BIGFS: UM SISTEMA DISTRIBUÍDO BASEADO EM CHUNKS COM TOLERÂNCIA A FALHAS E REPLICAÇÃO ATIVA

BIGFS: A CHUNK-BASED DISTRIBUTED SYSTEM WITH FAULT TOLERANCE AND ACTIVE REPLICATION

LUCAS VITOR DE SOUZA[1]; JUAN CARLOS VIEIRA DE QUEIROZ[2] 1;2
UNIVERSIDADE FEDERAL DE GOIÁS- GOIÂNIA-GO
lucasvitor@discente.ufg.br; juanqueiroz@discente.ufg.br

Resumo - BigFS é um sistema distribuído de armazenamento de arquivos, proposto como método avaliativo da disciplina de Sistemas Distribuídos, desenvolvido em Python e inspirado no HDFS. Ele utiliza o Pyro5 para comunicação entre cliente, NameNode e DataNodes, permitindo chamadas remotas a objetos. Os arquivos são divididos em chunks e replicados automaticamente em múltiplos nós, garantindo alta disponibilidade e tolerância a falhas. O NameNode gerencia os metadados e coordena operações como escrita, leitura, listagem e deleção. O sistema implementa verificação de integridade com checksum e leitura paralela para desempenho otimizado. É escalável horizontalmente e lida com falhas via heartbeat e reposição automática. Apesar de robusto, ainda carece de autenticação e controle de acesso.

Palavras-chave: Sistemas Distribuídos; Armazenamento em Chunks; Tolerância a Falhas; Replicação de Dados

Abstract – BigFS is a distributed file storage system, developed in Python and inspired by HDFS. It uses Pyro5 for communication between clients, NameNodes and DataNodes, allowing remote calls to objects. Files are divided into chunks and automatically replicated across multiple nodes, ensuring high availability and fault tolerance. The NameNode manages metadata and coordinates operations such as writing, reading, listing and deleting. The system implements integrity verification with checksum and parallel reading for optimized performance. It is horizontally scalable and handles failures via heartbeat and automatic replacement. Although robust, it still lacks authentication and access control.

Keywords: Distributed Systems; Chunk Storage; Fault Tolerance; Data Replication

1. INTRODUÇÃO

O crescimento exponencial do volume de dados digitais impulsionou o desenvolvimento de soluções distribuídas para armazenamento eficiente, tolerante a falhas e com alta disponibilidade. Neste contexto, o BigFS surge como uma proposta de sistema de arquivos distribuído desenvolvido em Python, inspirado no Hadoop Distributed File System (HDFS). O objetivo é proporcionar uma solução leve, escalável e de fácil implementação para cenários acadêmicos e experimentais.

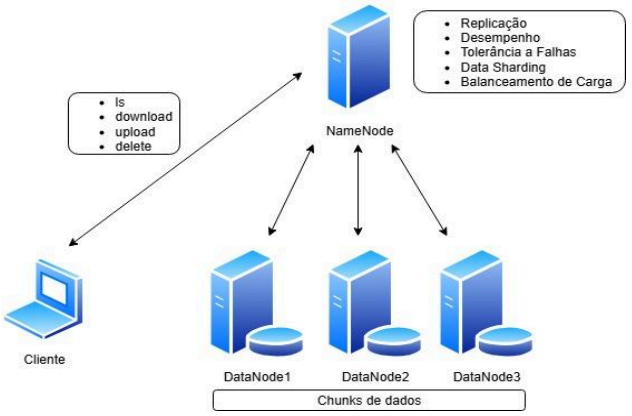
2. ARQUITETURA GERAL DO SISTEMA

O BigFS é composto por três principais componentes:

Cliente: Interface para envio e recebimento de arquivos. NameNode: Responsável pelos metadados, orquestração e verificação. DataNodes: Armazenamento físico dos chunks de arquivos.

