro	50,60%	53,10%	58,15%
Segundo	80,66%	88,04%	89,86%
Terceiro	85,61%	90,77%	92,50%
Quarto	86,94%	92,07%	93,48%

Fonte: O próprio autor.

Torna-se visível a diferença do consumo de CPU entre as arquiteturas pelo banco de dados. A partir destes dados pode-se deduzir que:

- Arquitetura Rudy consome menos CPU do banco de dados persistentes; e
- Arquitetura Salz e Willson consomem CPU na mesma proporção, com a arquitetura Salz tendo uma leve tendência a consumir menos CPU.

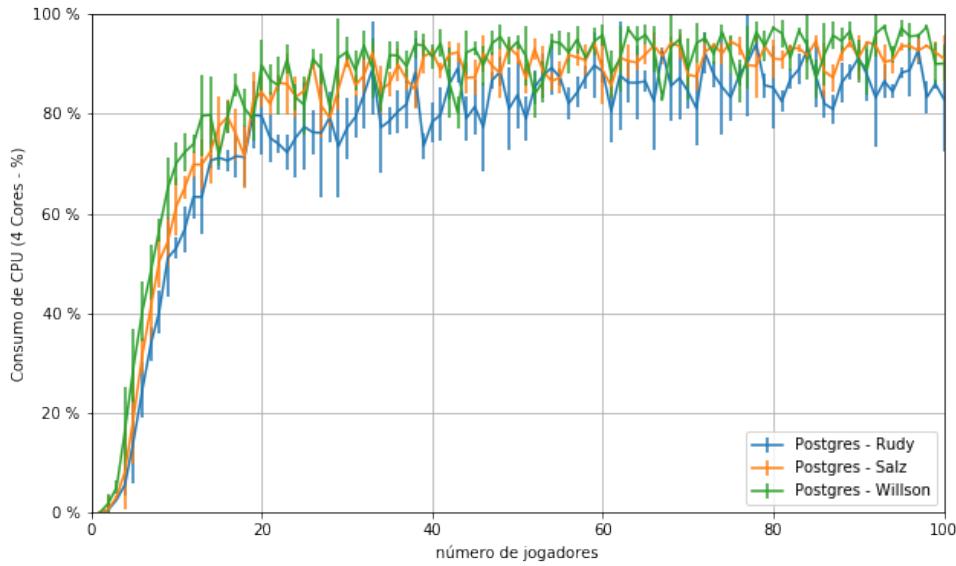
Para tornar esta diferença visível ao longo de toda a progressão da carga, faz sentido exibir esta informação em um gráfico de linha. A Figura 6.12 exibe a média de consumo de CPU pelo contêiner PostgreSQL comparado ao número de jogadores simultâneos.

Conforme exibido na Figura 6.12, pode-se deduzir que em todo caso as arquiteturas Rudy, Salz e Willson seguem a seguinte expressão:

$$CPU(db_postgresql_r) < CPU(db_postgresql_s) \approx CPU(db_postgresql_w)$$

Na qual entende-se:

Figura 6.12: Consumo de CPU pelo PostgreSQL comparado ao número de jogadores simultâneos.



Fonte: O próprio autor.

- $CPU(db_postgresql_r)$: Curva de consumo de CPU pelo PostgreSQL da arquitetura Rudy;
- $CPU(db_postgresql_s)$: Curva de consumo de CPU pelo PostgreSQL da arquitetura Salz; e
- $CPU(db_postgresql_w)$: Curva de consumo de CPU pelo PostgreSQL da arquitetura Willson.

O banco de dados temporários consome pouca CPU, comparado ao banco de dados persistentes. Entretanto, faz sentido realizar a mesma análise para validar se existe alguma diferença no seu uso conforme a arquitetura empregada. Dessa forma, foi dividido em quadrantes para analisar o consumo médio de CPU, a qual é exibido na Tabela 6.7.

Tabela 6.7: Consumo de CPU por quadrante pelo Redis.

Quadrante	Rudy	Salz	Willson
Primeiro	1,24%	1,46%	1,60%
Segundo	1,32%	1,79%	1,81%
Terceiro	1,28%	1,75%	1,75%
Quarto	1,26%	1,73%	1,72%

Fonte: O próprio autor.

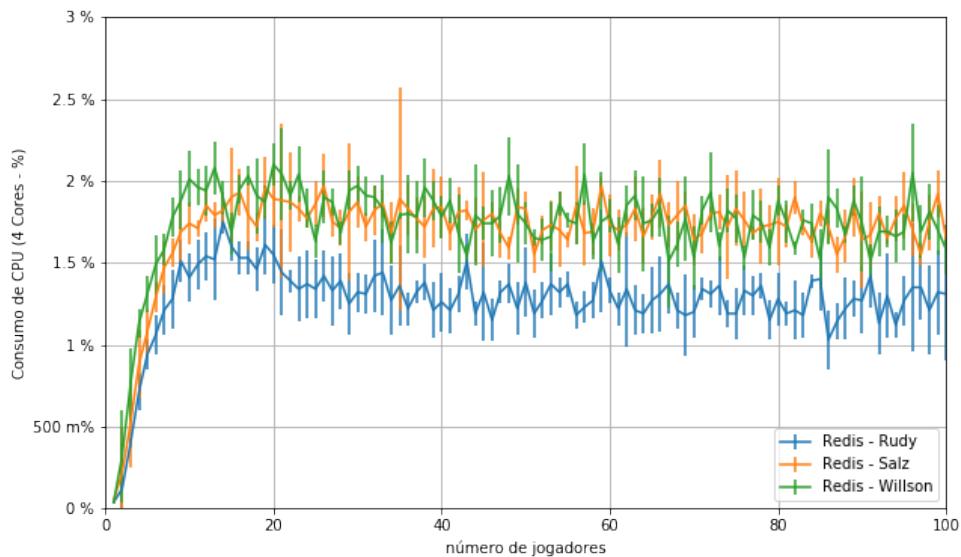
Dado os valores da Tabela 6.7, pode-se obter as seguintes conclusões:

- Rudy consome menos CPU do banco de dados temporários; e

- A arquitetura Salz e Willson consomem CPU com o mesmo comportamento.

Para tornar esta diferença visível ao longo de toda a progressão da carga, faz sentido exibir esta informação em um gráfico de linha. A Figura 6.13 exibe um gráfico a qual compara a média de consumo de CPU conforme o número de jogadores simultâneos.

Figura 6.13: Consumo de CPU pelo PostgreSQL comparado ao número de jogadores simultâneos.



Fonte: O próprio autor.

Conforme exibido na Figura 6.13, pode-se deduzir que em todo caso as arquiteturas Rudy, Salz e Willson seguem a seguinte expressão:

$$CPU(db_redis_r) < CPU(db_redis_s) \approx CPU(db_redis_w)$$

Na qual entende-se:

- $CPU(db_redis_r)$: Curva de consumo de CPU pelo Redis da arquitetura Rudy;
- $CPU(db_redis_s)$: Curva de consumo de CPU pelo Redis da arquitetura Salz; e
- $CPU(db_redis_w)$: Curva de consumo de CPU pelo Redis da arquitetura Willson.

Nota-se que o consumo de recurso do banco de dados temporários e banco de dados persistentes exibiram o mesmo comportamento. Dessa forma, pode-se generalizar

o comportamento do consumo de CPU das arquiteturas Rudy, Salz e Willson para banco de Dados:

$$CPU(db_r) < CPU(db_s) \approx CPU(db_w)$$

Na qual entende-se:

- $CPU(db_r)$: Curva de consumo de CPU pelos banco de dados da arquitetura Rudy;
- $CPU(db_s)$: Curva de consumo de CPU pelos banco de dados da arquitetura Salz; e
- $CPU(db_w)$: Curva de consumo de CPU pelos banco de dados da arquitetura Willson.

Este comportamento é dado pela característica do microsserviço *rcrud* que realiza uma multiplexação de conexões entre o banco de dados e a arquitetura. Entretanto, esta característica pode ser por dois motivos:

- O banco de dados está otimizado para responder uma única conexão, consumindo menos CPU; ou
- O banco de dados não consome tanta CPU por ter um microsserviço que tornou-se um gargalo como frente do banco de dados.

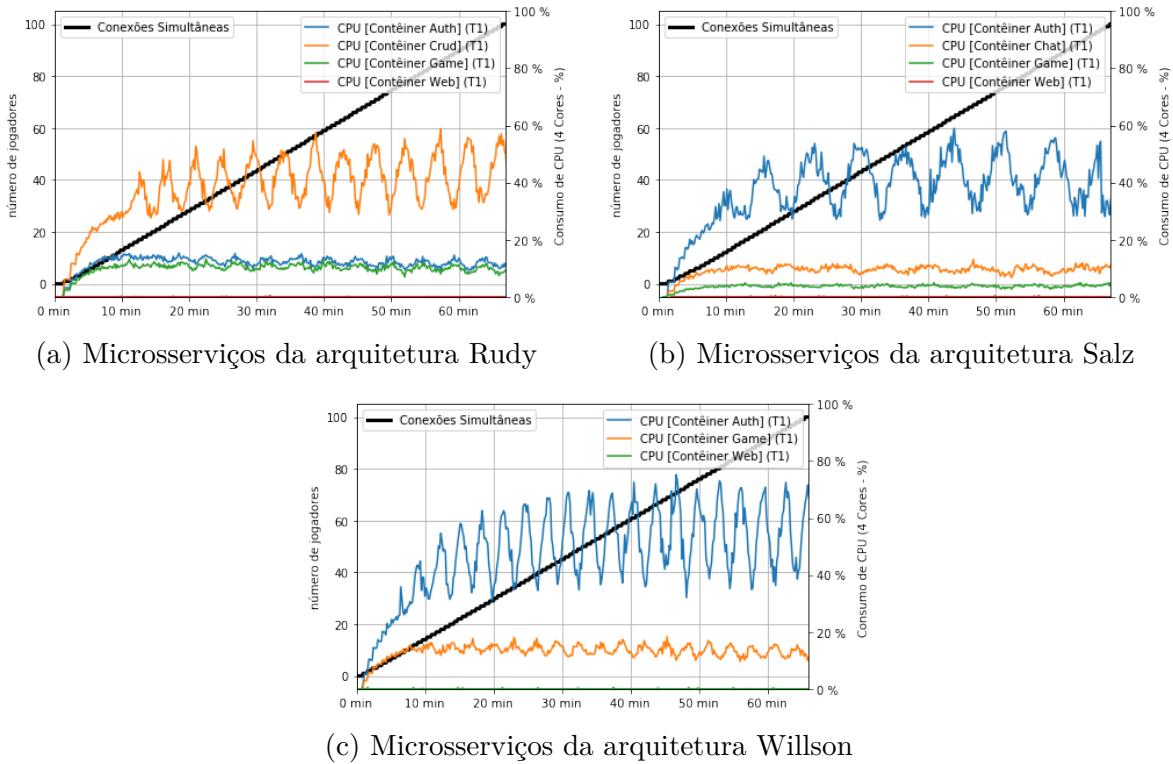
Dado estas duas hipóteses, faz sentido mensurar a qualidade de serviço entre-gue ao usuário. Pode-se mensurar qual das hipóteses é mais valiosa para a qualidade de serviço com base no tempo de resposta do usuário.

Microsserviços

Considerando o ambiente de banco de dados, pode-se realizar a associação de número de jogadores simultâneos e CPU consumida pelos contêineres de banco de dados. Essa associação é realizada pelo tempo de registro das métricas. O resultado desta associação pode ser visualizado na Figura 6.14.

Conforme exibido nas Figuras 6.14a, 6.14b e 6.14c, pode-se observar as seguintes características:

Figura 6.14: Consumo de CPU dos microsserviços



Fonte: O próprio autor.

- Os microsserviços *rweb*, *sweb* e *wweb* consomem pouca CPU comparado a todos os outros microsserviços das arquiteturas; e
- Os microsserviços *rcrud*, *sauth* e *wauth* são os microsserviços que mais consomem CPU nas arquiteturas Rudy, Salz e Willson.

Faz sentido analisar a média do consumo de CPU dos microsserviços dividindo em quadrantes. Dessa forma, pode-se obter dados significantes para realizar comparações a partir destes dados. A Tabela 6.8 exibe a média de consumo de CPU por quadrante de todos os microsserviços.

