Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Next Generation File System
Um framework para sistemas de arquivos distribuídos

Luis Augusto Toscano Guimarães

JUIZ DE FORA
NOVEMBRO, 2020

Next Generation File System
Um framework para sistemas de arquivos distribuídos

Luis Augusto Toscano Guimarães

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Orientador: Marcelo Ferreira Moreno

JUIZ DE FORA
NOVEMBRO, 2020

NEXT GENERATION FILE SYSTEM
Um framework para sistemas de arquivos distribuídos

Luis Augusto Toscano Guimarães

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Marcelo Ferreira Moreno
Doutorado em Informática/PUC-Rio, Brasil

Eduardo Pagani Julio
Doutorado em Computação/UFF, Brasil

Mario Antonio Ribeiro Dantas
Doutorado em Computer Science/University of Southampton, Inglaterra

JUIZ DE FORA
19 DE NOVEMBRO, 2020

Para a minha família e para todos que de alguma forma fazem parte da minha trajetória até aqui.

Resumo

O crescente volume de dados gerados e consumidos por aplicações distribuídas, que traz consigo um aumento dos requisitos de processamento e armazenamento, vem motivando avanços no desenvolvimento de sistemas de arquivos distribuídos.

Um sistema de arquivos distribuído possui as mesmas responsabilidades de um sistema de arquivos comum, porém os seus componentes estão distribuídos por uma rede de computadores. Desta forma, há outras características que um sistema deste tipo deve se preocupar, como os conceitos de transparência, desempenho, escalabilidade, entre outros.

Devido aos diferentes requisitos que as aplicações exigem, a criação de um único sistema de arquivos que atenda bem a todas as necessidades e casos de uso se torna difícil. Por isso, é fácil encontrar diversos trabalhos que focam em um determinado aspecto de interesse em um dado cenário, em detrimento de outros. Porém, apesar desta diferenciação entre os sistemas, é possível perceber semelhanças entre eles.

Desta forma, este trabalho propõe o desenvolvimento de um *framework* para sistemas de arquivos distribuídos, com o foco em flexibilidade, além de acelerar o desenvolvimento de sistemas deste tipo.

Palavras-chave: *framework*; sistemas distribuídos; sistema de arquivos.

Abstract

The growing volume of data generated and consumed by distributed applications, which also brings an increase on processing and storage requirements, has been motivating advances in the development of distributed file systems.

A distributed file system has the same responsibilities as a common file system, however its components are distributed over a computer network. Thus, there are other characteristics that such a system should be concerned with, such as the concepts of transparency, performance, scalability, among others.

Due to the different requirements that applications demand, creating a single file system that meets all necessities and use cases becomes difficult. In this way, it is easy to find several works that focus on a certain aspect of interest for given scenario, to the detriment of others. However, despite this differentiation between systems, it is possible to notice similarities among them.

Thus, this work proposes the development of a framework for distributed file systems, with a focus on flexibility, in addition to accelerating the development of systems of this type.

Keywords: framework; distributed systems; file system.

Agradecimentos

Agradeço, em especial, a minha mãe e a minha irmã por terem me dado os ensinamentos necessários para que eu conseguisse ser quem eu sou e chegar onde estou. Enquanto minha mãe cuidou de garantir todo o meu sustento, minha irmã cuidou de me incentivar com os estudos. Desta forma, as considero como duas grandes mulheres que conseguiram sozinhas me transformar em quem sou hoje. Infelizmente, meu pai faleceu quando eu ainda era criança, mas acredito que estaria orgulhoso se ainda estivesse por aqui.

Agradeço também aos amigos, por também serem fundamentais nesta trajetória de vida. As horas de bate-papo diário, os momentos de diversão, os momentos de conselhos e apoios em tomadas de decisão e todos os outros momentos envolvidos no que podemos definir simplesmente com a palavra "amizade".

Em especial, agradeço ao meu amigo Igor Pires por ter colaborado comigo desde o início do curso em vários trabalhos da graduação, sendo que muitos destes trabalhos pareciam "impossíveis" de serem realizados durante o tempo disponível da disciplina, mas que ao final sempre conseguimos concluir com resultados satisfatórios graças a essa grande parceria. Inclusive, o interesse no tema deste trabalho teve origem em uma destas colaborações, em que propus a ele o desenvolvimento de um sistema de arquivos em um trabalho para a disciplina de Sistemas Distribuídos e ele aceitou. Assim, tive contato com o tema ao qual mantenho grande interesse até hoje.

Outra pessoa que tem grande importância para este trabalho e também para a minha vida é a minha amiga de longa data, Mariana Sanguinete. Apesar da computação não ser a sua área de formação, sendo ela engenheira agrônoma, considero que possui um grande papel neste trabalho, afinal, cansou de estar ao meu lado me apoiando em decisões e escutando minhas lamentações quando as coisas não iam bem. Este apoio emocional é fundamental para que todo o resto funcione bem e, desta forma, termos a tranquilidade

para pensar e desenvolver o que queremos.

Não posso esquecer dos professores que fazem parte da minha construção e identidade, como minha irmã gosta de falar. Os seus ensinamentos são essenciais para que as portas do conhecimento sejam abertas e, desta forma, novas oportunidades de caminho a serem trilhados surjam para você. Considero que é somente através da educação que podemos tornar o mundo algo melhor.

Em especial, agradeço ao professor Eduardo Pagani por ter confiado em mim para diversos projetos que realizei durante a graduação, ao professor Rafael Bonfim por toda a dedicação e paciência em minha orientação no meu trabalho de conclusão de curso do bacharelado em Ciências Exatas e ao professor Marcelo Moreno por enxergar potencial em mim, fornecendo o suporte necessário para que eu consiga ir sempre além, também por toda a dedicação e paciência em minha orientação neste trabalho de conclusão de curso.

Por fim, agradeço a todas as pessoas que talvez nem saibam da importância que possuem, mas que para toda esta instituição chamada "Universidade Federal de Juiz de Fora" funcione, são essenciais.

O conjunto de todo o apoio e de todas essas pessoas que fazem parte da sua vida é que formam a base essencial para que grandes resultados sejam obtidos. Desta forma, eu deixo aqui o meu muito obrigado a todos que fazem parte da minha vida e que me apoiam para que eu consiga alcançar os meus objetivos.

"Ao infinito e além!"

Buzz Lightyear (Toy Story)

Conteúdo

Lista de Figuras **9**

Lista de Tabelas **10**

Lista de Abreviações **11**

1 Introdução **12**
1.1 Apresentação do tema . . . 12
1.2 Problema . . . 15
1.3 Proposta e Justificativas . . . 15
1.4 Objetivos . . . 16
1.5 Estrutura do trabalho . . . 16

2 Referencial Bibliográfico **17**
2.1 Framework . . . 17
2.2 Sistemas de arquivos . . . 19
2.3 Padrões de projeto . . . 24
2.4 Redes ad hoc e comunicação oportunista . . . 25

3 Framework Next Generation File System **27**
3.1 Arquitetura . . . 29
3.2 Base . . . 31
3.2.1 Mecanismo de configuração . . . 32
3.2.2 Mecanismo de *logging* . . . 33
3.3 Utilitários . . . 34
3.3.1 Comunicação em rede . . . 36
3.3.2 Eventos . . . 39
3.3.3 Segurança . . . 40
3.4 Componentes . . . 43
3.4.1 Componentes estáticos . . . 47
3.4.1.1 Carregadores de componentes . . . 48
3.4.1.2 Comunicação entre componentes . . . 50
3.4.1.3 Canais de comunicação . . . 51
3.4.2 Componentes Dinâmicos . . . 53
3.4.2.1 Service . . . 56
3.4.2.2 Proxy . . . 58
3.4.2.3 Manager . . . 62
3.4.2.4 Filesystem . . . 63
3.5 Aplicações . . . 64
3.5.1 Servidores . . . 65
3.5.2 Console . . . 66

4 Provas de conceito **67**
4.1 Storage filesystem . . . 67
4.2 Music filesystem . . . 71

4.3 Experimentos 75

5 Considerações Finais **80**
5.1 Conclusões 80
5.2 Trabalhos futuros 81

Bibliografia **83**

Lista de Figuras

1.1 Exemplo de árvore de diretórios e arquivos. 13

2.1 Diferença do fluxo de controle entre *frameworks* e bibliotecas de classes . . 17
2.2 Processo de desenvolvimento de um *framework*. 18
2.3 Comparativo entre diferentes sistemas de arquivos distribuídos. 23

3.1 Visão simplificada das áreas funcionais que compõem o *framework* NGFS. . 27
3.2 Relacionamento em camadas dos componentes de Serviços, *Proxies*, Gerentes, Sistema de Arquivos e Aplicação. 29
3.3 Principais interfaces do *framework*. 30
3.4 Principais métodos da interface *"Base"*. 31
3.5 Exemplo do arquivo de configuração "Ngfs.cfg". 33
3.6 Exemplo de arquivo global de configuração do sistema de *logging*. 34
3.7 Exemplo de arquivo de configuração do sistema de *logging* da aplicação "NgfsServer". 34
3.8 Visão geral sobre as bibliotecas POCO. 36
3.9 Interfaces para a implementação de sockets TCP e UDP. 38
3.10 Interfaces para a implementação de servidores de rede. 39
3.11 Relacionamento entre interfaces e classes do mecanismo de eventos. 41
3.12 Interfaces que possibilitam o desenvolvimento de componentes para o *framework* NGFS. 45
3.13 Interfaces IComponent e ILoadableComponent que possibilitam a implementação dos componentes estáticos e dinâmicos do *framework* NGFS. . . 47
3.14 Interfaces "ILoader", "ICommunication" e "IChannel" e especializações já presentes no *framework* NGFS. 48
3.15 Definição da classe "TimeService", um serviço tempo do *framework* NGFS. 50
3.16 Interface "ICommunication" do *framework* NGFS. 51
3.17 Interfaces disponíveis no framework NGFS para a especialização de componentes dinâmicos. 55
3.18 Interfaces alternativas para a implementação de serviços de tempo usando o protocolo TCP ou UDP. 58

4.1 Arquitetura do sistema de arquivos GFS. 68
4.2 Principais componentes utilizados no *Storage Filesystem*. Os componentes com a palavra "*Default*" no nome são componentes já presentes no *framework* NGFS, já os que contém a palavra "*Storage*" indicam componentes que foram especializados para este sistema de arquivos distribuído. . 70
4.3 Arquitetura do DroidPlayer. 73
4.4 Interface gráfica do DroidPlayer. 73
4.5 Principais componentes utilizados no *Music Filesystem*. 74
4.6 Tempo total da operação de leitura no *Storage Filesystem*. 78
4.7 Tempo total da operação de escrita no *Storage Filesystem*. 78
4.8 Tempo total para a replicação de um arquivo em todos os nós do *Music Filesystem*. 79

Lista de Tabelas

4.1 Configuração das máquinas virtuais utilizadas nos testes das provas de conceito. 76
4.2 Configuração do ambiente de rede para o cenário do *Storage Filesystem*. . . 76
4.3 Configuração do ambiente de rede para o cenário do *Music Filesystem*. . . 76

Lista de Abreviações

UFJF	Universidade Federal de Juiz de Fora
DCC	Departamento de Ciência da Computação
NGFS	Next Generation File System
NTFS	New Technology File System
EXT4	Fourth Extended File System
NFS	Network File System
GFS	Google File System
KFS	Kosmos File System
PVFS	Parallel Virtual File System
RDFS	Red Hat Global File System
HDFS	Hadoop Distributed File System
API	Application Programming Interface
IP	Internet Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
SSL	Secure Sockets Layer
AES	Advanced Encryption Standard
RSA	Rivest-Shamir-Adleman
SHA	Secure Hash Algorithm
IoT	Internet of Things

1 Introdução

1.1 Apresentação do tema

As aplicações que estão em execução, ou seja, os processos que estão em execução em um sistema operacional, estão constantemente manipulando informações.

O armazenamento destas informações pode ser realizado, por exemplo, no espaço de endereçamento do processo na memória principal do computador. Porém, além deste espaço ser limitado, quando o processo fosse encerrado, toda a informação seria perdida, já que se trata de um tipo de armazenamento temporário.

Além disso, também é comum que seja necessário o compartilhamento de informações entre as aplicações, mas, novamente, usar apenas a memória principal não é o suficiente para atender a todos os casos, pois este endereçamento é privado e não pode ser compartilhado com outros processos.

Como esta restrição não é adequada para diversos tipos de aplicação, algo a mais é necessário para o armazenamento de informações de longo prazo, em memória secundária. Segundo TANENBAUM e BOS (2015), existem três requisitos para o armazenamento de informações em memória secundária, sendo eles:

- Deve ser possível armazenar uma grande quantidade de informação;
- A informação deve ser mantida mesmo após o término do processo;
- Múltiplos processos devem ser capazes de acessar uma informação ao mesmo tempo.

Antes de compreender o que é um sistema de arquivos distribuído, é necessário entender o que são os arquivos. Assim como outros tipos de abstrações que um sistema operacional faz uso, o arquivo é uma abstração utilizada para representar toda a informação de longo prazo criada por um processo.

Os arquivos podem ser de diferentes tipos como arquivos binários, de texto, entre outros. São normalmente identificados por um nome e uma extensão, sendo a extensão útil para a identificação de qual aplicação consegue manipular o arquivo em questão.

É comum também que possuam atributos que identificam o seu proprietário, marcações de tempo da criação, da última modificação e do último acesso, informações sobre localização, tamanho, *flags* de apenas leitura, temporário, entre outros vários atributos que acabam formando metadados sobre os dados que o arquivo contêm.

Um sistema de arquivos é o sistema responsável por todo o gerenciamento destes arquivos. Desta forma, ele deve prover mecanismos que possibilitem a criação, a leitura, a escrita, a remoção, além de muitas outras operações que estão disponíveis neste tipo de sistema.

Além disso, também é responsabilidade do sistema de arquivos a definição de uma estrutura de organização e localização dos arquivos. Por exemplo, pode-se ter uma estrutura hierárquica de diretórios contendo arquivos. Se esta hierarquia assumir uma estrutura árvore, pode-se pensar que o caminho de um arquivo pode ser dado pelo caminho realizado a partir da raiz da árvore até um determinado nó dela.

Figura 1.1: Exemplo de árvore de diretórios e arquivos.

Pode-se pensar ainda em atribuir um nome a cada elemento deste caminho, para facilitar o trabalho que humanos teriam ter para lembrar a localização de um determinado arquivo. Usando como exemplo o caminho de um arquivo armazenado no sistema de arquivos de um sistema operacional *Windows*, tem-se: "D:\Documentos\UFJF\Graduação

\TCC\Ciência da Computação\Monografia.pdf". Neste exemplo, "D:" está referenciando um dispositivo de armazenamento ao qual o arquivo está armazenado, enquanto "Documentos\UFJF\Graduação\TCC\Ciência da Computação" são todos diretórios que fazem parte do caminho até o arquivo "Monografia.pdf". Todos estes elementos possuem seus próprios atributos que formam metadados sobre os dados armazenados.

Já em um sistema de arquivos distribuído, não há exatamente as mesmas responsabilidades de um sistema de arquivos comum. Os seus componentes estão distribuídos por uma rede de computadores. Desta forma, há outras características que um sistema deste tipo também deve se preocupar como os conceitos de transparência, desempenho, escalabilidade, controle de concorrência, tolerância a falhas, entre outros que são pertinentes aos sistemas distribuídos.

A motivação para a criação de sistemas deste tipo decorre do volume crescente dos dados, fazendo com que os requisitos de processamento e armazenamento também cresçam. Sendo assim, com um sistema distribuído, há o compartilhamento de recursos computacionais que estão disponíveis nos vários nós que fazem parte do sistema, facilitando assim o atendimento a esta crescente demanda.

Outro fator que estimula a criação deste tipo de sistema é o fato de que um sistema centralizado está mais suscetível a falhas ou sobrecargas que geram interrupção no fornecimento do serviço. Isto é diferente do que pode ocorrer em um sistema distribuído que, dependendo do tipo de falha, somente parte dos serviços será afetada ou nem mesmo isto, já que é possível fazer uso de diferentes tipos de replicação (TANENBAUM; STEEN, 2007) para que o sistema se torne mais tolerante a falhas.

São exemplos de sistemas de arquivos o NTFS (MICROSOFT, 2009) em sistemas operacionais *Windows* e o EXT4 (MATHUR et al., 2007) em sistemas operacionais *Linux*. Já o NFS (SANDBERG et al., 1985), o Coda (SATYANARAYANAN et al., 1989), o GFS (GHEMAWAT; GOBIOFF; LEUNG, 2003) e o HDFS (BORTHAKUR, 2007) são alguns exemplos de sistemas de arquivos distribuídos.

1.2 Problema

Devido aos diferentes requisitos dos diversos tipos de aplicações existentes, a criação de um único sistema de arquivos que atenda bem a todos estes requisitos se torna difícil, sendo que o mesmo vale para os sistemas de arquivos distribuídos.

Desta forma, é fácil encontrar diversos trabalhos focados em um determinado aspecto como o desempenho, a escalabilidade, a tolerância a falhas e a segurança em detrimento de outros. Porém, apesar desta diferenciação entre os sistemas, é possível perceber um núcleo comum entre eles, já que apesar do foco de cada um incidir sobre certos aspectos, ainda é necessário que outras funcionalidades comuns sejam atendidas.

Sendo assim, surge a questão: Não seria interessante a construção de um *framework* para sistemas de arquivos distribuídos que reúna estes mecanismos comuns, mas que forneça, a partir da especialização, a possibilidade de se atender aos diversos cenários de uso?

1.3 Proposta e Justificativas

Tendo, na seção anterior, introduzido o tema de sistemas de arquivos distribuídos e identificado o problema, este trabalho propõe a criação de um *framework* para sistemas de arquivos distribuídos, denominado *Next Generation File System*, tendo como foco a alta flexibilidade, tornando-o adaptável para diferentes cenários de uso.

Desta forma, torna-se possível reduzir o tempo de desenvolvimento deste tipo de sistema, já que os desenvolvedores poderão se concentrar na elaboração de novos recursos ao invés de sempre "reinventarem a roda". Com menos código novo sendo necessário para o desenvolvimento de um sistema de arquivos distribuído, espera-se uma diminuição da manutenção de código, novamente liberando o desenvolvedor para outras tarefas mais importantes. Por fim, pode-se ter a melhoria da qualidade de código destes sistemas, uma vez que a intenção do *framework* é a reutilização de componentes, tendo um código que será mais testado e com o tempo tenderá a uma estabilidade, podendo se tornar um padrão. É importante mencionar que os sistemas desenvolvidos usando o *framework* serão compatíveis entre si, já que utilizarão as mesmas abstrações e interfaces nele definidas.

Com isso, pode-se pensar que este *framework* trará benefícios para a área de sistemas de arquivos distribuídos, tornando-a mais ágil em desenvolvimento e, desta forma, será possível uma maior quantidade de pesquisas envolvendo o tema, além de uma maior equidade nos testes comparativos entre estes trabalhos. Afinal, se toda a implementação de sistema de arquivos distribuído for feita do zero, pode-se ter mais do que um fator influenciando determinada observação quando comparando este sistema a outro que também foi implementado do zero e de maneira totalmente diferente do primeiro.

1.4 Objetivos

O objetivo principal deste trabalho é a construção de um *framework* altamente flexível, permitindo a sua utilização para a criação de sistemas de arquivos distribuídos para diversos cenários de uso, promovendo uma redução no tempo de desenvolvimento de sistemas deste tipo. Como objetivo secundário, a fim de se testar o uso do *framework*, este foi especializado para diferentes cenários de uso. Para que estes objetivos sejam alcançados, se faz necessário:

- Determinar os requisitos comuns e novos de sistemas de arquivos distribuídos;
- Determinar os mecanismos de extensibilidade do *framework*;
- Implementar a base fixa do *framework* (*Frozenspots*);
- Implementar algumas instâncias para as partes flexíveis do *framework* (*Hotspots*);
- Definir os cenários de uso para prova de conceito do *framework*;
- Especializar o *framework* para implementação da prova de conceito;
- Analisar a prova de conceito nos cenários de uso.

1.5 Estrutura do trabalho

A estrutura deste trabalho é composta pelas referências bibliográficas (Capítulo 2), pela descrição sobre o *framework* desenvolvido (Capítulo 3), pelas provas de conceito (Capítulo 4) e, por fim, pelas considerações finais sobre o trabalho (Capítulo 5).

2 Referencial Bibliográfico

2.1 Framework

Segundo FAYAD e SCHMIDT (1997), um *framework* é um conjunto de classes que incorpora um design abstrato para soluções de um domínio de software, ou seja, é um software "semi-completo" que pode ser especializado para a construção de softwares personalizados. Portanto, um *framework* é reutilizável.

Outro ponto a ser destacado sobre os *frameworks* é a diferença em relação às bibliotecas de classes. Enquanto nas bibliotecas de classe a arquitetura do software final e o seu fluxo de execução são determinados pelo próprio software que faz uso da biblioteca, são os *frameworks* que determinam como será a arquitetura e o fluxo de execução do software construído sobre o *framework*, sendo isto chamado de inversão de controle.

