Lucas Piske

# Sandgrain: Gerador Escalável de Identificadores Numéricos de 64-bits

Vitória, ES

2019

Lucas Piske

# Sandgrain: Gerador Escalável de Identificadores Numéricos de 64-bits

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Departamento de Informática

Orientador: Prof. Dr. Vítor E. Silva Souza

Vitória, ES

2019

Lucas Piske
Sandgrain: Gerador Escalável de Identificadores Numéricos de 64-bits/ Lucas Piske. – Vitória, ES, 2019-
47 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Vítor E. Silva Souza

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Departamento de Informática, 2019.

1. Sistemas Distribuídos. 2. Banco de Dados. 3. Identificadores. 4. Escalabibli-dade. 5. Disponibilidade. I. Souza, Vítor Estêvão Silva. II. Universidade Federal do Espírito Santo. IV. Sandgrain: Gerador Escalável de Identificadores Numéricos de 64-bits

CDU 02:141:005.7

Lucas Piske

# Sandgrain: Gerador Escalável de Identificadores Numéricos de 64-bits

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Trabalho aprovado. Vitória, ES, 3 de dezembro de 2019:

**Prof. Dr. Vítor E. Silva Souza**
Orientador

**Prof. Dr. Rodrigo Laiola Guimarães**
Universidade Federal do Espírito Santo

**Camila Zacche de Aguiar, MSc**
Instituto de Tecnologia da Informação e Comunicação do Espírito Santo (PRODEST)

Vitória, ES
2019

# Resumo

Neste trabalho é discutida uma implementação de geração de identificadores 64-bits, geralmente usados como chaves primárias em registros de banco de dados, para situação em que o requisito de *throughput* seja muito alto para soluções comuns e já estabelecidas, como usar um sistema de gerenciamento de banco de dados para gerar estes identificadores. Este tipo de contexto pode aparecer em projetos Web com um número grande de acessos simultâneos, como são o caso de plataformas como Instagram, Twitter e etc. Esta solução aproveita ideias do projeto Snowflake, desenvolvido pelo Twitter para solucionar este problema para sistemas utilizando a sua infraestrutura interna (TWITTERENGINEERING, 2010).

Para chegar a implementação final deste sistema foram executados diferentes processos, já bem estabelecidos, na área de Engenharia de Software. Foi realizado o levantamento e a especificação dos requisitos do sistema, foi definida uma arquitetura que melhor desse suporte aos objetivos que o sistema desejava alcançar e por fim foi realizada a implementação e testes.

Durante as fases de implementação e testes foi possível aprender sobre a influência que a troca do protocolo HTTP para TCP teve sobre a transmissão de requisições e respostas, como utilizar uma arquitetura de eventos para diminuir a dependência entre módulos, como utilizar plataformas *cloud* para efetuar testes de desempenho de forma eficiente em relação a custo e etc.

Ao longo de todo este projeto foram utilizados conhecimentos adquiridos durante o curso de Ciência de Computação, principalmente os conceitos e as experiências adquiridos durante os seguintes cursos: Sistemas Distribuídos, Engenharia de Software, Engenharia de Requisitos, Banco de Dados e Programação.

**Palavras-chaves**: Sistemas Distribuídos. Banco de Dados. Identificadores. Escalabiblidade. Disponibilidade.

# Lista de ilustrações


Figura 1 – Diagrama de Casos de Uso do sistema Sandgrain . . . . . . . . . . . . . 23
Figura 2 – Diagrama de Segmentos dos Identificadores . . . . . . . . . . . . . . . 27
Figura 3 – Diagrama da Arquitetura Física do Sistema. . . . . . . . . . . . . . . . . . 30
Figura 4 – Diagrama de Pacotes do Sistema. . . . . . . . . . . . . . . . . . . . . . . . 31
Figura 5 – Relação do *throughput* de geração de identificadores e número de núcleos no processador. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
Figura 6 – Relação do *throughput* de geração de identificadores e número de instâncias no *cluster*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
Figura 7 – Relação da latência de geração de identificadores e número de núcleos no processador. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
Figura 8 – Relação da latência de geração de identificadores e número de instâncias no *cluster*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

# Lista de tabelas

Tabela 1 – Atores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

# Lista de abreviaturas e siglas

| | |
|---|---|
| UML | Unified Modeling Language |

# Sumário


**1 INTRODUÇÃO . . . 10**
**1.1 Objetivos . . . 11**
**1.2 Metodologia . . . 12**
**1.3 Organização da Monografia . . . 12**

**2 REFERENCIAL TEÓRICO . . . 14**
**2.1 Engenharia de Software . . . 14**
**2.2 Especificação e Análise de Requisitos . . . 14**
**2.3 Projeto de Sistemas e Implementação . . . 16**
**2.4 Testes Funcionais . . . 17**
**2.5 Testes de Desempenho . . . 17**
**2.6 Sistemas Distribuídos . . . 18**
**2.7 Trabalhos Relacionados . . . 19**
2.7.1 Flickr . . . 19
2.7.2 Twitter . . . 19
2.7.3 Instagram . . . 20

**3 ESPECIFICAÇÃO DE REQUISITOS . . . 21**
**3.1 Descrição do Escopo . . . 21**
**3.2 Casos de Uso . . . 23**
**3.3 Requisitos Não-Funcionais . . . 24**
3.3.1 Baixa Latência . . . 24
3.3.2 Alta Disponibilidade . . . 24
3.3.3 Escalabilidade . . . 25
3.3.4 Segurança . . . 25

**4 PROJETO ARQUITETURAL E IMPLEMENTAÇÃO . . . 26**
**4.1 Visão Geral da Solução . . . 26**
4.1.1 Geração de Identificadores . . . 26
4.1.2 Sequências e Schemas . . . 28
**4.2 Tecnologias Utilizadas . . . 29**
**4.3 Arquitetura Física . . . 30**
**4.4 Arquitetura do Servidor . . . 31**
4.4.1 Namespace Ledger . . . 32
4.4.2 Namespace Quarantine . . . 32
4.4.3 Namespace Container . . . 33

4.4.4 Namespace Distribution . . . 33
4.4.5 Id Generator . . . 34
4.4.6 Identity Access . . . 34
4.4.7 Health Report . . . 34
4.4.8 Service Discovery . . . 35
4.4.9 Netty Server . . . 35
4.4.10 Métricas . . . 36
**4.5 Biblioteca Cliente . . . 36**

**5 TESTES . . . 38**
**5.1 Testes Funcionais . . . 38**
**5.2 Testes de Desempenho . . . 38**
5.2.1 Plano de Testes . . . 39
5.2.2 Execução . . . 39
5.2.3 Ambiente de Testes . . . 40
5.2.4 Resultados . . . 40

**6 CONSIDERAÇÕES FINAIS . . . 44**
**6.1 Conclusões . . . 44**
**6.2 Limitações e Próximos Trabalhos . . . 45**

**REFERÊNCIAS . . . 46**

**APÊNDICES 47**

# 1 Introdução

Com o advento da Internet puderam ser desenvolvidos sistemas capazes de serem usados por usuários independente da sua localização, desde que estes possuam um navegador. Com o passar do tempo o número de experiências que usuários podem ter usando a Internet aumentou e a qualidade destas experiências também melhoraram, fazendo assim a Internet mais útil para cada vez um grupo maior de pessoas. Em maio de 2019 a porcentagem de pessoas que possuem acesso a Internet mundialmente era de 56,8% (INTERNETWORLDSTATS, 2019).

Assim, os sistemas que usam a Internet como meio de interação puderam ter uma base de usuários muito maior do que se era comum antes dela. Um exemplo é a rede social Facebook, que em abril de 2019 reportou possuir 2,38 bilhões usuários ativos mensais (ZEPHORIA, 2019).

Com aplicações que possuem quantidades de usuários grandes, como a que foi citada anteriormente, se tornando possíveis, novos desafios surgiram e outros que não eram muito comuns começaram a afligir mais projetos do que no passado. Um exemplo de desafio é gerenciar os grandes volumes de dados que são produzidos atualmente. Para lidar com esse problema novas tecnologias vem sendo desenvolvidas, como bancos de dados NoSQL, que são desenvolvidos em sua maioria com esse problema em mente.

O desafio que é o foco deste trabalho é a geração de identificadores de 64-bits para situações que requerem um *throughput* de geração alto. Identificadores tem muitas utilidades, uma das principais em sistemas de informação é a capacidade de anexá-los a um registro do banco de dados, quando inserido, para que depois este registro possa ser referenciado de forma única entre todos os registros contidos em um conjunto, o tipo de atributo que possui esta característica em banco de dados relacionais geralmente é chamado de chave primária.

Alguns tipos de registros possuem atributos que naturalmente serviriam como chave primária, um exemplo seria um atributo que contém um CPF num registro que representa uma pessoa física. Mas outros tipos de registros não possuem um atributo que os identifica de forma natural, mesmo que estes precisem serem identificados de forma única. Para solucionar este problema são gerados identificadores únicos seguindo alguma estratégia e um identificador gerado é adicionado como um atributo extra ao registro e este atributo servirá como sua chave primária.

Neste trabalho é apresentado o processo de desenvolvimento de um sistema que implementa uma estratégia de geração de identificadores capaz de gerar identificadores numéricos de 64-bits e dá suporte a casos que tenham como requisito um *throughput* de

geração de identificadores muito alto. Esse desafio já foi identificado por outras organizações (INSTAGRAMENGINEERING, 2019) e diferentes soluções já foram propostas, mas nenhuma implementação pode ser facilmente implantada em novos ambientes como é comum ser feito com banco de dados de prateleira (MySQL, PostgreSQL e etc), por exemplo.

O sistema proposto é um serviço de rede, que futuramente será publicado de forma a ser *open source*. Esse serviço funcionará de forma distribuída e terá a capacidade de escalar linearmente horizontalmente, ou seja o sistema deverá ser capaz de receber novos nós ao *cluster* e ao receber um novo nó a sua capacidade de gerar identificadores por segundo deve aumentar de forma linear.

## 1.1 Objetivos

Este trabalho tem como objetivo o desenvolvimento de um sistema distribuído linearmente escalável capaz de gerar *surrogates* numéricos de 64-bits, fazendo uso do conhecimento adquirido durante o curso de Ciência da Computação. São objetivos específicos deste projeto:

- Levantamento e documentação dos requisitos, de forma a levar em conta as possíveis necessidades da maioria de usuários em potencial, assim reduzindo a probabilidade de novas soluções semelhantes serem necessárias no futuro;
- Uso de estruturas e arquiteturas do sistema que atendam as necessidades do usuário descobertas durante o levantamento de requisitos e formem uma fundação sólida que aceite evoluções futuras do software;
- Implementação do sistema seguindo as estruturas e arquiteturas definidas de forma precisa para que se tenha certeza que o código sendo desenvolvido esteja alinhado com as necessidades do usuário final;
- Realização de testes de performance para que o comportamento do sistema em diferentes cenários possa ser conhecido e essa informação possa ser usada por futuros usuários quando estes implantarem o sistema em seus ambientes;
- Aplicação do conhecimento obtido durante o Curso de Computação, em particular nas disciplinas de Sistemas Distribuídos, Banco de Dados, Programação III, Engenharia de Software e Projeto de Sistemas de Software.