A Tabela 6.8 mostra o crescimento do consumo de carga médio dos quadrantes. Estes dados são importantes, haja vista que os mesmos permitem analisar o comportamento das curvas removendo oscilações notórias na visualização do consumo de CPU. A partir desta tabela pode-se realizar as seguintes conclusões:

- Entre os microsserviços *rcrud*, *sauth* e *wauth*, o com maior consumo é o microsserviço *wauth*; e
- Os microsserviços *rweb*, *sweb* e *wweb* possuem o mesmo consumo, em média.

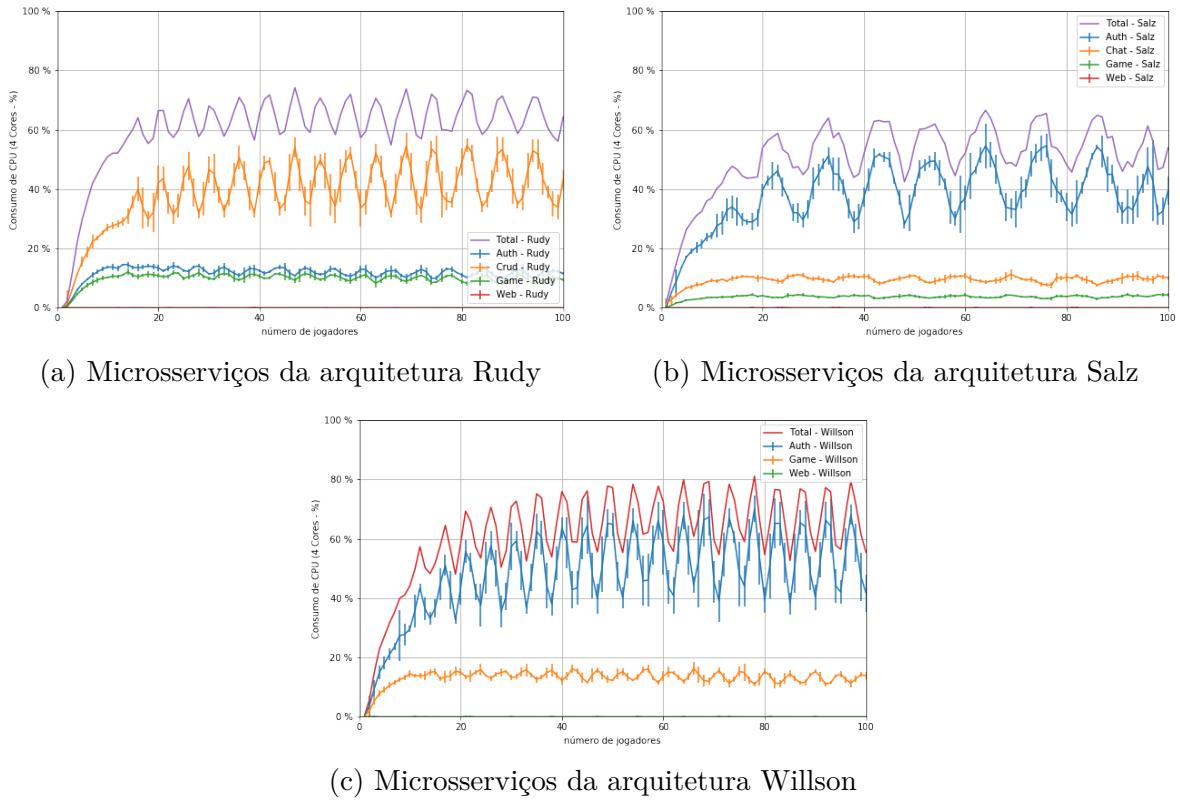
Tabela 6.8: Consumo de CPU por quadrante dos microsserviços.

Microsserviço	Primeiro Quadrante	Segundo Quadrante	Terceiro Quadrante	Quarto Quadrante
rauth	11,22%	12,52%	11,88%	11,56%
rcrud	25,97%	41,83%	42,50%	43,62%
rgame	8,92%	10,54%	10,14%	9,58%
rweb	0,01%	0,01%	0,01%	0,01%
sauth	26,08%	39,95%	43,22%	40,96%
schat	8,34%	9,97%	9,59%	9,61%
sgame	3,12%	3,87%	3,72%	3,85%
sweb	0,01%	0,01%	0,01%	0,01%
wauth	23,79%	50,73%	57,11%	56,69%
wgame	9,68%	14,20%	13,66%	13,42%
wweb	0,01%	0,01%	0,01%	0,01%

Fonte: O próprio autor.

É relevante, a partir destas conclusões, analisar a média em relação a progressão de usuários. A visualização da média de consumo de CPU por usuário simultâneo tem como objetivo remover ruídos que possam afetar a visualização dos dados.

Figura 6.15: Média do consumo de CPU dos microsserviços por jogador simultâneo



Fonte: O próprio autor.

A Figura 6.15 tem como objetivo exibir a média de consumo de CPU por jogador, assim permitindo analisar o comportamento das curvas. Entretanto, nota-se que todos os microsserviços contém uma característica senoidal, onde sua amplitude aumenta com a quantidade de jogadores, na qual já era visível na Figura 6.14. Porém, neste gráfico

também é visível o erro de cada média. Este erro é dado pela variância do conjunto de dados obtidos na determinada faixa de jogadores simultâneos. Ter um valor de variância alto significa que, por mais que a média seja um bom valor para a métrica, diversos pontos da mesma faixa estão longe deste valor. Dessa forma, pode-se interpretar como um erro, neste contexto.

Pode-se deduzir as seguintes características a partir da Figura 6.15:

- Os microsserviços que contém maior erro são os que estão relacionados a acesso de dados para autenticação ou/e geração de assinaturas de dados para outros serviços (*rcrud, sauth e wauth*);
- Os microsserviços que contém maior erro, são os que lideram a lista de maior consumo de CPU, porém nota-se a partir dos demais microsserviços que esta relação não é proporcional a carga; e
- O valor total de consumo de CPU mudou conforme a arquitetura.

Dado o último critério, utilizando a curva de total consumido das Sub figuras 6.15a, 6.15b e 6.15c, pode-se comparar o total de consumo de CPU por cada arquitetura. A Tabela 6.9 exibe os valores baseados em quadrantes, para realizar a comparação com valores.

Tabela 6.9: Consumo total de CPU por quadrante dos microsserviços.

Arquitetura	Primeiro Quadrante	Segundo Quadrante	Terceiro Quadrante	Quarto Quadrante
Rudy	47,05%	64,51%	65,25%	64,31%
Salz	38,74%	53,87%	56,74%	54,27%
Willson	44,60%	65,76%	67,94%	66,98%

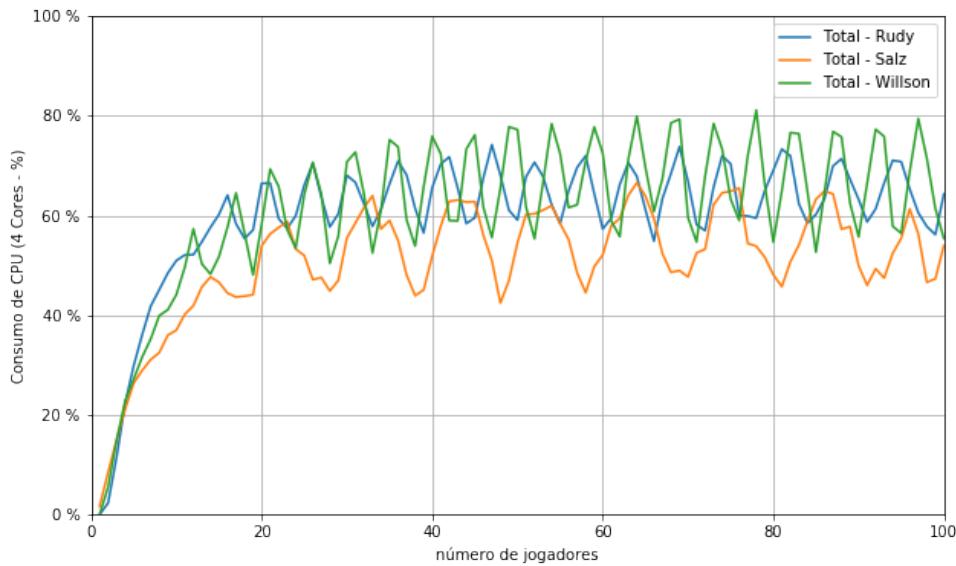
Fonte: O próprio autor.

A partir dos dados da Tabela 6.9, pode-se referenciar que a arquitetura Salz consome menos CPU que a arquitetura Rudy e Willson. Faz sentido realizar a visualização das três curvas de totais em um único gráfico a fim de exibir esta relação. A Figura 6.16 exibe esta comparação.

Dado a Figura 6.16 e a Tabela 6.9, pode-se afirmar que:

$$CPU(microsservicos_s) < CPU(microsservicos_r) < CPU(microsservicos_w)$$

Figura 6.16: Comparaçāo de consumo total de CPU pelas arquiteturas.



Fonte: O próprio autor.

Na qual entende-se:

- $CPU(microservices_r)$: Curva de consumo de CPU pelos microsserviços da arquitetura Rudy;
- $CPU(microservices_s)$: Curva de consumo de CPU pelos microsserviços da arquitetura Salz; e
- $CPU(microservices_w)$: Curva de consumo de CPU pelos microsserviços da arquitetura Willson.

Torna-se notório que tal sequência respeita a ordem de interconexões dos microsserviços. Nesse sentido, a arquitetura Salz possui mais conexões que a arquitetura Rudy, que possui mais conexões que a arquitetura Willson.

Nota-se, que possuir um consumo de CPU maior que outra arquitetura não implica em ser uma arquitetura melhor. Pode-se encontrar dois casos para um consumo de CPU ao comparar duas arquiteturas:

- A arquitetura possui pouca carga para estressar a CPU; ou
- A arquitetura converge para tal valor, visto que está com gargalo em outro recurso necessário.

Para o caso desta análise, pode-se descartar a primeira alternativa. Para este caso pode-se analisar que a curva estabiliza o seu crescimento entre 20 e 40 jogadores simultâneos, a mesma faixa na qual os bancos de dados estabilizaram seu crescimento (visível na Figura 6.12). Dessa forma, pode-se justificar o aumento de erros nas médias nos microsserviços com acesso ao banco conforme o aumento da carga no serviço.

Outra característica das curvas obtidas é a frequência da oscilação encontrada, característica visível na Figura 6.16. Nota-se que a frequência é escalada tal qual a crista de uma onda entra em contato com o vale da outra onda. Tal característica das curvas não é coincidência, sendo resultado do escalonador de contêineres. A Figura 6.15c é a que melhor exemplifica este comportamento, haja vista que a arquitetura Willson possui somente dois microsserviços que são impactados diretamente por este escalonador.