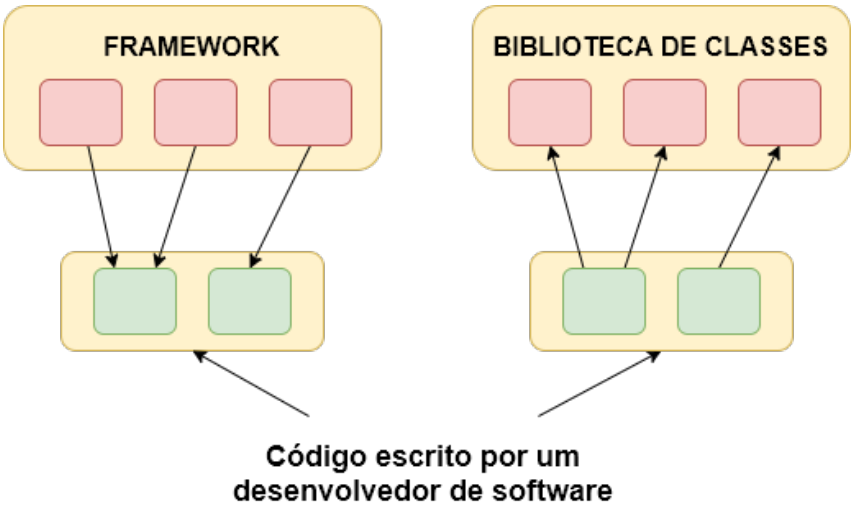

Figura 2.1: Diferença do fluxo de controle entre *frameworks* e bibliotecas de classes

Um *framework* deve fornecer interfaces estáveis e definir componentes genéricos que servirão para a criação dos componentes para o software final. Também deve fornecer métodos explícitos que permitam a extensão das interfaces estáveis (Hotspots).

Além da extensibilidade garantir a personalização do *framework*, ela também separa a parte estável de um domínio de aplicação das variações exigidas pelas instanciações

de um contexto particular (FAYAD; SCHMIDT, 1997).

Por fim, a reutilização de código provida pelo *framework* traz melhorias de qualidade, desempenho, confiabilidade e interoperabilidade de software, além de trazer considerável melhoria na produtividade do desenvolvedor do software (FAYAD; SCHMIDT, 1997).

Segundo LANDIN e NIKLASSON (1998), as dificuldades encontradas no desenvolvimento de um *framework* são relacionadas à dificuldade em se projetar a arquitetura e componentes com a reutilização em foco.

Desta forma, o projeto leva mais tempo do que aquele sem qualquer pensamento em reutilização. Para que o desenvolvimento do *framework* seja bem sucedido, é necessário que haja um processo organizado de desenvolvimento e também tempo de amadurecimento do *framework*.

Outro ponto que não deve ser desconsiderado durante o desenvolvimento é a documentação do *framework*, segundo BOOCH (1994), "O *framework* mais profundamente elegante nunca será reutilizado, a menos que o custo de compreendê-lo e usar suas abstrações seja menor do que o custo percebido pelo programador de escrevê-lo a partir do zero". Assim, a documentação deverá descrever detalhadamente o propósito, o design e como fazer uso do framework.

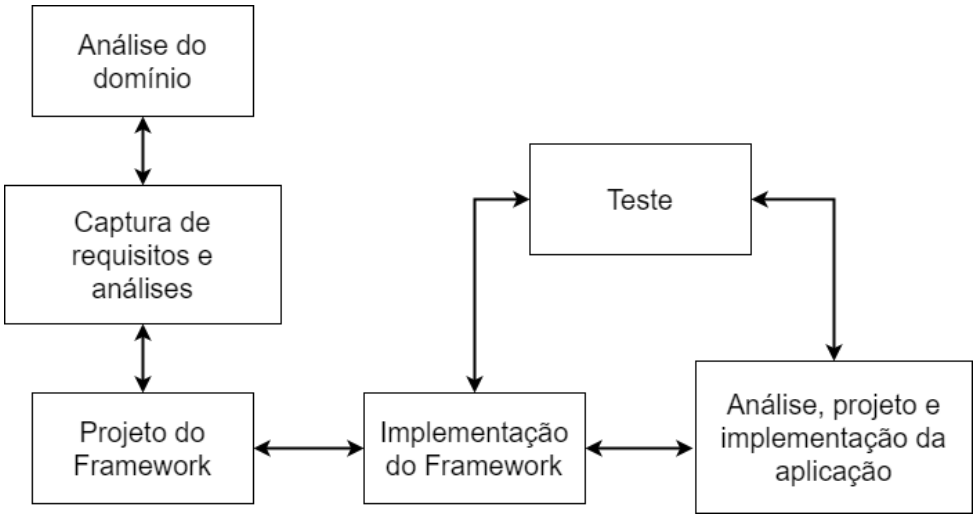

Figura 2.2: Processo de desenvolvimento de um *framework*.

A Figura 2.2 mostra quais devem ser as etapas do processo de desenvolvimento de um *framework*, sendo elas: Análise de domínio, captura de requisitos e análises, projeto

do *framework*, implementação do *framework*, implementação de uma aplicação e teste.

A etapa de análise de domínio ocorre como qualquer processo de desenvolvimento de software, ou seja, o domínio do software deverá ser estudado e entendido para o levantamento de requisitos. Justamente, esta é a próxima seção deste trabalho.

Tendo definidos os requisitos que devem ser atendidos, deve-se projetar o *framework* como uma base sólida para as aplicações que serão construídas por cima, mas que ainda permita uma boa flexibilidade de implementação. Entretanto, deve-se tomar o cuidado para que a hierarquia construída não seja tão flexível que crie problemas de controle. Também é interessante nesta fase, que sejam usados padrões de projetos que definirão como o *framework* será utilizado.

Na fase de implementação do *framework*, a preocupação deve ser a de se criar um código que seja fácil de entender e com uma boa estrutura, facilitando a documentação do mesmo.

Na etapa de validação e teste, ou seja, quando deve-se implementar uma aplicação que faça uso do *framework* e testá-la, ocorrendo da mesma maneira que o processo de teste de software.

Deve-se destacar que a validação do *framework* é baseada na reutilização, sendo que um *framework* reutilizado não precisa ser novamente testado depois de já ter sido testado uma vez. Desta forma, é esperado que um *framework* maduro tenha a necessidade de testes reduzida, uma vez que já devem existir diversas classes concretas reutilizadas e testadas.

2.2 Sistemas de arquivos

Como este trabalho se propõe à criação de um *framework* para sistemas de arquivos distribuídos, o primeiro passo a ser tomado no desenvolvimento é a análise de domínio e definição de requisitos deste domínio de software, como relatado anteriormente.

É possível observar em SATYANARAYANAN (1990) que já havia a preocupação na década de 90 com o crescimento do volume de dados armazenados e também com a necessidade destes dados estarem disponíveis em outro local além do armazenado. Um apanhado histórico mostra a evolução dos problemas encontrados em sistemas de arquivos

para computadores pessoais, tendo começado em 1983 com o IBM PC-DOS e em 1985 com o Apple Macintosh, nos quais havia um único usuário com um único processo, assim os sistemas de arquivos deveriam se preocupar apenas com a estrutura de nomes, o fornecimento de uma API, uma abstração para os dados armazenados no disco (Arquivo) e a garantia de integridade dos dados em caso de falha de hardware, software ou energia.

O próximo passo foi o OS/2 em 1988, com um usuário e múltiplos processos, surgindo então o problema de controle de concorrência para os sistemas de arquivos. Quando surgiu a possibilidade de múltiplos usuários, surgiu também a necessidade de segurança dos dados armazenados.

Por fim, quando os usuários eram múltiplos e se deslocavam fisicamente por uma rede de computadores, surgiram os sistemas de arquivos distribuídos, que permitiam o acesso independente do local aonde os arquivos estivessem armazenados. Assim, a eficiência, segurança e robustez se tornaram requisitos dos sistemas de arquivos. Além disso, surgiram também problemas relacionados à forma de localização e disponibilidade dos arquivos.

O estudo feito por AGRAWAL et al. (2007) entre 2000 e 2004 com 60 mil computadores corporativos com sistema operacional Windows, reuniu dados sobre como sistemas de arquivos se comportam ao longo do tempo.

Os dados coletados se referem ao tamanho, quantidade, tipos e grau de modificação de arquivos, além do tamanho, quantidade e hierarquia de diretórios e, por fim, a capacidade de armazenamento e total ocupado desse armazenamento.

Foi observado que a quantidade de arquivos cresceu consideravelmente ao longo dos anos e que a modificação de arquivos caiu. O tamanho médio dos arquivos começou a sofrer alteração ao longo dos anos devido ao armazenamento de conteúdo multimídia, como vídeos e áudio. A idade média dos arquivos ficou entre 80 e 160 dias. Em relação aos diretórios, houve um crescimento constante ao longo dos anos, 23 a 25% dos diretórios não continham arquivos, sendo 2 a média de arquivos por diretório e 65 a 67% dos diretórios não continham subdiretórios. Por fim, o crescimento da capacidade de armazenamento cresceu consideravelmente.

Em 2008, THANH et al. (2008) fizeram uma pesquisa sobre as características de

arquitetura, de estado dos processos, do tipo de comunicação, do sistema de nomeação, de sincronização, de consistência e replicação, de tolerância a falhas e de segurança dos seguintes sistemas de arquivos distribuídos: Google File System (GHEMAWAT; GOBIOFF; LEUNG, 2003), Lustre (SCHWAN, 2003), Kosmos File System (OVSIANNIKOV et al., 2013), Hadoop Distributed File System (BORTHAKUR, 2007), Panasas (NAGLE; SERENYI; MATTHEWS, 2004), Parallel Virtual File System (CARNS et al., 2000) e Redhat Global File System (SOLTIS; OKEEFE, 1996).

Segundo THANH et al. (2008), os tipos de arquitetura possíveis são:

- **Cliente-Servidor:** Um exemplo deste tipo de arquitetura é o NFS, no qual há vários servidores de arquivos pela rede e os clientes devem fazer as requisições a eles para obter uma resposta para determinada operação;
- **Baseada em cluster:** Um exemplo deste tipo é o GFS, no qual um único servidor mestre gerencia a operação de vários servidores (Chunk servers);
- **Simétrica:** Nesta arquitetura baseada em tecnologia *peer-to-peer*, todos os nós possuem as mesmas responsabilidades, o que resulta que todos conhecem a estrutura dos dados;
- **Assimétrica:** Em oposição à arquitetura simétrica, na asssimétrica tem-se nós que desempenham funções diferentes, por exemplo, nós que gerenciam os metadados e aqueles que armazenam os dados;
- **Paralela:** Nesta arquitetura, os dados são divididos em blocos e armazenados em diferentes servidores, para que seja possível a leitura/escrita paralela, melhorando o desempenho destas operações.

Em relação aos processos de um sistema de arquivos distribuído, eles podem ser com ou sem estado. A abordagem sem estado tem como vantagem a simplicidade e também a retomada de nós com falhas sem causar perturbações no sistema como um todo. Porém, cria problemas relacionados à concorrência de acesso a um mesmo dado, já que não há informação se existem clientes fazendo acesso a um arquivo. Portanto, pode haver

problemas de consistência dos dados. A maioria dos sistemas de arquivos distribuídos adota a abordagem com estado.

Sobre a comunicação entre nós de um sistema de arquivos distribuído, a maioria deles usa mecanismos de chamada remota de procedimento (RPC) sobre o TCP ou UDP. O TCP é usado na maioria dos casos e o UDP é utilizado quando se deseja melhorar a performance. Um caso especial é observado no Lustre, que independe da rede, pois fornece uma camada de abstração que permite ser usado em uma variedade de tipos de redes.

O sistema de nomes pode ser centralizado em um servidor de metadados, o que facilita no tratamento de problemas de sincronização. Alternativamente, pode estar distribuído por todos os servidores, resultando que todos os nós conheçam a estrutura dos dados armazenados.

A necessidade de mecanismos de sincronização surge quando há mais de um usuário acessando o mesmo arquivo. Neste caso, deve haver controle sobre as operações de leitura/escrita de arquivos. Para isso, deve ser definido o modelo de acesso que o sistema de arquivos adotará e qual será o mecanismo de controle utilizado. Vários sistemas de arquivos distribuídos utilizam o modelo de acesso *Write-once-read-many* e o mecanismo de *lock* para controle.

O mecanismo de *cache* melhora o desempenho do sistema de arquivos, além de reduzir o consumo de banda de rede, porém, pode trazer problemas de inconsistência de dados. Portanto, deve haver mecanismos que verifiquem a consistência dos dados.

Já o mecanismo de replicação, além de trazer maior tolerância a falhas, aumentando a disponibilidade do sistema, também pode ser usada para fornecer as operações paralelas de leitura/escrita, melhorando ainda o desempenho. Novamente, mecanismos de verificação de consistência, neste caso das réplicas, são necessários.

As falhas em um sistema de arquivos distribuído podem ser tratadas como uma exceção, ou seja, o sistema deverá isolar o nó com falha e recuperá-lo ao estado normal. Alternativamente, as falhas podem ser tratadas como norma, sendo que neste caso, a replicação é a ferramenta utilizada para solucionar as falhas que ocorrerem, assim, nós essenciais e os dados são replicados.

Por fim, um sistema de arquivos distribuído pode adotar mecanismos de segurança, para garantir a confidencialidade, integridade e disponibilidade dos dados.

A Figura 2.3 resume as características presentes nos sistemas de arquivos distribuídos analisados por THANH et al. (2008).

Sistema de arquivos	GFS	KFS	HDFS	Lustre	Panasas	PVFS2	RGFS
Arquitetura	Cluster, assimétrica, paralela	Cluster, assimétrica, paralela	Cluster, assimétrica, paralela	Cluster, assimétrica, paralela	Cluster, assimétrica, paralela	Cluster, simétrica, paralela	Cluster, simétrica, paralela
Processos	Com estado	Com estado	Com estado	Com estado	Com estado	Sem estado	Com estado
Comunicação	RPC/TCP	RPC/TCP	RPC/TCP/UDP	Camada de abstração	RPC/TCP	RPC/TCP	RPC/TCP
Nomeação	Servidor central de metadados	Servidor central de metadados	Servidor central de metadados	Servidor central de metadados	Servidor central de metadados	Metadados distribuídos em todos os nós	Metadados distribuídos em todos os nós
Sincronização	Write-once-read-many, múltiplos produtores/um consumidor, possui mecanismo de lock	Write-once-read-many, possui mecanismo de lock	Write-once-read-many, possui mecanismo de lock	Possui mecanismo de lock	Possui mecanismo de lock	Não possui mecanismo de lock	Possui mecanismo de lock
Consistência e Replicação	Replicação assíncrona, uso de checksum, critérios flexíveis sobre a consistência dos dados replicados	Replicação assíncrona, uso de checksum para validação de consistência	Replicação assíncrona, uso de checksum para validação de consistência	Replicação somente de metadados e cache no lado cliente, uso para validação de consistência	Replicação somente de metadados	Sem replicação, critérios de consistência flexíveis	Sem replicação
Tolerância a falhas	Falhas são a norma	Falhas são a norma	Falhas são a norma	Falhas são exceção	Falhas são exceção	Falhas são exceção	Falhas são exceção
Segurança	Não possui mecanismo dedicado à segurança	Não possui mecanismo dedicado à segurança	Não possui mecanismo dedicado à segurança	Autenticação, autorização e privacidade	Autenticação, autorização e privacidade	Autenticação, autorização e privacidade	Autenticação, autorização e privacidade

Figura 2.3: Comparativo entre diferentes sistemas de arquivos distribuídos.

2.3 Padrões de projeto

Segundo SOMMERVILLE (2005), os padrões de projeto foram criados a partir da ideia de que certos padrões eram comuns e inerentemente agradáveis e eficientes.

Um padrão descreve um problema e a sua solução, sendo possível, desta forma, a reutilização desta solução em outros projetos. É importante destacar que um padrão não é uma solução detalhada de como resolver um problema. Pode-se pensar que ele é resultado do conhecimento obtido a partir de experiências que foram acumuladas ao longo de soluções encontradas em diferentes projetos, até se obter uma solução comum para o problema.

Em outras palavras, "Padrões e linguagens de padrões são maneiras de descrever melhores práticas, bons projetos e captam a experiência de uma maneira possível de ser reusada por outros" (SOMMERVILLE, 2005).

Quatro características são consideradas essenciais para a definição de um padrão de projeto: Nome do padrão; Problema a ser resolvido; Solução dada pelo padrão; Consequências. Além disso, padrões de projeto costumam ser classificados em padrões de criação, padrões estruturais ou padrões comportamentais.

Os padrões de criação são padrões relacionados a soluções dadas para problemas na criação de objetos. São exemplos deste tipo padrões como *Factory*, *Builder* e *Singleton*.

Já os padrões estruturais são padrões relacionados a soluções que podem ser dadas para problemas na forma como os objetos podem ser compostos para formar estruturas maiores. São exemplos deste tipo, padrões como *Composite*, *Facade* e *Proxy*.

Por fim, os padrões comportamentais são os padrões relacionados a soluções que podem ser dadas para problemas com a comunicação entre objetos. São exemplos deste tipo padrões como *Mediator*, *Observer* e *Visitor*.

Três padrões de projeto foram essenciais para o desenvolvimento deste trabalho, sendo eles o padrão *Mediator*, *Plugin* e *Proxy*. O uso destes padrões será evidenciado nas explicações das Seções 3.4.1.3, 3.4.2 e 3.4.2.2.

O padrão *Mediator* é um padrão do tipo comportamental utilizado quando deseja-se intermediar a comunicação entre dois ou mais objetos para reduzir o acoplamento entre eles. Desta maneira, os objetos não farão referências explícitas um ao outro, mas sim ao

Mediator que expõe uma interface para a comunicação.

O padrão *Plugin* é um padrão do tipo estrutural usado quando se deseja adicionar funcionalidades a um programa, sendo isto feito somente sob demanda. O *plugin* é um componente de software pequeno e leve, que implementa uma interface bem definida, para que a aplicação possa utilizá-lo adequadamente.

Por fim, o padrão *Proxy* também é um padrão estrutural e é usado em diversos casos, como:

- O objeto real é caro de ser criado. Portanto, com um proxy o objeto real pode ser criado apenas quando for realmente acessado;
- O objeto real reside em um local remoto. Portanto, o proxy deve reencaminhar as chamadas para o objeto remoto;
- O objeto real deve possui controle de acesso. Portanto, o proxy deve fazer verificações antes de liberar o acesso ao objeto.

2.4 Redes ad hoc e comunicação oportunista

Redes ad hoc são redes sem fio que não necessitam de uma infraestrutura pré-existente ou de um controle centralizado. Estas redes possuem como característica (BASAGNI et al., 2004):

- Sem fio: Os nós da rede se comunicam sem fios, usando radiofrequência, infravermelho, etc;
- Ad hoc: São formadas dinamicamente e de maneira arbitrária por uma coleção de nós, conforme necessário, ou seja, é uma rede temporária;
- Autônoma e sem infraestrutura: Não dependem de qualquer infraestrutura ou administração centralizada. Assim, cada nó opera de modo ponto-a-ponto e atua como um roteador independente;
- Roteamento de múltiplos saltos: Como não há necessidade de nenhum roteador dedicado, cada nó atua como um roteador, encaminhando os pacotes de uns para os outros, permitindo o compartilhamento de dados entre os dispositivos;

- Mobilidade: Os nós podem se mover livremente enquanto se comunicam com outros nós. Assim, a topologia deste tipo de rede é dinâmica e, portanto, a comunicação entre os nós pode mudar constantemente.

Devido a estas características, as redes ad hoc possuem benefícios como a auto-reconfiguração ou ainda a facilidade de implantação, porém, acabam tendo os mesmos problemas pelas quais uma rede sem fio tradicional enfrenta.

Seus maiores diferenciais, que são a ausência de infraestrutura e um controle centralizado, trazem problemas em relação a uma maior complexidade de design em comparação a redes fixas ou ainda sobre a detecção e gerenciamento de falhas.

Outro problema comum que também decorre das características que as redes ad hoc possuem, é que devido à grande mobilidade dos nós, as rotas de comunicação podem se alterar frequentemente e de forma imprevisível. Assim, é frequente que a rede esteja desconexa e que também ocorra perda de pacotes.

Por fim, devido às variações das capacidades dos nós, pode haver problemas como a baixa largura de banda, o que pode acabar acarretando problemas de congestionamento na rede. Além disso, também existem questões relacionadas ao maior gasto energético devido à necessidade dos nós terem que atuar como roteadores. Assim, é importante o controle deste gasto para que nós não sejam perdidos prematuramente.