A comunicação entre os componentes é realizada via Pyro5, permitindo chamadas remotas com objetos Python reais. A arquitetura se apoia na divisão dos arquivos em chunks de até 10MB, replicados entre diferentes DataNodes para garantir disponibilidade.

Figura 1 - Arquitetura Geral.



Fonte: Elaboração Própria.

3. COMUNICAÇÃO COM PYRO5

O sistema utiliza Pyro5 (Python Remote Objects) como protocolo de comunicação entre Cliente, NameNode e DataNodes. O Pyro5 permite chamada remota de métodos como se fossem locais, com suporte a objetos Python complexos e integração transparente entre múltiplos processos distribuídos.

Pyro5 oferece maior flexibilidade e robustez em comparação a soluções mais simples como XML-RPC, incluindo melhor desempenho, suporte a objetos reais (não apenas tipos básicos) e facilidade de implementação de servidores concorrentes.

Apesar da simplicidade no desenvolvimento, Pyro5 não oferece autenticação nem criptografia por padrão. Para produção, recomenda-se configurar canais seguros com SSL ou camadas adicionais de segurança.

4. ESTRUTURA DE METADADOS

Os metadados do sistema são armazenados diretamente no NameNode, em memória e com persistência em disco, com estrutura semelhante a:

```
metadados = {
  "relatorio.txt": {
    "relatorio.txt_chunk1": ["DN1", "DN2", "DN3"],
    "relatorio.txt_chunk2": ["DN1", "DN2", "DN4"]
  }
}
```

Essa estrutura baseada em dicionários Python permite acesso rápido e controle simples durante o ciclo de desenvolvimento. Os metadados são persistidos em disco em um arquivo do tipo JSON.

5. POLÍTICA DE FUNCIONAMENTO E DISTRIBUIÇÃO DE CHUNKS DO NAMENODE

O NameNode se insere no nameserver. Datanodes se inserem no nameserver e no namenode (namenode tem uma lista dos datanodes que foram registrados e quando foi o último heartbeat (a cada 3 segundos), essa lista é atualizada a cada heartbeat, se o tempo for maior que 10 minutos sem heartbeat, o datanode é considerado morto). Quando o cliente envia um arquivo ao NameNode e esse arquivo é direcionado aos DataNodes, faz-se a verificação se ele está ativo ou não.

No momento da escrita, o NameNode seleciona DataNodes baseando-se na disponibilidade e na carga (Awareness de espaço em disco) para armazenar os chunks.

Estratégia simples e eficaz para o escopo atual. Awareness de espaço em disco significa que o sistema conhece e considera a quantidade de espaço livre disponível em cada nó (DataNode) ao tomar decisões sobre onde armazenar novos dados.

6. FUNCIONALIDADES E OPERAÇÕES DO SISTEMA

6.1 Escrita de Arquivos

O cliente localiza o NameNode via NameServer. É solicitado a referência remota para escrita de arquivos. O arquivo é enviado em pedaços de 60 Kb(+ checksum) para o NameNode e esse arquivo é remontado no namenode e persistido em disco (temp_upload no NameNode). É feita a validação de integridade com checksum. O arquivo, já no NameNode, é particionado em chunks de 10Mb (e.g., relatorio.txt_chunk1, relatorio.txt_chunk2, ...), e enviado aos DataNodes. O chunk é armazenado no DataNode indicado. Após a transferência ao DataNode ser completada, o arquivo é excluído do disco do NameNode e o serviço de replicação faz as réplicas (fator 3).

Caso o tamanho do arquivo não exceda os 10Mb, o arquivo terá apenas um chunk, caso exceda, exemplo um arquivo de 100Mb ele terá 10 chunks, onde cada um será armazenado em um DataNode escolhido levando em consideração o awareness de disco (dos DataNodes Ativos) pelo NameNode.