6.1.3 Consumo de Memória

Este experimento visa analisar o consumo de memória unitariamente, em relação ao número de jogadores simultâneos. Espera-se que o seu crescimento seja linear seguindo o crescimento de jogadores concorrentes. Neste contexto existem os seguintes valores:

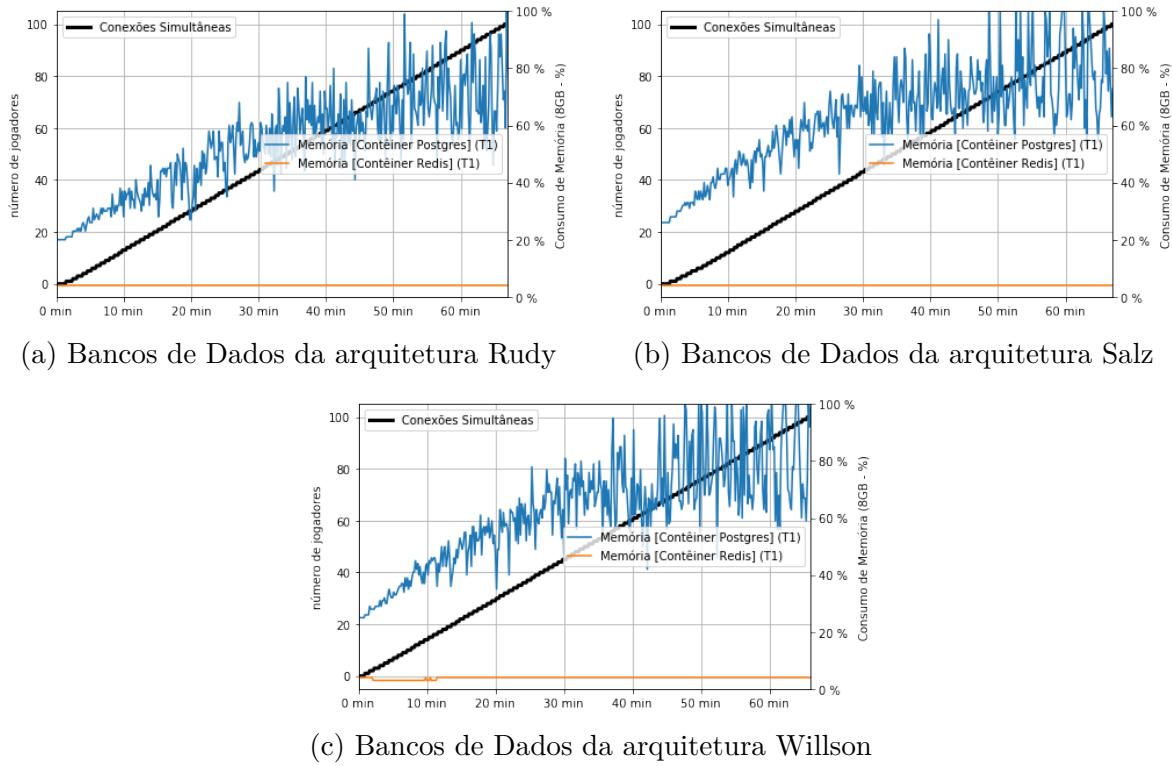
- Jogadores simultâneos: Variável capturada a partir do microsserviço de jogo; e
- Consumo de memória: Variável capturada a partir do monitor de recursos do Docker.

Nota-se que o número de jogadores simultâneos e o consumo de memória são indexados pelo tempo a qual tais dados foram capturados. Nesse sentido, pode-se relacionar, o número de jogadores simultâneos ao consumo de memória dos microsserviços e do banco de dados. Dado tal contexto, faz sentido realizar uma análise separando os ambientes de Banco de Dados de Microsserviços.

Banco de Dados

Considerando o ambiente de banco de dados, pode-se realizar a associação de número de jogadores simultâneos e memória consumida pelos contêineres de banco de dados. Essa associação é realizada pelo tempo de registro das métricas. O resultado desta associação pode ser visualizado na Figura 6.17.

Figura 6.17: Consumo de memória dos bancos de dados



Fonte: O próprio autor.

Percebe-se, a partir da Figura 6.17, que o banco com maior estresse é o PostgreSQL. O banco de dados Redis consome uma quantidade de memória visivelmente fixa. Entretanto, o banco de dados Redis é utilizado para armazenamento de dados da sessão do cliente em memória. Assim, faz sentido calcular a quantidade de memória consumida por cada usuário para garantir que na totalidade este valor não é significativo em porcentagem.

Para realizar a autenticação é utilizado JWT. Contudo, é necessário garantir uma única conexão por usuário. A unicidade é realizada utilizando o Redis, armazenando uma sequência de caracteres aleatórios em um campo formatado com o nome do usuário. O trecho de código que realiza esta operação é exibido na Figura 6.1.

Listing 6.1: Informações do Bloco

```

1 package merepositories
2
3 // mmosandbox/infra/merepositories/token_repository.go
4
5 // GenerateToken based in username from account
6 func (repository *TokenRepository) GenerateToken(username string) string {
7     token := repository.randomToken()
8     repository.set(repository.keyPattern(username), token)

```

```

9
10    return token
11 }
12
13 // keyPattern to store into redis
14 func (repository *TokenRepository) keyPattern(username string) string {
15     return fmt.Sprintf("session.%s", username)
16 }
17
18 // randomToken with 256 runes
19 func (repository *TokenRepository) randomToken() string {
20     return randomize.RandStringRunes(256)
21 }
```

O repositório de dados *TokenRepository*¹ contém as operações exibidas na Listagem 6.1. Este repositório gera a chave com o padrão *session.%s* e uma sequência de caracteres aleatória de 256 bytes.

Para realizar este cálculo é necessário estimar o custo de armazenamento das chaves. Por padrão, os clientes de ataque geram nomes de usuários de 32 caracteres. Com tais dados, pode-se estimar o custo de memória por usuário. Pode-se calcular o consumo de memória por um único jogador simultâneo utilizando a seguinte fórmula:

$$m_j = m_{chave} + m_{padrao} + m_{valor}$$

$$\Rightarrow m_j = 32\text{bytes} + 8\text{bytes} + 256\text{bytes} = 296\text{bytes} \quad (6.1)$$

Na qual entende-se:

- m_j : Tamanho dos dados armazenados em memória no Redis por jogador autenticado;
- m_{chave} : Tamanho da chave armazenada na estrutura chave-valor Redis; e
- m_{valor} : Tamanho do valor armazenado na estrutura chave-valor do Redis.

Logo, ao ter 100 jogadores simultâneos, o serviço Redis consumirá aproximadamente 29Kb para armazenamento de memória. Tal valor é insignificante comparado

¹ *TokenRepository*: https://github.com/schweigert/mmosandbox/blob/master/infra/merepositories/token_repository.go

ao total de memória disponível no hospedeiro, dessa forma exibindo um comportamento retilíneo na Figura 6.17.

Entretanto o PostgreSQL exibe um crescimento linear visível na Figura 6.17. Faz sentido sanitizar estas curvas com a média dos quadrantes. A Tabela 6.10 exibe o consumo de memória médio por quadrante.

Tabela 6.10: Consumo médio de memória por quadrante do PostgreSQL.

Arquitetura	Primeiro Quadrante	Segundo Quadrante	Terceiro Quadrante	Quarto Quadrante
Rudy	32,41%	54,07%	65,92%	73,57%
Salz	39,56%	61,52%	71,84%	76,29%
Willson	38,83%	61,72%	72,43%	79,25%

Fonte: O próprio autor.

A partir da Tabela 6.10, pode-se mensurar a distância de consumo de memória por parte do PostgreSQL. No quarto quadrante também pode-se observar uma distância, em média, dentre as curvas de consumo. Entretanto não pode-se mensurar que é uma tendência, visto que o segundo e terceiro quadrante ficam próximo. Dado tais informações pode-se parcialmente concluir que o consumo de memória por parte do banco de dados persistentes é menor na arquitetura Rudy. Contudo, o consumo é demasiadamente próximo entre as arquiteturas Willson e Salz, tendo a arquitetura Salz uma leve tendência a ser maior.

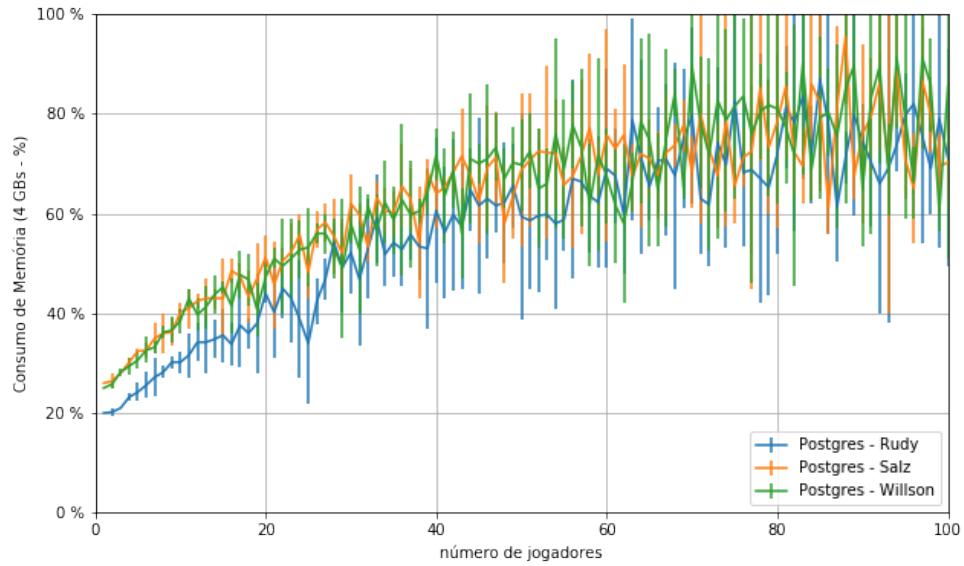
Tal comportamento faz sentido, levando em consideração o número de conexões simultâneas que cada arquitetura realiza ao banco de dados. A arquitetura Rudy contém poucas conexões simultâneas ao banco de dados persistentes haja vista que concentra todas suas conexões ao microsserviço *rcrud*, o qual ocorre a partir de todos os microsserviços das demais arquiteturas.

Para confirmar que esta conclusão é correta, faz sentido correlacionar a média de consumo de memória pelo banco de dados persistentes correlacionado ao número de jogadores simultâneos. Tal correlação é visível na Figura 6.18.

A partir da Figura 6.18 pode-se confirmar as conclusões parciais obtidas da Tabela 6.6. Além das conclusões parciais, é possível visualizar o crescimento da média a cada incremento de jogador simultâneo. Pode-se concluir que o serviço de fato foi estressado, haja vista que pode-se perceber inicialmente um comportamento na qual não corresponde a uma curva multimodal.

A partir destes dados pode-se concluir que:

Figura 6.18: Consumo de memória média do PostgreSQL comparado ao número de jogadores simultâneos.



Fonte: O próprio autor.

$$MEM(db_postgresql_r) < MEM(db_postgresql_s) \approx MEM(db_postgresql_w)$$

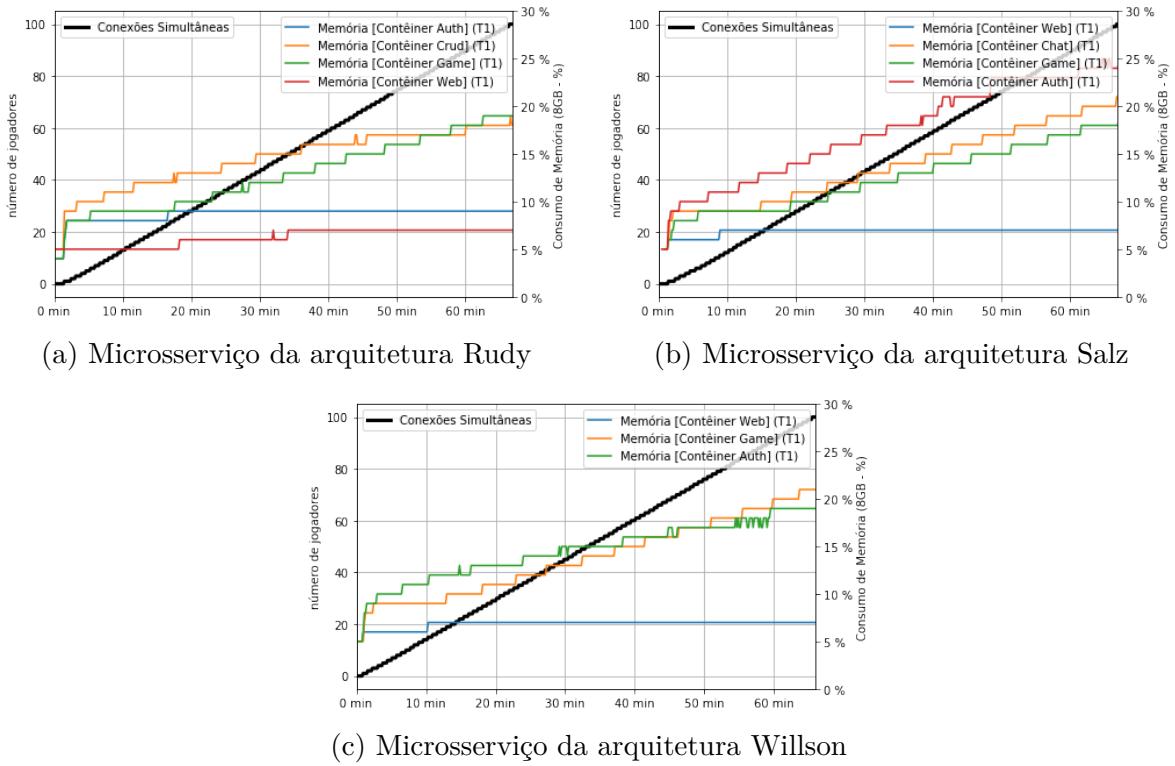
Na qual entende-se:

- $MEM(db_postgresql_r)$: Curva de consumo de memória do PostgreSQL pela arquitetura Rudy;
- $MEM(db_postgresql_s)$: Curva de consumo de memória do PostgreSQL pela arquitetura Salz; e
- $MEM(db_postgresql_w)$: Curva de consumo de memória do PostgreSQL pela arquitetura Willson.