As redes ad hoc possuem diversos tipos de aplicação, segundo BASAGNI et al. (2004):

- Redes táticas: Redes para comunicação militar, operações autônomas em campos de batalha, etc;

- Redes de sensores: Redes formadas por uma coleção de sensores que coletam dados em tempo real para monitoramento do clima, tremores de terra, furacões, entre outros;

- Comércio e entretenimento: Sistemas de pagamento podem ser criados usando redes ad hoc como, por exemplo, para táxis. Também podem ser usadas para transmissão de notícias, condições de estrada, condições climáticas, compartilhamento de conteúdos multimídia, jogos multiplayer, acesso à Internet em áreas sem cobertura.

3 Framework Next Generation File System

O *framework* desenvolvido neste trabalho se denomina *Next Generation File System* (NGFS), nome surgido a partir da ideia de "iteratividade" no processo de desenvolvimento de sistemas de arquivos distribuídos que fazem o seu uso. Ou seja, foi pensado que sempre será possível desenvolver um novo sistema através da reutilização dos componentes já existentes e/ou com a implementação de novos componentes, sendo este novo sistema a "próxima geração" do sistema de arquivos.

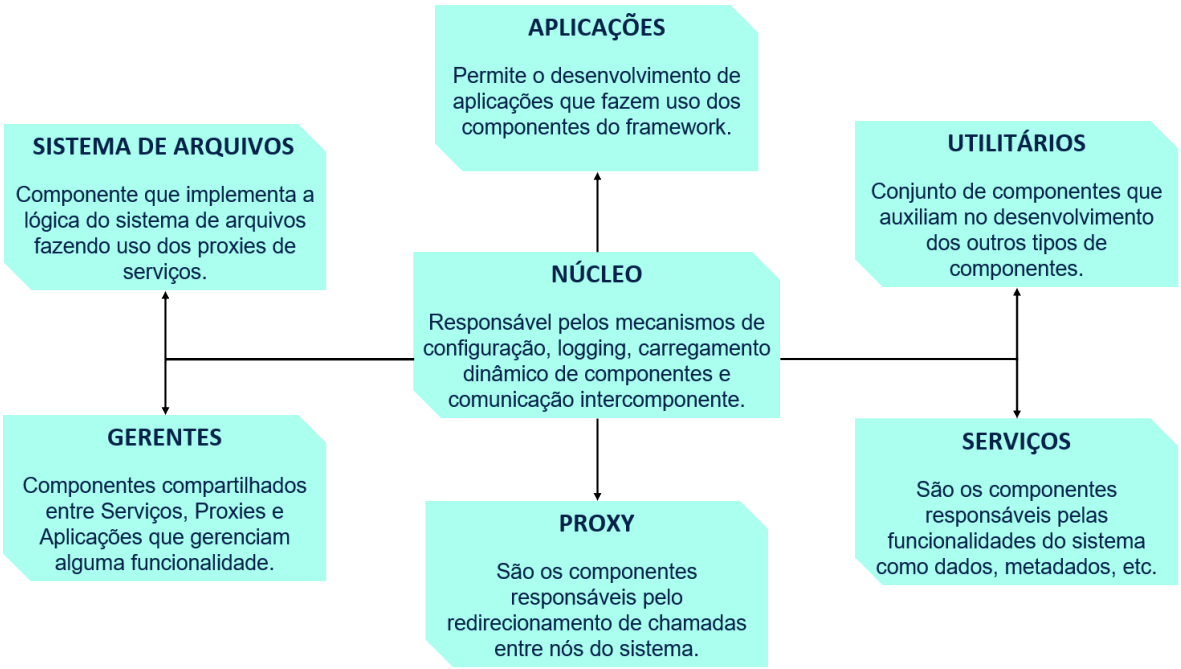

Figura 3.1: Visão simplificada das áreas funcionais que compõem o *framework* NGFS.

A Figura 3.1 mostra de forma simplificada as partes que compõem o *framework* NGFS, sendo elas: Núcleo, Utilitários, Serviços, Proxy, Gerentes, Sistema de Arquivos e Aplicações.

A área funcional do Núcleo é responsável por fornecer mecanismos de configuração e *logging*, carregamento dinâmico de componentes, ou seja, durante a execução de uma aplicação, pode-se trocar as partes que compõem o sistema sendo. Desta forma, é possível mudar o comportamento da aplicação enquanto ela ainda está em execução, tornando-a adaptável a certas condições de uso. Além disso, o núcleo também é responsável pela

comunicação entre estes componentes que formam o sistema, possuindo assim o papel de intermediador da comunicação.

A área funcional de Utilitários, é formada por um grande conjunto de classes que foram incluídas no *framework* para auxiliarem no desenvolvimento dos outros tipos de componente. A área Utilitários inclui classes relacionadas à comunicação em rede, à segurança, ao gerenciamento de banco de dados, entre outros.

Os componentes da área funcional de Serviços possuem realmente as funcionalidades típicas do sistema. Para o caso de sistemas de arquivos distribuídos, são necessárias funcionalidades para armazenamento de dados e metadados, replicação de dados e metadados para aumentar a tolerância a falhas, autenticação de usuário, entre outras. Estas funcionalidades podem ser implementadas com os componentes utilitários já presentes no *framework*, reduzindo assim o tempo de desenvolvimento do sistema.

Os componentes da área funcional *Proxy* representam a solução dada para transformar os sistemas de arquivos desenvolvidos que utilizam este *framework* em um sistema distribuído. O seu papel é o de redirecionamento das chamadas que são feitas aos serviços para outros nós do sistema. Assim, fazem o envio destas requisições através da rede para serviços que se encontram em outros dispositivos, obtendo a resposta e devolvendo para quem fez a chamada.

Os componentes da área funcional Gerentes gerenciam algum tipo de funcionalidade local, como o acesso a um banco de dados instalado localmente, a instalação de pacotes de atualização do *framework*, os processos em execução no local, etc., sendo estes gerenciadores compartilhados com os demais componentes e a aplicação.

Os componentes da área funcional "Sistema de Arquivos" implementam toda a lógica de um sistema de arquivos, fazendo isso através das funcionalidades fornecidas pelos serviços. É importante destacar que o acesso aos serviços é feito de forma indireta através dos *proxies* que podem redirecionar a chamada para um nó remoto, caso o serviço não esteja presente no local.

Por fim, as aplicações em si são implementadas fazendo uso da combinação destes componentes para entregar a funcionalidade completa de um sistema de arquivos distribuído, formando a área funcional Aplicações. Desta forma, este *framework* é capaz

de permitir o desenvolvimento de aplicações distribuídas e que através do uso de componentes já existentes, ou através da implementação de novos, consegue promover um desenvolvimento mais rápido.

3.1 Arquitetura

Como mencionado anteriormente, a Figura 3.1 mostra o NGFS composto por algumas áreas funcionais. Já na Figura 3.2 é ilustrado o relacionamento em camadas dos componentes do tipo Serviços, *Proxy*, Gerentes, Sistema de Arquivos e Aplicação. Desta forma, é possível perceber que a camada de Aplicação consegue ter acesso aos componentes do tipo Sistema de Arquivos, a camada de Sistema de Arquivos tem acesso aos *Proxies*, já os *Proxies* conseguem acessar os Serviços e todos conseguem acessar os Gerentes.

Os utilitários não são mostrados nesta figura e o mesmo acontece para os componentes do núcleo, pois estão incorporados pelos demais. Nas seções 3.3 e 3.4.1, maiores detalhes são dados sobre estes elementos.

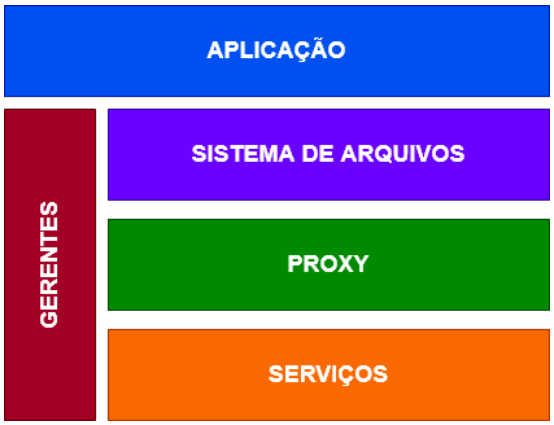

Figura 3.2: Relacionamento em camadas dos componentes de Serviços, *Proxies*, Gerentes, Sistema de Arquivos e Aplicação.

Com o aumento no nível de detalhamento que está presente na Figura 3.3, pode-se observar as principais interfaces de programação criadas no NGFS. A partir delas que é possível fazer a especialização de componentes e aplicações, constituindo a parte flexível do *framework* (*Hotspots*).

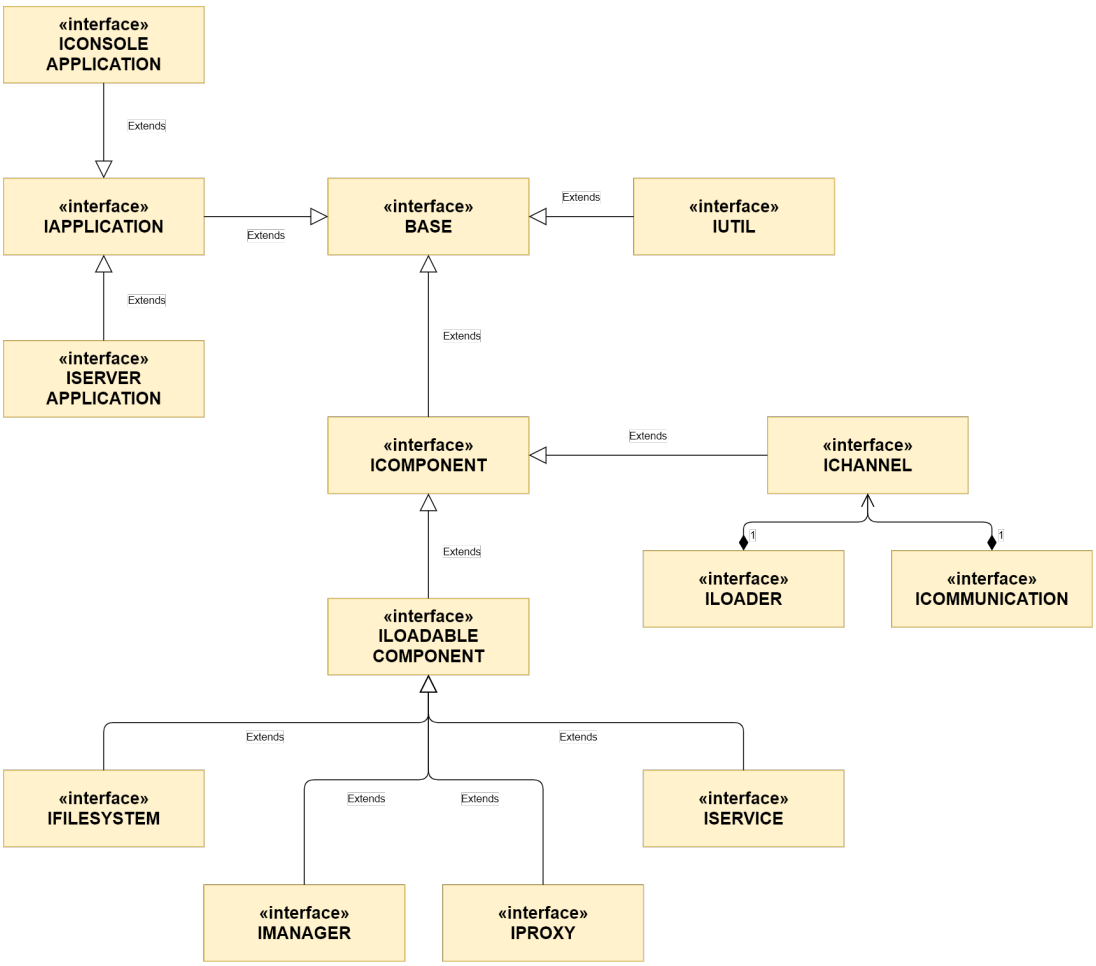

Figura 3.3: Principais interfaces do *framework*.

A interface *"Base"* é responsável por fornecer os mecanismos de configuração e *logging* do *framework*. Como todas as demais interfaces derivam dela, pode-se afirmar que todos os utilitários, componentes e aplicações do sistema possuem estas características. Por sua vez, como o mecanismo de configuração está presente em todo o sistema, é possível construir componentes e aplicações que são altamente configuráveis, trazendo assim a característica de flexibilidade ao *framework*.

A interface *"IComponent"* é outra importante interface do sistema, pois é a partir dela que todas as outras interfaces de componentes, sejam eles estáticos ou dinâmicos, são derivadas. A função desta interface é fornecer, além dos recursos da *"Base"*, a possibilidade de executar os componentes em *threads*. Ou seja, o NGFS é preparado para que cada um de seus componentes sejam executados por uma *thread* própria. As interfaces derivadas de *"IComponent"* são explicadas nas seções do Capítulo 3.4.

A interface *"IApplication"* fornece recursos da *"Base"* e recursos como parâmetros

de inicialização. Assim, se torna fácil a criação de novos parâmetros, a validação dos valores de entrada destes, além de já ser incluído de forma automática o parâmetro de "ajuda". As interfaces derivadas de *"IApplication"* são explicadas nas seções do capítulo 3.5.

3.2 Base

A interface *"Base"* é responsável, como já dito anteriormente, pelos mecanismos de configuração e *logging*. Por consequência, todos os demais elementos do sistema também os possuem, devido à herança de classes.

A Figura 3.4 mostra os principais métodos que estão presentes nesta interface, sendo eles os métodos para gerenciar as configurações ou para fazer algum tipo de *log*.

BASE

```
std::string getInstanceId();
std::string getId();
std::string getName();

void loadConfigFile(const std::string& configFilePath, bool writable = true);
void loadConfigFile(const std::string& configFilePath, const std::string& configSetName, bool writable = true);
void writeConfigFile(std::string configFilePath, ConfigSetPtr configSet);

void addConfigSet(ConfigSetPtr configSetPtr, const std::string configSetName, bool writable = true);
ConfigSetPtr getConfigSet(const std::string& configSetName);
ConfigSetPtr getConfigSet();
void removeConfigSet(const std::string& configSetName);
bool containsConfigSet(const std::string& configSetName);

std::string getBaseDirectoryPath();
std::string getBinariesDirectoryPath();
std::string getComponentsDirectoryPath();
std::string getConfigurationsDirectoryPath();
std::string getDataDirectoryPath();
std::string getKeysDirectoryPath();
std::string getLogsDirectoryPath();
std::string getPackagesDirectoryPath();
std::string getTemporaryDirectoryPath();
std::string getCacheDirectoryPath();

void logTrace(const std::string& msg);
void logDebug(const std::string& msg);
void logInfo(const std::string& msg);
void logNotice(const std::string& msg);
void logWarning(const std::string& msg);
void logError(const std::string& msg);
void logCritical(const std::string& msg);
void logFatal(const std::string& msg);
```

Figura 3.4: Principais métodos da interface *"Base"*.

3.2.1 Mecanismo de configuração

O mecanismo de configuração faz o armazenamento em um mapa das chaves-valor associadas a uma propriedade. Este estilo de mecanismo é baseado em arquivos no formato *"properties"* na plataforma Java. A Figura 3.5 mostra um exemplo de arquivo neste formato.

No NGFS esta ideia foi expandida para um conjunto de mapas de propriedades, sendo que este conjunto de mapas foi dado o nome de *"ConfigSet"*. Desta forma, é possível reunir diferentes conjuntos de configurações como, por exemplo, configurações relacionadas ao sistema de *logging*, do componente em si, da aplicação e até mesmo do próprio *framework*.

Os *"ConfigSet"* são identificados por um nome e possuem níveis de prioridade, além de poderem ser colocados como somente-leitura. Por padrão, o nome de identificação é dado pelo nome do arquivo de configuração e o nível de prioridade pela ordem de carregamento dos arquivos, sendo o primeiro arquivo carregado o de configuração do *framework*, que é chamado de "Ngfs.cfg".

Os níveis de prioridade dos *"ConfigSet"* servem, por exemplo, para resolver conflitos de propriedades. Estes conflitos podem ocorrer quando dois arquivos de configuração possuem a mesma propriedade, mas com valores diferentes. Neste caso, prevalecerá o valor da propriedade que possui a maior prioridade.

A Figura 3.5, além de mostrar um exemplo de arquivo no formato *"properties"*, também mostra um exemplo do arquivo de configuração "Ngfs.cfg". É possível perceber que há diversas propriedades relacionadas a caminhos de diretórios. Estes diretórios são sempre criados durante a inicialização do *framework* e são fundamentais para a sua organização. Por exemplo, é no diretório indicado por "ngfs.serviceDir" que o *framework* irá buscar os componentes do tipo serviço para o seu carregamento dinâmico.

Todo o gerenciamento destas configurações fica sob responsabilidade da interface *"Base"*, que expõe métodos para a manipulação daquelas como é mostrado na Figura 3.4.

```
Ngfs.cfg
1  ngfs.baseDir = Ngfs
2  ngfs.appDir = Ngfs/Application
3  ngfs.certDir = Ngfs/Certificate
4  ngfs.compDir = Ngfs/Component
5  ngfs.filesystemDir = Ngfs/Component/Filesystem
6  ngfs.managerDir = Ngfs/Component/Manager
7  ngfs.proxyDir = Ngfs/Component/Proxy
8  ngfs.serviceDir = Ngfs/Component/Service
9  ngfs.cfgDir = Ngfs/Configuration
10 ngfs.dataDir = Ngfs/Data
11 ngfs.tmpDir = Ngfs/Data/tmp
12 ngfs.cacheDir = Ngfs/Data/cache
13 ngfs.logDir = Ngfs/Logging
14 ngfs.pkgDir = Ngfs/Package
15 ngfs.createConfigFiles = true
```

Figura 3.5: Exemplo do arquivo de configuração "Ngfs.cfg".

3.2.2 Mecanismo de *logging*

A implementação deste mecanismo do *framework* faz uso de dois arquivos de configuração, um servindo para a configuração das propriedades globais de *log* e outro para a configuração das propriedades específicas de *log* do componente.

As Figuras 3.6 e 3.7 mostram exemplos destes dois arquivos. Como o arquivo de configurações globais é sempre carregado primeiro, possui maior prioridade e, portanto, irá sobrescrever configurações de todos os demais arquivos de configuração de *logging*. É por isso que o mecanismo faz uso dos dois arquivos para a configuração, já que assim é possível alterar facilmente uma propriedade no arquivo global de configuração e esta servirá para todos os outros.

Duas saídas de *log* foram implementadas, a principal delas é a saída no console que já é configurada por padrão. A segunda saída é para arquivo de texto, sendo esta opcional.

As Figuras 3.6 e 3.7 também mostram várias propriedades que podem ser configuradas em relação a essas saídas. É possível alterar completamente como as mensagens aparecem nas saídas a partir da alteração no padrão presente na propriedade "logging.formatters.f0.pattern" ou, ainda, pode-se usar propriedades como "logging.channels.c2.rotation" que fazem o *backup* automático dos arquivos de *log*, optando por comprimi-los com "logging.channels.c2.compress" e apagá-los, caso a quantidade de *backups* ultrapasse a quantidade definida em "logging.channels.c2.purgeCount".

Isto também acaba servindo como exemplo de uso do mecanismo de configuração do *framework*. Neste caso, todo o mecanismo de *logging* está sendo configurado através de arquivos de configuração, não sendo necessária nenhuma alteração no código para que isso tudo funcione como esperado.

GlobalLogging.cfg

```
1  logging.channels.c1.class = ConsoleChannel
2  logging.channels.c1.formatter = f0
3  logging.formatters.f0.class = PatternFormatter
4  logging.formatters.f0.pattern = [%p] [%d-%m-%Y %H:%M:%S] [%s] > %t
5  logging.formatters.f0.times = UTC
```

Figura 3.6: Exemplo de arquivo global de configuração do sistema de *logging*.

NgfsServerLogging.cfg

```
1  logging.channels.c0.channels = c1, c2
2  logging.channels.c0.class = SplitterChannel
3  logging.channels.c2.archive = timestamp
4  logging.channels.c2.class = FileChannel
5  logging.channels.c2.compress = true
6  logging.channels.c2.formatter = f0
7  logging.channels.c2.path = Ngfs/Logging/NgfsServer.log
8  logging.channels.c2.purgeCount = 10
9  logging.channels.c2.rotation = 100M
10 logging.channels.c2.times = utc
11 logging.loggers.l0.channel = c0
12 logging.loggers.l0.level = information
13 logging.loggers.l0.name = NgfsServer
```

Figura 3.7: Exemplo de arquivo de configuração do sistema de *logging* da aplicação "NgfsServer".

3.3 Utilitários

Os utilitários são um conjunto de classes implementadas que formam a base fixa do *framework* (*Frozenspots*). Após pesquisas realizadas para se conhecer o domínio de sistemas de arquivos distribuídos, foram pensadas quais seriam as funcionalidades úteis para a fácil implementação dos componentes de sistemas deste tipo. Desta forma, chegou-se então em um conjunto de implementações que entregam funcionalidades prontas para uso, sendo elas, por exemplo: Compressão de arquivos, uso de banco de dados, codificação de dados, mecanismo de eventos, algoritmos de hashing, comunicação em rede, gerenciamento de processos, geração de identificadores e números aleatórios, segurança,

serialização de dados, streams, tratamento de strings, tarefas agendadas, manipulação de tempo e observadores.