## 1.2 Metodologia

Para atingir os objetivos apresentados na seção anterior, os seguintes passos foram realizados:

1. Revisão Bibliográfica: pesquisar o conteúdo necessário para realizar os objetivos do projeto. Isso inclui estudar temas de Sistemas Distribuídos, Redes, entre outros. Além disso também pesquisar possíveis ferramentas, bibliotecas e *frameworks* que podem auxiliar na fase de implementação;
2. Design do Sistema: levantar e analisar de maneira aprofundada todos os requisitos das funcionalidades do sistemas, bem como definir as tecnologias e arquiteturas a serem utilizadas no desenvolvimento;
3. Estudo das Tecnologias: estudar a fundo as tecnologias pesquisadas durante a revisão bibliográfica. Alguns exemplos destas tecnologias são: Apache ZooKeeper, Apache Curator, Netty, Protocol Buffers, entre outras;
4. Implementação e Testes Funcionais: realizar a implementação do sistema seguindo as descrições da fase de projeto e, para cada grupo de funcionalidades implementado, efetuar testes para avaliar a exatidão do código;
5. Testes de Desempenho: realizar testes para avaliar o desempenho do sistema desenvolvido em diferentes cenários;
6. Redação da Monografia: escrever a monografia usando a linguagem de marcação *LaTeX*[1] com o editor *TexStudio*[2] a partir do *template abnTeX*,[3] que segue as normas impostas pela ABNT (Associação Brasileira de Normas Técnicas) para documentos técnicos e científicos brasileiros.

## 1.3 Organização da Monografia

Além desta introdução, esta monografia é composta por outros cinco capítulos.

Capitulo 2: é realizada uma discussão sobre os diversos fundamentos teóricos usados para desenvolver este trabalho. Sendo estes fundamentos: Engenharia de Software, Especificação e Análise de Requisitos, Projeto de Sistemas e Implementação, Testes de Desempenho e Sistemas Distribuídos.

Capitulo 3: este capítulo contém a descrição do escopo do problema que este projeto se propõe a resolver e a especificação dos requisitos funcionais e não-funcionais do sistema.

[1] <https://www.latex-project.org/>.
[2] <https://www.texstudio.org/>.
[3] <https://www.abntex.net.br/>.

Capitulo 4: descreve a solução do problema para geração de identificadores de forma distribuída e os componentes relevantes da arquitetura do sistema desenvolvido.

Capitulo 5: explica os testes funcionais executados e o processo realizado para efetuar os testes de desempenho, descreve o ambiente usado e apresenta seus resultados.

Capitulo 6: apresenta a conclusão do trabalho, dificuldades encontradas, limitações e propostas de trabalhos futuros.

# 2 Referencial Teórico

Este capítulo apresenta os principais conceitos teóricos que fundamentaram o desenvolvimento deste projeto e trabalhos relacionados com este. Este capítulo está organizado em 7 seções, sendo estas: Engenharia de Software (2.1), Especificação e Análise de Requisitos (2.2), Projeto de Sistemas e Implementação (2.3), Testes Funcionais (2.4), Testes de Desempenho (2.5), Sistemas Distribuídos (2.6) e Trabalhos Relacionados (2.7).

## 2.1 Engenharia de Software

Infraestruturas e serviços nacionais são controlados por sistemas computacionais e a maioria dos produtos elétricos inclui um computador e um software que o controla. A manufatura e a distribuição industriais são totalmente informatizadas, assim como o sistema financeiro. Por este motivo a Engenharia de Software se tornou uma área da computação tão importante para praticamente toda a sociedade moderna (SOMMERVILLE, 2003).

A Engenharia de Software surgiu visando melhorar a produtividade do processo de desenvolvimento e, ao mesmo tempo, a qualidade dos produtos de software (BARCELLOS, 2018). A Engenharia de Software auxilia na produção de software provendo ferramentas de apoio para as atividades, métodos para orientar a realização das atividades, processo para definir as atividades e os produtos e a qualidade de processo e de produto de software (HIRAMA, 2011).

Ao longo da história da Engenharia de Software foram sendo construídas ferramentas computadorizadas para apoiar o desenvolvimento. Essas iniciativas avançaram bastante, mas ainda assim necessitam da intervenção humana. Foram concebidos vários modelos de processos de software e nenhum pode ser considerado o ideal, devido às suas divergências. Entretanto, todos compartilham de atividades fundamentais como especificação, projeto e implementação, validação e evolução (SOMMERVILLE, 2003).

## 2.2 Especificação e Análise de Requisitos

O termo 'requisito' não é usado de forma consistente pela indústria de software. Em alguns casos, o requisito é apenas uma declaração abstrata em alto nível de um serviço que o sistema deve oferecer ou uma restrição a um sistema. No outro extremo, é uma definição detalhada e formal de uma função do sistema (SOMMERVILLE, 2003).

Os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento. Esses requisitos refletem as necessidades

dos clientes para um sistema que serve a uma finalidade determinada, como controlar um dispositivo, colocar um pedido ou encontrar informações (SOMMERVILLE, 2003).

O levantamento de requisitos (também chamado elicitação de requisitos) combina elementos de resolução de problemas, elaboração, negociação e especificação. Para encorajar uma abordagem colaborativa e orientada às equipes em relação ao levantamento de requisitos, os interessados trabalham juntos para identificar o problema, propor elementos da solução, negociar diferentes abordagens e especificar um conjunto preliminar de requisitos da solução (PRESSMAN, 2016).

Há várias técnicas utilizadas para isso, como, por exemplo: leitura de obras de referência e livros-texto, observação do ambiente do usuário, realização de entrevistas com os usuários, entrevistas com especialistas do domínio, reutilização de análises anteriores, comparação com sistemas preexistentes do mesmo domínio do negócio. O produto do levantamento de requisitos é o documento de requisitos, que declara os diversos tipos de requisitos do sistema (BEZERRA, 2006). As seções essenciais deste documento são:

1. Requisitos funcionais: definem funcionalidades oferecidas pelo sistema. Um exemplo seria oferecer o horário atual aos usuários;

2. Requisitos não-funcionais: estabelecem restrições de qualidade para a implementação do sistema. Por exemplo: usabilidade e segurança;

3. Requisitos normativos: são restrições impostas sobre o processo de desenvolvimento de software. Como prazos, por exemplo.

No contexto dos sistemas de software, a etapa de análise de requisitos é a etapa em que os analistas realizam um estudo detalhado dos requisitos levantados na atividade anterior. A partir desse estudo, são construídos modelos para representar o sistema a ser construído (BEZERRA, 2006).

De maneira simples, um modelo é uma simplificação da realidade enfocando certos aspectos considerados relevantes, segundo a perspectiva do modelo, e omitindo os demais. Modelos são construídos para se obter uma melhor compreensão da porção da realidade sendo modelada (BARCELLOS, 2018).

Estes modelos produzidos são essenciais para o processo de desenvolvimento de sistemas. Estes geralmente são criados para enfocar os aspectos chave, possibilitar o estudo do comportamento do sistema, facilitar a comunicação entre membros da equipe de desenvolvimento e clientes e usuários, possibilitar a discussão de correções e modificações com o usuário, servir como base para a tomada de decisão e formar a documentação do sistema (BARCELLOS, 2018).

## 2.3 Projeto de Sistemas e Implementação

Após a Especificação de Requisitos de Software, complementada por diagramas e protótipos da atividade de Análise, a atividade de Projeto representa a ligação entre a codificação do software e os seus requisitos. A atividade de Projeto visa estabelecer uma arquitetura do software que seja realizável, apoiada em definições de módulos funcionais, detalhamento dos procedimentos ou algoritmos e interfaces entre os módulos (HIRAMA, 2011).

Inicialmente, o projeto é representado em um nível alto de abstração, enfocando a estrutura geral do sistema. Definida a arquitetura, o projeto passa a tratar do detalhamento de seus elementos. Esses refinamentos conduzem a representações de menores níveis de abstração, até se chegar ao projeto de algoritmos e estruturas de dados. Assim, independentemente do paradigma adotado, o processo de projeto envolve as seguintes atividades (FALBO, 2014):

1. Projeto da Arquitetura do Software: determina as estruturas arquiteturais do sistema e seus relacionamentos;

2. Projeto dos Elementos da Arquitetura: o design de cada estrutura arquitetural definidas é realizado em mais detalhes, o que requer subdividir essas estruturas em outras estruturas menores;

3. Projeto Detalhado: entra em mais detalhes sobre os elementos fundamentais da arquitetura, como por exemplo, interfaces. Também estabelece como os componentes de software se comunicarão, como este sistema se integrará como outros sistemas, como pessoas irão operar o software, além de projetar detalhes de algoritmos e estruturas de dados.

Um projeto trata de como resolver um problema, por isso, sempre existe um processo de projeto. No entanto, nem sempre é necessário ou apropriado descrever o projeto em detalhes usando a UML ou outra linguagem de descrição (SOMMERVILLE, 2003).

A atividade de Implementação tem por objetivo traduzir as especificações de software em códigos de programa que possam ser processados por um sistema computacional. Naturalmente, esses códigos são dependentes da linguagem de programação escolhida. As características da linguagem e o estilo de codificação podem afetar profundamente a qualidade e a manutenibilidade do software (HIRAMA, 2011).

As atividades de projeto e implementação de software são invariavelmente intercaladas. O projeto de software é uma atividade criativa em que você identifica os componentes de software e seus relacionamentos com base nos requisitos do cliente. A implementação é

o processo de concretização do projeto como um programador ou esboçado em um quadro ou em papel (SOMMERVILLE, 2003).

## 2.4 Testes Funcionais

Os testes funcionais, ou de caixa-preta, utilizam as especificações (de requisitos, análise e projeto) para definir os objetivos do teste e, portanto, para guiar o projeto de casos de teste. Os testes são conduzidos na interface do software e o programa ou sistema é considerado uma caixa-preta. Para testar um módulo, programa ou sistema, são fornecidas entradas e avaliadas as saídas geradas para verificar se estão em conformidade com a correspondente especificação (FALBO, 2014).

Os testes caixa-preta são empregados para demonstrar que as funções do software estão operacionais, que a entrada é adequadamente aceita e a saída é corretamente produzida, e que a integridade da informação externa (uma base de dados, por exemplo) é mantida (FALBO, 2014).

Em princípio, o teste funcional pode detectar todos os defeitos, submetendo o programa ou sistema a todas as possíveis entradas, o que é denominado teste exaustivo. No entanto, o domínio de entrada pode ser infinito ou muito grande, de modo a tornar o tempo da atividade de teste inviável, fazendo com que essa alternativa se torne impraticável (DELAMARO; MALDONADO; JINO, 2007).

Essa limitação da atividade de teste, que não permite afirmar, em geral, que o programa esteja correto, fez com que fossem definidas as técnicas de teste e os diversos critérios pertencentes a cada uma delas. Assim, é possível conduzir essa atividade de maneira mais sistemática, podendo-se inclusive, dependendo do critério utilizado, ter uma avaliação quantitativa da atividade de teste (DELAMARO; MALDONADO; JINO, 2007).

## 2.5 Testes de Desempenho

Uma vez que o sistema tenha sido totalmente integrado, é possível testá-lo para propriedades emergentes, como desempenho e confiabilidade. Os testes de desempenho precisam ser projetados para assegurar que o sistema possa processar a carga a que se destina. Isso normalmente envolve a execução de uma série de testes em que você aumenta a carga até que o desempenho do sistema se torne inaceitável (SOMMERVILLE, 2003).