Microsserviços

Considerando o ambiente de microsserviços, pode-se realizar a associação de número de jogadores simultâneos e memória consumida pelos contêineres de banco de dados. Essa associação é realizada pelo tempo de registro das métricas. O resultado desta associação pode ser visualizado na Figura 6.19.

Figura 6.19: Consumo de memória dos microsserviços



Fonte: O próprio autor.

Ao escalar jogadores simultâneos ao serviço, a memória consumida pelas arquiteturas de microsserviços cresce linearmente, com saltos. Tais saltos, visíveis na Figura 6.19, são provenientes do gerenciador de memória da linguagem de implementação. Este método de alocação de memória tenta evitar a comunicação com o sistema operacional para esta alocação de memória.

Faz sentido analisar o comportamento em quadrantes, já que o seu crescimento é linear. Dessa forma podemos analisar o comportamento de crescimento dos microsserviços e realizar uma análise comparativa sobre os dados obtidos. Pode-se visualizar estes dados na Tabela 6.11.

A partir da Tabela 6.11 podemos deduzir diversas características dos microsserviços e das arquiteturas. Dessa forma, deduz-se sobre os microsserviços que:

- Ao comparar os microsserviços de autenticação, percebe-se que o microsserviço *rauth* não aumenta o seu consumo de memória, repassando o seu consumo de memória ao microsserviço *rcrud*;
- O microsserviço *rweb* consome menos memória comparado aos microsserviços *sweb* e *wweb* pois não realiza conexão com o banco de dados; e

Tabela 6.11: Consumo médio de memória pelos microsserviços.

Microsserviços	Quadrante 1		Quadrante 2		Quadrante 3		Quadrante 4	
	Média	Total	Média	Total	Média	Total	Média	Total
rauth	8 %	33 %	9 %	41 %	9 %	49 %	9 %	53 %
rcrud	11 %		14 %		17 %		18 %	
rgame	9 %		12 %		16 %		19 %	
rweb	5 %		6 %		7 %		7 %	
sauth	9 %	34 %	12 %	41 %	16 %	51 %	19 %	60 %
schat	9 %		11 %		14 %		17 %	
sgame	9 %		11 %		14 %		17 %	
sweb	7 %		7 %		7 %		7 %	
wauth	10 %	26 %	12 %	31 %	16 %	39 %	20 %	47 %
wgame	9 %		12 %		16 %		20 %	
wweb	7 %		7 %		7 %		7 %	

Fonte: O próprio autor.

- O microsserviço *schat* e *sgame* consomem menos memória ao comparar com os microsserviços *wgame* e *rgame*. Entretanto a soma de ambos consome muito mais.

A partir do consumo total, exibido na Tabela 6.11, pode-se concluir que:

$$\overline{Memoria_{gs_w}} < \overline{Memoria_{gs_r}} < \overline{Memoria_{gs_s}}$$

Na qual entende-se:

- $\overline{Memoria_{gs_r}}$: Média de consumo de memória dos microsserviços pela arquitetura Rudy;
- $\overline{Memoria_{gs_s}}$: Média de consumo de memória dos microsserviços pela arquitetura Salz; e
- $\overline{Memoria_{gs_w}}$: Média de consumo de memória dos microsserviços pela arquitetura Willson.

A arquitetura Willson obteve melhor desempenho para consumo de memória, visto que tal arquitetura possui um componente a menos para desempenhar o mesmo papel comparado as outras arquiteturas. No geral, seus microsserviços consumiram mais memória unitariamente, talvez sendo um problema para escalar esta arquitetura em múltiplas máquinas com recursos escassos.

A arquitetura Rudy obteve o resultado mediano, se comparado as arquiteturas selecionadas. O seu consumo foi maior que a arquitetura Willson por contemplar um microsserviço a mais. Entretanto, a arquitetura economizou memória ao concentrar todas as conexões no microsserviço *rcrud*, porém centralizou todo o processamento de CPU para isto no mesmo local.

A arquitetura Salz distribui as consultas ao banco de dados de forma que os serviços que precisam destes dados realizam consultas diretamente ao banco de dados. Esta estratégia maximiza o número de conexões ao banco e, consequentemente, aumenta o consumo de memória. Além da estratégia, a arquitetura contém 4 microsserviços, o que também colabora com o aumento do consumo de memória.

6.1.4 Entrada de Rede

Este experimento visa analisar o consumo da entrada de rede unitariamente, em relação ao número de jogadores simultâneos. Espera-se que o seu crescimento seja de tendência linear junto ao crescimento de jogadores concorrentes. Neste contexto existem os seguintes valores:

- Jogadores simultâneos: Variável capturada a partir do microsserviço de jogo;
- Entrada do serviço: Variável capturada a partir do monitor de recursos do Docker; e
- Entrada do ambiente: Variável capturada a partir do sistema operacional.

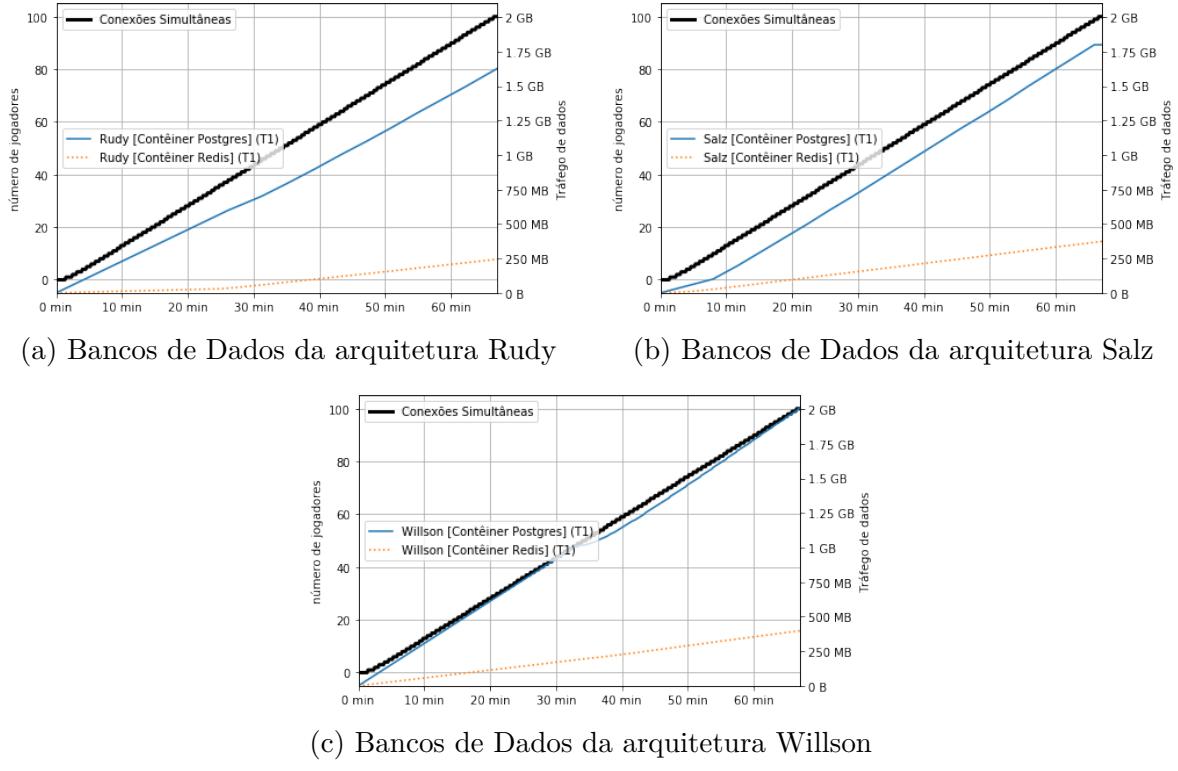
Nota-se que o número de jogadores simultâneos e o consumo entrada de rede são indexados pelo tempo a qual tais dados foram capturados. Nesse sentido, pode-se relacionar o número de jogadores simultâneos ao consumo de entrada de rede dos microsserviços e do banco de dados. Dado tal contexto, faz sentido realizar uma análise separando os ambientes de Banco de Dados de Microsserviços.

Banco de Dados

Considerando o ambiente de banco de dados, pode-se realizar a associação de número de jogadores simultâneos e entrada de rede consumida pelos contêineres de banco de

dados. Essa associação é realizada pelo tempo de registro das métricas. O resultado desta associação pode ser visualizado na Figura 6.20.

Figura 6.20: Entrada de dados da rede dos bancos de dados



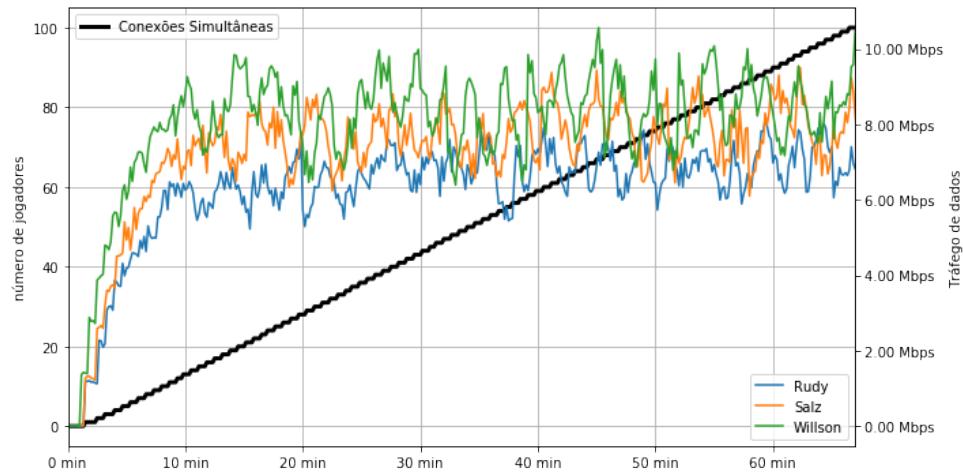
Fonte: O próprio autor.

A Figura 6.20 exibe o comportamento da entrada dos dados para os bancos de dados. Ele segue um padrão visivelmente linear. Ambas as arquiteturas consomem o banco de dados com o mesmo comportamento linear, entretanto em proporções diferentes. Faz sentido analisar a vazão da rede do banco de dados. Esta visualização está disponível na Figura 6.21.

A Figura 6.21 exibe claramente que existe uma vazão máxima para entrada de dados aos bancos de dados. Nota-se que existe diferença entre as arquiteturas, a qual já era visível na Figura 6.20.

Como existem pontos de contato entre as curvas, faz sentido realizar uma análise do comportamento médio. A partir destes dados pode-se comparar as curvas e validar qual banco de dados recebeu mais requisições. A Tabela 6.14 exibe a média por quadrante das curvas comparado a vazão da rede por segundo.

A partir da Tabela 6.14 percebe-se que a arquitetura Willson mesmo com características próximas a arquitetura Rudy, performou com melhor qualidade, recebendo

Figura 6.21: Vazão da entrada de dados para a rede *databases*.

Fonte: O próprio autor.

Tabela 6.12: Consumo médio da entrada da rede por quadrante do PostgreSQL.

Arquitetura	Primeiro Quadrante	Segundo Quadrante	Terceiro Quadrante	Quarto Quadrante
Rudy	4,94 Mbps	6,67 Mbps	6,91 Mbps	7,03 Mbps
Salz	6,06 Mbps	7,61 Mbps	7,88 Mbps	7,88 Mbps
Willson	6,67 Mbps	8,34 Mbps	8,41 Mbps	8,55 Mbps

Fonte: O próprio autor.

mais dados de requisição. O microsserviço *rcrud* a qual poderia tornar-se um gargalo para a entrada de rede tem esta característica confirmada.

Ao comparar as arquiteturas Salz e Willson, perce-se que, por mais que todos os seus microsserviços realizam conexão direta ao banco, existe uma diferença notória entre ambas. Esta diferença é pela quantidade de microsserviços na qual utilizam determinado banco de dados.