Para auxílio da implementação destes recursos, foram utilizadas as bibliotecas POCO[1]. Segundo informações que estão disponíveis no site do projeto, este conjunto de bibliotecas teve seu desenvolvimento iniciado em 2004 pelo engenheiro de software Günter Obiltschnig. Sua motivação estava no fato de que, na época, o C++ estava perdendo espaço devido a uma tendência de uso de linguagens de programação que geram código gerenciado. Porém, como ele gostava da linguagem C++, decidiu criar um conjunto abrangente de bibliotecas para atender aos requisitos de vários tipos de aplicações.

Primeiramente, criou bibliotecas para ele mesmo, mas depois acabou fornecendo-as para outros programadores e, assim, o projeto POCO (*POrtable COmponents*) foi criado no ano de 2005, sendo que mais de 180 desenvolvedores colaboraram no código do projeto até o momento.

As bibliotecas POCO estão licenciadas pela *Boost Software License*, que permite tanto o uso não-comercial quanto comercial.

A Figura 3.8 mostra as principais partes do conjunto de bibliotecas POCO. Bibliotecas como a "Net", "NetSSL" auxiliaram na implementação dos recursos de rede do *framework* NGFS, enquanto bibliotecas como "Crypto" auxiliaram na implementação de algoritmos de criptografia. Também há muitos recursos interessantes incluídos nas bibliotecas "Foundation" e "Util".

Desta forma, várias destas bibliotecas são utilizadas no prsente trabalho para facilitar o desenvolvimento do mesmo. Tais implementações se transformaram nos utilitários do *framework*.

Como este conjunto de classes utilitárias que foram implementadas é extenso, as seções a seguir dão destaque para as partes de grande importância no *framework* NGFS, pois são utilizadas em mecanismos essenciais para o seu funcionamento.

[1]https://pocoproject.org/

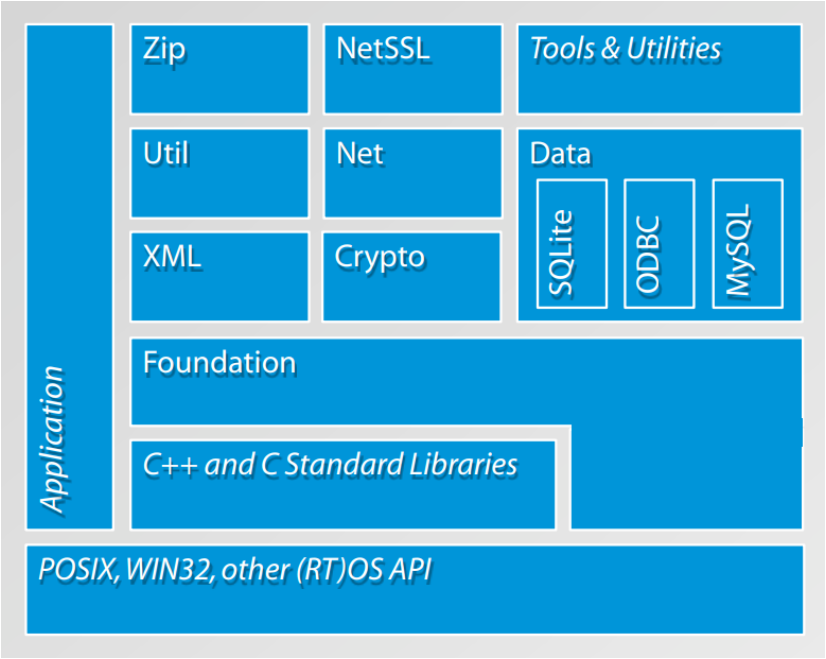

Figura 3.8: Visão geral sobre as bibliotecas POCO.

3.3.1 Comunicação em rede

Um sistema de arquivos distribuído é um sistema de arquivos que, especificamente, possui seus componentes distribuídos em uma rede de computadores. Portanto, se deseja-se a criação de um sistema que seja distribuído, é necessário pensar na comunicação via rede.

Os protocolos TCP e UDP são os principais protocolos que se encontram na camada de transporte em redes TCP/IP (TANENBAUM, 2003). Enquanto o TCP fornece um serviço de transferência confiável, trazendo garantias como a entrega ordenada, o controle de fluxo, o controle de congestionamento, além de ser orientado à conexão, o UDP se apresenta como o inverso disto, não oferecendo um serviço confiável de entrega, não tendo qualquer garantia. Porém, costuma ser interessante sua utilização em aplicações que demandam pouco atraso, como aplicações multimídia, ficando a cargo da aplicação a implementação de mecanismos para tratar as possíveis perdas de mensagens, inversão na ordem de recebimento, entre outros problemas que podem acontecer.

Um interessante recurso do UDP é a possibilidade de comunicação do tipo um para muitos, como a comunicação *broadcast* e a *multicast*, sendo a primeira do tipo "um para todos" e a segunda do tipo "um para alguns". Na comunicação *multicast* os nós

formam os chamados "grupos *multicast*" e, assim, a comunicação ocorre dentro deste grupo.

Os sistemas operacionais fornecem uma API para comunicação via rede que, comumente, se baseia na abstração chamada *socket*. O *socket* representa uma extremidade de uma comunicação que ocorre entre dois processos através de uma rede de computadores.

Através do uso de *sockets*, é possível representar um processo na rede como sendo um endereço IP e um identificador númerico chamado de "porta". O endereço IP serve para que o protocolo IP faça o roteamento dos dados enviados de um *host* de origem até o *host* de destino. Chegando no seu destino, pode-se utilizar o número de porta para identificar o processo que está aguardando o recebimento de dados da rede.

Neste contexto, é esperado que o *framework* desenvolvido neste trabalho, por ser um sistema distribuído, faça uso de *sockets*. Para facilitar a implementação do uso de *sockets*, sem que fosse necessária a utilização direta da API disponibilizada por um sistema operacional, foram utilizadas as implementações de *socket* disponíveis na biblioteca POCO Net e POCO Net SSL. Esta última é dedicada para os casos de *sockets* usando o protocolo SSL (CHOU, 2002), trazendo assim uma comunicação segura através do uso de criptografia.

A Figura 3.9 mostra as interfaces disponíveis no *framework* para o uso de sockets TCP e UDP. As interfaces "ITCPClientSocket" e "ITCPServerSocket" servem para o envio e recebimento de dados usando o protocolo TCP, enquanto as interfaces "ITCPSecureClientSocket" e "ITCPSecureServerSocket" servem para o mesmo propósito, porém, com o uso do protocolo SSL. Já as interfaces "IUDPSocket" e "IMulticastSocket" fazem uso do protocolo UDP e UDP com comunicação *multicast*.

Da mesma forma que as interfaces de *socket* que foram criadas, também foram criadas interfaces para a implementação de servidores de rede. Estas interfaces são mostradas na Figura 3.10, sendo possível ver os componentes que fazem parte de um servidor de rede, que são o despachante e o filtro de entrada.

Por padrão, os servidores de rede implementados com a interface "INetworkServer" são executados em uma *thread* própria. Ao receberem uma conexão no caso do uso de um *socket* TCP, ou um datagrama no caso do uso de um socket UDP, os servidores

devem encapsular estes objetos em um "ITCPServerHandler" ou "IUDPServerHandler", respectivamente, sendo estes enfileirados. Enquanto isso, um objeto do tipo "ITCPServer-Dispatcher" ou "IUDPServerDispatcher" é o responsável por desenfileirar os *handlers* e despachá-los através do disparo de evento do tipo "Nova conexão"ou "Nova mensagem". Caso exista algum ouvinte registrado para este evento, este deverá fazer o tratamento adequado, retornando a resposta, se assim for necessário. Cada despacho é executado em uma *thread* própria, ou seja, o *framework* do NGFS faz uso de servidores de rede que são do tipo *multithread*.

Por fim, a interface "INetworkServerFilter" permite a implementação de filtros de entrada que podem aceitar ou recusar estas conexões ou mensagens recebidas pelo servidor de rede, seguindo alguma regra própria. Pode-se pensar, por exemplo, na implementação de algum mecanismo de *firewall* embutido diretamente no servidor de rede.

Desta forma, este conjunto de interfaces relacionadas a *sockets* e servidores de rede foram especializados em componentes que se encontram prontos para uso, sendo utilizados para a implementação dos serviços e *proxies* do *framework*.

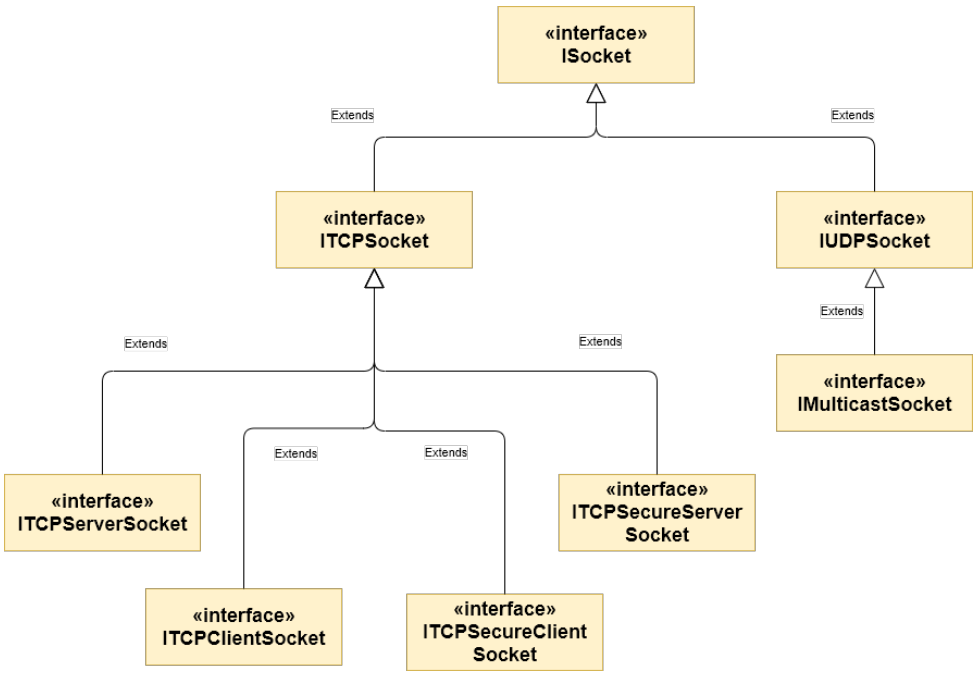

Figura 3.9: Interfaces para a implementação de sockets TCP e UDP.

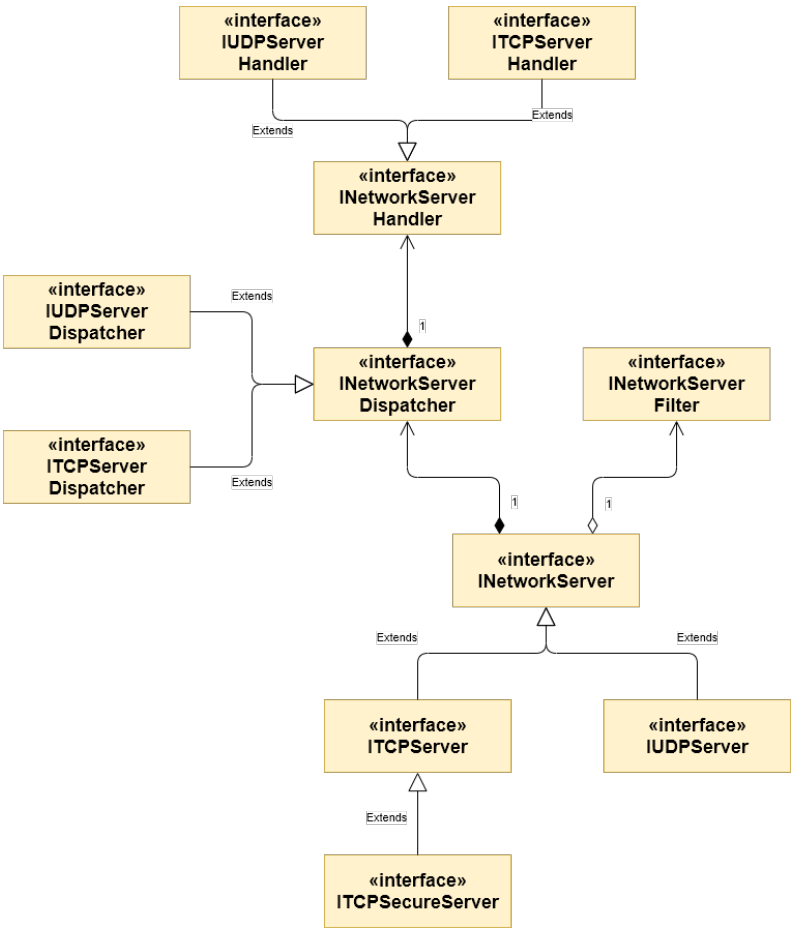

Figura 3.10: Interfaces para a implementação de servidores de rede.

3.3.2 Eventos

Um evento é o resultado de uma determinada ação, por exemplo, ao pressionar uma tecla no computador, um evento de "tecla pressionada" é gerado pelo sistema operacional e as aplicações interessadas neste tipo de evento podem se inscrever como ouvintes, sendo informadas quando este evento ocorrer.

É esperado que um evento carregue consigo informações que são relevantes para que se entenda o que ocorreu. Retomando o exemplo da tecla pressionada, pode-se presumir que um evento deste tipo carrega a informação da identificação de qual tecla foi pressionada. Assim, a aplicação poderá decidir exatamente qual tratamento deve dar a ele.

Muitos dos componentes que foram implementados no *framework* NGFS fazem uso de eventos, ou seja, seguem o paradigma de programação orientada a eventos. Um exemplo de componente deste *framework* que é orientado a eventos são os serviços. Os

serviços aguardam por eventos que são disparados pelo servidor de rede e então reagem a isto tratando a mensagem recebida.

Os ouvintes de eventos são compostos por métodos tratadores de eventos. Pode-se ter, ainda, o caso de uma cascata de eventos, no qual um evento é gerado e disparado pelo despachante de eventos. Isto significa que estes métodos tratadores dos ouvintes serão chamados e, então, estes tratadores podem acabar disparando novos eventos, formando uma cascata. Isto também ocorre nos serviços do *framework*, nos quais um evento de mensagem recebida se transforma, a partir do reconhecimento do tipo de mensagem, em um evento mais específico que, por sua vez, irá disparar um tratador mais específico.

A implementação de eventos no *framework* NGFS pode ser realizada através das interfaces "ngfs::IEvent" e "ngfs::IPriorityEvent". Já a classe "ngfs::EventArgs" serve para passar as informações relacionadas ao evento em questão, fazendo uso de uma estrutura de mapa para armazenar chaves-valor. A Figura 3.11 mostra o relacionamento entre essas interfaces e classes do mecanismo de eventos que foi implementado no *framework*.

Com as interfaces "ngfs::IEvent" e "ngfs::IPriorityEvent" é possível a implementação de tipos diferentes de eventos. Este uso acabou não sendo muito explorado, sendo implementada apenas um tipo genérico de evento. Esse evento genérico é utilizado para todos os eventos que são gerados no *framework*.

Em relação à criação de eventos prioritários, o seu diferencial é que estes eventos podem ser tratados de maneira especial. São úteis, por exemplo, no caso em que um evento prioritário é disparado e este deverá ser tratado imediatamente, antes de outros eventos que estão aguardando na fila de despacho.

Por fim, a classe "ngfs::EventDispatcher" possui como responsabilidade o gerenciamento de eventos, sendo possível criar eventos, do tipo prioritário ou não, registrar ouvintes desses eventos e disparar eventos de maneira síncrona ou assíncrona.

3.3.3 Segurança

Um ponto fundamental de um sistema distribuído é a sua segurança. Como os seus componentes estão distribuídos em uma rede, acabam estando sujeitos a uma variedade de ataques possíveis.

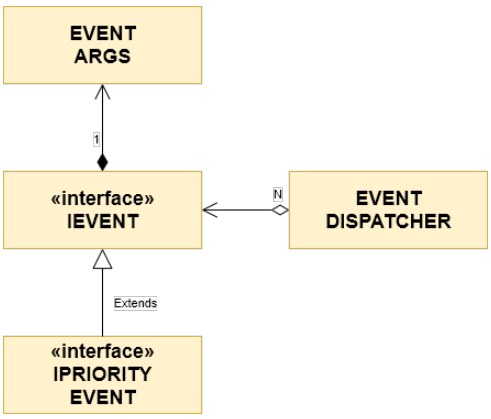

Figura 3.11: Relacionamento entre interfaces e classes do mecanismo de eventos.

Pode-se citar, como exemplo, o tipo de ataque *man-in-the-middle*, em que um atacante pode interceptar os dados que trafegam na rede, podendo roubá-los ou ainda podem adulterá-los, sem que os envolvidos na comunicação percebam que isto esteja ocorrendo. Para evitar que problemas do tipo aconteçam, faz-se uso de criptografias e assinaturas digitais, garantindo assim o sigilo e autenticidade da informação transferida pela rede.

Todo processo de cifragem, que é o processo de transformação de um texto inteligível em um texto ininteligível, se baseia em um algoritmo e uma chave. O algoritmo poderá utilizar a chave, por exemplo, para definir a quantidade de vezes que uma determinada letra será trocada de posição. Por sua vez, o processo de decifragem de um texto cifrado se baseia na reversão destas operações que foram realizadas.

Existem dois tipos de criptografia, sendo eles a simétrica e a assimétrica. No caso da criptografia simétrica, uma única chave é utilizada no processo de cifragem/decifragem, sendo esta chave compartilhada entre emissor e receptor da informação. Desta forma, cuidados especiais devem ser tomados para que esta chave não seja vazada e toda a segurança seja perdida. Já no caso da criptografia assimétrica, existe um par de chaves envolvidos no processo, sendo uma chave privada, que deve ser mantida em sigilo, e uma chave pública, que pode ser compartilhada livremente. Enquanto a chave pública pode ser utilizada no processo de cifragem, apenas a chave privada pode ser utilizada para decifrar a informação. Desta forma, neste tipo de criptografia, cada entidade envolvida na comunicação deve possuir um par de chaves, armazenar sua chave privada e compar-

tilhar a sua chave pública com quem deseja compartilhar informações. O emissor de uma mensagem deve fazer uso da chave pública do destinatário, pois, assim, somente este, que é o possuidor da chave privada, deve ser capaz de descriptografá-la.

São exemplos de criptografia simétrica o algoritmo AES e de criptografia assimétrica o algoritmo RSA (STALLINGS, 2006). Estes algoritmos foram implementados no *framework* NGFS nas classes "ngfs::AESEncryptor" e "ngfs::RSAEncryptor" que, por sua vez, dependem de implementações de cifras simétricas/assimétricas que estão disponíveis na biblioteca do OpenSSL[2].

Por sua vez, a criptografia de chave pública, como também é conhecida a criptografia assimétrica, tem outro uso interessante para a área de segurança, que é o processo de assinatura digital. A assinatura digital é uma técnica empregada nas situações em que não existe confiança entre emissor e receptor, ou seja, um não consegue garantir o que o outro diz ser e vice-versa. Ou, ainda, não é possível garantir que o conteúdo da mensagem não foi alterado por um terceiro durante a sua transferência. Este costuma ser exatamente o cenário em uma rede de computadores, onde nem sempre é possível garantir que se está comunicando com quem se espera e muito menos pode-se garantir que as informações trafegadas não foram alteradas no caminho.

A técnica de assinatura digital consiste no uso combinado de algoritmos de hash com algoritmos de criptografia assimétrica. A primeira etapa do processo deve ser a aplicação de uma função de hash sobre o conteúdo de uma mensagem. O valor de saída desta função deve então ser criptografado utilizando a chave privada do emissor, sendo o resultado da criptografia chamado de "assinatura digital". A mensagem, a assinatura e a chave pública do emissor devem ser então enviadas para o destinatário. O destinatário, por sua vez, repetirá o procedimento, aplicando a função de *hash* no conteúdo da mensagem original, aplicando a criptografia sobre o resultado do *hash* e utilizando a chave do emissor para, então, comparar os resultados finais das assinaturas gerada e recebida. Caso sejam iguais, o conteúdo da mensagem não foi alterado e aquela mensagem foi realmente enviada pelo emissor esperado.

Ainda, se deseja-se garantir o sigilo das informações, pode-se criptografar o conteúdo

[2]https://www.openssl.org/

da mensagem com a chave pública do destinatário. Desta forma, somente o destinatário poderia descriptografar a mensagem e fazer o processo de verificação da assinatura digital.

No *framework*, o mecanismo de assinatura digital está implementado na classe "ngfs::RSASigner", que faz uso dos algoritmos de *hash* SHA e do algoritmo de criptografia assimétrica RSA.