O teste de desempenho é feito em todas as etapas no processo de teste. Até mesmo em nível de unidade, o desempenho de um módulo individual pode ser avaliado durante o teste. No entanto, o verdadeiro desempenho de um sistema só pode ser avaliado depois que todos os elementos do sistema estiveram totalmente integrados (PRESSMAN, 2016).

Os testes de desempenho muitas vezes são acoplados ao teste de esforço e usualmente requerem instrumentação de hardware e software. Isto é, frequentemente é necessário medir a utilização de recursos (por exemplo, ciclos de processador) de forma precisa. Instrumentação externa pode monitorar intervalos de execução, log de eventos (por exemplos, interrupções) à medida que ocorrem, e verificar os estados da máquina regularmente. Monitorando o sistema com instrumentos, o testador pode descobrir situações que levam à degradação e possível falha do sistema (PRESSMAN, 2016).

## 2.6 Sistemas Distribuídos

Nos dias de hoje, muitas aplicações precisam estar funcionando 24 horas por dia, todos os dias do ano, e possuem requisitos de disponibilidade e confiabilidade que antes eram necessários em somente um pequeno conjunto de serviços críticos existentes no mundo. Da mesma forma, o potencial de crescimento rápido também significa que aplicações devem ser construídas para que possam escalar de forma quase instantânea em resposta à demanda de usuários. Essas necessidades fazem com que grande parte das aplicações construídas nos dias de hoje precisem ser um sistema distribuído (BURNS, 2018).

Várias definições de sistemas distribuídos já foram dadas na literatura, nenhuma delas satisfatória e de acordo com nenhuma das outras (TANENBAUM, 2007). Para a finalidade deste projeto foi suficiente dar uma caracterização sem ser muito específica:

> *Um sistema distribuído é uma coleção de computadores independentes que aparentam ser um único sistema coerente para seus usuários.*

Essa definição tem vários aspectos importantes. O primeiro é que um sistema distribuído consiste em componentes (isto é, computadores) autônomos. Um segundo aspecto é que os usuários, sejam pessoas ou programas, acham que estão tratando com um único sistema. Isso significa que, de um modo ou de outro, os componentes autônomos precisam colaborar (TANENBAUM, 2007).

Os desafios advindos da construção de sistemas distribuídos são a heterogeneidade dos componentes, ser um sistema aberto, o que permite que componentes sejam adicionados ou substituídos, a segurança, a escalabilidade – isto é, a capacidade de funcionar bem quando a carga ou o número de usuários aumenta –, o tratamento de falhas, a concorrência de componentes, a transparência e o fornecimento de um serviço de qualidade (COULOURIS JEAN DOLLIMORE, 2013).

## 2.7 Trabalhos Relacionados

Durante a realização deste trabalho foram analisadas múltiplos projetos que se propõe a resolver o desafio apresentado neste trabalho. Todos estes trabalhos estudados foram desenvolvidos pelos seguintes serviços *web*: Flickr, Instagram e Twitter. Como todos os trabalhos foram desenvolvidos internamente por estas empresas, a informação sobre o seu desenvolvimento é limitada e somente foi possível ter acesso foi ao que as empresas compartilharam publicamente. Nas 3 seguintes subseções são discutidos os trabalhos por empresa e seus problemas identificados, que o projeto discutido nesta monografia busca resolver.

### 2.7.1 Flickr

Flickr utiliza as capacidades "auto increment"do sistema de gerência de banco de dados MySQL. A empresa cria uma tabela, com um campo que possui o atributo *auto_increment*, em duas instâncias de banco de dados MySQL diferentes e esta tabela é usada para criar uma sequência de números únicos que servem como identificadores.

Em ambas as tabelas o campo é incrementado em dois a cada geração de identificador, mas em uma o seu valor inicial é 1 e na outra dois, assim uma instância MySQL gerará uma sequência de números pares e outra uma sequência de números impares. Dessa forma as duas instâncias podem gerar identificadores de forma concorrente sem a ameaça de gerar dois identificadores iguais.

Um problema com esta estratégia é que não poderia ser usada com mais máquinas caso seja necessário uma capacidade maior de geração de identificadores, podendo se tornar um gargalo para operações que precisem de gerar identificadores. Outro problema é a necessidade de um banco de dados, que na sua implementação tem várias funcionalidades que não seriam usadas e geram um *overhead* para geração de identificadores, aumentando a latência das requisições.

### 2.7.2 Twitter

Para gerar identificadores 64-bits de uma forma não coordenada Twitter criou uma aplicação de rede que produz identificadores compondo os bits de seus identificadores usando uma combinação de um número sequencial, um número único para o servidor que está rodando a aplicação geradora de identificadores e o *timestamp* do momento que o identificador foi gerado.

O número sequencial é atribuído as *threads* dentro da aplicação que gera identificadores e nenhuma *thread* recebe os mesmo números. O número único do servidor é escolhido usando Zookeeper de forma a que nenhuma servidor escolha o mesmo número.

O problema dessa solução é que sua implementação depende altamente na infraestrutura criada internamente pelo Twitter e dessa forma não pode ser facilmente por outras organizações.

### 2.7.3 Instagram

Instagram possui uma arquitetura aonde seu dados são repartidos em múltiplas instâncias de banco de dados, aonde cada instância contém um conjunto de dados que não possui intercessão com as outras instâncias.

Para realizar a geração dos identificadores uma função escrita usando a linguagem PG/PGSQL é adicionada a todas as instâncias quando estas são implantadas. Com esta função instalada esta pode ser invocada dentro da instância, por exemplo quando um registro esta sendo inserido em uma tabela.

A função escrita retorna um identificador de 64-bit que é composto por um *timestamp* do momento em que o identificador foi gerado, um identificador que é único a instância de banco de dados que está executando a função e um número sequencial que é incrementado a cada geração dentro de um milissegundo, permitindo gerar múltiplos identificadores por milissegundo por instância de banco de dados.

Essa implementação se mostra dependente na infraestrutura já existente internamente na organização assim como a solução discutida na seção 2.7.2. Além disso da forma que o identificador é composto esta solução só permite a geração de 1024 identificadores por milissegundo em cada instância.

# 3 Especificação de Requisitos

Neste capítulo são apresentados alguns resultados do processo de Engenharia de Requisitos realizado durante a criação deste sistema. Na Seção 3.1 é descrito o escopo do projeto; na Seção 3.2 são apresentados os diagramas dos casos de uso identificados e na Seção 3.3 são discutidos os requisitos não-funcionais mais relevantes para o desenvolvimento do sistema.

## 3.1 Descrição do Escopo

Atualmente praticamente todos os sistemas de informações desenvolvidos precisam de uma forma de salvar dados. Essa necessidade pode aparecer durante o projeto de um sistema por diferentes motivos, como por exemplo, as leis de um país podem estabelecer que as ações de seus usuários devam ser arquivadas para que no futuro caso seja necessário elas possam ser auditadas por algum orgão do governo.

Para suprir esta necessidade é comum que estes sistemas de informação usem sistemas de gerenciamento de banco de dados de prateleira, como MySQL ou PostgreSQL. Isso acontece pois implementar persistência e recuperação de dados poderia levar muito tempo e iria requerer pessoas com muita experiência nesta área da Computação para desenvolver uma solução satisfatória, como a oferecida por produtos de prateleira já existentes.

Muitos tipos destes dados que os sistemas atuais precisam salvar e recuperar necessitam de uma forma de serem referenciados de forma única e que a informação usada para referenciá-los se mantenha a mesma. Mas para muitos destes tipos de dados não existe um atributo natural que satisfaça estes requisitos, um exemplo, seria um usuário em uma plataforma social, já que suas informações podem a qualquer momento mudar, como nome de usuário, email e etc, não existe nenhuma informação deste usuário que pode ser usada para referenciá-lo.

Portanto, para que seja possível referenciar estes tipos de dados, são gerados identificadores artificiais que são adicionados junto a estes registros e, no futuro, caso seja necessário referenciá-los, este identificador artificial pode ser usado. Uma estratégia comum para gerar estes identificadores artificiais é guardar um número inteiro que é atribuído a cada novo registro e depois incrementado, assim impedindo que dois registros possuam o mesmo identificador e estes registros nunca precisarão modificar os valores destes identificadores.

Estes identificadores podem ser gerados pelos próprios gerenciadores de banco

de dados, mas quando uma aplicação necessita de um *cluster* de banco de dados com múltiplos nós que aceitem operações de escrita, de forma concorrente, e seguindo uma arquitetura de particionamento horizontal, esta operação de geração de identificadores, da forma que é implementada geralmente pelos gerenciadores de banco de dados, não consegue ser escalada assim como as escritas de registros nesta topologia proposta.

Isso acontece porque os gerenciadores de banco de dados implementam a geração de identificadores usando números inteiros que são incrementados a cada vez que o último valor é usado. Para isso somente uma instância do *cluster* do banco de dados pode manter o último valor usado e todas as outras teriam que comunicar com esta instância toda vez que fosse necessário gerar um identificador. Criando um único ponto de falha e um gargalo de performance para todas as operações que necessitem gerar um identificador.

O sistema Sandgrain será criado para permitir a geração de identificadores 64-bits de forma simples para quando desenvolvedores precisam da topologia de sistemas de gerenciamento de banco de dados citada anteriormente, mas seus sistemas não podem sofrer com os problemas gerados pelas soluções comuns atuais. Os usuários do sistema Sandgrain poderão ignorar os métodos de geração de identificadores do sistema de banco de dados e utilizar somente o Sandgrain.

As aplicações dos usuários que se integrarem com Sandgrain poderão gerar um conjunto de identificadores fornecendo o nome da sequência a qual esses identificadores pertencem e o nome do *schema* ao qual esta sequência pertence. Neste contexto, *schemas* são recipientes lógicos utilizados para evitar colisões de nomes de sequências.

Usuários devem criar o *schema* anterior ao seus uso, do contrário a geração de identificadores resultará em erro. *Schemas* poderão ser criados passando somente os seus nomes, sendo que os nomes devem ser únicos. *Schemas* atualmente existentes podem também serem listados e excluídos.

Usuários também têm a possibilidade de gerenciar as permissões que os outros usuários possuem, podendo escolher quais deles podem efetuar operações específicas em recursos designados. Um exemplo seria permitir que somente o usuário **usuario_42** possa utilizar sequências contidas no *schema* **produtos** para gerar identificadores.

Os operadores do sistemas terão a capacidade de recuperar as métricas do sistema enquanto este está funcionando, para que desta forma possam monitorar a saúde do sistema e, caso o sistema se mostre em más condições, medidas poderão ser tomadas para fazer com que este volte para um estado estável.