O NameNode atualiza os metadados do arquivo, contendo:

```
relatorio.txt = {
  relatorio.txt_chunk1: [DN1, DN2, DN3],
  relatorio.txt_chunk2: [DN1, DN2, DN4]
}
```

6.2 Download de Arquivos

O cliente faz uma chamada remota da função de download com o nome do arquivo ao NameNode. O NameNode busca a lista de chunks do arquivo e para cada chunk, a lista de DataNodes que armazenam a cópia. O NameNode lê os chunks paralelamente dos primeiros DataNodes disponíveis. Em caso de falha, tenta o próximo DataNode da lista. O NameNode reagrupa os chunks e reconstrói o arquivo original persistindo o arquivo em disco no NameNode. Em seguida, o arquivo é enviado para o cliente em blocos de 60 Kb (+ checksum). Após envio concluído e feito

a validação do checksum, o arquivo é excluído do disco no NameNode.

6.3 Listagem e Deleção

Listagem: O cliente realiza uma chamada remota ao NameNode, utilizando o método listar_arquivos(). O NameNode acessa sua estrutura de metadados em memória, onde cada chave representa um arquivo armazenado no sistema. O NameNode retorna ao cliente a lista com os nomes dos arquivos disponíveis, exemplo; relatorio.txt

Deleção: O cliente solicita ao NameNode a exclusão de um arquivo. O NameNode consulta os metadados e identifica: Todos os chunks do arquivo; Todos os DataNodes que armazenam cada chunk.

O NameNode faz uma chamada remota da função de deleção aos respectivos DataNodes. Os DataNodes confirmam a remoção dos chunks. O NameNode apaga os metadados do arquivo. O cliente recebe a confirmação de deleção bem-sucedida. Em caso de falha de algum DataNode durante o processo, o sistema tentará novamente até que todos os chunks sejam efetivamente removidos.

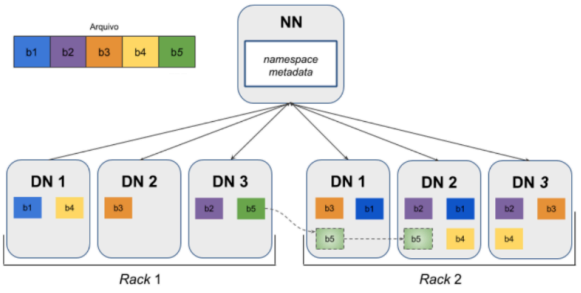
7. REQUISITOS NÃO FUNCIONAIS

7.1 Replicação Ativa

Protocolo de Verificação Contínua: Um serviço no NameNode verifica periodicamente (a cada poucos segundos) os metadados dos arquivos para garantir que cada chunk esteja replicado em pelo menos dois DataNodes.

Ação de Replicação: Se for detectado que um chunk tem menos de duas cópias, o serviço solicita ao NameNode novos DataNodes disponíveis e envia cópias adicionais do chunk.

Figura 2 - Replicação ativa.



Fonte: Elaboração Própria.

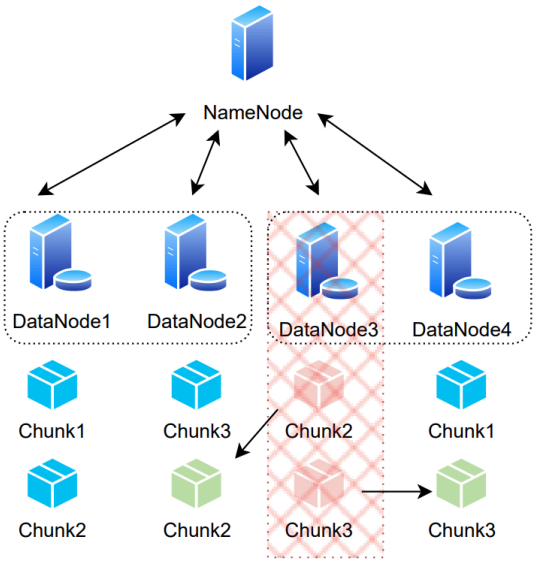
7.2 Tolerância a Falhas

Detecção de falhas via Heartbeat: Cada DataNode envia um heartbeat ao NameNode a cada 3 segundos. Se não houver resposta em até 10 minutos, o nó é considerado inativo.