3.4 Componentes

Segundo SOMMERVILLE (2005), há um consenso de que "um componente é uma unidade de software independente que pode ser composta por outros componentes para criar um sistema de software". A partir daí, tem-se duas características que um componente possui:

- Por ser uma entidade independente, seu código não precisa ser compilado juntamente com os demais componentes do sistema para ser executado. Desta forma, não é necessário fornecer o seu código-fonte, apenas a sua API;
- Todos os serviços fornecidos pelo componentes estão disponíveis através desta API e todas as interações ocorrem somente por essa interface, com o estado interno do componente não sendo exposto.

A ideia principal do uso de componentes é o reúso de software. Com o uso de componentes, pode-se facilmente reutilizá-los, atualizá-los e compô-los. Além disso, um software pode ser inteiramente composto por componentes.

Como o componente é uma unidade independente que expõe seus serviços através de uma interface, seus detalhes de implementação internos são ocultados e podem ser alterados sem que isso afete o restante do sistema, desde que a interface seja mantida.

Também devido a esta interface, pode-se abstrair as complexidades de implementação abaixo e voltar a preocupação somente ao alto nível, na integração destes componentes.

Pode-se destacar que, com o uso de componentes, o tempo de desenvolvimento é reduzido devido à redução na necessidade de implementações, sendo o desenvolvimento focado apenas na integração e uso destes componentes prontos.

Como não é necessário que o código-fonte do componente esteja disponível para compilação do programa em si, pode-se pensar também na possibilidade da criação de um negócio que envolva a comercialização de componentes por terceiros.

A aplicação por sua vez, deve fornecer mecanismos de carregamento e execução, além de também fornecer mecanismos para a comunicação entre estes componentes.

Apesar das grandes vantagens que o uso de componentes trazem, alguns problemas existem. Por exemplo, se a implementação do componente é uma caixa-preta, como garantir que o componente é confiável? Que não possui falhas que levem a brechas de segurança ou até mesmo que contenha código malicioso?

Desta forma, deve-se ter a preocupação de usar apenas componentes que foram certificados, assegurando que são confiáveis. Porém, é claro, que isto traz custos para o desenvolvimento do projeto.

Pensando nas características de independência, reutilização de código, redução no tempo de desenvolvimento e possibilidade de desenvolvimento e distribuição por parte de terceiros é que o *framework* NGFS optou pelo paradigma de desenvolvimento baseado em componentes.

Sendo assim, existem interfaces no *framework* que possibilitam o desenvolvimento de componentes para o sistema de arquivos distribuído. A Figura 3.12 mostra as interfaces que possibilitam o desenvolvimento de todos os tipos de componente do *framework* NGFS.

É possível perceber que a interface "IComponent" herda da interface "Base" e, portanto, todos os componentes do sistema também possuem os mecanismos de configuração e *logging* explicados no capítulo 3.2. Também é possível perceber que há uma distinção entre os tipos de componente, havendo uma interface "ILoadableComponent" das quais as interfaces "IFileSystem", "IManager", "IProxy" e "IService" herdam. Esta diferenciação ocorre porque o *framework* NGFS faz uso de dois tipos de componentes, os componentes estáticos e os componentes dinâmicos. Enquanto os componentes estáticos não podem ser alterados durante o tempo de execução da aplicação, os componentes dinâmicos podem ser alterados a qualquer momento. Desta forma, os componentes que herdam apenas da interface "IComponent" somente podem ser alterados em tempo de compilação, sendo assim, os tipos de componentes estáticos do *framework*.

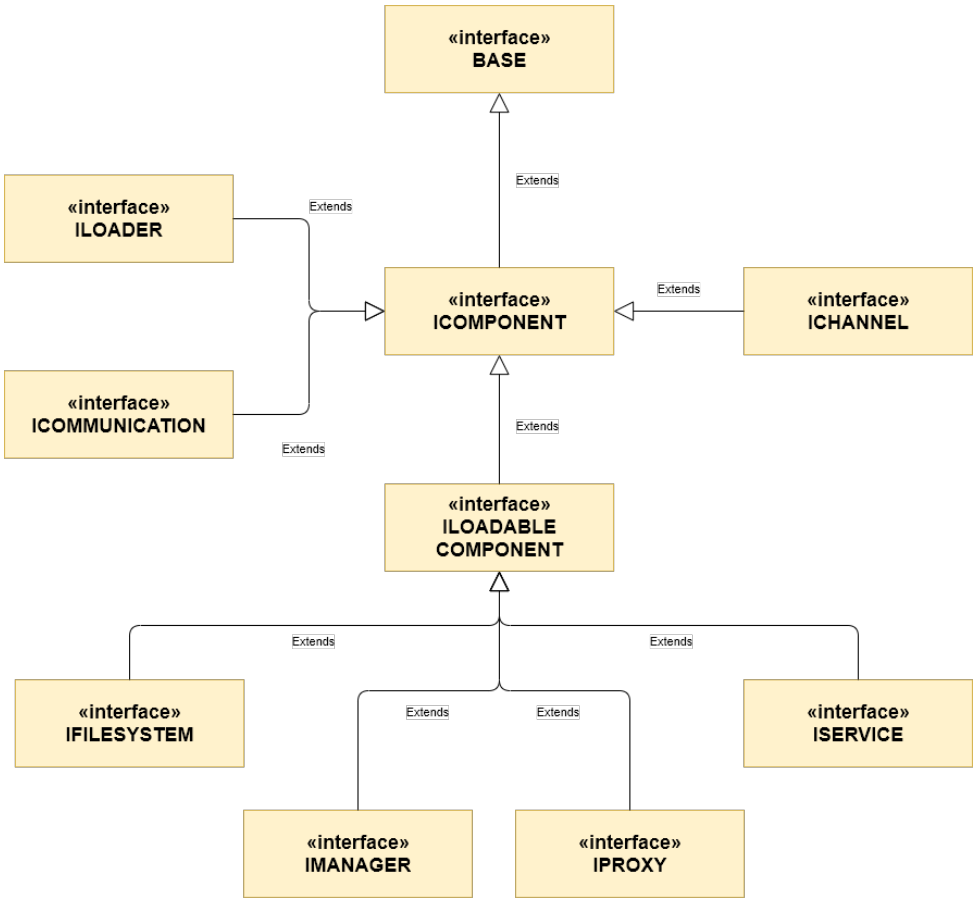

Figura 3.12: Interfaces que possibilitam o desenvolvimento de componentes para o *framework* NGFS.

A Figura 3.13 mostra os principais métodos das interfaces "IComponent" e "ILoadableComponent". A Interface "IComponent" força com que todos os componentes possuem métodos de inicialização e desinicialização, além de um método principal chamado *"componentMain"*. A intenção destes métodos é transformar a execução dos componentes em "mini-programas", ou seja, cada componente do *framework* NGFS é executado em sua própria thread.

A partir desta interface, pode-se criar os componentes estáticos do tipo "ILoader", "ICommunication", que são componentes responsáveis pelo carregamento de componentes dinâmicos e pela comunicação entre estes componentes. Por fim, a interface "IChannel" é uma composição destes outros dois tipos de componente, possibilitando um gerenciamento mais fácil destas duas características do *framework*, sendo isto explicado na seção 3.4.1.3.

Enquanto isso, o destaque para a interface "ILoadableComponent" é a possibilidade de fornecer uma API extensível, através dos métodos *"registerMethod"*, *"containsMethod"* e *"invokeMethod"*. Maiores explicações sobre estes mecanismos serão dados na seção 3.4.2.

A partir desta interface, pode-se criar os componentes dinâmicos do tipo "IFilesystem", "IManager", "IProxy" e "IService", que são os principais componentes deste *framework* e são os componentes que o tornam um *framework* para sistemas de arquivos distribuídos, por assim dizer, já que componentes do tipo "IFilesystem" são os componentes que implementam a própria lógica de um sistema de arquivos, fazendo uso para isto de serviços que são fornecidos pelos componentes do tipo "IService". Porém, como deseja-se ter um sistema que seja distribuído, os componentes do tipo "IProxy" adicionam esta característica ao serem os responsáveis por fazer a "interligação" entre um componente que está em execução em um determinado nó a outro componente que está em execução em outro nó do sistema, fazendo isto através da comunicação em rede. Por fim, pensando que poderiam haver componentes que são compartilhados pelos demais, é que os componentes do tipo "IManager" foram criados.

Assim, este conjunto de interfaces que possibilitam a criação de diferentes tipos de componentes, sendo esta característica o "coração" deste *framework*, pois é isto que adiciona a característica de alta flexibilidade que é o foco deste trabalho.

Desta forma, é neste ponto que atinge-se o objetivo de determinar os mecanismos de extensibilidade disponíveis no *framework*, ou seja, os seus *hotspots*.

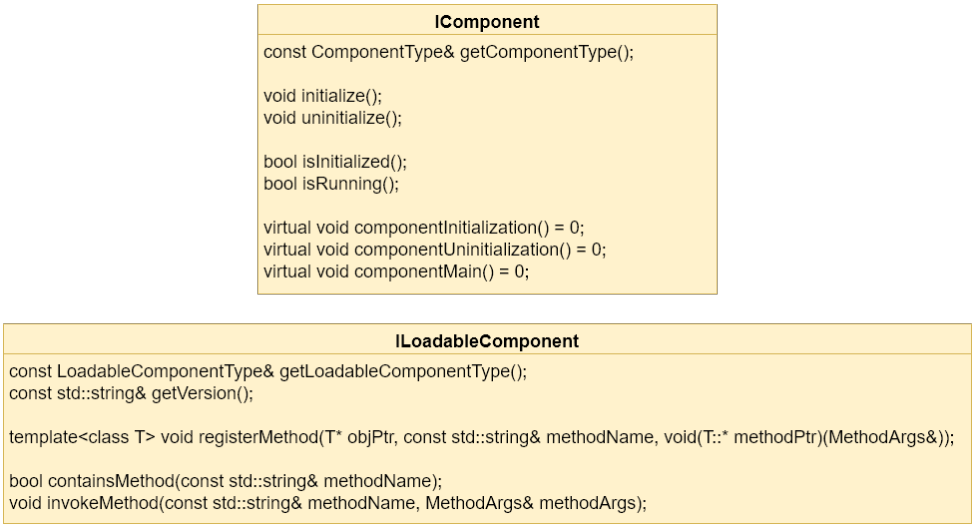

Figura 3.13: Interfaces IComponent e ILoadableComponent que possibilitam a implementação dos componentes estáticos e dinâmicos do *framework* NGFS.

3.4.1 Componentes estáticos

Os componentes estáticos do *framework* são os componentes que herdam somente da interface "IComponent" e que não podem ser alterados em tempo de execução, apenas no tempo de compilação.

Três tipos de componentes estáticos foram definidos no *framework* e podem ser implementados a partir das interfaces "ILoader", "ICommunication" e "IChannel". Estes tipos de componente são o suporte necessário para que os componentes do tipo dinâmico funcionem no sistema.

Os componentes do tipo "ILoader", por exemplo, são os responsáveis pelo carregamento e descarregamento em tempo de execução dos componentes dinâmicos. Já os componentes "ICommunication" são os responsáveis por intermediar a comunicação entre estes componentes, promovendo assim o desacoplamento entre eles, sendo isto necessário para que a característica de carregamento/descarregamento de componentes funcione de maneira adequada, caso contrário, se os componentes possuíssem um alto acoplamento, dificilmente seria possível trocar um componente pelo outro ou até mesmo não ter o componente presente em execução ali.

Por fim, os componentes do tipo "IChannel" são uma composição destes outros dois componentes, sendo usados para facilitar o gerenciamento destas funcionalidades por outro elemento do sistema como os componentes do tipo "sistema de arquivos" que possuem um canal para serviços, onde é possível carregar/descarregar serviços e onde estes serviços podem comunicar entre si. Além disso, este canal também possibilita que *"proxies"* sejam carregados dinamicamente no lugar dos serviços, permitindo que chamadas aos serviços sejam redirecionadas para outros nós.

A Figura 3.14 mostra o relacionamento destas interfaces e as especializações já presentes no *framework* desenvolvido neste trabalho. É possível notar todos os carregadores dos tipos de componentes dinâmico, um sistema de comunicação entre componentes que funciona via troca de mensagens e três tipos de canais que servem para componentes do tipo "sistema de arquivos", gerentes (*Common Channel*) e um canal que é capaz de gerenciar dois tipos de componentes, sendo eles do tipo serviço e *proxy* (*Service Channel*).

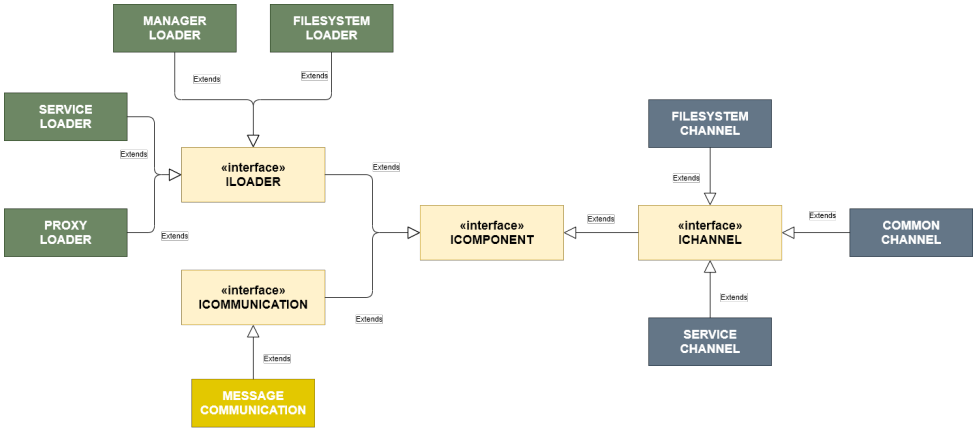

Figura 3.14: Interfaces "ILoader", "ICommunication" e "IChannel" e especializações já presentes no *framework* NGFS.

3.4.1.1 Carregadores de componentes

Os carregadores de componentes são os componentes que são derivados da interface "ILoader" e possuem como responsabilidade o gerenciamento de todo o tempo de vida de um componente dinâmico.

Este tempo de vida dos componentes dinâmicos começa com o seu carregamento, ao qual o componente no formato de arquivo da biblioteca compartilhada (Arquivos do

tipo "*.dll" em sistemas operacionais Windows ou "*.so" em sistemas operacionais Linux) são carregados e colocados em execução.

Para o carregamento destas bibliotecas, a classe "Poco::ClassLoader" é utilizada. Esta classe permite o carregamento de bibliotecas compartilhadas em tempo de execução e independente de sistema operacional. Os requisitos para que este mecanismo funcione, é que os objetos que devem ser carregados a partir da biblioteca possuam uma mesma interface e esta biblioteca deve exportar um manifesto que exporta as classes disponíveis. Maiores informações estão disponíveis na documentação da biblioteca POCO[3].

A POCO fornece macros que facilitam a criação destes manifestos de exportação de classes. Para facilitar um pouco mais, novas macros foram definidas no *framework* NGFS para indicar qual tipo de componente está sendo exportado. Assim, há macros como "BEGIN_SERVICE_EXPORT_MANIFEST" para exportação de serviços e a macro "BEGIN_PROXY_EXPORT_MANIFEST" para a exportação de *proxies*.

A Figura 3.15 exemplifica como o uso destas macros deve ocorrer. É válido destacar que é possível que uma biblioteca compartilhada possa conter mais de um componente exportado, mas este uso não foi explorado até o momento deste trabalho.

Após o carregamento da biblioteca do componente, uma instância para cada classe exportada é criada e colocada em execução em *pool de threads*, permanecendo assim até o fim do seu tempo de vida.

Quando o descarregamento da biblioteca é solicitado, é chamado o método de desinicialização do componente, o encerramento de sua thread é aguardado e, após isto, a sua biblioteca compartilhada é descarregada, terminando o tempo de vida do componente.

O funcionamento deste carregamento/descarregamento dinâmico só é possível graças ao desacoplamento existente entre estes componentes, consequência da intermediação na comunicação fornecida pelos componentes de comunicação que serão explicados na seção a seguir.

[3]https://pocoproject.org/docs/Poco.ClassLoader.html

```
#pragma once

#include "Ngfs/Component/Loadable/Service/Interface/ITCPTimeService.h"

#include "Ngfs/Util/Configuration/DefaultConfigurator.h"
#include "Ngfs/Util/Logging/DefaultLogger.h"

#include "Ngfs/Util/Identificator/UUID.h"

#include "Ngfs/Util/Time/Time.h"

namespace ngfs
{
    class NGFSAPI TCPTimeService : public ITCPTimeService
    {
        private:
            Time time;
        protected:
            void componentInitialization();
            void componentUninitialization();
            void componentMain();
        public:
            TCPTimeService(bool enableFileLogger = false);
            ~TCPTimeService();

            std::string getTimezoneName();
            int getTZD();

            std::int64_t getUTCTimestamp();
            std::string getUTCTimeString(const std::string& fmt = "%d-%m-%Y %H:%M:%S");

            std::int64_t getLocalTimestamp();
            std::string getLocalTimeString(const std::string& fmt = "%d-%m-%Y %H:%M:%S");
    };
}

BEGIN_SERVICE_EXPORT_MANIFEST
    EXPORT_COMPONENT(ngfs::TCPTimeService)
END_COMPONENT_EXPORT_MANIFEST
```

Figura 3.15: Definição da classe “TimeService”, um serviço tempo do *framework* NGFS.

3.4.1.2 Comunicação entre componentes

A intermediação da comunicação entre os componentes é uma característica essencial para promover o baixo acoplamento entre eles e, desta forma, tornar possível que o seu carregamento/descarregamento dinâmico seja realizado sem que isto provoque falhas críticas e interrupções no funcionamento da aplicação.

Para a criação de componentes que façam a intermediação desta esta comunicação, a interface “ICommunication” foi definida. Ela é responsável pelo registro de tipos de mensagens que são aceitas por um determinado componente, aceita o registro de manipuladores destas mensagens, permite que os componentes se associem ou desassociem de um tipo de mensagem e, é claro, permite a troca de mensagens entre os componentes.

A Figura 3.16 mostra a interface “ICommunication” e os serviços disponíveis através dos componentes especializados a partir desta interface.

Os métodos de troca de mensagens *“sendMessage()”* e *“sendMessageTo()”* são deixados para implementação dos componentes derivados, pois foi pensado que poderia

haver uma vasta gama de meios de troca de mensagens entre os componentes.

Apesar de estar-se falando aqui de troca de mensagens entre componentes que estão em execução em uma mesma aplicação, pode-se pensar em implementações alternativas que fizessem, por exemplo, a troca de mensagens entre componentes de aplicações diferentes ou algo semelhante.

Por fim, o *framework* já possui uma especialização para esta interface, chamada de "MessageCommunication". Esta classe faz uma implementação simples de troca de mensagens, através do enfileiramento de mensagens e disparo dos manipuladores correspondentes, caso eles existam. Caso contrário, a mensagem se manterá em um *buffer* até que seja removida após um tempo de espera ser expirado. Esta *bufferização* das mensagens foi implementado pensando na possibilidade do tratamento "tardio" de uma mensagem, caso um componente que não estava disponível no momento do envio, se torne disponível posteriormente. Assim, fica a cargo de quem enviou a mensagem, escolher se deseja este comportamento de tratamento tardio ou não, definindo o tempo de expiração adequado.

ICOMMUNICATION

```
void registerMessageType(const std::string& msgType);
void unregisterMessageType(const std::string& msgType);
std::list<std::string> getMessageTypes();
bool containsMessageType(const std::string& msgType);

template<class T> void registerMessageHandler(T* handlerPtr, void(T::* handlerMethod)(Message&), std::list<std::string>& msgTypeList);
template<class T> void registerMessageHandler(T* handlerPtr, void(T::* handlerMethod)(Message&));
template<class T> void unregisterMessageHandler(T* handlerPtr);

template<class T> void associateMessageType(T* handlerPtr, std::list<std::string>& msgTypeList);
template<class T> void associateMessageType(T* handlerPtr, const std::string& msgType);
template<class T> void disassociateMessageType(T* handlerPtr, const std::string& msgType;

bool hasMessageHandler(const std::string& msgType);

virtual void sendMessage(Message& msg) = 0;
virtual void sendMessageTo(Message& msg, const std::string& recipientId) = 0;
```

Figura 3.16: Interface "ICommunication" do *framework* NGFS.

3.4.1.3 Canais de comunicação

Foi pensando em canais de comunicação, nos quais há um ou vários transmissores e, assim como um ou vários receptores, além de um meio de transmissão é que a interface "IChannel" do *framework* NGFS foi elaborada, compondo as interfaces "ILoader" e "ICommunication".

Melhor explicando, se pensar nos transmissores como sendo os componentes que

estão enviando uma mensagem em um determinado instante de tempo e sendo os componentes que receberão essas mensagens os receptores, sendo isto feito pela passagem do endereço ao qual a mensagem está armazenado na memória do computador. Pode-se pensar que trata-se de um tipo de canal de comunicação entre estes componentes.