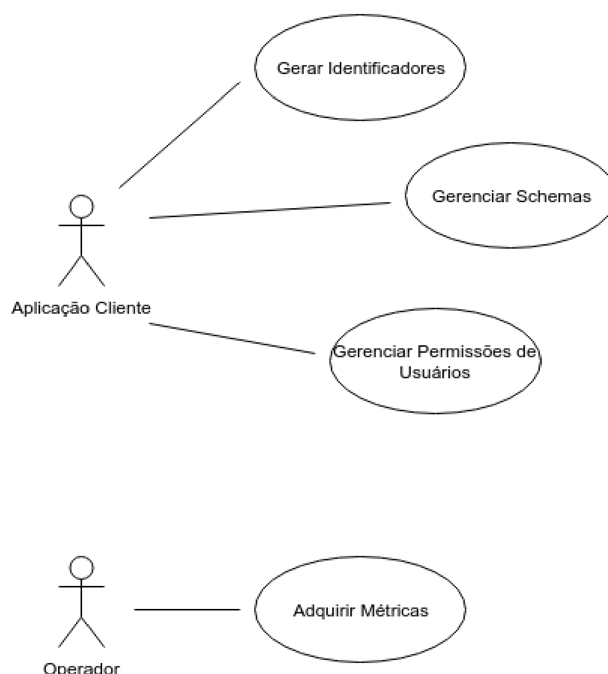


Figura 1 – Diagrama de Casos de Uso do sistema Sandgrain

## 3.2 Casos de Uso

O modelo de casos de uso tem como propósito capturar e descrever as funcionalidades que um sistema deve prover para os atores que interagem com o mesmo. Os atores que foram identificados para este sistema são descritos na Tabela 1. Os casos de uso de cada ator são exibidos na Figura 1.

| **Ator** | **Descrição** |
|---|---|
| Aplicação Cliente | Uma aplicação integrada com o sistema Sandgrain com a capacidade de gerar identificadores, criar schemas, gerenciar permissões e etc. |
| Operador | Um usuário ou aplicação com a responsabilidade de gerenciar uma implantação do sistema Sandgrain |

Tabela 1 – Atores

**Aplicações Cliente** ao se integrarem com Sandgrain tem a capacidade de **Gerar Identificadores**, **Gerenciar Schemas**, **Gerenciar Estado do Cluster** e **Gerenciar Permissões de Usuários**. Mas a capacidade de uma instância deste tipo de ator realizar uma destas ações pode ser restringida caso o mesmo não possua a permissão necessária.

Ao usar Sandgrain para **Gerar Identificadores** a **Aplicações Cliente** deve fornecer o nome da sequência utilizada para gerar o identificador, o *schema* a qual essa sequência deve pertencer e quantos identificadores devem ser gerados. Como resultado desta operação serão retornados a **Aplicações Cliente** um conjunto de identificadores.

Como citado na Seção 3.1, um *schema* deve ser criado no sistema antes que este

possa ser usado para gerar identificadores. Estes podem ser criados por uma **Aplicação Cliente** fornecendo um nome para o novo *schema* a ser criado e este nome deve ser único entre todos os *schemas* criados em uma implantação do sistema. Além de criar *schemas*, **Aplicação Cliente** também pode listá-los e remove-los.

**Aplicações Cliente** podem dar ou restringir permissões a outras instâncias deste tipo de ator para que possam realizar ações sobre recursos do sistema, como por exemplo gerar um identificador usando uma dada sequência. Somente **Aplicações Cliente** que possuem permissão podem editar as dos outros usuários e, quando o sistema é implantado, é configurado um usuário *super* que tem permissão para efetuar todas as operações possíveis no sistema.

**Operadores** são seres humanos ou programas de computador que possuem a capacidade de receber métricas sobre o estado de funcionamento do sistema a qualquer momento de sua execução sem influenciar o seu funcionamento. Assim, o operador pode monitorar o funcionamento do Sandgrain, identificar possíveis problemas que possam estar acontecendo e tomar medidas para voltar o sistema para um estado normal.

## 3.3 Requisitos Não-Funcionais

Nas seguintes subseções serão discutidos os requisitos não-funcionais que foram considerados como os que terão maior influência sobre o processo de desenvolvimento. A ordem das subseções não possui relação com a ordem de importância dos requisitos listados.

### 3.3.1 Baixa Latência

Com a geração de identificadores sendo uma operação extremamente essencial e sendo bastante provável que uma aplicação cliente precise de executar a operação de geração de identificadores em uma mesma transação em um pequeno intervalo de tempo ficou claro que é de muita importância para os usuários que o intervalo entre a estimulação para o início da operação e a sua resposta deve ser muito pequena, para que os usuários deste sistema não percam desempenho em suas aplicações. No contexto deste trabalho foi considerada uma latência como baixa se esta possui o 99º inferior a 6 milissegundos quando o sistema e a rede em que se encontra estão em estado normal.

### 3.3.2 Alta Disponibilidade

Um dos casos de uso mais importantes para os identificadores gerados por esse sistema é a possibilidade de atribuí-los a registros para distinguir um registro do outro. Esta atribuição acontece geralmente quando um registro é criado e caso essa atribuição

não seja possível a criação irá falhar.

Como criar novos registros em um grande número de sistemas é uma operação muito importante, se mostra essencial que este sistema seja capaz de continuar realizando as suas operações, principalmente realizar a geração de identificadores, mesmo que aconteçam falhas de *hardware*, *software* ou energia.

### 3.3.3 Escalabilidade

Como citado na Subseção 3.3.2, um dos casos de uso mais importantes requer que o sistema seja capaz de gerar identificadores ou, do contrário, a aplicação cliente não será capaz de gerar identificadores para atribuí-los a registros de banco de dados e assim distingui-los um dos outros.

Uma razão para que o sistema não consiga gerar identificadores, além do motivo discutido na Subseção 3.3.2, é a taxa de requisições para criação de identificadores por tempo aumentar de forma que uma máquina não seja capaz de lidar com esta carga.

Isso pode acontecer, por exemplo, caso uma notícia que fale sobre a sua aplicação *web* seja publicada e a sua quantidade de usuários cresça muito e rapidamente por causa desta notícia. Se o seu sistema não estiver preparado para lidar com este crescimento a qualidade do serviço muito provavelmente irá se degradar, já que os recursos necessários para lidar com estes novos usuários será bem maior que anteriormente.

Por esta razão se tornou de suma importância que o sistema seja apto a lidar com um aumento de trabalho a ser realizado simplesmente aumentando a quantidade de recursos de processamento e memória disponíveis. Isso pode ser feito melhorando o *hardware* da máquina executando o sistema ou usando diversas máquinas trabalhando em conjunto.

### 3.3.4 Segurança

A fim de prover segurança para seus usuários o sistema irá fornecer uma forma de exigir que todas as operações possíveis de serem feitas por **Aplicações Clientes** sejam autenticadas, ou seja, deverá ser possível saber quem está executando dada operação.

Além disso, usuários serão capazes de especificar quais usuários são permitidos ou não a executarem certas operações. Um exemplo seria não permitir proibir um dado usuário de gerar identificadores usando uma sequência específica ou permitir um usuário a criar *schemas*.

Por fim também será oferecido ao usuário a possibilidade de proteger os dados enviados e recebidos do sistema Sandgrain de possíveis atacantes que tentem visualizar estas informações sem permissão ou modificá-las.

# 4 Projeto Arquitetural e Implementação

Após a fase de especificação e análise de requisitos, foi realizado o projeto do sistema. Nesta fase é realizada a modelagem da estrutura geral deste projeto e como estas estruturas interagem umas com as outras. Tais estruturas podem ser diferentes dependendo da perspectiva que se esteja analisando o sistema, como uma perspectiva física por exemplo.

O projeto e implementação de software é um estágio do processo no qual um sistema de software executável é desenvolvido. O projeto de software é uma atividade criativa em que se identificam os componentes de software e seus relacionamentos com base nos requisitos do cliente (SOMMERVILLE, 2003).

Este capítulo discute de forma breve o Projeto Arquitetural e a Implementação do sistema Sandgrain. Na Seção 4.1 é discutida a solução do problema de geração de identificadores por uma visão geral, na Seção 4.2 são descritas as tecnologias usadas para a implementação deste projeto, na Seção 4.3, a arquitetura do sistema é analisada de uma perspectiva física, nas seções 4.4 e 4.5 são discutidas as implementações do servidor Sandgrain e da biblioteca usada pelos clientes para comunicar com o servidor, respectivamente.

## 4.1 Visão Geral da Solução

A fim de facilitar o entendimento da arquitetura e implementação deste projeto, esta seção discute a solução elaborada para a geração dos identificadores e todos os conceitos essenciais para o seu entendimento, criando assim uma base para a compreensão das responsabilidades dos diferentes componentes deste sistema.

### 4.1.1 Geração de Identificadores

Com objetivo de gerar identificadores, enquanto respeitando os requisitos não-funcionais, o sistema deve ser composto de um *cluster* de instâncias que consigam se coordenar, para realizar esta tarefa, mas também devem ser pouco dependente umas nas outras, de forma que, por exemplo, caso uma instância pare de funcionar e seja encerrada, isso não afete o funcionamento das outras instâncias.

A coordenação entre as instâncias no *cluster*, neste projeto, é feita usando o serviço distribuído ZooKeeper, explicado em mais detalhes na Seção 4.2. A autonomia necessária, citada anteriormente, é uma característica que se dá pelo método usado neste projeto para gerar os identificadores.

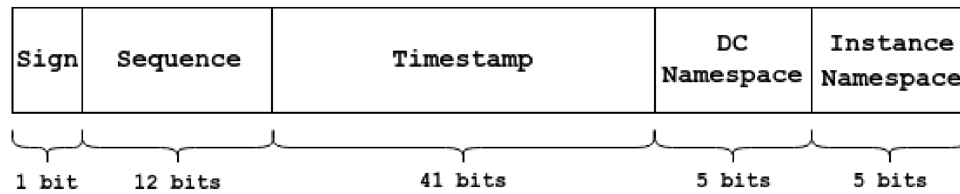


Figura 2 – Diagrama de Segmentos dos Identificadores

O identificadores possuem 64-bits por múltiplos motivos, o primeiro é o fato de ser comum que sistemas usem banco de dados para gerarem seus identificadores inicialmente e estes geralmente são inteiros de 64-bits, portanto manter o tipo e o tamanho facilita na transição entre estratégias de geração de identificadores.

A segunda é que um tamanho maior necessitaria de mais espaço de memória ou disco para o seu armazenamento e, para casos nos quais é necessário armazenar um número muito grande destes identificadores, um aumento mesmo de poucos bits pode fazer uma grande diferença.

A terceira é que um tamanho menor prejudicaria a capacidade de geração de identificadores por milissegundo que o sistema tem, pois diminuiria o número de possíveis valores para um identificador.

Ao gerar os identificadores 64-bits estes são montados a partir de múltiplos segmentos de bits, cada um com um propósito diferente. O diagrama com estes segmentos representados pode ser visto a seguir:

Os segmentos **DC Namespace** e **Instance Namespace** são os responsáveis por prover ao sistema a autonomia discutida. Garantindo que as instâncias de um *cluster* não compartilhem **Instance Namespaces** estas podem gerar identificadores sem a necessidade de comunicação, pois não haverá a possibilidade de dois identificadores iguais em instâncias diferentes serem gerados.

Removendo a necessidade de comunicação entre as instâncias durante o processo de geração de identificadores faz com que caso uma instância tenha problemas, as outras possam continuar gerando identificadores sem problemas.

**DC Namespace** tem a função similar à **Instance Namespace**, mas é usado em casos que existem múltiplos *clusters*, em diferentes *data centers* por exemplo. Assegurando que cada *cluster* tenha um valor único para **DC Namespace** estes *clusters* podem gerar identificadores sem se comunicarem, mas também sem a possibilidade de gerarem identificadores duplicados.

O segmento **Timestamp** é a quantidade de milissegundos decorridos desde de primeiro de janeiro de 1970 até o momento em que o identificador foi gerado. **Sequence** é um valor inteiro que é incrementado a cada geração que ocorre no mesmo instante de

tempo, ou seja, que tenha o mesmo valor para **Timestamp**.