Reposição Automática: Caso um DataNode falhe, os chunks que estavam sob sua responsabilidade são automaticamente replicados para novos DataNodes disponíveis.

Rebalanceamento Pós-Retorno: Quando um DataNode retorna, o sistema ajusta o número de cópias, removendo cópias excedentes para manter o fator de replicação fixo em 3.

Figura 3 - Tolerância a falhas.



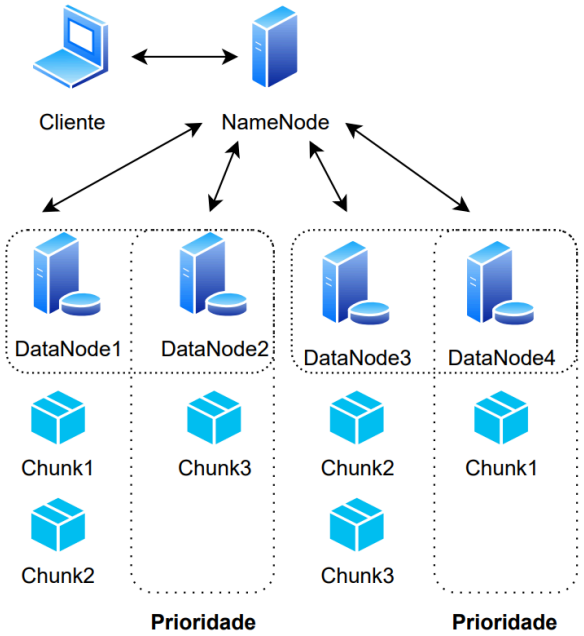
Fonte: Elaboração Própria.

Com isso, se um DataNode falhar, o mesmo serviço de monitoramento identificará a ausência e solicitará ao NameNode outro DataNode saudável para manter o fator de replicação. Quando o DataNode que falhou retornar, o sistema remove os chunk do DataNode que caiu a fim de garantir que não tenham arquivos não utilizados nele, garantindo também que o fator de replicação continue fixo em 3.

O protocolo de Verificação Contínua (PVC) é o serviço responsável por verificar os arquivos para garantir a replicação, ação de replicação, reposição automática e rebalanceamento pós retorno.

Uma forma que propomos para o balanceamento de carga foi colocar prioridade de escrita em DataNodes que possuem baixa utilização.

Figura 4 - Prioridade de Escrita.



Fonte: Elaboração Própria.

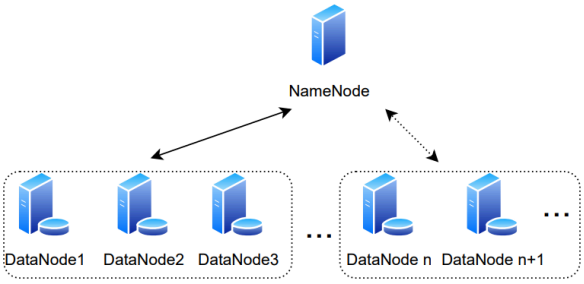
5.3 Escalabilidade e Concorrência

Com a Escalabilidade Horizontal é possível adicionar centenas ou milhares de DataNodes para suportar crescimento no volume de dados e acesso simultâneo sem reconfiguração manual do sistema.

Com múltiplos DataNodes e fator de replicação, as leituras de diferentes chunks são feitas simultaneamente, reduzindo a latência. Os servidores (NameNode e DataNodes) utilizam múltiplas threads para lidar com requisições simultâneas.

Ou seja, permitir leitura paralela é essencial para desempenho e escalabilidade de sistemas distribuídos.

Figura 4 - Escalabilidade.



Fonte: Elaboração própria.