Nesse sentido, pode-se deduzir:

$$\overline{\text{Entrada}_{db_r}} < \overline{\text{Entrada}_{db_s}} < \overline{\text{Entrada}_{db_w}}$$

Deste modo, entende-se:

- $\overline{\text{Entrada}_{db_r}}$: Vazão de entrada médio do banco de dados na arquitetura Rudy;
- $\overline{\text{Entrada}_{db_r}}$: Vazão de entrada médio do banco de dados na arquitetura Salz; e
- $\overline{\text{Entrada}_{db_r}}$: Vazão de entrada médio do banco de dados na arquitetura Willson.

A Arquitetura Rudy contempla a menor vazão de rede entre as arquiteturas

selecionadas. A sua menor vazão, comparada as demais, é pelo afunilamento de requisições em um unico microsserviço. Dessa forma, o microsserviço *rcrud* é um gargalo aparente nesta arquitetura.

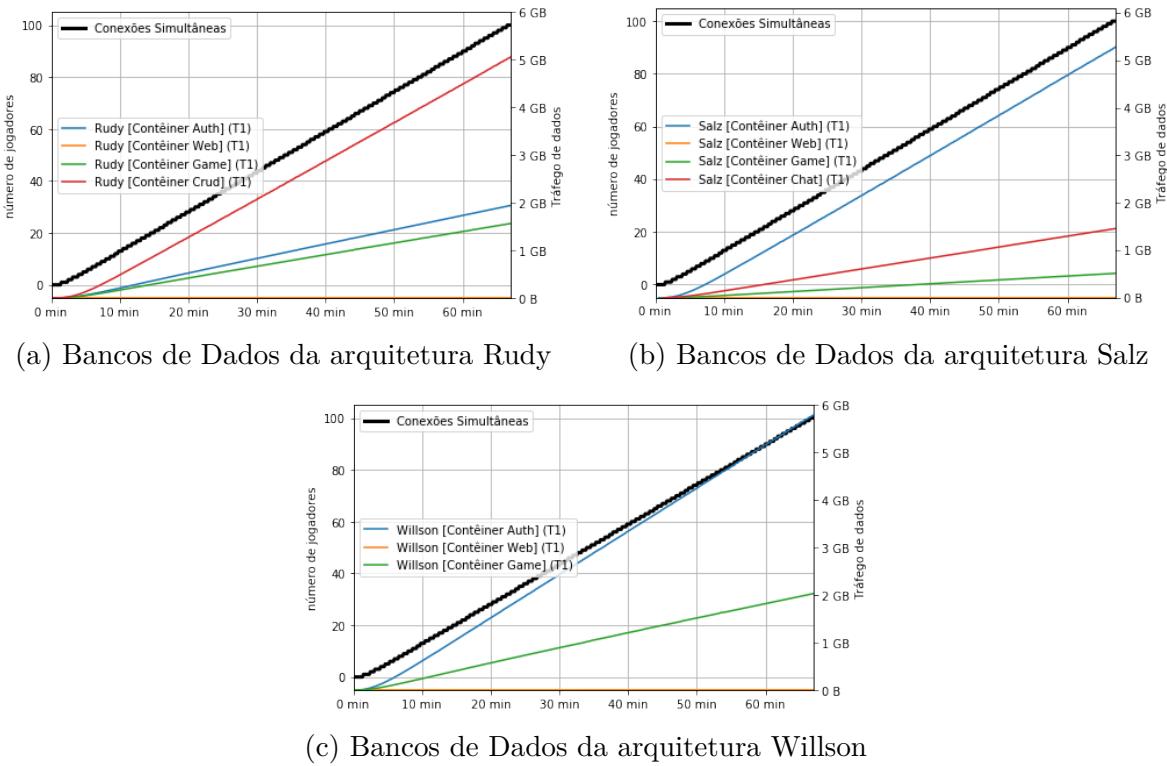
A arquitetura Salz possui uma vazão de entrada maior comparado a arquitetura Rudy, porém menor que a arquitetura Willson. Esta característica é dada pela saturação da CPU do ponto de vista da arquitetura Salz, a qual obstruiu a vazão de dados.

A arquitetura Willson possui a maior vazão de entrada de dados entre as arquiteturas selecionadas. Esta característica provém de um melhor aproveitamento da CPU pela distribuição do processamento na arquitetura Willson.

Microsserviços

Considerando o ambiente de microsserviços, pode-se realizar a associação de número de jogadores simultâneos e entrada de rede consumida pelos contêineres de microsserviços. Essa associação é realizada pelo tempo de registro das métricas. O resultado desta associação pode ser visualizado na Figura 6.22.

Figura 6.22: Entrada de dados da rede das arquiteturas



Fonte: O próprio autor.

A Figura 6.22 exibe o comportamento do consumo de entrada da rede, na qual visivelmente todos os serviços são lineares. Dessa forma, pode-se deduzir os seguintes comportamentos para as arquiteturas de microsserviços:

$$Entrada_{rweb} < Entrada_{rgame} < Entrada_{rauth} < Entrada_{rcrud}$$

$$Entrada_{sweb} < Entrada_{sgame} < Entrada_{schat} < Entrada_{sauth}$$

$$Entrada_{wweb} < Entrada_{wgame} < Entrada_{wauth}$$

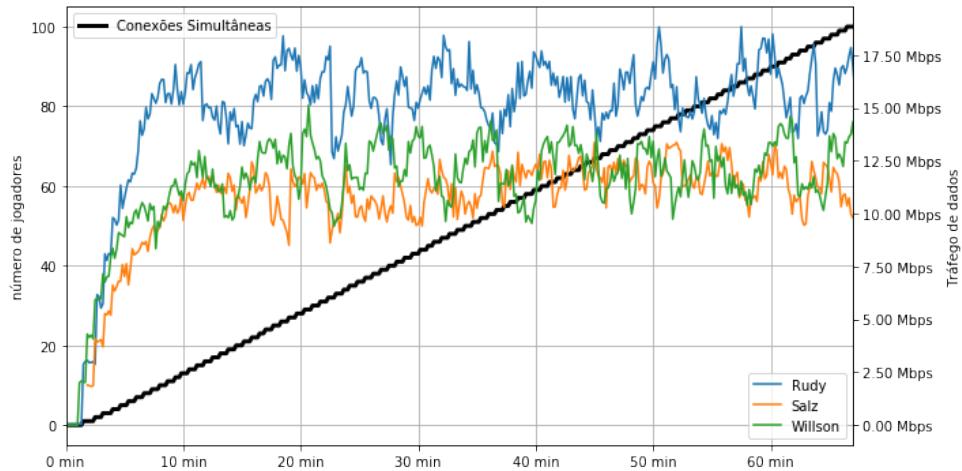
Na qual entende-se:

- $Entrada_{rweb}$: Curva de entrada do microsserviço web na arquitetura Rudy;
- $Entrada_{rgame}$: Curva de entrada do microsserviço game na arquitetura Rudy;
- $Entrada_{rauth}$: Curva de entrada do microsserviço auth na arquitetura Rudy;
- $Entrada_{rcrud}$: Curva de entrada do microsserviço crud na arquitetura Rudy;
- $Entrada_{sweb}$: Curva de entrada do microsserviço web na arquitetura Salz;
- $Entrada_{sgame}$: Curva de entrada do microsserviço game na arquitetura Salz;
- $Entrada_{sauth}$: Curva de entrada do microsserviço auth na arquitetura Salz;
- $Entrada_{schat}$: Curva de entrada do microsserviço chat na arquitetura Salz;
- $Entrada_{wweb}$: Curva de entrada do microsserviço web na arquitetura Willson;
- $Entrada_{wgame}$: Curva de entrada do microsserviço game na arquitetura Willson; e
- $Entrada_{wauth}$: Curva de entrada do microsserviço auth na arquitetura Willson.

Este comportamento está diretamente relacionado a distribuição das funcionalidades dentro das arquiteturas. Este comportamento é notório na Sub figura 6.22a, a qual exibe a demanda exercida sobre o microsserviço *rcrud* a qual está distribuído entre diversos processos nas demais arquiteturas.

A partir do comportamento exibido nas Sub figuras 6.22a, 6.22b e 6.22c, faz sentido analisar a vazão de rede de entrada. Dessa forma pode-se analisar a limitação da entrada da rede. O comportamento da vazão da rede de entrada de dados é visível na Figura 6.27.

Figura 6.23: Vazão da entrada de dados para a rede *gameservers*.



Fonte: O próprio autor.

Conforme exibido na Figura 6.27, a arquitetura Rudy contempla um maior consumo da entrada da rede. Entretanto, tanto a arquitetura Salz e Willson possuem, visualmente, o mesmo comportamento. Dessa forma, faz sentido analisar a média os quadrantes para quantificar e verificar se os valores médios são realmente próximos na arquitetura Salz e Willson. A Tabela 6.13 exibe estes dados.

Tabela 6.13: Consumo médio da entrada da rede por quadrante do servidor de jogo.

Arquitetura	Primeiro Quadrante	Segundo Quadrante	Terceiro Quadrante	Quarto Quadrante
Rudy	12,58 Mbps	15,62 Mbps	15,84 Mbps	15,66 Mbps
Salz	9,22 Mbps	10,63 Mbps	11,97 Mbps	11,42 Mbps
Willson	9,12 Mbps	12,30 Mbps	12,27 Mbps	12,23 Mbps

Fonte: O próprio autor.

A Tabela 6.13 exibe a média dos quadrantes da vazão de entrada dos servidores de jogo. Dessa forma, podemos confirmar que a arquitetura Salz possui uma carga de entrada menor que a arquitetura Willson. Entretanto, este aumento é um reflexo direto da comunicação interna da arquitetura ou desempenho por uma otimização

Nesse sentido, pode-se deduzir:

$$\overline{Entrada_{gs_r}} < \overline{Entrada_{gs_s}} < \overline{Entrada_{gs_w}}$$

Deste modo, entende-se:

- $\overline{Entrada}_{gs_r}$: Vazão de entrada médio dos microsserviços na arquitetura Rudy;
- $\overline{Entrada}_{gs_r}$: Vazão de entrada médio dos microsserviços na arquitetura Salz; e
- $\overline{Entrada}_{gs_r}$: Vazão de entrada médio dos microsserviços na arquitetura Willson.

A arquitetura Rudy possui um aumento excessivo em relação as demais arquiteturas selecionadas pela separação do microsserviço *rcrud*. Todas as chamadas, tanto ao Redis e PostgreSQL passa por tal serviço. Dessa forma, temos um aumento elevado em relação as demais arquiteturas.

A arquitetura Salz possui o menor valor de entrada. Tal característica ocorre mesmo com a sincronização de posição de personagens entre os microsserviços *sgame* e *schat*. Esta característica é resultante da comunicação direta com os microsserviços a qual a arquitetura necessita de informações, evitando reinjeção de dados na arquitetura de microsserviços.

A arquitetura Willson possui um valor intermediário em relação as demais arquiteturas selecionadas. Este valor intermediário é dada pela vazão de requisições que a arquitetura Willson consegue exercer ao cliente. Dessa forma, percebemos o tempo de resposta da arquitetura Willson menor que as demais, e consecutivamente recebendo mais requisições, e por sua vez elevando a entrada do serviço como um todo.