**Sign** é o segmento de bits referente ao sinal do número. Neste método este bit é mantido constante, fazendo que os números gerados sempre sejam positivos.

### 4.1.2 Sequências e Schemas

Com intuito de aumentar a quantidade de identificadores que podem ser gerados em único milissegundo, foi introduzido o conceito de **Sequências** ao sistema. Uma **Sequência** é uma *string* de caracteres que é passada como argumento para geração de identificadores.

Deve-se garantir que um conjunto de identificadores gerados usando a mesma **Sequência** não conterá dois elementos iguais. Consequentemente, no caso que um conjunto de identificadores seja criado usando **Sequências** diferentes, este naturalmente não terá duplicatas.

Fazendo uso de **Sequências** a quantidade possível de identificadores gerados por milissegundo no sistema pode aumentar além do que 64 bits permite, pois entre sequências não é necessário a garantia de unicidade de identificadores.

Um exemplo de uso desse conceito seria um sistema no qual seja necessário gerar identificadores únicos para referenciar as entidades usuário e produto. Poderiam ser usadas sequências diferentes para as entidades, já que só é necessário garantir a unicidade dos identificadores para uma única entidade, mas não para as duas.

**Schemas** foram introduzidos com propósito de resolver um pequeno problema criado com a introdução do conceito de **Sequências**. Como **Sequências** são somente *strings* de caracteres passados como argumento para a criação de identificadores, é possível que dois usuários estejam usando a mesma sem saber, e assim disputem a capacidade de geração de identificadores de uma única sequência.

Para resolver isto, **Schemas** foram introduzidos de forma a desambiguar os casos nos quais múltiplas **Sequências** com o mesmo nome sejam necessárias. Estes funcionam como recipientes lógicos, em que duas **Sequências** com o mesmo nome, usadas em **Schemas** diferentes se comportaram como se tivessem nomes diferentes.

Com objetivo de fazer com que **Schemas** não tenham o mesmo problema que **Sequências**, em que duas pessoas possam estar usando a mesma sem saber, estas devem ser criadas antes de serem usadas e como parte do processo de criação é verificado se um **Schema** com o mesmo nome já existe e, caso exista, a criação irá falhar.

## 4.2 Tecnologias Utilizadas

A linguagem de programação utilizada para implementar este projeto foi Java EE 8 (Java Enterprise Edition 8), devido ao seu desempenho ajudar na latência e *throughput* da geração de identificadores e por seu ecossistema oferecer diversas bibliotecas e *frameworks* necessários para o desenvolvimento desta aplicação. Dentre os mais importantes estão: Spring Boot, Netty, Protocol Buffers, ZooKeeper, Apache Curator e Dropwizard Metrics.

Spring Boot é um *framework* que auxilia na criação de uma aplicação autônoma baseada em Spring, que toma uma posição mais opinativa e torna o desenvolvimento mais rápido. O uso deste *framework* se deve à necessidade do uso do padrão de projeto injeção de dependência, para unir os componentes desenvolvidos, ter a capacidade de gerenciar o ciclo de vida destes componentes, permitir a externalização das configurações de forma padronizada, etc.

Netty é um *framework* de I/O não bloqueante que permite o desenvolvimento rápido e fácil de aplicações de rede que seguem a arquitetura de cliente e servidor. Este foi utilizado tanto no desenvolvimento da biblioteca cliente quanto no servidor para que a comunicação entre eles pudesse ser realizada utilizando o protocolo TCP.

Protocol Buffers é um projeto desenvolvido pelo Google com o propósito de serializar dados estruturados de maneira extensível e neutro quanto a linguagens de programação e plataformas, gerando dados serializados compactos e de forma rápida. Neste projeto esta tecnologia foi usada para serializar as requisições e respostas trocadas pela biblioteca cliente e o servidor.

ZooKeeper é um serviço centralizado que provê coordenação distribuída de forma altamente confiável. Este essencialmente oferece um armazenamento de chave-valor hierárquico, que pode ser usado como um serviço de sincronização em um sistema distribuído, por exemplo. É um projeto *open source*, maduro e já usado em diversos projetos conhecidos, como Kafka e Hadoop. Por essas razões, ele foi escolhido para resolver o problema de coordenação distribuída deste projeto, ao invés de seus alternativas ou mesmo uma implementação de consenso realizada como parte do sistema desenvolvido.

Apache Curator é uma biblioteca cliente para ZooKeeper, que inclui diversas APIs de alto nível e utilidades para tornar a interação com ZooKeeper mais simples e confiável. O uso desta biblioteca foi necessária pois ZooKeeper foi utilizado para realizar coordenação entre as instâncias do *cluster*, como citado anteriormente.

Google Guava é um conjunto de bibliotecas *open source* Java que possuem novos tipos de classes de coleção, utilitários para lidar com concorrência, *strings*, *hashing*, etc. Neste projeto foi utilizado o EventBus, fornecido pelo Guava, com o propósito de realizar a comunicação assíncrona que ocorre entre os diferentes componentes do sistema.

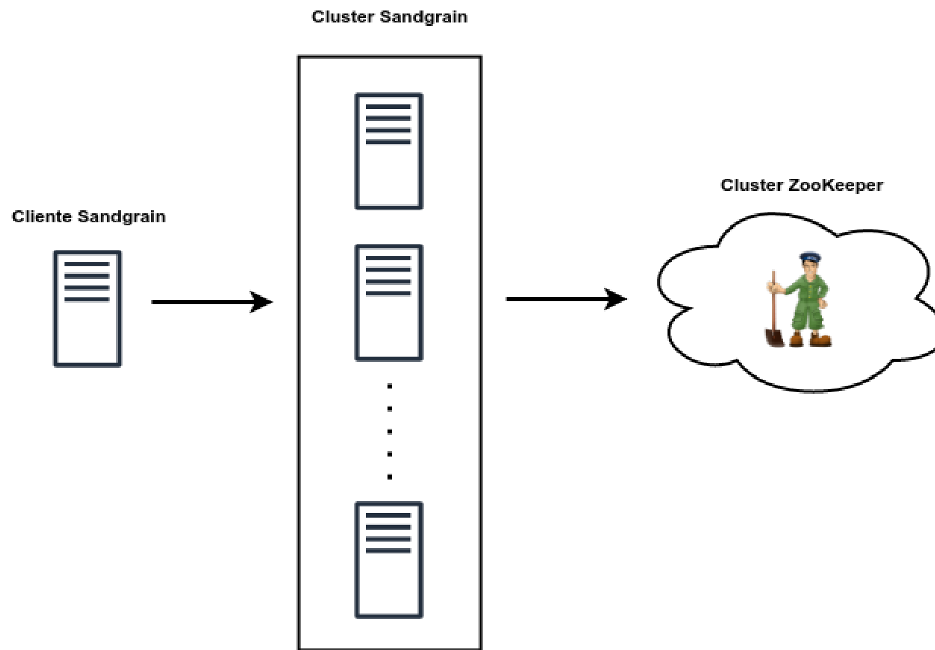


Figura 3 – Diagrama da Arquitetura Física do Sistema.

Dropwizard Metrics é uma biblioteca que provê instrumentação para medir e publicar o estado de uma aplicação Java. Esta biblioteca oferece um conjunto de instrumentos de medida que facilitam, por exemplo, a medição da taxa de um certo evento nos últimos 15 minutos, além disso ela também fornece formas de acessar estas medições em tempo real através de JMX, HTTP, etc.

## 4.3 Arquitetura Física

Nesta seção é apresentada a arquitetura física deste sistema, projetada com objetivo de prover todas as funcionalidades enquanto respeitando todos os requisitos não-funcionais levantados e discutidos na Seção 3.3. Um diagrama mostrando uma visão global da arquitetura física pode ser vista na Figura 3.

A fim de cumprir os requisitos de escalabilidade e alta disponibilidade foi decidido que seria necessário que múltiplos servidores Sandgrain funcionassem como um *cluster*. Desta forma caso algum dos servidores tivesse problemas, outros poderiam assumir o seu trabalho, assim fornecendo alta disponibilidade ao sistema.

Com este *cluster* tendo a capacidade de receber novas instâncias a qualquer momento, que é a forma com que as instâncias foram projetadas neste projeto, o requisito de escalabilidade, como descrito na Subseção 3.3.3, também pode ser cumprido.

O *cluster* ZooKeeper como mostrado no diagrama da Figura 3 é usado somente pelas instâncias do *cluster* Sandgrain, com o intuito de ser comunicaram e, assim, chegarem em consenso, que é necessário em algumas tarefas. Isso é discutido em mais detalhes na

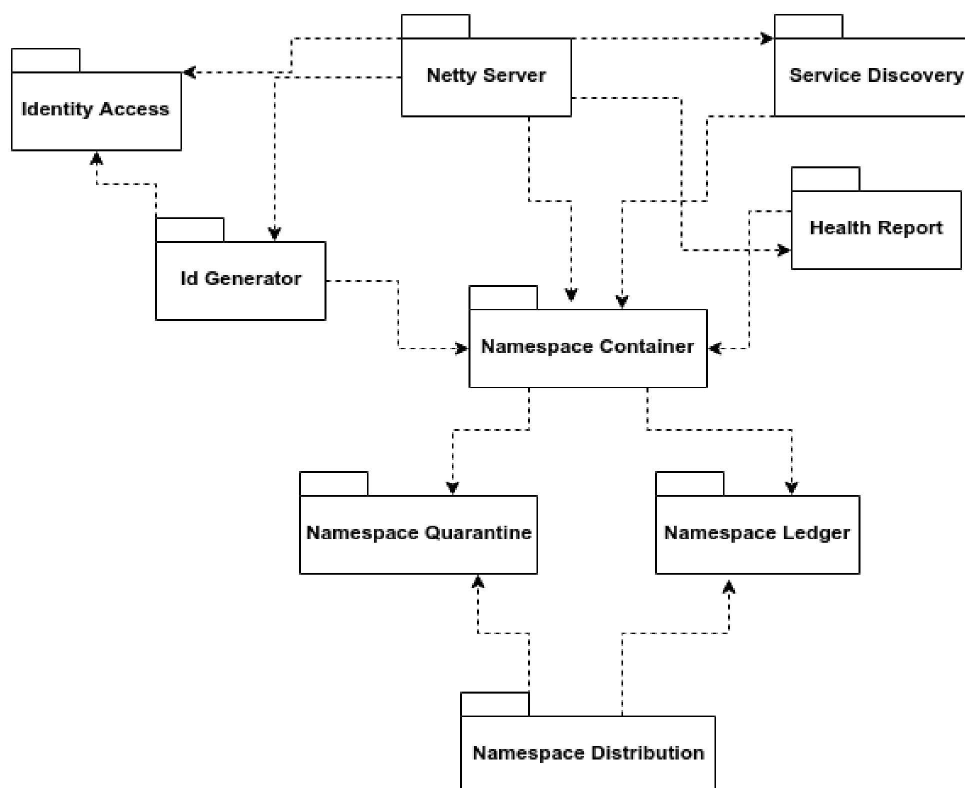


Figura 4 – Diagrama de Pacotes do Sistema.

Seção 4.4, que discute os componentes dos servidores.

O último componente desta arquitetura física é a aplicação que faz uso de uma biblioteca que a permita se comunicar com os servidores do *cluster* Sandgrain, para que possa invocar a API oferecida pelo serviço de forma simples.