Indo um pouco além disso, e se os participantes deste canal de comunicação podem entrar e sair dele a qualquer momento? É necessário então fornecer mecanismos para que estes componentes entrem e saiam do canal.

É a partir de ideias assim que a abstração "IChannel" surgiu, sendo ela a composição das outras interfaces que já fornecem os mecanismos que seriam necessários para o funcionamento desta ideia, ou seja, "entrar/sair" do canal, "transmitir" mensagens a partir dos "transmissores" para os "receptores".

O uso desta abstração tem por objetivo a simplificação no uso das funcionalidades relacionadas ao carregamento/descarregamento dos componentes do sistema e a comunicação entre eles.

Desta forma, foram implementados canais para serviços, para gerentes e para "sistemas de arquivos".

O canal para serviços, possui um comportamento especial, já que ao realizar o carregamento de um serviço, automaticamente o *proxy* correspondente será buscado no caminho da pasta referenciada pela propriedade "ngfs.proxyDir". Esta busca é simples, por exemplo, para o carregamento de um serviço "ngfs_time_service" a busca na pasta indicada por "ngfs.proxyDir" será por um arquivo com nome "ngfs_time_proxy'. Caso o arquivo do *proxy* correspondente não seja encontrado, o serviço será carregado e poderá operar normalmente, porém, ele não será acessível pelos demais componentes da aplicação, já que este acesso é feito apenas via *proxy*. Desta forma, este serviço responderá apenas por solitações que chegam de outro nó da rede, sendo funcionamento útil para aplicações do tipo servidor, já que neste caso, pode-se construir uma aplicação que na verdade faz apenas o carregamento dos componentes de serviço, não sendo necessário os demais componentes, pois apenas requisições externas são esperadas.

O canal para gerentes é um canal compartilhado por todos, ou seja, todos os componentes e a aplicação podem acessá-lo, além disso, é possível que componentes do

tipo gerente entrem e saiam dele. Este canal foi pensado para que houvesse um mecanismo de comunicação comum no *framework* servindo, por exemplo, para que mensagens de aviso de encerramento da aplicação chegue para todos. Em relação aos gerentes, estes serão explicados na seção 3.4.2.3, mas são componentes que gerenciam algum tipo de recurso, como banco de dados, processos em execução, entre outros.

O canal de sistema de arquivos permite que vários componentes do tipo "sistema de arquivos" sejam carregados e se comuniquem entre si. Este canal foi pensado para permitir que uma aplicação carregue diferentes sistemas de arquivos ao mesmo tempo e que dependendo da implementação que estes sistemas possuem, seja possível uma colaboração entre eles, através da comunicação no canal.

Deve ser esclarecido que o canal de serviços é de propriedade de um componente do tipo "sistema de arquivos", ou seja, cada sistema de arquivos pode carregar seus próprios serviços e *proxies*, com a comunicação ocorrendo entre eles apenas. Já o canal de sistema de arquivos é de propriedade da aplicação, ou seja, apenas a aplicação tem capacidade para carregar diferentes sistemas de arquivos e se comunicar com eles, não tendo acesso direto aos serviços ou *proxies* carregados por um sistema de arquivos. Por fim, o mesmo canal de gerentes está disponível para todos os outros componentes e, assim, todos os componentes podem fazer uso dos gerentes e trocar mensagens entre os participantes deste canal.

3.4.2 Componentes Dinâmicos

Os componentes dinâmicos do *framework* são os componentes que herdam da interface "ILoadableComponent" e, assim, podem ser carregados e descarregados em tempo de execução.

A ideia para a implementação destes componentes surgiu a partir dos estudos realizados sobre o padrão de projeto *plugin*, pelo qual é possível desenvolver componentes independentes que podem ser "plugados" ao sistema, fornecendo funcionalidades adicionais para ele.

Os componentes deste tipo, também são os que transformam este *framework* em um sistema de arquivos distribuído. Afinal, os componentes de serviços, *proxies* e

"sistemas de arquivos" são deste tipo.

Estes componentes são gerenciados pelo carregador de componentes do sistema e podem se comunicar uns com os outros através do sistema de comunicação entre componentes do canal do qual está participando.

A Figura 3.17 mostra todas as interfaces disponíveis para a criação de componentes dinâmicos usando o frmaework NGFS. É possível perceber a grande quantidade de interfaces que estão disponíveis, graças ao nível de granularidade que foi dado para os serviços do sistema e, consequentemente, dos *proxies*, já que a proporção dada foi de um *proxy* para cada serviço.

Ao todo, são definidos 9 tipos de serviço, sendo eles: Roteamento, descoberta, gerenciamento, segurança, configuração, tempo, dados, metadados e réplica. Estas são as funcionalidades que foram definidas como necessárias para a implementação de um sistema de arquivos distribuído, sendo possível atender a diversos cenários de uso, a partir da implementação de diferentes tipos de serviço.

A decisão por usar a proporção um para um em relação aos *proxies* foi pensando na possibilidade de haver especificidades que cada tipo de serviço poderia ter. Além disso, estes são componentes-chave para o *framework*, pois implementam um mecanismo de redirecionamento de chamadas do nó local para um nó remoto. Desta forma, são os *proxies* que transformam os sistemas desenvolvidos com este *framework* em sistemas distribuídos.

Outro componente também muito importante para este *framework*, são os componentes do tipo "sistema de arquivos". Estes componentes são os responsáveis por implementar toda a lógica por trás de um sistema de arquivos, fazendo uso dos *proxies* que dão acesso aos serviços do sistema.

A Figura 3.13 mostrou os principais métodos da interface "ILoadableComponent". Lá foi mostrado que existem métodos como "registerMethod", "containsMethod" e "invokeMethod". Estes métodos são a implementação muito simples de um mecanismo de *"reflection"* no *framework*.

O mecanismo de *"reflection"* é um mecanismo comum em linguagens de alto nível como o Java e possibilita que um programa seja capaz de obter informações sobre a sua

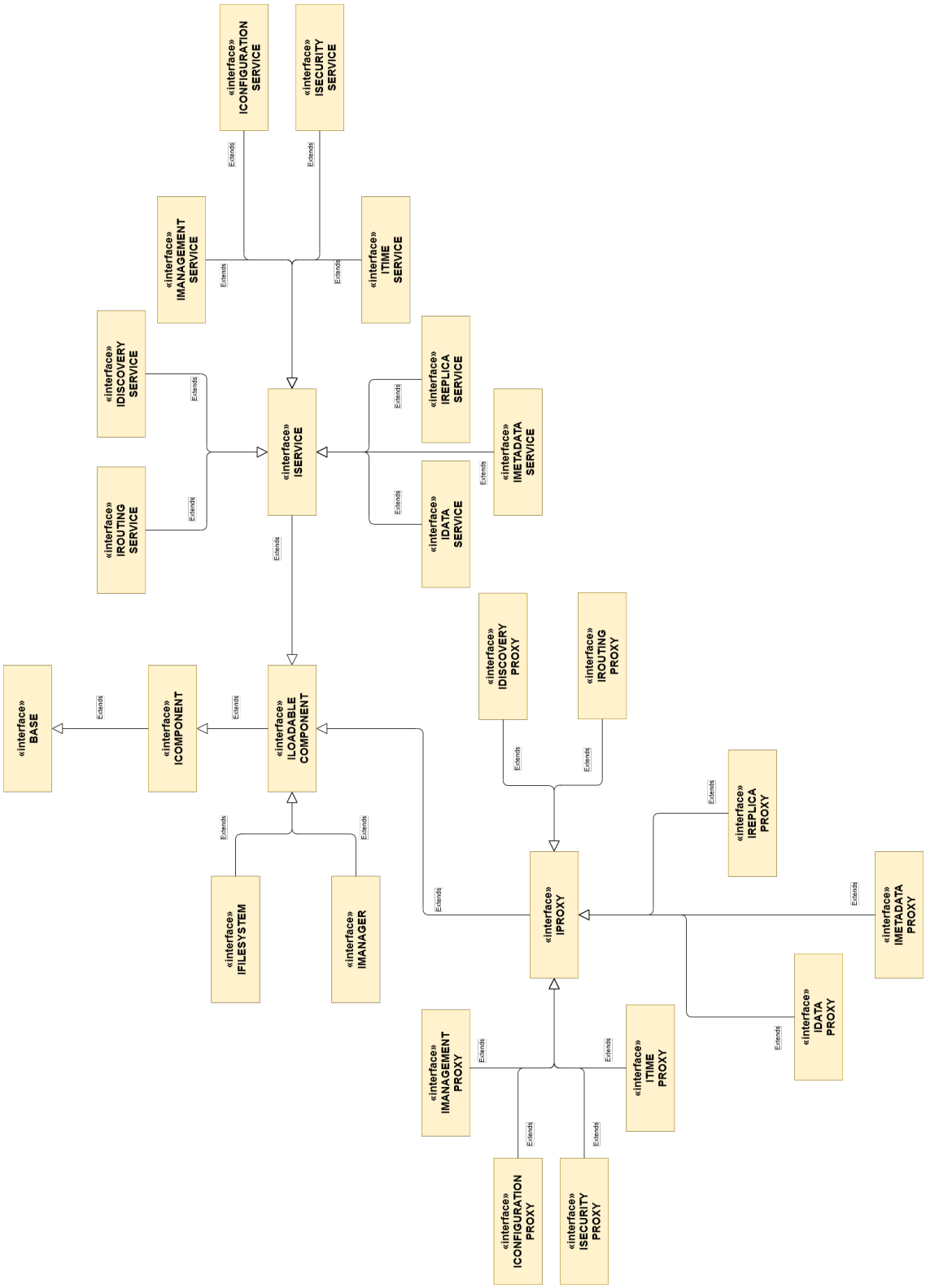

Figura 3.17: Interfaces disponíveis no framework NGFS para a especialização de componentes dinâmicos.

própria estrutura, como quais campos e métodos possui, os tipos de parâmetros, entre várias outras características. Por isso, o nome "reflexão", que o dicionário[4] define como: "Ação ou efeito de refletir(-se); repercussão; Ato de pensar o próprio pensamento; ato do conhecimento que se volta sobre si mesmo, tendo como objeto seu próprio ato".

Este mecanismo que foi implementado no *framework* faz uso da classe presente no cabeçalho "functional.h" chamada "std::function". Esta classe funciona como um *wrapper* para qualquer tipo de elemento executável como funções e expressões lambda.

Fazendo uso desta classe, foi implementado um mapa que armazena o protótipo de funções, sendo possível especificar o seu retorno e os seus parâmetros. Sendo o nome da função utilizado como chave do protótipo armazenado no mapa. Assim, a partir do nome da função e da passagem de parâmetros, é possível invocá-la.

Este mecanismo foi desenvolvido pensando na possibilidade de que a API fornecida pelos componentes dinâmicos fosse também dinamicamente extensível, ou seja, é possível que um determinado componente exporte as funções que deseja expôr além da API previamente definida em sua interface. Outro componente, tendo conhecimento do nome e dos parâmetros necessários para invocar estas funções, pode fazê-la.

Desta forma, todos os componentes dinâmicos deste *framework* possuem esta característica de possuírem uma API padrão, mas que se assim desejarem, podem fazer sua extensão a partir deste mecanismo simples de *reflection* que foi implementado.

3.4.2.1 Service

Os serviços do *framework* são os componentes dinâmicos responsáveis por fornecer toda a funcionalidade esperada de um sistema de arquivos, ou seja, é através deles que os mecanismos de dados, metadados, replica, entre outros, devem ser implementados.

Ao todo, são 9 os tipos de serviços que possuem interfaces definidas no *framework* para especialização: Roteamento, descoberta, gerenciamento, segurança, configuração, tempo, dados, metadados e réplica.

O serviço de roteamento foi pensado para cenários onde a rede disponível não provê este serviço. Imagina-se, por exemplo, cenários de redes ad hoc, onde não há

[4]http://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/reflex%C3%A3o/

qualquer infraestrutura pré-existente e nem uma administração centralizada, tendo cada nó a responsabilidade de atuar como um roteador da rede. Assim, se deseja-se desenvolver um sistema de arquivos distribuído para redes ad hoc, deve-se pensar no roteamento que será feito nesta rede. Daí a presença de uma interface para implementação de serviços de roteamento neste *framework*.

Como o *framework* NGFS possibilita o desenvolvimento de sistemas de arquivos distribuídos que são muito dinâmicos, poderia haver dificuldades para configurar os componentes do sistema. Afinal, é possível configurar que um determinado nó é um servidor de metadados, mas este nó só vai ser um nó de metadados se o serviço correspondente estiver carregado no momento. Pensando desta forma, em alguns cenários, onde pode-se ter servidores mudando de configuração dinamicamente, ou seja, alterando os componentes que estão atualmente carregados, é necessário um serviço de descoberta que seja capaz de informar quais endereços da rede possuem serviços daquele tipo. Daí a presença de uma interface para a implementação de serviços de descoberta no *framework*.

Os serviços de gerenciamento e configuração, são serviços voltados para a administração remota do sistema. O serviço de gerenciamento tem como objetivo fornecer informações de estado de um nó como, por exemplo, quais componentes estão atualmente carregados nele. Já o serviço de configuração tem como objetivo obter configurações necessárias para o funcionamento adequado do nó do sistema, ou seja, deve obter arquivos de configuração, componentes para serem instalados localmente, entre outros.

No momento, a interface disponível para a implementação de serviços de segurança é voltada exclusivamente para a implementação de serviços de autenticação de usuários. Mas, no futuro, pode-se pensar em diversos tipos de serviços de segurança que poderiam ser implementados aqui.

O serviço de tempo ganha um papel importante se pensar-se que a sincronização de relógios é algo essencial em um sistema distribuído, podendo afetar, por exemplo, até mecanismos de segurança que usam como referência o tempo. É devido a isto, que foi pensada uma interface exclusiva para a implementação de serviços de tempo, servindo para que os relógios dos nós estejam sempre sincronizados em relação a um mesmo referencial.

A tríade de interfaces que servem para a implementação de serviços de dados,

metadados e réplica são os serviços essenciais para que seja possível a construção de sistemas de arquivos utilizando este *framework*.

Por fim, além destas interfaces de serviço, ainda há interfaces alternativas que já possuem as implementações necessárias para o uso dos protocolos TCP e UDP. A Figura mostra um exemplo dessas interfaces alternativas para a construção de serviços de tempo que usam o protocolo TCP ou protocolo UDP, através da especialização das interfaces "ITCPTimeService" e "IUDPTimeService", respectivamente.

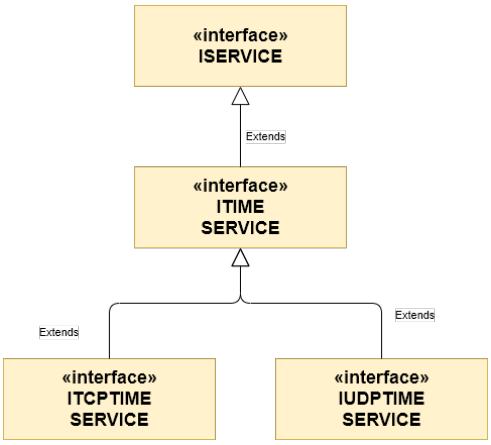

Figura 3.18: Interfaces alternativas para a implementação de serviços de tempo usando o protocolo TCP ou UDP.

3.4.2.2 Proxy

Os *proxies* são componentes dinâmicos que fazem o controle de acesso aos serviços do sistema. A ideia para a implementação destes componentes surgiu a partir dos estudos sobre o padrão de projeto que possui o mesmo nome do componente, ou seja, *proxy*.

O maior desafio encontrado no desenvolvimento deste *framework* foi relacionado às seguintes questões: "Como se comunicar com os componentes que não estão carregados localmente, mas sim em outro nó do sistema?", "Como fornecer um mecanismo amigável ao desenvolver que está fazendo uso do *framework* de comunicação entre os componentes distribuídos?", sendo a resposta para isto dada pelo uso dos *proxies*, pois são mecanismos que resolvem o problema de comunicação entre componentes que se encontram distribuídos e são um mecanismo amigável, já que para o desenvolvedor, suas chamadas estão ocorrendo todas localmente, sendo toda a complexidade do processo abstraída pelos

proxies.

Pelo padrão de projeto *proxy*, este deve implementar a mesma interface do objeto real, sendo neste caso, um serviço. Desta maneira, para quem faz as chamadas, nada muda. Em sua implementação, o *proxy* pode simplesmente reencaminhar a chamada para o objeto real ou fornecer alguma funcionalidade extra antes de fazer este encaminhamento. É neste ponto que a sua utilização começa a ser bem interessante. Pode-se, por exemplo, fazer o controle de acesso ao serviço através dos *proxies*, fazendo a verificação de permissões quando uma chamada for realizada ou ainda fazer o cache dos resultados de algumas operações de dados e metadados. Por fim, pode-se pensar no caso do *proxy* remoto, que é o caso da implementação dos *proxies* deste *framework*.

A implementação de um *proxy* deste *framework* funciona como um *wrapper* de serviço. Quando um serviço é carregado pelo canal de serviços, este também tentará carregar o *proxy* correspondente da forma já explicada na seção 3.4.1.3. Caso seja encontrado, será carregado e a instância do serviço será passada para este *proxy*. O inverso ocorrerá se o serviço for descarregado. Também é possível fazer o carregamento direto do *proxy*, sem que um serviço correspondente esteja disponível.

Quando um sistema de arquivos realizar chamadas a um serviço, que na verdade estará sendo realizada a um *proxy*, este irá verificar se possui uma instância do serviço ou não. Caso tenha, ele fará o simples redirecionamento da chamada ao serviço e retornará a resposta ao sistema de arquivos. Caso contrário, a chamada deverá ser reencaminhada para um serviço disponível na rede.

Este processo de redirecionamento via rede já se encontra implementado na interface "IProxy" do *framework*, no método *forwardRequest()*. Desta maneira, todos os *proxies* suportam este redirecionamento.

O Algoritmo 1, Algoritmo 2 e Algoritmo 3 mostram o pseudo-código das operações envolvidas nesta tarefa de redirecionamento de requisições realizadas pelos *proxies*.

A primeira tarefa é solicitar aos serviços de descoberta os endereços dos serviços que atendem à requisição solicitada. Uma função auxiliar é chamada para requisitar aos serviços de descoberta os endereços dos serviços que atendem esta solicitação. Com a lista formada, é feita a requisição para um dos serviços, se houver resposta, ela será retornada

imediatamente. Caso contrário, novas requisições serão feitas até que um dos serviços respondam ou então, se nenhum responder, uma mensagem de erro será retornada.

É importante destacar que o comportamento do reencaminhamento é diferente devido às peculiaridades de cada protocolo de transporte, sendo eles o TCP e o UDP. Como o TCP é orientado à conexão e possui alguns tipos de garantia, pode-se conectar à outra extremidade, obter a resposta e retornar. Já no caso do UDP, não há uma conexão e não há garantias de que a mensagem chegou na outra extremidade. Desta forma, se faz necessário aguardar por uma resposta e até mesmo fazer o reenvio da mensagem.

São tarefas que parecem simples, porém, exigem que vários componentes do sistema se comuniquem para que tudo isso seja possível de acontecer. Desta forma, é possível considerar que os *proxies* são uma solução elegante para um problema complexo.

Algoritmo 1: Fazer encaminhamento de requisição

```
Entrada: Mensagem de requisição(request) e protocolo de
         encaminhamento (protocol)
Saída: Mensagem de resposta retornada pelo serviço remoto
1  início
2  |  obter tipo de serviço da request;
3  |  obter lista de endereços dos serviços remotos;
4  |  para cada remoteAddress faça
5  |  |  response = reencaminhar request para remoteAddress usando
   |  |  protocol;
6  |  |  se response = ok então
7  |  |  |  retornar response;
8  |  |  fim se
9  |  fim para
10 |  retornar erro;
11 fim
```

Algoritmo 2: Obter lista de endereços de serviço

```
Entrada: Tipo de serviço (serviceType)
Saída: Lista de endereços dos serviços remotos
1  início
2      addressList = vazio;
3      obter lista de endereços dos serviços de descoberta;
4      para cada discoveryAddress de serviço de descoberta faça
5          response = requisitar para discoveryAddress endereços de
           serviceType;
6          se response = ok então
7              adicionar endereços a addressList;
8          fim se
9      fim para
10     retornar addressList;
11 fim
```

Algoritmo 3: Reencaminhar requisição

```
Entrada: Endereço do serviço remoto (remoteAddress), Mensagem de
         requisição(request) e protocolo de encaminhamento (protocol)
Saída: Mensagem de resposta retornada pelo serviço remoto
1  início
2      se protocol = tcp então
3          response = reencaminhar request para remoteAddress usando
           TCP;
4          retornar response ;
5      senão
6          porta_udp = escolher porta UDP disponível;
7          tentativas = 0;
8          timeout = 30;
9          escutar na porta_udp;
10         para tentativas < 3 faça
11             reencaminhar request para remoteAddress usando UDP;
12             aguardar no máximo timeout;
13             se response = ok então
14                 retornar response ;
15             senão
16                 tentativas = tentativas + 1;
17             fim se
18         fim para
19         retornar erro;
20     fim se
21 fim
```

3.4.2.3 Manager

Os gerentes são componentes dinâmicos pensados para a necessidade de se compartilhar recursos entre os demais componentes. Por exemplo, pode-se pensar em um acesso a banco de dados, onde, ao invés de ter-se vários acessos ao banco vindo de cada componente da aplicação, um único componente faria todo o gerenciamento do acesso ao banco de dados. É daí que surge a ideia de um componente compartilhado por todos.