6.1.5 Saída de Rede

Este experimento visa analisar o consumo da saída de rede unitariamente, em relação ao número de jogadores simultâneos. Espera-se que o seu crescimento seja de tendência linear junto ao crescimento de jogadores concorrentes. Neste contexto existem os seguintes valores:

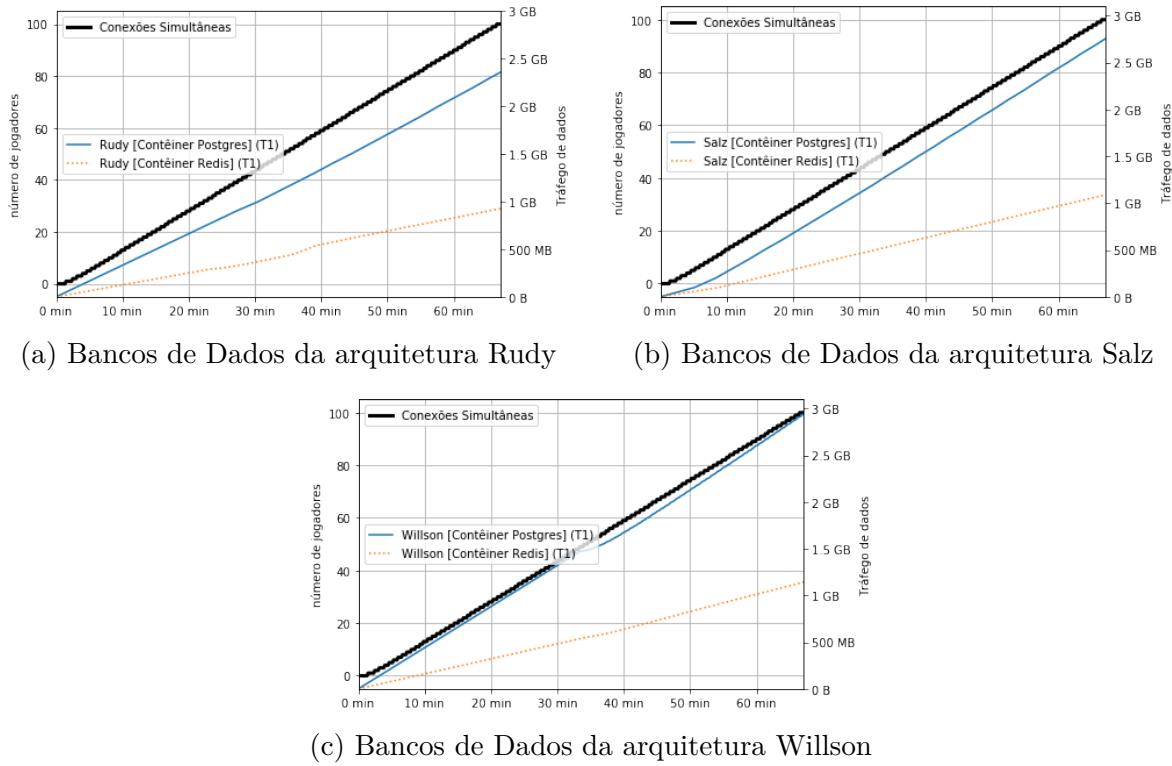
- Jogadores simultâneos: Variável capturada a partir do microsserviço de jogo;
- Saída do serviço: Variável capturada a partir do monitor de recursos do Docker; e
- Saída do ambiente: Variável capturada a partir do sistema operacional.

Nota-se que o número de jogadores simultâneos e o consumo da saída de rede são indexados pelo tempo a qual tais dados foram capturados. Nesse sentido, pode-se relacionar o número de jogadores simultâneos ao consumo de entrada de rede dos microsserviços e do banco de dados. Dado tal contexto, faz sentido realizar uma análise separando os ambientes de Banco de Dados de Microsserviços.

Banco de Dados

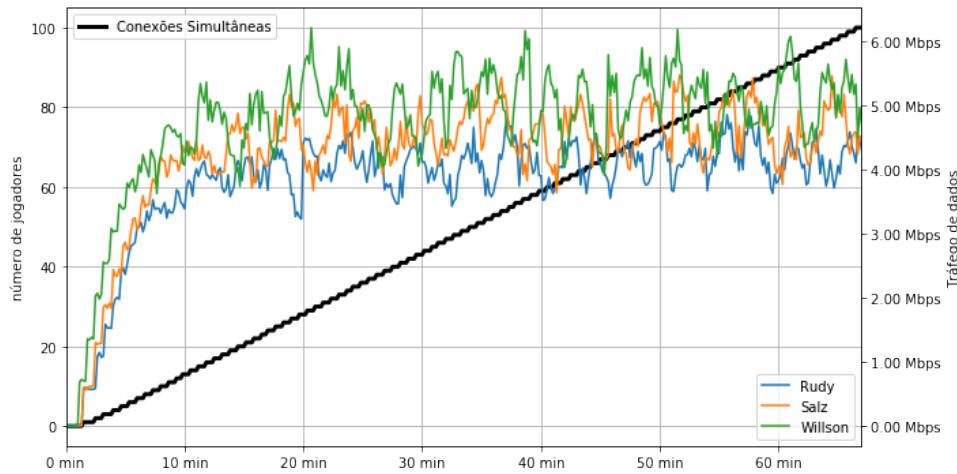
Considerando o ambiente de banco de dados, pode-se realizar a associação de número de jogadores simultâneos e saída de rede consumida pelos contêineres de banco de dados. Essa associação é realizada pelo tempo de registro das métricas. O resultado desta associação pode ser visualizado na Figura 6.24.

Figura 6.24: Saída de dados da rede dos bancos de dados



Fonte: O próprio autor.

Conforme visível na Figura 6.24, o comportamento da saída de dados segue a tendência da entrada e dados ao banco de dados. Este comportamento era previsível, haja vista que o banco de dados recebe requisições a qual geram um determinada saída. Faz sentido analisar a taxa de vazão total do ambiente de dados para fins comparativos a vazão de entrada. O comportamento da vazão da rede de saída de dados é visível na Figura 6.25.

Figura 6.25: Vazão da saída de dados para a rede *gameservers*.

Fonte: O próprio autor.

Percebe-se que o comportamento de vazão de saída da rede segue o mesmo padrão da entrada de rede para a rede de banco de dados, visível ao comparar a Figura 6.25 e 6.21. Faz sentido analisar os quadrantes, validando se o comportamento das médias dos quadrantes acompanham as mesmas proporções conforme a entrada de rede. A Tabela 6.14 exibe os valores de média de vazão de rede dos quadrantes.

Tabela 6.14: Consumo médio da saída da rede por quadrante do servidor de jogo.

Arquitetura	Primeiro Quadrante	Segundo Quadrante	Terceiro Quadrante	Quarto Quadrante
Rudy	2,93 Mbps	3,98 Mbps	4,10 Mbps	4,27 Mbps
Salz	3,42 Mbps	4,57 Mbps	4,62 Mbps	4,63 Mbps
Willson	3,65 Mbps	5,02 Mbps	5,10 Mbps	5,18 Mbps

Fonte: O próprio autor.

A partir da Tabela 6.14, percebe-se uma sequência para comparação entre as arquiteturas no critério referente a consumo de saída de rede. Nesse sentido, pode-se deduzir:

$$\overline{Saida_{db_r}} < \overline{Saida_{db_s}} < \overline{Saida_{db_w}}$$

Deste modo, entende-se:

- $\overline{Saida_{db_r}}$: Vazão de saída média dos bancos de dados na arquitetura Rudy;
- $\overline{Saida_{db_s}}$: Vazão de saída média dos bancos de dados na arquitetura Salz; e
- $\overline{Saida_{db_w}}$: Vazão de saída média dos bancos de dados na arquitetura Willson.

O comportamento visual e a expressão deduzida a partir da média dos quadrantes entre arquiteturas extraído é um reflexo direto da entrada de rede de banco de dados. Dessa forma, podemos comparar a entrada e saída, a fim de ter uma única comparação para vazão de rede sobre o ambiente de banco de dados. Faz sentido comparar a expressão deduzida da Tabela ?? com a expressão deduzida a partir da Tabela 6.14 da seguinte forma:

$$\overline{Entrada_{db_r}} < \overline{Entrada_{db_s}} < \overline{Entrada_{db_w}}$$

$$\overline{Saida_{db_r}} < \overline{Saida_{db_s}} < \overline{Saida_{db_w}}$$

Deduz que:

$$\overline{Rede_{db_r}} < \overline{Rede_{db_s}} < \overline{Rede_{db_w}}$$

Na qual entende-se:

- $\overline{Rede_{db_r}}$: Vazão de rede média dos bancos de dados na arquitetura Rudy;
- $\overline{Rede_{db_s}}$: Vazão de rede média dos bancos de dados na arquitetura Salz; e
- $\overline{Rede_{db_w}}$: Vazão de rede média dos bancos de dados na arquitetura Willson.

Microsserviços

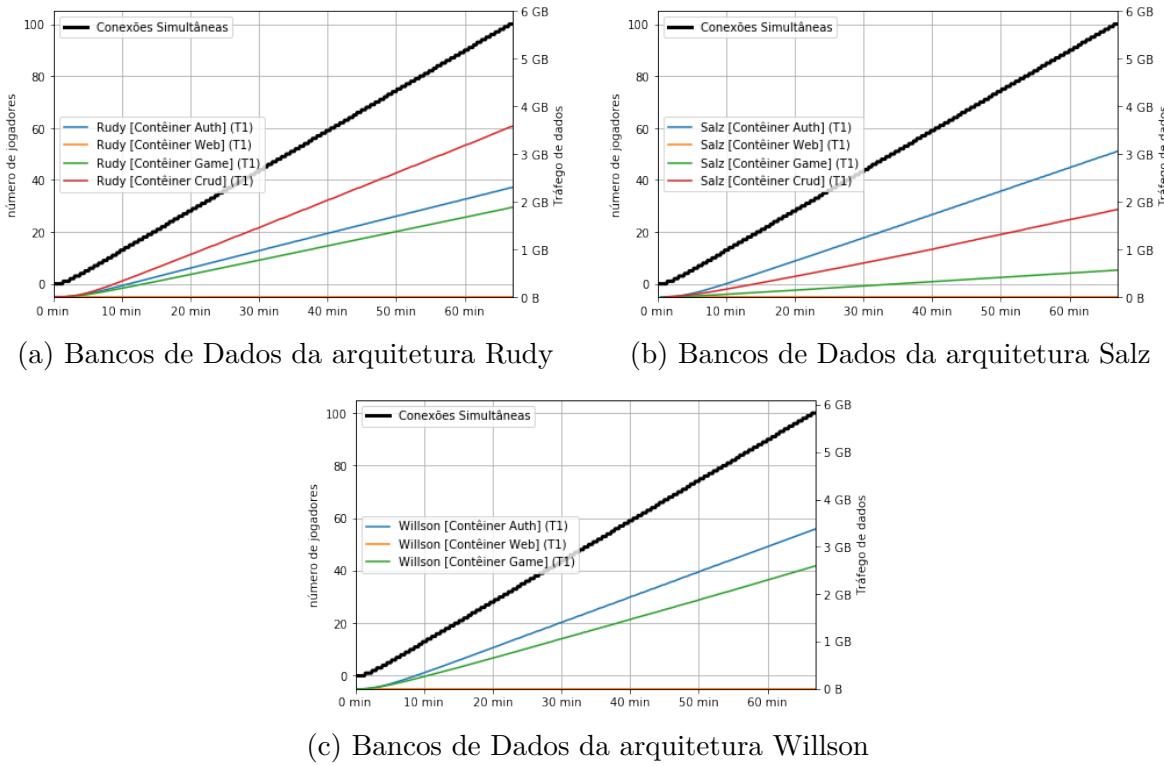
Considerando o ambiente de banco de dados, pode-se realizar a associação de número de jogadores simultâneos e saída de rede consumida pelos contêineres de banco de dados. Essa associação é realizada pelo tempo de registro das métricas. O resultado desta associação pode ser visualizado na Figura 6.26.

A Figura 6.26 exibe os mesmos comportamentos da entrada de rede. A partir da figura, pode-se deduzir os seguintes comportamentos para as arquiteturas de microsserviços:

$$Saida_{rweb} < Saida_{rgame} < Saida_{rauth} < Saida_{rcrud}$$

$$Saida_{sweb} < Saida_{sgame} < Saida_{schat} < Saida_{sauth}$$

Figura 6.26: Saída de dados da rede dos microsserviços



Fonte: O próprio autor.

$$Saida_{wweb} < Saida_{wgame} < Saida_{wauth}$$

Na qual entende-se:

- $Saida_{rweb}$: Curva de saída do microsserviço web na arquitetura Rudy;
- $Saida_{rgame}$: Curva de saída do microsserviço game na arquitetura Rudy;
- $Saida_{rauth}$: Curva de saída do microsserviço auth na arquitetura Rudy;
- $Saida_{rcrud}$: Curva de saída do microsserviço crud na arquitetura Rudy;
- $Saida_{sweb}$: Curva de saída do microsserviço web na arquitetura Salz;
- $Saida_{sgame}$: Curva de saída do microsserviço game na arquitetura Salz;
- $Saida_{sauth}$: Curva de saída do microsserviço auth na arquitetura Salz;
- $Saida_{schat}$: Curva de saída do microsserviço chat na arquitetura Salz;
- $Saida_{wweb}$: Curva de saída do microsserviço web na arquitetura Willson;
- $Saida_{wgame}$: Curva de saída do microsserviço game na arquitetura Willson; e

- $Saida_{wauth}$: Curva de saída do microsserviço auth na arquitetura Willson.