Inicialmente a comunicação iria ser implementada usando o protocolo HTTP, mas alguns testes iniciais foram feitos e estes mostraram que para atingir a latência ideal para este tipo de projeto seria necessário um protocolo mais leve e rápido, por isso no final a comunicação foi realizada utilizando o protocolo TCP.

## 4.4 Arquitetura do Servidor

Nesta seção serão discutidos os componentes que formam a arquitetura do servidor, a responsabilidade individual de cada um, como estes se comunicam uns com os outros e por fim como as métricas de funcionamento do sistemas são medidas e expostas.

No diagrama a seguir é possível ter uma visão global de todos estes componentes e das dependências que estes possuem uns dos outros. As subseções a seguir discutem cada um dos componentes.

### 4.4.1 Namespace Ledger

Este componente é responsável por gerenciar os *namespaces* que as instâncias do *cluster* podem utilizar para geração de identificadores ao receber requisições. Cada nó participante do *cluster* recebe um conjunto de *namespaces* de modo que dois conjuntos não possuam interseção, para que não ocorra colisão de identificadores.

Quando uma instância é inicializada o método **addManager** da interface deste componente é invocado, o que irá iniciar um processo que redistribuirá os *namespaces* que cada nó pode utilizar para gerar identificadores.

Essa redistribuição ocorre de modo que cada instância possua um número de *namespaces* mais próximos aos outros o possível. Como ocorrerá uma redistribuição o novo nó inicializado também irá receber *namespaces*, para que possa começar a gerar identificadores.

Ao finalizar de forma normal a aplicação também invocará **removeManagers**, para que os *namespaces* dos nós sejam redistribuídos, e como um nó está deixando o *cluster* os outros iram repartir os *namespaces* antes usados pelo nó que está de saída.

Quando um nó deixa o *cluster* de forma inesperada, um dos nós que foi eleito como líder invocará o método **removeAllManagersExceptFor** para que os *namespaces* antes usados pelo nó que saiu possam ser redistribuídos e assim utilizados pelos nós que continuam servindo requisições.

Através destes três métodos as instâncias mantém os seus *namespaces* atribuídos a cada uma, sempre atualizados e repartidos de forma mais igual o possível. Isso é possível através de uma estrutura de dados que foi chamada de *ledger* que fica nos servidores ZooKeeper e é manipulada pelos nós Sandgrain afim de chegar em um consenso sobre quais nós podem utilizar quais *namespaces*.

Por meio de eventos este componente comunica ao restante dos componentes do servidor os *namespaces* que foram atribuídos ou desatribuídos a este servidor, para que o sistema possa gerar identificadores de forma correta a partir destas informações.

### 4.4.2 Namespace Quarantine

Quando uma instância tem *namespaces* atribuídos a ela, esta não pode começar a fazer uso destes para gerar identificadores imediatamente. Isso se dá pelo fato que outros nós do *cluster* ainda podem estar usando os *namespaces* e, caso dois *namespaces* iguais sejam usados ao mesmo tempo, identificadores repetidos poderiam ser gerados.

Para evitar que *namespaces* não sejam utilizados até outras instâncias tenham parado de usá-los, este componente armazena os *namespaces* atribuídos a uma instância até que outras instâncias confirmem que pararam de fazer uso destes. Quando isso acontece,

este componente dispara eventos dizendo quais *namespaces* podem ser usados de forma segura pela instância.

Este componente também faz uso de uma estrutura de dados armazenada no ZooKeeper chamada de *quarantine* que mantém informação sobre quais instâncias ainda podem estar fazendo uso de quais *namespaces*. Quando o componente recebe um evento informando que um *namespace* não está mas em uso, a estrutura é atualizada e os nós são notificados desta atualização.

### 4.4.3 Namespace Container

Este componente tem como principais objetivos manter uma coleção dos *namespaces* que podem ser usados de forma segura pela instância e oferecer uma interface que permita outros módulos acessarem os *namespaces* armazenados, enquanto rastreia se um *namespace* está sendo usado no momento.

Este rastreamento se faz necessário pois, ao ter um *namespace* desatribuído de si, uma instância precisa parar de fazer uso deste *namespace* e, ao parar, deve informar aos outros nós que fez isto. Como a instância só pode notificar os outros métodos quando não estiver mas fazendo uso do *namespace*, é necessário rastrear o uso de *namespaces* dentro de uma instância.

O gerenciamento de *namespaces* é realizado escutando eventos publicados pelos outros componentes do sistema e manipulando a coleção interna. Quando um evento, por exemplo, com informação que um *namespace* pode sair com segurança da quarentena, o componente deve adicioná-lo à coleção interna.

Para oferecer acesso aos *namespaces* armazenados, a interface do componente permite a criação de *Pools* de *namespaces*. Com o uso do padrão de projeto *Pool* é possível que os usuários da interface usem os *namespaces* e o componente consiga rastrear quais destes estão sendo usados no momento.

### 4.4.4 Namespace Distribution

Como os nós do *cluster* podem parar de funcionar a qualquer momento, é necessário que, caso isto aconteça, outra instância possa realizar ou finalizar tarefas que um nó deveria ter realizado antes de deixar o *cluster*.

A principal tarefa neste caso é a redistribuição de *namespaces*, que deve ocorrer quando um nó deixa o *cluster*. Caso essa redistribuição não ocorra entre os nós restantes, estes não poderiam fazer uso dos *namespaces*, assim diminuindo a capacidade de geração de identificadores por milissegundo.

Para lidar com esse desafio, este componente utiliza o *framework* Apache Curator

e ZooKeeper para realizar eleições de nós líderes, de modo a sempre ter no máximo um nó líder em dado momento da execução do *clutser*. Este nó líder tem como responsabilidade efetuar a redistribuição de *namespaces* caso um nó deixe o *cluster* sem fazê-lo.

### 4.4.5 Id Generator

Este componente foi criado com a responsabilidade de gerar os identificadores a serem retornados aos usuários e criar, remover e listar todos os *schemas* do *cluster*.

A partir dos métodos **create**, **delete** e **list** da interface **SchemaService**, oferecida por este componente, uma estrutura de dados armazenada no ZooKeeper é manipula a fim de gerenciar os *schemas* que podem ser usados para gerações de identificadores em todo *cluster*.

A geração de identificadores é provida, por este componente, pelo método **generateId** que, a partir de um *schema* existente no *cluster*, o nome de uma *sequence* e a quantidade desejada de identificadores a serem gerados, gera e retorna os identificadores.

### 4.4.6 Identity Access

Este componente tem como funcionalidades o gerenciamento de regras de acesso a recursos do *clutser*, como *schemas*, e a autorização de operações feitas por usuários sobre estes recursos.

Através dos métodos **createAclRule**, **deleteAclRule** e **describe**, o componente oferece a capacidade de gerenciar o acesso de usuários a certos recursos do *cluster*. Os métodos **createAclRule** e **deleteAclRule** permitem a criação e remoção de regras de acesso a recursos, respectivamente, enquanto **describe** provê uma forma de ler todas as regras de acesso criadas.

A persistência destas regras criadas pelos usuários é feita por meio de uma estrutura de dados simples que é mantida nos servidores ZooKeeper.

A autorização é desempenhada através do uso do método **checkGranted** que é capaz de verificar se, dadas as regras de acesso registradas no sistema, é permitido ou não a um usuário realizar uma dada operação sobre um recurso especificado. A autenticação de usuários necessária para realizar a verificação de autorização é discutida na Seção 4.4.9.

### 4.4.7 Health Report

Este componente oferece, por meio de sua interface, um relatório sobre a saúde da instância. Ao invocar o método **getReport**, diferentes informações sobre a capacidade da instância de realizar suas funcionalidades como se é esperado por seus usuários são retornadas.

Essas informações podem ser usadas por clientes para determinar se certas instâncias poderão responder às suas requisições ou se será necessário parar de rotear requisições para esta instância temporariamente ou de forma permanente.

Essas informações também são importantes para futuras ferramentas que monitorem o *cluster*, para que seus operadores saibam se algum problema está acontecendo e se é necessária a sua intervenção.

### 4.4.8 Service Discovery

O objetivo deste componente é fazer com que os clientes do *cluster* Sandgrain consigam, ao se conectar, saber quais nós existem e ter informações para que possam se conectar a eles. Três métodos fazem parte da interface deste componente: **registerService**, **unregisterService** e **getOnlineSandgrainHosts**.

Quando a instância foi inicializada completamente e está pronta para servir requisições, o método **registerService** é invocado e este modificará uma estrutura de dados no ZooKeeper, inserindo as informações necessárias para que clientes consigam se conectar ao nó. Ao modificar esta estrutura de dados todos os nós do *cluster* serão notificados e também receberão as informações inseridas pelo nó ingressante.

O método **unregisterService** será invocado quando a instância inciar o seu processo de desligamento, assim notificando os outros nós e clientes do *cluster* que ele não estará mais respondendo requisições. Este método modifica a mesma estrutura de dados que **registerService**, desta vez removendo as informações antes adicionadas pela mesma instância.

O método **getOnlineSandgrainHosts** retorna as informações registradas pelos nós através de **registerService**. Já **getOnlineSandgrainHosts** é utilizada atualmente pelos clientes do *cluster* para que possam, a partir da conexão com um nó, ter informação suficiente para se conectar com os demais.

### 4.4.9 Netty Server

Este componente tem como responsabilidade receber as requisições, enviadas pelo cliente através da rede e, baseado na requisição, invocar operações implementadas nos outros componentes. Além disso, também é o dever deste componente realizar o gerenciamento do ciclo de vida dos outros componentes do servidor.

Para implementar as funcionalidades deste componente foi feito uso extensivo dos *frameworks* Netty e Spring Boot. Netty foi utilizado para implementar todos os aspectos da comunicação que o servidor faz com o cliente, enquanto Spring Boot foi usado para implementar a externalização das configuração do servidor, instanciar todos os componentes e gerenciar o ciclo de vida destes.

Seguindo o padrão de projeto MVC, para cada operação que usuários podem realizar através de requisições, existe uma classe que irá receber a requisição, delegar a realização da operação para outro módulo, receber o resultado desta operação e retornar uma resposta.

A serialização e desserialização destas requisições e respostas é efetuada usando Protocol Buffers. Compartilhando uma especificação entre a biblioteca cliente e o servidor, estes conseguem interpretar as mensagens enviadas entre eles. A partir desta especificação uma série classes Java são compiladas e proveem as estruturas das requisições e respostas definidas na especificação e os mecanismos capazes de realizar a serialização e desserialização destas.

Também utilizando Netty foi possível implementar criptografia para a comunicação entre clientes e servidores, verificação de integridades dos dados enviados e autenticação de clientes. Todas estas funcionalidades proveem do uso do protocolo de segurança SSL, implementado por este componente com suporte da classe **SslHandler**, provida por Netty.

### 4.4.10 Métricas

Em todos os componentes citados nas subseções anteriores, métricas são medidas e capturadas para aumentar a capacidade dos operadores do *cluster* de servidores Sandgrain detectarem qualquer anomalia e poderem tomarem decisões sobre o que deve ser feito para voltar este *cluster* para um estado normal.

Usando as estruturas de medição fornecidas pela biblioteca Dropwizard Metrics cada componente mede a ocorrência de eventos importantes em seu contexto usando a estrutura de medida que faz mais sentido para cada tipo de evento. Um exemplo seria a métrica **idGenerationThroughput.meter**, que mede quantos identificadores foram gerados por segundo nos últimos 5,10 e 15 minutos.