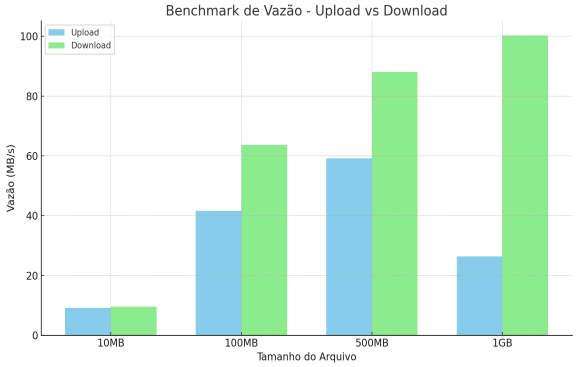
5.4 Verificação de Integridade

O sistema aplica checksum (SHA-256) tanto na escrita quanto na leitura, validando cada chunk transmitido e detectando falhas.

8. DESEMPENHO

Foram realizados testes de benchmark para avaliar o desempenho do BigFS em operações de upload e download com diferentes tamanhos de arquivos (10MB, 100MB, 500MB e 1GB). Os testes mediram o tempo de operação e a vazão (MB/s), considerando comunicação com múltiplos DataNodes e verificação de integridade com checksum.

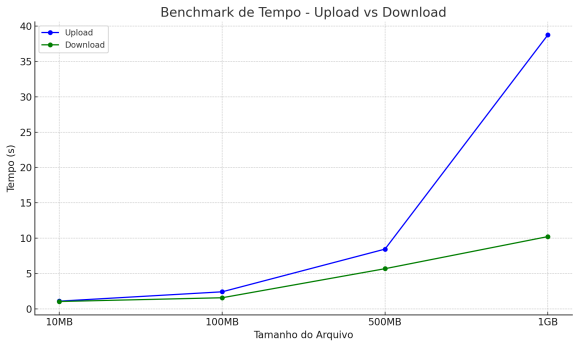
Figura 5 - Benchmark de Vazão.



Fonte: Elaboração Própria.

Resultados principais: A leitura (download) apresentou desempenho superior à escrita (upload) em arquivos grandes, beneficiando-se do paralelismo e da replicação. Arquivos de 1GB foram lidos a uma vazão de até 100 MB/s, enquanto uploads ficaram em torno de 26 MB/s para o mesmo tamanho.

Figura 5 - Benchmark de Tempo.



Fonte: Elaboração Própria.

Arquivos menores, como 10MB, mantiveram desempenho mais linear, com média de 9 MB/s para ambas operações. Esses resultados indicam que o sistema escala bem para operações de leitura, sendo adequado para workloads com alta demanda de acesso a dados.

9. CONCLUSÃO

O BigFS demonstra que é possível construir uma solução distribuída de armazenamento com tolerância a falhas, replicação ativa e integridade garantida, utilizando tecnologias acessíveis como Python e Pyro5. É uma plataforma ideal para fins educacionais, podendo ser expandida para incluir segurança, balanceamento de carga e autenticação.

10. REFERÊNCIAS

- SHVACHKO, Konstantin et al. The Hadoop Distributed File System. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2010. p. 1–10.

- TANENBAUM, Andrew S.; VAN STEEN, Maarten. Sistemas Distribuídos: Princípios e Paradigmas. 2. ed. São Paulo: Pearson Prentice Hall, 2007.

- OLIVEIRA, Alexandre; SILVA, João Paulo. Projetos de Sistemas Distribuídos com Python: comunicação remota e concorrência com Pyro. Revista Brasileira de Computação Aplicada, v. 11, n. 2, 2019.

- BISHOP, Matt. Introduction to Computer Security. Boston: Addison-Wesley, 2005.

- PYRO5 Documentation. Python Remote Objects Documentation. Disponível em: https://pyro5.readthedocs.io. Acesso em: 01 jul. 2025.

- FAZUL, Rhauani Weber Aita; CARDOSO, Paulo Vinicius; BARCELOS, Patrícia Pitthan. Análise do impacto da replicação de dados implementada pelo Apache Hadoop no balanceamento de carga. Santa Maria: Universidade Federal de Santa Maria.