A partir da dedução da Figura 6.26, pode-se dizer que o comportamento dos microsserviços, tanto quanto entrada e saída contemplam o mesmo comportamento. Nesse sentido, pode-se apresentar a seguinte dedução:

$$Rede_{rweb} < Rede_{rgame} < Rede_{rauth} < Rede_{rcrud}$$

$$Rede_{sweb} < Rede_{sgame} < Rede_{schat} < Rede_{sauth}$$

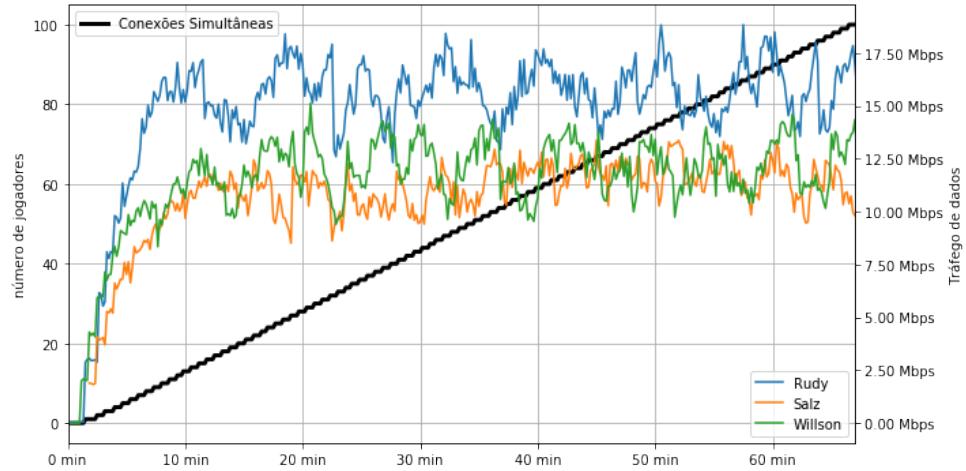
$$Rede_{wweb} < Rede_{wgame} < Rede_{wauth}$$

Na qual entende-se:

- $Rede_{rweb}$: Consumo da rede do microsserviço web na arquitetura Rudy;
- $Rede_{rgame}$: Consumo da rede do microsserviço game na arquitetura Rudy;
- $Rede_{rauth}$: Consumo da rede do microsserviço auth na arquitetura Rudy;
- $Rede_{rcrud}$: Consumo da rede do microsserviço crud na arquitetura Rudy;
- $Rede_{sweb}$: Consumo da rede do microsserviço web na arquitetura Salz;
- $Rede_{sgame}$: Consumo da rede do microsserviço game na arquitetura Salz;
- $Rede_{sauth}$: Consumo da rede do microsserviço auth na arquitetura Salz;
- $Rede_{schat}$: Consumo da rede do microsserviço chat na arquitetura Salz;
- $Rede_{wweb}$: Consumo da rede do microsserviço web na arquitetura Willson;
- $Rede_{wgame}$: Consumo da rede do microsserviço game na arquitetura Willson; e
- $Rede_{wauth}$: Consumo da rede do microsserviço auth na arquitetura Willson.

Além da análise do comportamento de consumo de rede, a partir da saída da rede, faz sentido analisar a vazão da rede por todos os microsserviços operando juntos. A Figura 6.27 exibe a vazão de rede ao longo do tempo do experimento.

A partir da Figura 6.27, percebe-se um maior consumo pela arquitetura Rudy. Este comportamento era previsível pela relação com a rede de entrada. Entretanto, não

Figura 6.27: Vazão da saída de dados para a rede *gameservers*.

Fonte: O próprio autor.

pode deduzir o comportamento entre a arquitetura Salz e Willson. Dessa forma, faz sentido realizar uma análise por quadrantes a partir da vazão de dados. Estes dados são visíveis na Figura 6.15.

Tabela 6.15: Consumo médio da saída da rede por quadrante do servidor de jogo.

Arquitetura	Primeiro Quadrante	Segundo Quadrante	Terceiro Quadrante	Quarto Quadrante
Rudy	13,91 Mbps	17,50 Mbps	17,08 Mbps	17,41 Mbps
Salz	12,34 Mbps	15,01 Mbps	13,85 Mbps	14,84 Mbps
Willson	12,93 Mbps	15,46 Mbps	15,39 Mbps	15,66 Mbps

Fonte: O próprio autor.

A partir dos valores obtidos na Tabela 6.15 pode-se deduzir a relação de consumo de rede. A partir da Tabela 6.14, percebe-se uma sequência para comparação entre as arquiteturas no critério referente a consumo de saída de rede. Nesse sentido, pode-se deduzir:

$$\overline{Saida_{gs_s}} < \overline{Saida_{gs_w}} < \overline{Saida_{gs_r}}$$

Deste modo, entende-se:

- $\overline{Saida_{gs_s}}$: Vazão de saída média dos microsserviços na arquitetura Rudy;
- $\overline{Saida_{gs_s}}$: Vazão de saída média dos microsserviços na arquitetura Salz; e
- $\overline{Saida_{gs_s}}$: Vazão de saída média dos microsserviços na arquitetura Willson.

7 Considerações & Trabalhos futuros

Este TCC tem como objetivo levantar questões relacionadas a complexidade de coordenação de arquiteturas de microsserviços para jogos MMORPG. Estas arquiteturas são complexas devido a quantidade de módulos e serviços necessários que visam suprir a demanda do mercado para esse gênero de jogo. Sendo assim, para melhor descrever as características de tais arquiteturas, este trabalho propõe a realização de uma análise sobre três arquiteturas distintas descritas na literatura.

Dentre as três arquiteturas selecionadas, é explícito o objetivo a qual cada arquitetura foi planejada. O principal objetivo da arquitetura Rudy é normalizar conexões de diferentes qualidades com o seu modelo de paralelização de requisições. Já para a arquitetura Salz o seu objetivo é escalabilidade para jogadores simultâneos em um único ambiente. A arquitetura Willson tem como objetivo reduzir o tempo de resposta aos clientes. Porém, este objetivo não é descrito de forma clara e objetiva pelos autores, os quais desenvolveram tais arquiteturas dentro de cenários de demanda específicos do mercado. Dessa forma, espera-se encontrar padrões de valores que venham de encontro com tais objetivos.

Com relação aos testes, os valores de recurso para o cenário não podem ser estipulados. Dessa forma, foi definido um teto a fim de limitar o consumo das arquiteturas. Porém, espera-se não utilizar o limite destes recursos, tornando o cenário de testes mais simples. Caso o consumo de recursos seja menor ao limite, todas as arquiteturas serão testadas utilizando o mesmo cenário.

Com relação aos testes, faz necessário verificar na totalidade se será viável desenvolver, implantar e testar todas as arquiteturas no período de tempo estipulado. Dessa forma, podem haver alterações para o TCC II. Espera-se para o TCC II projetar, desenvolver, implantar e analisar os dados, conforme o cronograma estipulado.

O atual trabalho, iniciado no primeiro semestre de 2018, atingiu seus objetivos. Dentre as atividades estipuladas para a primeira fase do TCC, todas foram concluídas, porém não dentro do prazo estipulado de um semestre.

As principais considerações acerca da pesquisa referenciada são relevantes as

características de projeto de um serviço distribuído em microsserviços para atender jogos MMORPG. Elas podem ser enumeradas da seguinte forma:

- Preocupação de consumo de recursos e escalabilidade para muitos jogadores, evitando a segregação de jogadores em seu ambiente;
- Preocupação dos autores das arquiteturas acerca do processamento em relação ao escalonador de processos do sistema operacional, cache de dados e/ou requisições e modelo de processamento paralelo das requisições, visando alto desempenho das arquiteturas; e
- Preocupação dos desenvolvedores das arquiteturas com o funcionamento independente do cliente, fomentando o desenvolvimento multiplataforma do gênero.

Estas preocupações ampliam a utilidade do atual trabalho. Entretanto, o conteúdo encontrado para a pesquisa referenciada dificultou o processo de desenvolvimento da referenciação teórica. Em específico, as dificuldades encontradas durante a elaboração da primeira parte do atual trabalho foram:

- Dificuldade de encontrar material científico ou acadêmico correlacionando arquiteturas de microsserviços e arquiteturas para jogos MMORPG.
- Os autores das arquiteturas especificam com exatidão determinados pontos relevantes ao projeto, mas não foi identificado um aprofundamento para determinados microsserviços da arquitetura. Dessa forma foi necessário realizar buscas na literatura a fim de viabilizar tais projetos, seguindo as especificações das arquiteturas.

Além das dificuldades com relação direta aos objetivos atuais deste trabalho, houve a necessidade de realizar um estudo sobre as tecnologias que permeiam o processo de implantação, desenvolvimento e testes de arquiteturas de microsserviços. Dessa forma espera-se utilizar a tecnologia *Docker Swarm* para implantação das arquiteturas, sobre uma nuvem de computadores *OpenStack*. A fim de desenvolver de forma ágil as arquiteturas, será utilizada a linguagem de programação *GoLang*, a qual se propõe ser rápida (compilada, fortemente tipificada, executado como código nativo), a qual inclui uma ampla biblioteca padrão para conexão com banco de dados e protocolo TCP, podendo

facilmente ser implantada utilizando a tecnologia *Docker*. A resolução destas dificuldades na atual etapa facilitará o desenvolvimento do TCC II.

Para o TCC-II serão desenvolvidas as arquiteturas de microsserviços para jogos MMORPG descritas na proposta do atual trabalho. Também serão desenvolvidos testes de carga automatizados a fim de facilitar a etapa de testes das arquiteturas. Por fim, os dados serão coletados por um sistema de análise de recursos desenvolvido no TCC-II, durante os testes, serão analisados a fim de gerar uma análise das arquiteturas de microsserviços para jogos MMORPG que levantará questões de desempenho presentes nela.

Referências

- ACEVEDO, C. A. J.; JORGE, J. P. G. y; PATIÑO, I. R. Methodology to transform a monolithic software into a microservice architecture. In: *2017 6th International Conference on Software Process Improvement (CIMPS)*. Zacatecas, Mexico: IEEE, 2017. v. 1, n. 17417259, p. 1–6.
- ADAMS, A. R. E. *Fundamentals of Game Design (Game Design and Development Series)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 0131687476.
- ADAMS, E. *Fundamentals of Game Design*. New Riders Publishing, 2014. ISBN 978-032192967-9. Disponível em: <<https://www.amazon.com.br/Fundamentals-Game-Design-Ernest-Adams/dp/0321929675>>.
- BARRI, I.; GINE, F.; ROIG, C. A hybrid p2p system to support mmorpg playability. In: *2011 IEEE International Conference on High Performance Computing and Communications*. Banff, Alberta, Canada: IEEE, 2011. p. 569–574.
- BILTON, N. *Search Bits SEARCH Video Game Industry Continues Major Growth, Gartner Says*. 2011. Acessado em: 19/01/2018. Disponível em: <<https://bits.blogs.nytimes.com/2011/07/05/video-game-industry-continues-major-growth-gartner-says/>>.
- BLIZZARD. *StarCraft II*. 2010. [Online; Acessado em: 8. Maio 2018]. Disponível em: <<https://starcraft2.com/en-us>>.
- BORELLA, M. Research note: Source models of network game traffic. v. 23, p. 403–410, 02 2000.
- BUCHINGER, D. *Sherlock Dengue 8: The Neighborhood - Um jogo sério colaborativo-cooperativo para combate à dengue*. 2014. Online; Acessado em: 17. Apr. 2018. Disponível em: <http://www.udesc.br/arquivos/cct/id_cpmenu/1024-diego_buchinger_1_15167055468902_1024.pdf>.
- CHADWICK, J.; SNYDER, T.; PANDA, H. *Programming ASP.NET MVC 4: Developing Real-World Web Applications with ASP.NET MVC*. O'Reilly Media, 2012. ISBN 978-144932031-7. Disponível em: <<https://www.amazon.com/Programming-ASP-NET-MVC-Developing-Applications/dp/1449320317>>.
- CLARKE, R. I.; LEE, J. H.; CLARK, N. Why Video Game Genres Fail: A Classificatory Analysis. *SURFACE*, 2015.
- CLARKE-WILLSON, S. *Guild Wars 2: Scaling from One to Millions*. 2013. [Online; Acessado em: 1. Setembro 2018]. Disponível em: <<https://archive.org/details/GDC2013Willson>>.
- CLARKE-WILLSON, S. *Guild Wars Microservices and 24/7 Uptime*. 2017. Disponível em: <http://twvideo01.ubm-us.net/o1/vault/gdc2017/Presentations/Clarke-Willson_Guild Wars 2 microservices.pdf>.