Com as medidas sendo feitas de forma correta, a última coisa a fazer com as métricas é expô-las aos usuários. Para isso foi usado um exportador JMX, fornecido por Dropwizard Metrics, que expõe as métricas, sem a necessidade de muita configuração, para serem consumidas por qualquer cliente JMX.

## 4.5 Biblioteca Cliente

Para que aplicações possam fazer uso das funcionalidades oferecidas pelo serviço desenvolvido, estas precisam se comunicar com o as instâncias do *cluster*. A fim de facilitar a implementação da comunicação que deve ocorrer, foi desenvolvida uma biblioteca que pode ser usada por aplicações clientes para invocar as funcionalidades desejadas no serviço.

A biblioteca é responsável somente por gerenciar a conexão com o *cluster* de forma

transparente e realizar as requisições ao serviço. Ela foi mantida o mais simples possível para que novas implementações possam ser criadas, sem muito trabalho e, assim, mais projetos possam ter acesso ao serviço desenvolvido neste trabalho.

Como parte do gerenciamento da conexão com o *cluster*, a biblioteca implementa o descobrimento das instâncias e a criação e manutenção das conexões feitas com estas instâncias.

O descobrimento é um processo executado pelo cliente continuamente para que este possa saber quais instâncias estão conectadas ao *cluster* a todo momento e assim possa se conectar e realizar as requisições desejadas.

Tendo estas informações, o cliente se conecta a múltiplas instâncias e supervisiona estas conexões para que, caso algum problema ocorra, este possa simplesmente criar uma nova conexão com outro nó que não esteja apresentando problemas e continuar realizando suas atividades.

Com as conexões estabelecidas, a API oferecida pela biblioteca pode ser usada para realizar as operações sobre o serviço, como gerar identificadores, sendo que cada método oferecido pela API resulta na criação de uma requisição, que é serializada, enviada a uma das instâncias do serviço e recebe uma resposta.

Para interoperar de forma fácil com as instâncias, o cliente também utiliza o projeto Protocol Buffers para criação, serialização e deserialização de requisições e respostas trocadas pelos dois, como já citado na Seção 4.4.9. Compartilhando a estrutura das mensagens que são trocadas entre clientes e nós, a implementação da interoperabilidade se tornou simples.

# 5 Testes

Neste capítulo são discutidos os dois principais tipos de testes que foram executados durante este projeto. Na Seção 5.1 são discutidos os testes funcionais realizados e na Seção 5.2 os testes de desempenho.

## 5.1 Testes Funcionais

A fim de testar que o sistema implementado respeita os requisitos funcionais levantados durante a fase de levantamento de requisitos foram realizados múltiplos conjuntos de testes manuais, com cada conjunto focando em um grupo de funcionalidades do sistema.

Os grupos definidos foram: gerenciamento de permissões de usuários, gerenciamento de *schemas*, coleta de métricas e geração de identificadores. O gerenciamento de permissões foi testado modificando as permissões de um usuário e realizando operações como este para verificar se as suas permissões estão sendo observadas.

O gerenciamento de *schemas* foi testado criando, listando e deletando um conjunto de schemas, avaliando se dois ou mais *schemas* não possuem o mesmo nome. A coleta de métricas foi testada usando um cliente JMX para monitorar as métricas enquanto diferentes operações eram feitas no sistema e foi analisado se os valores das métricas eram os esperados para cada contexto.

A geração de identificadores foi testada realizando várias requisições de gerações de forma concorrente em uma mesma sequência e em diferente sequências. Os resultados destas operações, ou seja, os identificadores gerados, foram armazenados e ao final foi verificado se toda requisição gerou identificadores e se não foram gerados identificadores iguais para uma mesma sequência, como se é esperado do sistema.

Durante o processo de desenvolvimento estes testes ajudaram a descobrir *bugs* na implementação do sistema, mas até o final do processo todos estes *bugs* foram eliminados e o comportamento do sistema foi observado em todos os testes funcionais.

## 5.2 Testes de Desempenho

De forma a avaliar a qualidade do desempenho do sistema desenvolvido testes foram realizados analisando o comportamento da latência e *throughput*, com o sistemas possuindo diferentes configurações. Nas próximas subseções são explicados como estes testes foram realizados, o seu ambiente de execução e seus resultados.

### 5.2.1 Plano de Testes

A fim de testar o comportamento da latência e do *throughput* da geração de identificadores do sistema, foi decidido realizar uma sequência de testes na qual requisições seriam feitas de forma concorrente e a cada teste seria observado o comportamento do *throughput* e da latência. Caso fosse percebido que o sistema se comportou de forma normal, a quantidade de requisições por segundo seria aumentada e um novo teste seria realizado, até que seja notada uma deterioração no comportamento do sistema.

Essa sequência de testes foi repetida múltiplas vezes, cada vez com o *cluster* Sandgrain possuindo características diferentes, de modo a mostrar como estas tem efeito no comportamento do sistema ao gerar identificadores. Foi avaliado o *cluster* sendo composto de uma única instância, mas com esta possuindo diferentes níveis de capacidades de processamento e o cluster tendo diferentes números de instâncias, com o mesmo nível de processamento e este nível sendo o menor usado para avaliar o comportamento de uma única instância.

### 5.2.2 Execução

A execução do plano de testes criado foi realizada utilizando o software *open source* JMeter, uma aplicação Java criada com propósito de efetuar testes de carga e medir performance. Esta foi originalmente criada para testar aplicações Web, mas expandiu suas funcionalidades e atualmente possui a capacidade de testar outros tipos de aplicações, incluindo a aplicação desenvolvida neste projeto.

Porém, para aplicações como a desenvolvida neste projeto, que possuem bibliotecas clientes dedicadas, é necessária a implementação de um Sampler para que JMeter consiga usar a biblioteca cliente para realizar as requisições durantes os testes. Com o Sampler desenvolvido e configurado para uso no JMeter os planos de testes puderam ser implementados usando a interface gráfica do JMeter.

O próximo passo foi realizar a execução dos planos de testes implementados. Para isso, em cada experimento foram implantados os *clusters* sujeitos aos testes, com seus tamanhos variando conforme a necessidade do experimento sendo executado. Além dos *clusters* sujeitos ao testes também foi se usado o JMeter de forma distribuída, assim a carga de requisições pôde aumentar sem ser limitada pelos recursos de uma única máquina.

Após a infraestrutura necessária estar ser implantada em cada experimento, o JMeter foi invocado para que o plano de teste pudesse ser executado, os dados do experimento coletados e armazenados em disco na máquina mestre do *cluster* JMeter. Estes dados armazenados depois foram compilados em um relatório e seus resultados são discutidos na Seção 5.2.4.

### 5.2.3 Ambiente de Testes

Para que o ambiente de testes se aproxime de um ambiente onde este sistema seria normalmente usado, é necessário que sejam utilizadas múltiplas máquinas. Para isso foi usada a suíte de computação em nuvem oferecida pelo Google, mais especificamente o seu serviço Google Kubernetes Engine, que oferece *clusters* Kubernetes gerenciados, para implantação de aplicações em containers.

Através deste serviço foram criadas especificações para as máquinas necessárias para os testes e a quantidade para cada especificação, e o serviço instanciou estas de forma automática. Foram definidas três especificações diferentes de máquinas, uma para cada conjunto que desempenou um papel distinto nos testes. Uma para as máquinas participantes do *cluster* Sandgrain, outra para participantes do *cluster* ZooKeeper e outra para integrantes do *cluster* JMeter.

Através deste serviço foram criadas especificações para as máquinas necessárias para os testes e a quantidade para cada especificação, o serviço então instanciou estas de forma automática. Foram definidos cinco especificações diferentes de máquinas, uma foi criada para especificar a configurações das máquinas que iriam executar o JMeter e outra para as que iriam compor o cluster ZooKeeper. As outras três especificam configurações para instâncias com diferentes capacidades de processamento de um *cluster* Sandgrain, que foram usadas em diferentes sequências de teste.

### 5.2.4 Resultados

Com os testes realizados e os dados deste processo coletados e armazenados o próximo passo foi organizar os resultados de uma forma a facilitar a análise destes. Para isso, organizamos os dados nos 4 gráficos que são encontrados nesta seção.

Na Figura 5 temos organizado os resultados dos testes feitos para analisar a relação de identificadores gerados por segundo e o número de núcleos em uma única máquina.

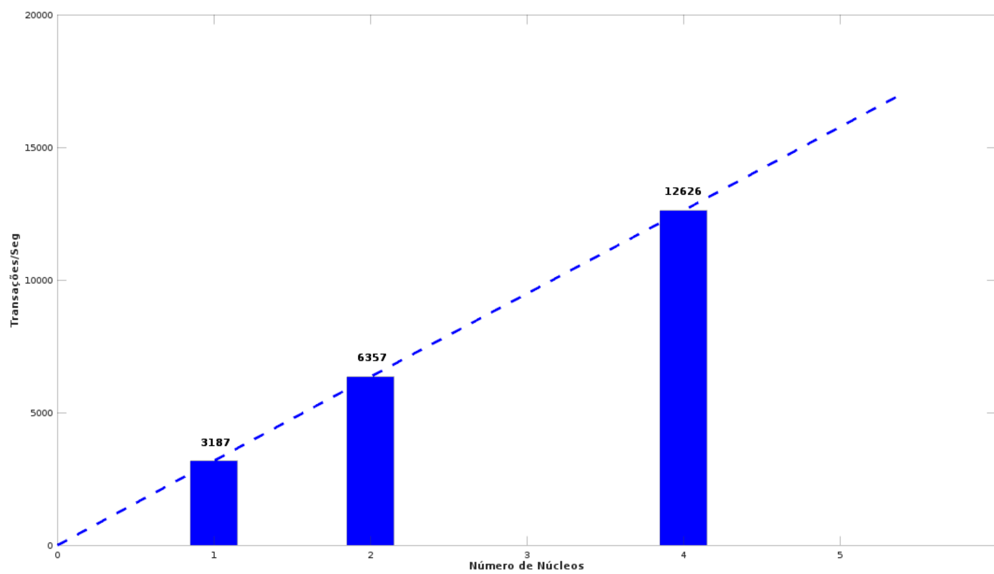


Figura 5 – Relação do *throughput* de geração de identificadores e número de núcleos no processador.

É possível notar no gráfico da Figura 5 que a medida que aumentamos o número de núcleos em uma máquina a quantidade de identificadores gerados por segundo aumenta de forma linear, para até 4 núcleos.

Na Figura 6 foram organizados os resultados dos testes feitos para analisar a relação de identificadores gerados por segundo e o número de instâncias em um *cluster*.