No momento atual, apenas três componentes do tipo gerente foram implementados no *framework*, servindo como uma prova do conceito. Estes componentes são o "ProcessManager", "PackageManager" e "PasswordManager".

O "ProcessManager" é responsável pelo monitorar os processos que estão em execução no sistema operacional. Por exemplo, é possível iniciar processos, obter a lista de processos e encerrar um processo. Este componente é utilizado pela aplicação "NgfsInit" que é uma aplicação do tipo servidora que foi implementada com o intuito de dar partida nas demais aplicações que fazem parte de um sistema implementado com o *framework* NGFS. Desta forma, através do "ProcessManager", esta aplicação pode monitorar se as aplicações do sistema estão em execução ou se forem encerradas, reiniciá-las, mantendo assim o sistema em operação.

O "PackageManager" é o responsável pelo gerenciamento dos pacotes de instalação do sistema. Inspirado nos pacotes DEB do sistema operacional *Debian* o *framework* NGFS faz a implementação de um tipo de pacote para a distribuição de componentes e arquivos de configuração. Estes pacotes são arquivos compactados, criptografados e assinados digitalmente. Internamente, eles possuem um descritor que contém informações como nome, tamanho, caminho e hash dos arquivos contidos no pacote. Por sua vez, este descritor é assinado digitalmente. Assim, qualquer modificação no pacote, a assinatura não será mais válida e, portanto, recusada pelo "PackageManager". Esta também foi a solução dada para manter a segurança dos componentes que são utilizados pelas aplicações. É esperado que apenas componentes que foram certificados, sejam usados pelas aplicações.

Por fim, o "PasswordManager" foi pensado para servir como um cofre virtual, podendo gerar senhas aleatórias e guardar senhas que podem ser acessadas por uma chave-

mestra. Como sua segurança não foi atestada, seu uso atual é pouco pelos componentes do *framework*, servindo mais para a geração de senhas aleatórias.

3.4.2.4 Filesystem

Os componentes do tipo *filesystem* foram pensados para retirar da aplicação a responsabilidade de implementar a lógica do sistema de arquivos. Além disso, com a implementação da lógica do sistema de arquivos em um componente dinâmico, pode-se pensar não somente no reúso por diversas aplicações, mas também no cenário de uma aplicação fazendo uso de vários sistemas de arquivos ao mesmo tempo.

O cenário de uma aplicação fazendo uso de vários sistemas de arquivos é possível graças a existência dos componentes *filesystem*. Estes componentes expõem uma API no padrão POSIX, ou seja, define-se operações como:

- mount, umount: Inicializar/desinicializar o sistema de arquivos;
- open, close: Operações que a criação de metadados e também para operações de *lock*/*unlock*;
- stat: Operação para a obtenção de informações sobre um metadado;
- write, read: Operações de leitura e escrita de dados;
- mkdir, rmdir: Operações de criação e remoção de diretórios;
- link, unlink: Operações de ligação entre caminhos;
- chmod, chown, chgrp: Operações relacionadas à permissões de arquivos.

Algumas operações extras também foram colocadas na API dos componentes *filesystem*, como "statfs()" para obter informações sobre o estado do sistema de arquivos e "sync()" para forçar que o sistema de arquivos faça a sincronização dos dados, caso a implementação do sistema de arquivos faça uso de cache.

Todas estas operações do componente *filesystem* devem ser implementadas através do uso dos *proxies* que dão acesso aos *serviços* do sistema. No caso de todos os componentes estarem disponíveis localmente, deixa-se de ter um sistema de arquivos distribuído

e passa-se a ter um sistema de arquivos local, já que todas as chamadas serão executadas localmente. O inverso também pode ocorrer, caso se trate de em um hardware com baixas capacidades de processamento, memória ou algo parecido, pode-se ter somente a aplicação, o *filesystem* e os *proxies*, fazendo com que todas as chamadas realizadas sejam reencaminhadas para nós remotos do sistema.

A utilidade do canal de *filesystem*, é a de permitir que a aplicação seja capaz de carregar vários sistemas de arquivos ao mesmo tempo e descarregá-los quando necessário. É pensado também no cenário em que através da troca de mensagens neste canal, se a implementação dos *filesystems* permitirem, ocorra uma colaboração entre eles.

Por fim, o *framework* NGFS não fornece nenhuma implementação padrão de *filesystem*, apesar dos sistemas *"Storage filesystem"* e *"Music filesystem"* terem sido implementados como prova de conceito do *framework*. Maiores detalhes sobre estes sistemas de arquivos se encontram no Capítulo 4.

3.5 Aplicações

Por fim, tem-se a última parte que envolve o *framework* NGFS. Durante o seu projeto, foi questionado se seria necessário ou não fornecer uma camada de desenvolvimento de aplicações que fazem uso dos mecanismos descritos nas seções anteriores.

Chegou-se então na conclusão de que fornecer uma camada deste tipo, traria benefícios para o desenvolvedor que fizesse uso do *framework*, pois seria mais uma facilidade disponibilizada por ele, já que as aplicações desenvolvidas usando as suas interfaces de aplicação chamadas de "IApplication", "IServerApplication" e "IClientApplication" já estariam preparadas para lidar com todos os mecanismos presentes no *framework*. Diminuindo assim ainda mais o tempo de desenvolvimento de sistemas de arquivos distribuídos.

Outro fator que foi um incentivo para o desenvolvimento destas interfaces, foram as classes "Poco::Util::Application"[5] e "Poco::Util::ServerApplication"[6]. Estas classes possuem algumas facilidades que são interessantes para as aplicações, como a criação e tratamento de parâmetros de entrada de uma maneira fácil. Assim, estes recursos foram

[5]https://pocoproject.org/docs/Poco.Util.Application.html
[6]https://pocoproject.org/docs/Poco.Util.ServerApplication.html

incorporados às interfaces de aplicação.

Outro recurso interessante, mas específico das aplicações servidoras, é que a classe "Poco::Util::ServerApplication" já possui toda a implementação necessária para que a aplicação seja executada como um serviço no sistema operacional Windows ou como um daemon no sistema operacional Linux.

Todos os recursos presentes nas classes da biblioteca POCO para aplicações estão disponíveis para as aplicações criadas utilizando as interfaces de aplicação do *framework* NGFS, pois são derivadas das classes da POCO. Porém, além destes recursos, entregam os mecanismos pensados para as aplicações do *framework*, ou seja, possuem acesso a um canal de *filesystem* e ao canal comum. Desta forma, a implementação de aplicações usando estas interfaces, possuem as características de configuração e *logging* herdadas da interface "Base", possuem os recursos disponíveis das classes "Poco::Util::Application" e "Poco::Util::ServerApplication" da biblioteca POCO e, por fim, possuem a capacidade de fazerem uso de sistemas de arquivos implementados pelos componentes *filesystem* que podem ser carregados/descarregados em tempo de execução através do canal de *filesystem* e, por fim, ainda podem fazer o carregamento de componentes do tipo gerente.

3.5.1 Servidores

As aplicações servidoras são as aplicações implementadas fazendo uso da interface "IServerApplication", tendo o seu propósito de funcionamento como sendo um serviço do sistema operacional em questão.

Devido às características do *framework* NGFS, pode-se implementar aplicações servidoras que nada mais fazem do que se instalarem como serviços do sistema operacional, realizarem o carregamento dos serviços e só. Este é exatamente o funcionamento de uma das aplicações deste tipo que foram implementadas no *framework*, sendo chamadas de "NgfsServer".

O outro tipo de aplicação, possui a função de inicializar as demais aplicações do sistema, sendo chamada "NgfsInit". Esta aplicação também funciona como um serviço do sistema operacional e possui como única função, dar início nas demais aplicações e garantir que estão sempre em funcionamento. Caso não estejam, fazem o reinício da

aplicação que foi encerrada. Para todo este gerenciamento, a aplicação "NgfsInit" faz uso do gerente de processos "ProcessManager".

3.5.2 Console

A interface "IConsoleApplication" é destinada para a criação da maioria das aplicações voltadas ao usuário final, como as aplicações clientes dos sistemas de arquivos distribuídos.

Desta maneira, foi desenvolvida a aplicação "NgfsClient" que funciona como um cliente genérico para sistemas de arquivos distribuídos implementados usando o *framework* NGFS. O funcionamento é genérico, pois devido à arquitetura do *framework*, pode-se fazer com que o cliente carregue apenas o componente do tipo *filesystem* e os *proxies*, mantendo os serviços distribuídos na rede. Desta maneira, todas as chamadas que a aplicação fizer para o sistema de arquivos, serão reencaminhadas para destinos remotos.

Assim, o diferencial entre o cliente do sistema de arquivos "Storage filesystem" para o "Music filesystem" das provas de conceito do *framework*, é que no primeiro caso é carregado um tipo de componente *filesystem* que contém a lógica daquele sistema de arquivos, já no outro caso é carregado um componente *filesystem* que contém a lógica do outro sistema de arquivos, nada mais do que isso.

4 Provas de conceito

As provas de conceito são implementações de sistemas de arquivos distribuídos que fazem o uso do *framework*, mostrando que é possível especializá-lo para diferentes cenários de uso, com a facilidade promovida pelo reúso dos componentes do *framework* e/ou algumas novas especializações. Para isto, dois cenários de uso foram escolhidos, sendo eles o de um serviço de armazenamento de arquivos em nuvem e um serviço de compartilhamento de músicas *peer-to-peer* (P2P).

O cenário de um serviço de armazenamento de arquivos em nuvem tem o seu funcionamento baseado em serviços como Google Drive, Microsoft OneDrive, Apple iCloud, etc., nos quais há múltiplos usuários, cada um possuindo uma cota de espaço de armazenamento e que através de uma aplicação cliente é possível a realização de operações sobre diretórios e arquivos. Desta forma, o cenário é ideal para se testar as partes do *framework* relacionadas à segurança como a autenticação, autorização, confidencialidade, além do próprio armazenamento de dados e seus metadados.

Já o serviço de compartilhamento de músicas possui inspiração em redes *peer-to-peer* que utilizam protocolos como o BitTorrent. Desta forma, os nós do sistema são tanto clientes como servidores e atuam para que a disponibilidade de um arquivo se torne maior através da replicação do mesmo para seus pares conectados. Além disso, a obtenção de um arquivo pode ser feita de maneira mais rápida, pois o arquivo pode ser obtido em partes vindas de diferentes pares e de forma simultânea. Desta forma, o cenário é ideal para se testar as partes do *framework* relacionadas à descoberta e comunicação em rede, além da replicação, consistência e sincronização dos dados.

4.1 Storage filesystem

O *Storage filesystem* implementa um sistema de arquivos distribuído baseado no GFS(GHEMAWAT; GOBIOFF; LEUNG, 2003), que é utilizado em serviços do Google como, por exemplo, o *Drive*. A Figura 4.1 mostra a arquitetura deste sistema.

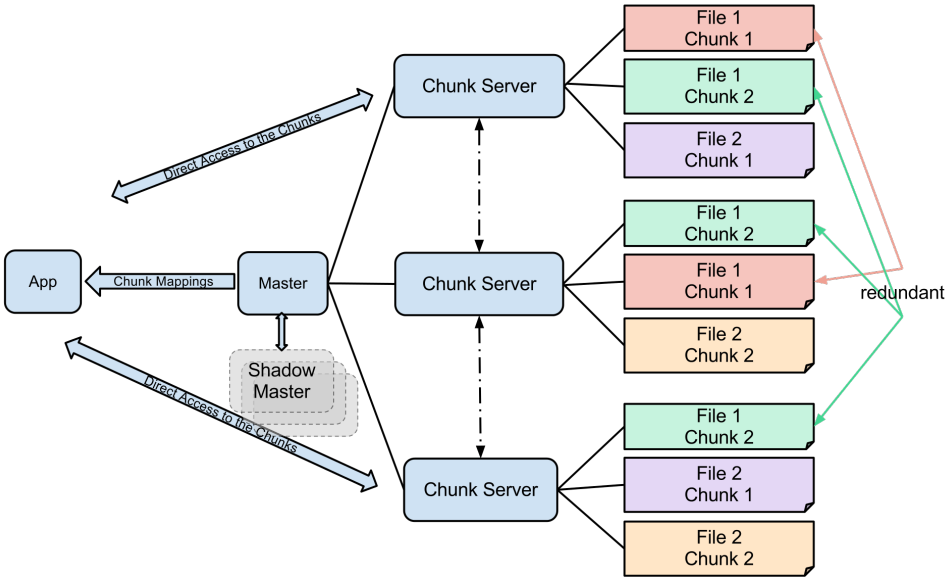

Figura 4.1: Arquitetura do sistema de arquivos GFS.

Neste sistema, os dados são armazenados em blocos de 64 MB que raramente são sobrescritos ou reduzem o seu tamanho, sendo normalmente incrementados ou lidos. Um cluster GFS é o conjunto de nós que podem ser divididos em dois tipos: *Master* e *Chunk Servers*.

Os nós mestres, são os nós de metadados do sistema de arquivos distribuído, desta forma, armazenam apenas metadados e não os dados em si. É sua responsabilidade fazer mapeamento de arquivo para rótulos de blocos e de rótulos de blocos para locais onde eles estão armazenados, que são nos *chunk servers*. Também fazem o controle de quais blocos estão sendo lidos ou gravados e também tomam a iniciativa de fazer réplicas quando o número de cópias cai de uma determinada quantidade.

As aplicações clientes devem primeiro fazer o acesso ao servidor mestre e após obterem a localização dos blocos, devem se comunicar com os *chunk servers* para a transferência dos dados.

Desta forma, tem-se uma arquitetura baseada em *cluster*, com nós assimétricos e com a possibilidade de leitura/escrita de dados em paralelo, além de fazer uso do protocolo TCP para comunicação em rede. Existe um servidor de metadados central e a replicação de dados e metadados é utilizada, pois consideram que a falha é a norma, já que fazem uso de hardware de baixa qualidade. Deve-se destacar que este sistema de arquivos não possui mecanismos dedicados à segurança.

Pensando nestas características do GFS, o *Storage filesystem* foi implementado utilizando o NGFS e pouco foi necessário ser feito, pois os componentes padrões implementados no *framework* já possuem a maioria destas características.

Desta forma, o *framework* NGFS já possui por padrão um serviço de dados que faz uso de comunicação TCP e que armazena os dados em blocos. Por sua vez, pode-se utilizar também o serviço de metadados padrão do *framework* para o servidor central de metadados.

Em relação à replicação, há um diferencial. Apesar do componente padrão disponível no *framework* atender ao requisito de replicação de metadados e dados, ele não possui o mesmo comportamento que está presente no GFS. Enquanto no GFS o nó mestre é que coordena todo o processo de replicação, no caso do *Storage filesystem* o componente de replicação deve ser carregado pelos nós de metadados e dados. Desta forma, cada nó será responsável por replicar seus dados ou metados para os demais pares.

Por fim, o maior trabalho de implementação no *Storage filesystem* foi a implementação de um serviço de autenticação de usuários e também a maior diferença em relação ao GFS, que não faz uso de nenhum sistema do tipo.

Desta forma, no *Storage filesystem* há a implementação de um serviço de autenticação de usuário baseado em token que faz uso de um banco de dados *MySQL*. Além disso, foi criado um filtro de entrada com autenticação para o servidor TCP dos serviços. Desta maneira, ao receber uma conexão o filtro de entrada solicita o token de autenticação e faz a validação com o serviço de autenticação. Caso o usuário esteja autorizado, a conexão é aceita pelo servidor, caso contrário, é recusada.

Com este cenário, é possível perceber o quanto o *framework* pode ajudar no desenvolvimento ágil de sistemas de arquivos distribuídos. Contando com componentes já disponíveis no *framework*, foi necessário apenas a implementação de um serviço de autenticação de usuários e um filtro de entrada para a validação do token de acesso.

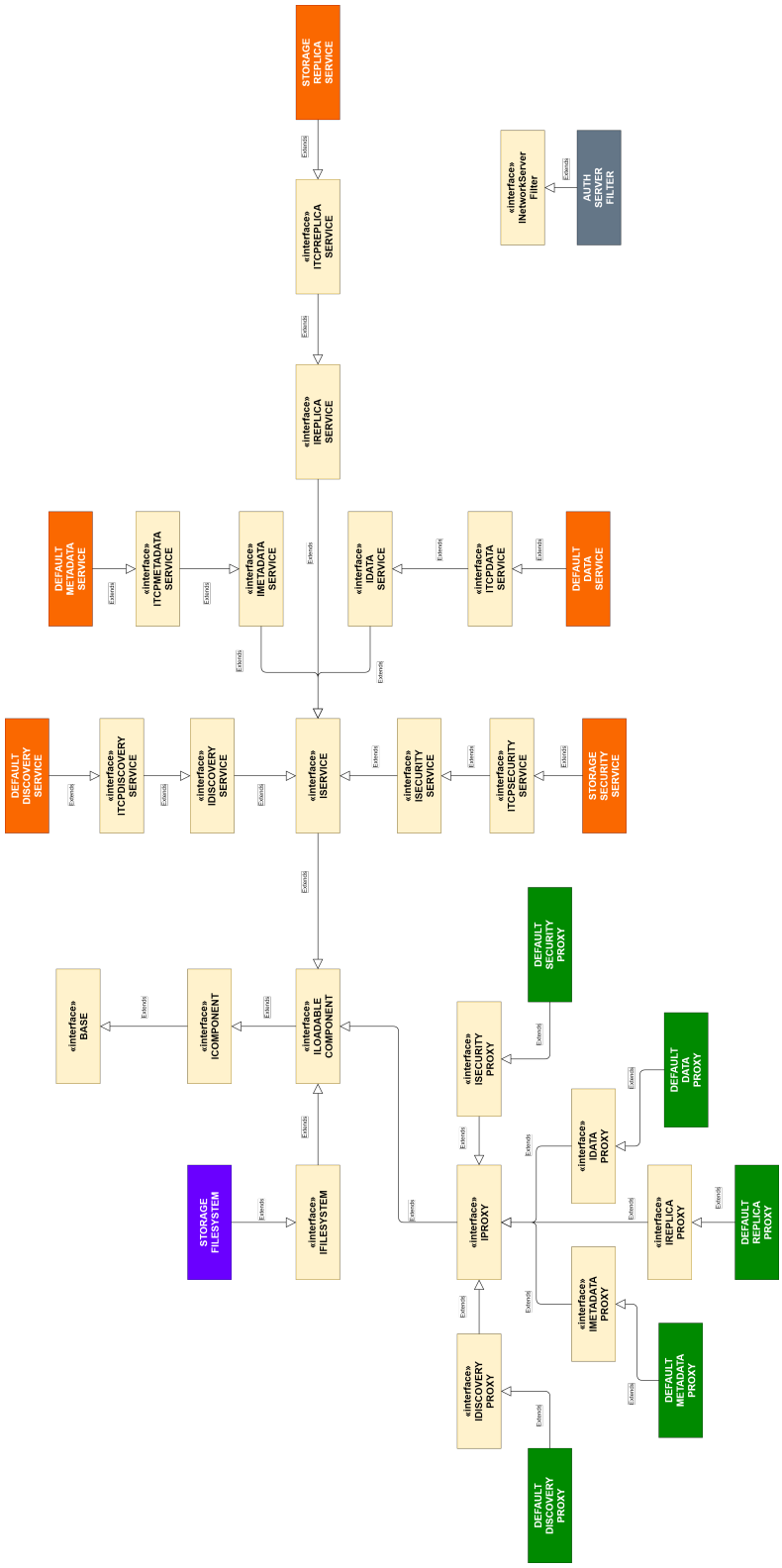

Figura 4.2: Principais componentes utilizados no *Storage Filesystem*. Os componentes com a palavra "*Default*" no nome são componentes já presentes no *framework* NGFS, já os que contém a palavra "*Storage*" indicam componentes que foram especializados para este sistema de arquivos distribuído.

4.2 Music filesystem

A motivação para a criação do sistema de arquivos Music filesystem surgiu a partir de trabalhos como (KOUYOUMDJIEVA; KARLSSON, 2016), em que é proposto o compartilhamento de músicas através de uma rede ad hoc oportunista e (KELLER; LE; CICI, 2012), em que é proposto o streaming de conteúdo de vídeo em um esquema também oportunista.