- DEZA, E. D. M. M. *Encyclopedia of Distances*. Springer, 2009. ISBN 978-364200233-5. Disponível em: <<https://www.amazon.com/Encyclopedia-Distances-Michel-Marie-Deza/dp/3642002331>>.
- DICE. *Battlefield*. 2013. [Online; Acessado em: 8. Maio 2018]. Disponível em: <<https://www.battlefield.com/pt-br>>.
- EA. *FIFA 18 - Soccer Video Game - EA SPORTS Official Site*. 2018. [Online; Acessado em: 8. Maio 2018]. Disponível em: <<https://www.easports.com/fifa>>.
- Exit Games. *Photon Unity Networking*. 2017. [Online; Acessado em: 15. Maio 2018]. Disponível em: <<https://doc-api.photonengine.com/en/pun/current/index.html>>.
- Exit Games. *Serialization in Photon Engine*. 2018. [Online; Acessado em: 29. Agosto 2018]. Disponível em: <<https://doc.photonengine.com/en-us/realtimedev/current/reference/serialization-in-photon>>.
- FARBER, J. Network game traffic modelling. In: *Proceedings of the 1st Workshop on Network and System Support for Games*. New York, NY, USA: ACM, 2002. (NetGames '02), p. 53–57. ISBN 1-58113-493-2. Disponível em: <<http://doi.acm.org/10.1145/566500.566508>>.
- FRANCESCO, P. D. Architecting microservices. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, Gothenburg, Sweden, v. 1, p. 224–229, Apr 2017.
- GOLDSMITH, T. "Cathode-ray tube amusement device". 1947. "Online; Acessado em: 15. Apr. 2018". Disponível em: <<https://patents.google.com/patent/US2455992>>.
- GUINNESS. *Greatest aggregate time playing an MMO or MMORPG videogame (all players)*. 2013. [Online; Acessado em: 23. Apr. 2018]. Disponível em: <<http://www.guinnessworldrecords.com/world-records/most-popular-free-mmorpg>>.
- HANNA, P. *Video Game Technologies*. 2015. Acessado em: 19/01/2018. Disponível em: <<https://www.di.ubi.pt/~agomes/tjv/teoricas/01-genres.pdf>>.
- HOWARD, E. et al. Cascading impact of lag on quality of experience in cooperative multiplayer games. In: *2014 13th Annual Workshop on Network and Systems Support for Games*. Nagoya, Japan: IEEE, 2014. v. 1, n. 14852575, p. 1–6. ISSN 2156-8138.
- HUANG, G.; YE, M.; CHENG, L. Modeling system performance in mmorpg. In: *IEEE Global Telecommunications Conference Workshops, 2004. GlobeCom Workshops 2004*. Northwestern University, USA: IEEE, 2004. v. 1, p. 512–518.
- HUANG, J.; CHEN, G. Digital stb game portability based on mvc pattern. In: *2010 Second World Congress on Software Engineering*. Wuhan, China: IEEE, 2010. v. 2, n. 11836781, p. 13–16. ISSN 978-1-4244-9287-9.
- IKEM, O. V. How We Solved Authentication and Authorization in Our Microservice Architecture. *Initiate*, Initiate, May 2018. Disponível em: <<https://initiate.andela.com/how-we-solved-authentication-and-authorization-in-our-microservice-architecture-994539d1b6e6>>.
- Internet Society. *XDR: External Data Representation Standard*. 2006. [Online; Acessado em: 19. Maio 2018]. Disponível em: <<https://tools.ietf.org/html/rfc4506.html>>.

- JAJEX. *RuneScape Online Community*. 2018. Acessado em: 01/02/2018 às 01:05. Disponível em: <<http://www.runescape.com/community>>.
- JON, A. A. The development of mmorpg culture and the guild. v. 25, p. 97–112, 01 2010.
- JONES, M. et al. *JSON Web Token (JWT)*. 2015. [Online; Acessado em: 1. Maio 2018]. Disponível em: <<https://tools.ietf.org/html/rfc7519>>.
- KHAZAEI, H. et al. Efficiency analysis of provisioning microservices. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Luxembourg, Austria: IEEE, 2016. v. 1, n. 16622230, p. 261–268. ISSN 2330-2186.
- KIM, J. Y.; KIM, J. R.; PARK, C. J. Methodology for verifying the load limit point and bottle-neck of a game server using the large scale virtual clients. In: *2008 10th International Conference on Advanced Communication Technology*. Phoenix Park, Korea: IEEE, 2008. v. 1, p. 382–386. ISSN 1738-9445.
- KLEINA, N. *8 dos maiores mundos virtuais que já conhecemos*. 2018. [Online; Acessado em: 17. Apr. 2018]. Disponível em: <<https://www.tecmundo.com.br/internet/129103-habbo-second-life-8-maiores-mundos-virtuais-conhecemos.htm>>.
- LENGYEL, E. *Mathematics for 3D Game Programming and Computer Graphics, Third Edition*. Cengage Learning PTR, 2011. ISBN 978-143545886-4. Disponível em: <<https://www.amazon.com/Mathematics-Programming-Computer-Graphics-Third/dp/1435458869>>.
- LINIETSKY, A. M. J. *Godot Docs 3.0*. 2018. [Online; Acessado em: 15. Maio 2018]. Disponível em: <<http://docs.godotengine.org/en/3.0>>.
- MDHR. *Cuphead*. 2017. [Online; Acessado em: 8. Maio 2018]. Disponível em: <<https://store.steampowered.com/app/268910/Cuphead>>.
- MICROSOFT. *Age of Empires III - Age of Empires*. 2005. [Online; Acessado em: 8. Maio 2018]. Disponível em: <<https://www.ageofempires.com/games/aoeiii>>.
- MOJANG. *How do I play multiplayer?* 2009. [Online; Acessado em: 8. Maio 2018]. Disponível em: <<https://help.mojang.com/customer/en/portal/articles/429052-how-do-i-play-multiplayer->>.
- NADAREISHVILI, I. et al. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, 2016. ISBN 978-149195625-0. Disponível em: <<https://www.amazon.com/Microservice-Architecture-Aligning-Principles-Practices/dp/1491956259>>.
- NEWMAN, S. *Building Microservices*. O'Reilly Media, 2015. ISBN 978-149195035-7. Disponível em: <<https://www.amazon.com.br/Building-Microservices-Sam-Newman-dp/1491950358>>.
- OpenStack Team. *What is OpenStack?* 2019. [Online; accessed 6. Oct. 2019]. Disponível em: <<https://www.openstack.org/software>>.
- RAYMOND, E. S. *The Art of UNIX Programming (The Addison-Wesley Professional Computing Series)*. Addison-Wesley, 2003. ISBN 978-013142901-7. Disponível em: <<https://www.amazon.com/UNIX-Programming-Addison-Wesley-Professional-Computng/dp/0131429019>>.

- RINGLER, R. *C# Multithreaded and Parallel Programming*. Packt Publishing - ebooks Account, 2014. ISBN 978-184968832-1. Disponível em: <<https://www.amazon.com/Multithreaded-Parallel-Programming-Rodney-Ringler/dp/184968832X>>.
- RIOT. *Guia do Novo Jogador*. 2009. [Online; Acessado em: 8. Maio 2018]. Disponível em: <<https://br.leagueoflegends.com/pt/game-info/get-started/new-player-guide>>.
- ROLLINGS, A.; ADAMS, E. *Andrew Rollings and Ernest Adams on Game Design*. New Riders, 2003. (NRG Series). ISBN 9781592730018. Disponível em: <<https://books.google.com.br/books?id=Qc19ChiOUI4C>>.
- RUDDY, M. *Inside Tibia, The Technical Infrastructure of an MMORPG*. 2011. Disponível em: <<http://twvideo01.ubm-us.net/o1/vault/gdceurope2011/slides-Matthias_Rudy_ProgrammingTrack_InsideTibiaArchitecture.pdf>>.
- SALDANA, J. et al. Traffic optimization for tcp-based massive multiplayer online games. In: *2012 International Symposium on Performance Evaluation of Computer Telecommunication Systems (SPECTS)*. Genoa, Italy: IEEE, 2012. p. 1–8.
- SALZ, D. *Albion Online - A Cross-Platform MMO (Unite Europe 2016, Amsterdam)*. 2016. Disponível em: <<https://www.slideshare.net/davidsalz54/albion-online-a-crossplatform-mmo-unite-europe-2016-amsterdam>>.
- SALZ, D. *Albion Online - Software Architecture of an MMO*. talk at Quo Vadis, 2016. [Online; Acessado em: 29. Agosto 2018]. Disponível em: <https://pt.slideshare.net/davidsalz54/albion-online-software-architecture-of-an-mmo-talk-at-quo-vadis-2016-berlin?from_action=save>.
- SCS. *Euro Truck Simulator 2*. 2016. [Online; Acessado em: 8. Maio 2018]. Disponível em: <<https://eurotrucksimulator2.com>>.
- SPORTV. *League of Legends ganha torneio de fim de ano organizado pela ABCDE*. 2018. [Online; Acessado em: 17. Apr. 2018]. Disponível em: <<https://sportv.globo.com/site-e-sportv/noticia/league-of-legends-ganha-torneio-de-fim-de-ano-organizado-pela-abcdes.html>>.
- STATISTA. *Games market revenue worldwide in 2015, 2016 and 2018, by segment and screen (in billion U.S. dollars)*. 2017. Acessado em: 19/01/2018. Disponível em: <<https://www.statista.com/statistics/278181/video-games-revenue-worldwide-from-2012-to-2015-by-source/>>.
- STATISTA. *Global internet gaming traffic 2021 | Statistic*. 2018. [Online; Acessado em: 19. Apr. 2018]. Disponível em: <<https://www.statista.com/statistics/267190/traffic-forecast-for-internet-gaming>>.
- STATISTA. *Global PC/MMO revenue 2015*. 2018. [Online; Acessado em: 1. Maio 2018]. Disponível em: <<https://www.statista.com/statistics/412555/global-pc-mmo-revenues>>.
- STATISTA. *LoL player share by region 2017*. 2018. Online; Acessado em: 17. Apr. 2018. Disponível em: <<https://www.statista.com/statistics/711469/league-of-legends-lol-player-distribution-by-region>>.

- SUZNJEVIC, M.; MATIJASEVIC, M. Towards reinterpretation of interaction complexity for load prediction in cloud-based mmorpgs. In: *2012 IEEE International Workshop on Haptic Audio Visual Environments and Games (HAVE 2012) Proceedings*. Munich, Germany: IEEE, 2012. v. 1, n. 13171916, p. 148–149. ISSN 978-1-4673-1567-8.
- THOMPSON, G. W. L. *Fundamentals of Network Game Development*. Cengage Learning, 2008. ISBN 978-158450557-0. Disponível em: <<https://www.amazon.com/Fundamentals-Network-Game-Development-Lecky-Thompson/dp/1584505575>>.
- VILLAMIZAR, M. et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. Cartagena, Colombia: IEEE, 2016. p. 179–182. ISSN 1863-2386.
- XEROX. *High-level framework for network-based resource sharing*. 1976. [Online; Acessado em: 19. Maio 2018]. Disponível em: <<https://tools.ietf.org/html/rfc707>>.
- YARUSSO, A. *2600 Consoles and Clones*. 2006. Disponível em: <<http://www.atariage.com/2600/archives/consoles.html>>.
- ZELESKO, M. J.; CHERITON, D. R. Specializing object-oriented rpc for functionality and performance. In: *Proceedings of 16th International Conference on Distributed Computing Systems*. Hong Kong, China: IEEE, 1996. v. 1, n. 5323443, p. 175–187. ISSN 0-8186-7399-0.