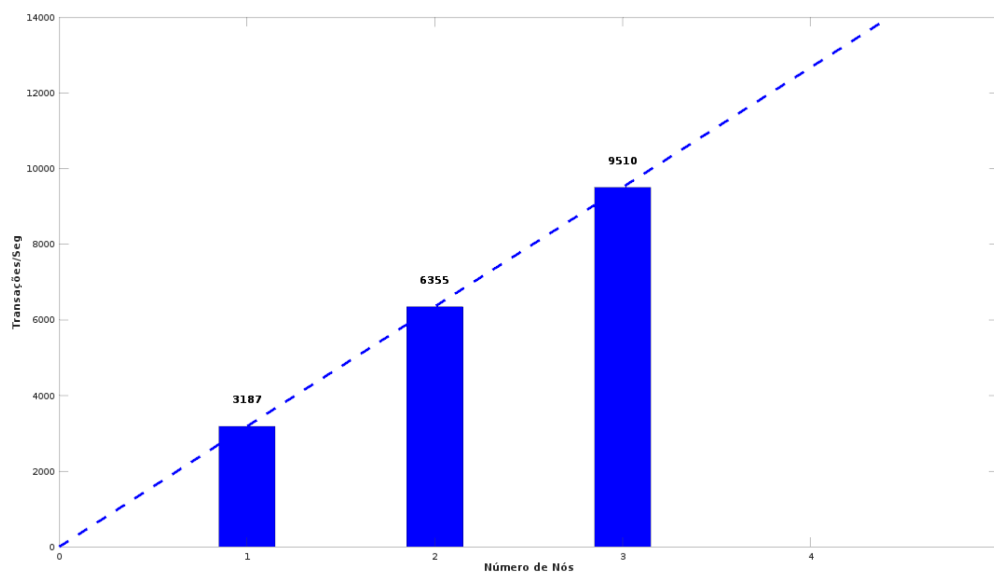


Figura 6 – Relação do *throughput* de geração de identificadores e número de instâncias no *cluster*.

O gráfico da Figura 6 mostra que à medida que o número de instâncias cresce a capacidade do *cluster* de gerar identificadores por segundo também cresce, de forma linear.

A Figura 7 mostra a latência à medida que a quantidade de núcleos em uma máquina aumenta, enquanto a Figura 8 mostra a latência à medida que a quantidade de instâncias pertencentes ao *cluster* aumenta.

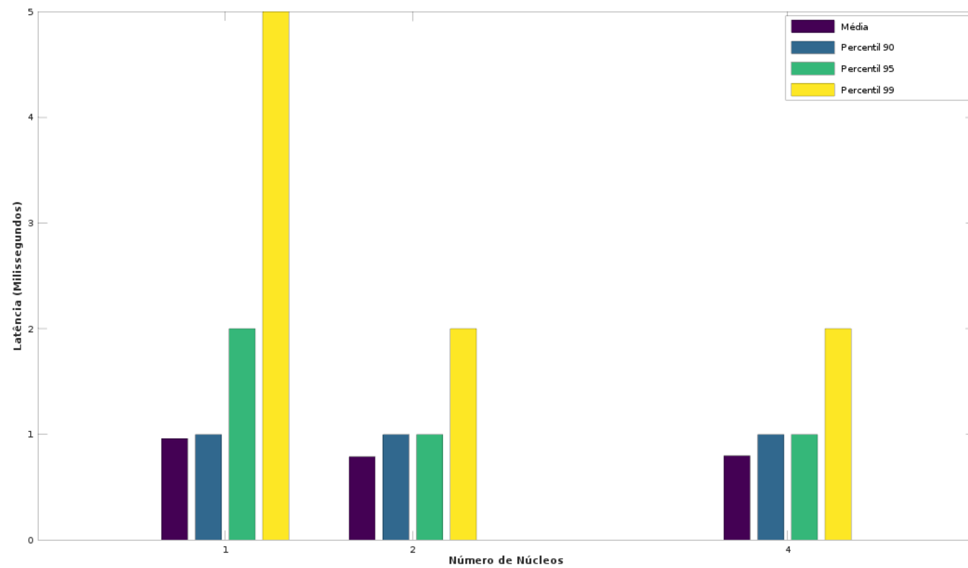


Figura 7 – Relação da latência de geração de identificadores e número de núcleos no processador.

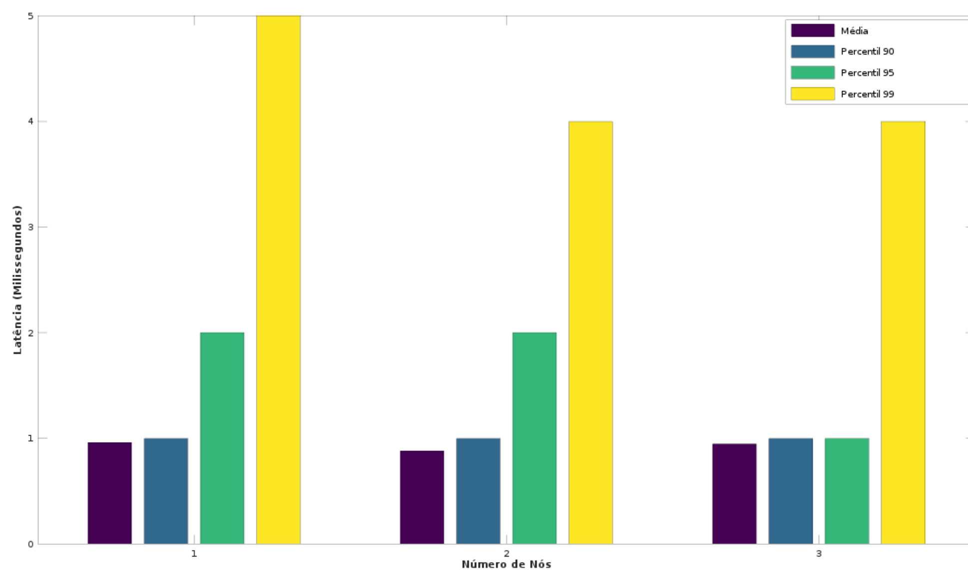


Figura 8 – Relação da latência de geração de identificadores e número de instâncias no *cluster*.

Observando ambos os gráficos podemos perceber que a latência não é afetada pelo número núcleos ou instâncias aumentando até as quantidades testadas.

# 6 Considerações Finais

Este capítulo discute as conclusões do trabalho desenvolvido, as dificuldades encontradas durante a sua realização, as limitações identificadas no sistema ao final do projeto e perspectivas de trabalhos futuros.

## 6.1 Conclusões

Com novas aplicações com um número grande de usuários surgindo a todo momento e a necessidade de identificar e referenciar entidades de forma única continuando sendo essencial para construir tais aplicações, surge a necessidade de uma forma de gerar os identificadores que seja capaz de suprir os requisitos impostos por estes novos sistemas emergentes.

Seguindo o roteiro proposto no Capítulo 1, os objetivos que foram listados foram alcançados. O levantamento dos requisitos foi realizado, foram estudadas as estruturas arquiteturais que serviriam melhor o tipo de sistema que foi desenvolvido, a implementação foi realizada seguindo a arquitetura para evitar um desalinhamento com as necessidades dos usuários, foram realizados testes de desempenho para verificar o comportamento do sistema em diferentes cenários e isso tudo foi feito utilizando os conhecimento obtido durante o curso de Ciência da Computação.

Uma dificultada que ocorreu durante o inicio deste projeto foi a aprendizagem das tecnologias ZooKeeper, Apache Curator e Netty. Isto foi um problema pois estas tecnologias não são tão utilizadas, como por exemplo *frameworks* Web geralmente são, o que faz com que materiais de pesquisa sejam mais escassos. Isso foi especial para o caso do *Apache Curator* para o qual, basicamente, o único recurso encontrado foi o site oficial. Mas estes problemas foram resolvidos por meio de pesquisas minuciosas feitas na Internet e da leitura de livros.

Após aprender a usar estas tecnologias, no entanto, elas permitiram o desenvolvimento de uma aplicação mais confiável, com um desempenho maior e isso tudo dentro de um tempo que seria impossível sem as mesmas.

Uma outra dificuldade foi que, inicialmente, se acreditava que seria possível atingir uma taxa de geração de identificadores por segundo e latência baixa usando o protocolo HTTP, mas após alguns testes foi possível perceber que para chegar às latências necessárias para este tipo de problema seria necessário usar um protocolo abaixo na camada de rede, o protocolo TCP. Isso fez com que fosse necessário reimplementar o módulo responsável por servir as requisições feitas pelos clientes da aplicação.

Porém, esta reimplementação se mostrou uma boa escolha pois, como pode-se notar nos resultados apresentados no Capítulo 5.2, foi possível alcançar latências menores que um milissegundo. Além disso foi possível aprender novas coisas ligadas a redes de computadores, desenvolvimento de protocolos, etc.

## 6.2 Limitações e Próximos Trabalhos

Ao final do ciclo de desenvolvimento que ocorreu durante a elaboração deste trabalho foram encontradas múltiplas limitações, que dão margem a mudanças futuras, podendo essas serem novas funcionalidades, correção de erros, etc.

São listadas a seguir algumas melhorias que poderiam ser feitas ao sistema desenvolvido a fim de suprir estas limitações observadas:

- Realizar mais testes a fim de aumentar a confiabilidade do sistema;
- Desenvolver ferramentas de linha de comando que facilitem no gerenciamento do sistema e permita automatização através de *scripts*;
- Suportar outros mecanismos de coordenação de sistemas distribuídos, como por exemplo etcd.

# Referências


BARCELLOS, M. P. *Notas de Aula: Engenharia de Software*. Vitória: [s.n.], 2018. Citado 2 vezes nas páginas 14 e 15.

BEZERRA, E. *Princípios de Análise e Projeto de Sistemas com UML*. [S.l.]: Elsevier, 2006. ISBN 8535216960. Citado na página 15.

BURNS, B. *Designing Distributed Systems*. [S.l.]: O'Reilly, 2018. ISBN 1491983647. Citado na página 18.

COULOURIS JEAN DOLLIMORE, T. K. e. G. B. G. *Sistemas Distribuídos: Conceitos e Projeto*. [S.l.]: Bookman, 2013. ISBN 8582600534. Citado na página 18.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao Teste de Software*. [S.l.]: Elsevier, 2007. ISBN 8535226346. Citado na página 17.

FALBO, R. de A. *Engenharia de Software - Notas de Aula*. Vitória: [s.n.], 2014. Citado 2 vezes nas páginas 16 e 17.

HIRAMA, K. *Engenharia de software: Qualidade e produtividade com tecnologia*. [S.l.]: GEN LTC, 2011. ISBN 853524882X. Citado 2 vezes nas páginas 14 e 16.

INSTAGRAMENGINEERING. *Sharding & IDs at Instagram*. 2019. <https://instagram-engineering.com/sharding-ids-at-instagram-1cf5a71e5a5c>. Acessado: 19/06/2019. Citado na página 11.

INTERNETWORLDSTATS. *The Internet Big Picture*. 2019. <https://www.internetworldstats.com/stats.htm>. Acessado: 29/05/2019. Citado na página 10.

PRESSMAN, R. *Engenharia de Software*. [S.l.]: AMGH, 2016. ISBN 8580555337. Citado 3 vezes nas páginas 15, 17 e 18.

SOMMERVILLE, I. *Engenharia de software*. [S.l.]: Addison Wesley, 2003. v. 6. ISBN 8543024978. Citado 5 vezes nas páginas 14, 15, 16, 17 e 26.

TANENBAUM, A. S. *Sistemas distribuídos: princípios e paradigmas, 2ed*. [S.l.]: Pearson, 2007. ISBN 8576051427. Citado na página 18.

TWITTERENGINEERING. *Announcing Snowflake*. 2010. <https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html>. Acessado: 2/12/2019. Citado na página 4.

ZEPHORIA. *The Top 20 Valuable Facebook Statistics – Updated May 2019*. 2019. <https://zephoria.com/top-15-valuable-facebook-statistics>. Acessado: 29/05/2019. Citado na página 10.

# Apêndices