Em (KOUYOUMDJIEVA; KARLSSON, 2016), é destacado a previsão de que até 2020, boa parte do tráfego na Internet será de conteúdo multimídia. Desta forma, surge a necessidade de se explorar novas possibilidades para reduzir este tráfego, sendo uma delas a disseminação deste conteúdo de dispositivo para dispositivo, ou seja, através de uma comunicação oportunista. A escolha pelo conteúdo de músicas se dá devido a necessidade do requerimento de uma baixa taxa de transferência de dados para este tipo de conteúdo, assim, não sobrecarregando os dispositivos e também não prejudicando a experiência dos usuários. É proposto a criação de um sistema de publicação/assinatura, onde os dispositivos devem publicar os conteúdos que possui e fazer a assinatura de categorias de conteúdos desejados. Desta forma, quando os nós se encontram, eles descobrem os conteúdos entre si e baixam o conteúdo desejado. A escolha por fazer download do conteúdo e não streaming é devido a possibilidade de haver interrupção do streaming, caso os nós servidores se afastem dos nós clientes ao ponto de não haver mais nenhuma disponibilidade do conteúdo no momento, prejudicando a experiência do usuário.

Em (KELLER; LE; CICI, 2012), o cenário considerado é de um grupo de usuários, próximos uns dos outros, interessados em assistir ao mesmo vídeo da Internet ao mesmo tempo. No cenário atual da Internet, cada usuário baixaria o vídeo de forma independente, usando sua própria conexão móvel, o que pode levar a uma baixa qualidade. O trabalho propõe então a criação de um sistema chamado "MicroCast", que usa os recursos de smartphones de forma cooperativa, de forma a melhorar a experiência do streaming de vídeo. Este sistema opera de forma a usar duas interfaces de rede do dispositivo, sendo uma a rede móvel e a outra o Wifi. A interface de rede móvel, serve para se conectar ao servidor de vídeo e fazer o download de um determinado trecho do vídeo, enquanto a interface Wifi serve para se conectar ao resto do grupo. Existe um subsistema chamado

"MicroDownload"que se encarrega de decidir e agendar quais partes do vídeo cada telefone deverá baixar do servidor, levando em consideração informações sobre a taxa de download que cada um possui. Além disso, outros dois subsistemas chamados "MicroNC-P2"e "MicroBroadcast"atuam para a entrega dos segmentos do vídeo na rede local, sendo o MicroNC-P2 responsável por definir, por exemplo, a ordenação dos segmentos disponíveis e o MicroBroadcast responsável pela entrega dos segmentos.

Durante uma disciplina da graduação, tendo em vista as tecnologias de redes ad hoc com comunicação oportunista e sistema de arquivos distribuído, foi proposto a criação de um aplicativo de compartilhamento de músicas para o sistema operacional Android fazendo uso da API Nearby Connections para a criação da rede ad hoc e o uso de um sistema de arquivos distribuído para o compartilhamento dos arquivos de música. O nome dado a esse aplicativo foi *"DroidPlayer"* e a Figura 4.3 mostra a sua arquitetura, enquanto a Figura 4.4 mostra a sua interface gráfica.

Os nós deste sistema eram simétricos, ou seja, todos possuíam os serviços de metadados, dados e replicação, além da comunicação em rede e roteamento. Por fim, a interface gráfica do aplicativo se comunicava com o serviço de mídia para a reprodução das músicas. Por sua vez, tanto a interface gráfica quanto o serviço de mídia fazem acesso ao sistema de arquivos distribuído.

Como o NGFS ainda não é um *framework* adaptado para dispositivos móveis, a implementação deste sistema de arquivos usando o framework passou por algumas adaptações. Para simular o comportamento de uma rede ad hoc, um serviço de roteamento especial foi criado. A partir da configuração feita via arquivo de configuração, o serviço sabe com quem deve se comunicar. Porém, durante a sua execução, realiza a troca do par de comunicação de tempos em tempos e de forma aleatória, na tentativa de simular a mobilidade dos dispositivos móveis. Além disso, em alguns momentos não faz conexão com nenhum par, simulando a indisponibilidade de conexão com pares por perto. Por fim, neste cenário, somente o protocolo UDP foi utilizado, justamente por ele não oferecer garantias na entrega das mensagens. Fora isto, nada mais foi necessário, pois os componentes padrões do *framework* já atendiam aos requisitos deste sistema de arquivos.

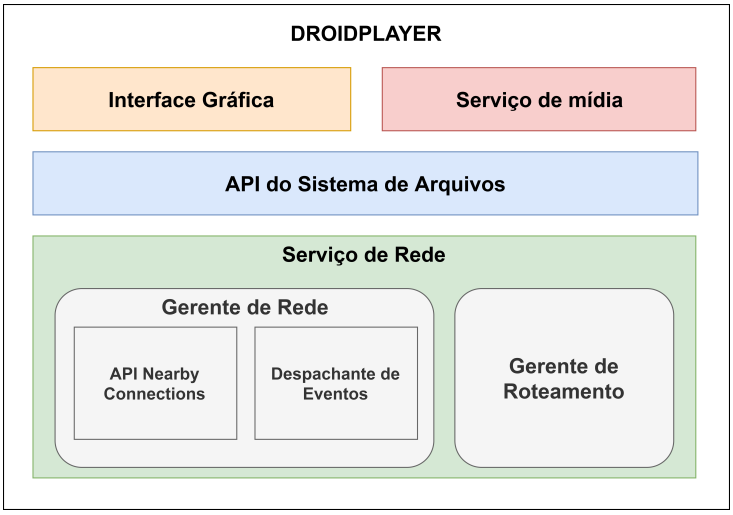

Figura 4.3: Arquitetura do DroidPlayer.

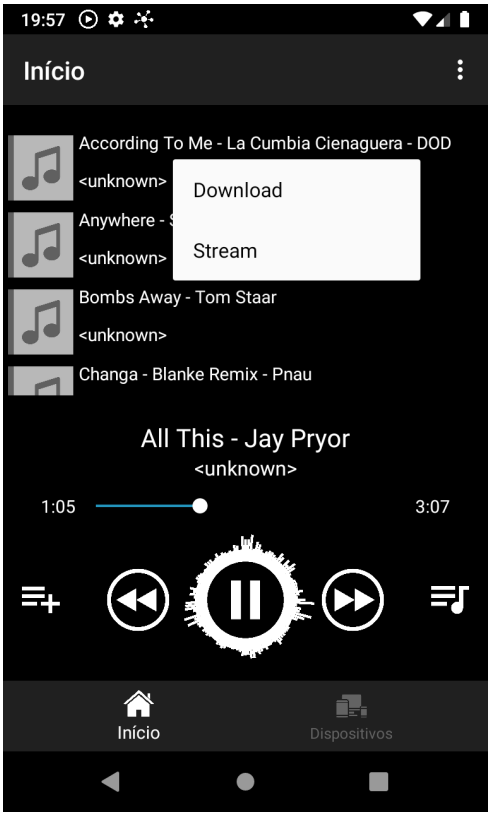

Figura 4.4: Interface gráfica do DroidPlayer.

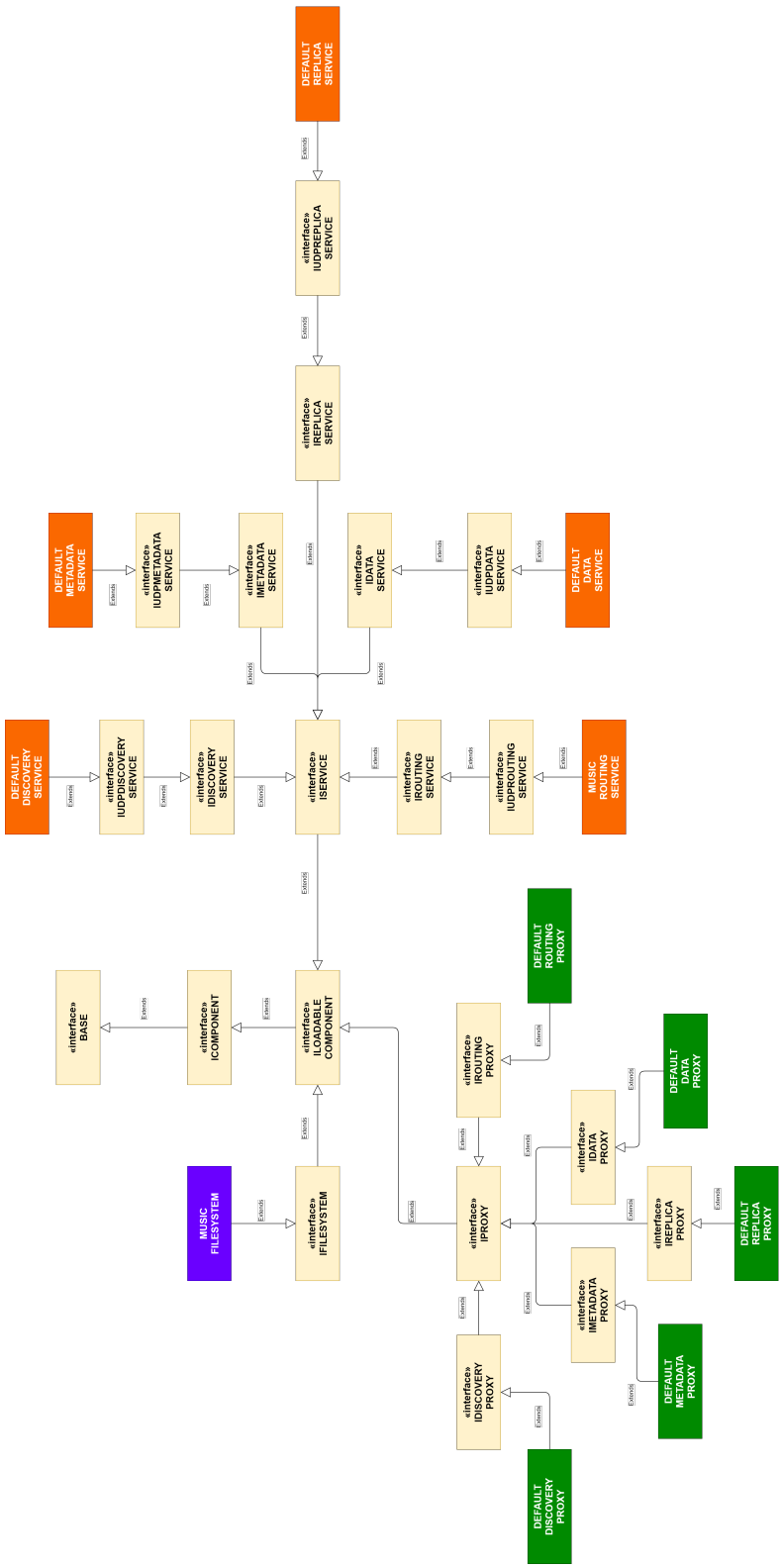

Figura 4.5: Principais componentes utilizados no *Music Filesystem*.

4.3 Experimentos

A fim de se analisar o desempenho das provas de conceito implementadas utilizando o *framework* NGFS, alguns experimentos foram realizados com as mesmas.

Para isto, um ambiente com seis máquinas virtuais foi utilizado, conforme a Tabela 4.1 mostra, todas utilizam o mesmo mecanismo de virtualização, o mesmo sistema operacional, a mesma configuração de CPU, quantidade de memória RAM e espaço de armazenamento. Porém, diferentes configurações de rede foram utilizadas para melhor se aproximar do cenário para qual cada prova de conceito foi pensada.

No caso do *Storage Filesystem*, trata-se de um sistema de arquivos distribuído pensado para um ambiente de nuvem de armazenamento. Desta forma, cinco máquinas foram escolhidas para formarem a nuvem de serviços e uma outra foi escolhida para funcionar como cliente. As máquinas servidoras possuem a mesma configuração de rede, sem grandes variações de latência, na tentativa de simular que estão em um mesmo ambiente físico, interligadas por uma boa infraestrutura de rede. Já a máquina cliente foi configurada com uma latência de rede maior e uma grande variação, na tentativa de simular um conexão não tão boa de acesso à Internet.

No caso do *Music Filesystem*, trata-se de um sistema de arquivos distribuído pensado para dispositivos móveis, fazendo uso de redes ad hoc. Desta forma, é esperado que cada nó do sistema possua uma condição de rede diferente, podendo haver grandes latências e com muita variação. Assim, os nós deste sistema receberam configurações de rede que poderiam ser consideradas boas, medianas ou ruins.

A Tabela 4.2 e a Tabela 4.3 mostram as configurações do ambiente de rede para o cenário de uso do *Storage Filesystem* e para o cenário de uso do *Music Filesystem*, respectivamente. Estas configurações foram realizadas utilizando-se o componente do *kernel* Linux, denominado "netem" (*Network Emulator*)[7]. Este componente possibilita a simulação de diferentes condições de rede como o atraso, a perda, a duplicação e reordenação de pacotes. Ainda em relação às tabelas de configuração do ambiente de rede, é importante destacar o que seriam a variância, correlação e distribuição informados nelas. Como redes reais apresentam variabilidade na propriedade de atraso, o "netem" possibi-

[7]https://www.linux.org/docs/man8/tc-netem.html

lita que atrasos variáveis sejam configurados. Enquanto isso, o valor de correlação indica quanto o próximo valor de atraso depende do resultado obtido anteriormente, desta forma o atraso da rede não se torna puramente aleatório. Por fim, o atraso em uma rede não segue uma distribuição uniforme, sendo comum o uso de uma distribuição normal para descrever o comportamento da variação do atraso de rede.

Virtualização	Sistema Operacional	CPU	RAM	Disco
KVM	Ubuntu Server 20.04.1 LTS	2 x Xeon 64 Bits	4 GB	30 GB
KVM	Ubuntu Server 20.04.1 LTS	2 x Xeon 64 Bits	4 GB	30 GB
KVM	Ubuntu Server 20.04.1 LTS	2 x Xeon 64 Bits	4 GB	30 GB
KVM	Ubuntu Server 20.04.1 LTS	2 x Xeon 64 Bits	4 GB	30 GB
KVM	Ubuntu Server 20.04.1 LTS	2 x Xeon 64 Bits	4 GB	30 GB
KVM	Ubuntu Server 20.04.1 LTS	2 x Xeon 64 Bits	4 GB	30 GB

Tabela 4.1: Configuração das máquinas virtuais utilizadas nos testes das provas de conceito.

Latência	Variância	Correlação	Distribuição
1ms	1ms	25%	Normal
1ms	1ms	25%	Normal
1ms	1ms	25%	Normal
1ms	1ms	25%	Normal
1ms	1ms	25%	Normal
100ms	25ms	25%	Normal

Tabela 4.2: Configuração do ambiente de rede para o cenário do *Storage Filesystem*.

Latência	Variância	Correlação	Distribuição
25ms	2ms	25%	Normal
50ms	5ms	25%	Normal
100ms	10ms	25%	Normal
100ms	10ms	25%	Normal
50ms	5ms	25%	Normal
25ms	2ms	25%	Normal

Tabela 4.3: Configuração do ambiente de rede para o cenário do *Music Filesystem*.

Os experimentos foram planejados para se avaliar as operações mais importantes em cada um dos sistemas de arquivos distribuídos implementados. É natural pensar que as operações mais importantes em um sistema de arquivos sejam as operações de leitura e escrita de arquivos, como no caso do *Storage Filesystem*, no qual estas são as operações mais frequentes. Portanto, para este sistema, os testes realizados foram voltados à obtenção do tempo total para a operação de leitura/escrita de um arquivo de 100 MB,

sendo que este tempo total é composto pelo tempo de verificação das informações de autenticação do cliente, pelo tempo de consulta aos metadados e pelo acesso aos dados para a transferência. Por outro lado, no caso do *Music Filesystem*, pensar apenas nas operações de leitura e escrita é insuficiente. Na forma pela qual este sistema foi implementado, com os nós sendo simétricos, as operações de leitura e escrita são feitas localmente, já que os serviços de metadados e dados estão presentes em cada nó. Desta forma, a operação que se torna mais importante de ser avaliada seria a de replicação. Assim, o teste realizado no sistema *Music Filesystem* é voltada à análise do tempo de replicação de um arquivo de 5 MB, ou seja, o tempo que é gasto até que um arquivo esteja presente em todos os nós do sistema. As Figuras 4.6, 4.7 e 4.8 mostram os resultados obtidos para 100 operações de leitura/escrita no *Storage Filesystem* e 100 operações de replicação no *Music Filesystem*.

Analisando os resultados obtidos nas operações de leitura/escrita no *Storage Filesystem*, tem-se um tempo mínimo de leitura de 12,192 segundos e um máximo de 60,238 segundos, com o tempo médio de 40,018 segundos. Já a operação de escrita teve um tempo mínimo de 36,533 segundos, um tempo máximo de 60,187 segundos e um tempo médio de 49,110 segundos. Desta forma, é possível perceber que a operação de escrita é mais demorada que a de leitura, o que é um comportamento esperado. Também é possível perceber que o tempo mínimo da escrita é, aproximadamente, três vezes maior do que a leitura. Possivelmente, a explicação para isso está relacionada à forma como o sistema foi implementado, na qual o cliente inicialmente quebra o arquivo em blocos, armazenando-os em um cache local para só depois passar a se comunicar com os servidores de metadados e dados para a realização da operação de escrita. Assim, esse tempo mínimo maior pode estar relacionado com esse processamento extra.

Analisando os resultados obtidos na operação de replicação no *Music Filesystem*, observa-se um tempo mínimo de 13,6 segundos e um máximo de 51,3 segundos, com um tempo médio de 30,4 segundos. A grande diferença entre os tempos mínimo e máximo eram esperados, já que o ambiente de rede do cenário deste sistema é muito variável, mas chama a atenção o tempo médio que foi obtido. A implementação do serviço de descoberta deste sistema está configurada para que, a cada 30 segundos, o nó verifique quais são os pares que estão disponíveis para conexão no momento. Quando um novo par

é descoberto, o serviço de replicação de um nó faz a comunicação com o mesmo serviço do outro nó, a fim de que informações sobre a disponibilidade de metadados e dados em cada um sejam obtidas e, assim, a replicação seja realizada se necessário. Como o tempo médio de replicação de um arquivo no sistema foi de 30,4 segundos, isto indica, possivelmente, que a maior parte das trocas está ocorrendo justamente nos momentos de encontro entre os nós do sistema.

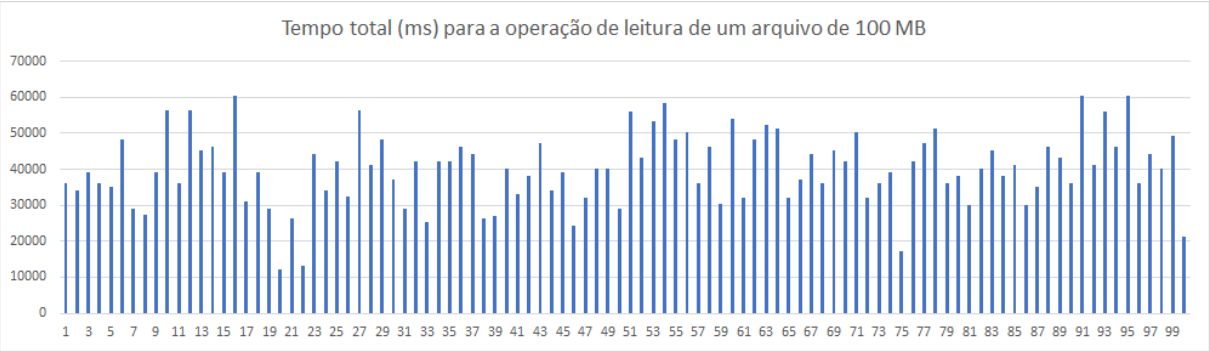

Figura 4.6: Tempo total da operação de leitura no *Storage Filesystem*.

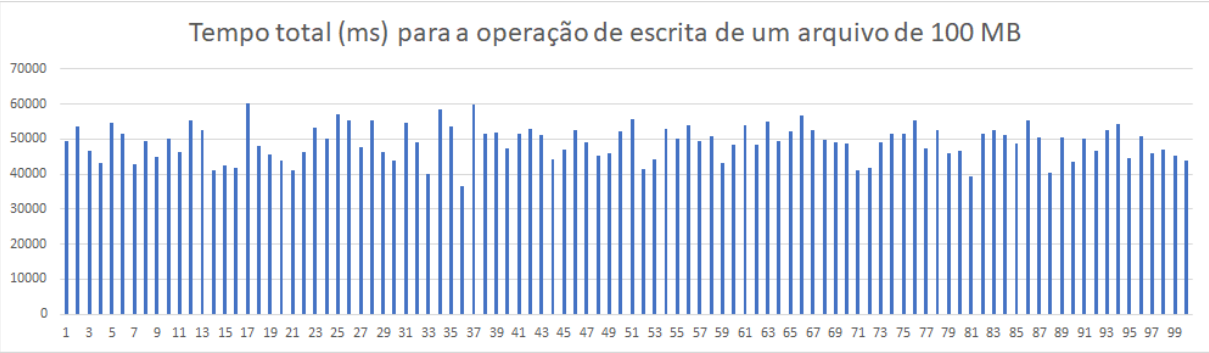

Figura 4.7: Tempo total da operação de escrita no *Storage Filesystem*.

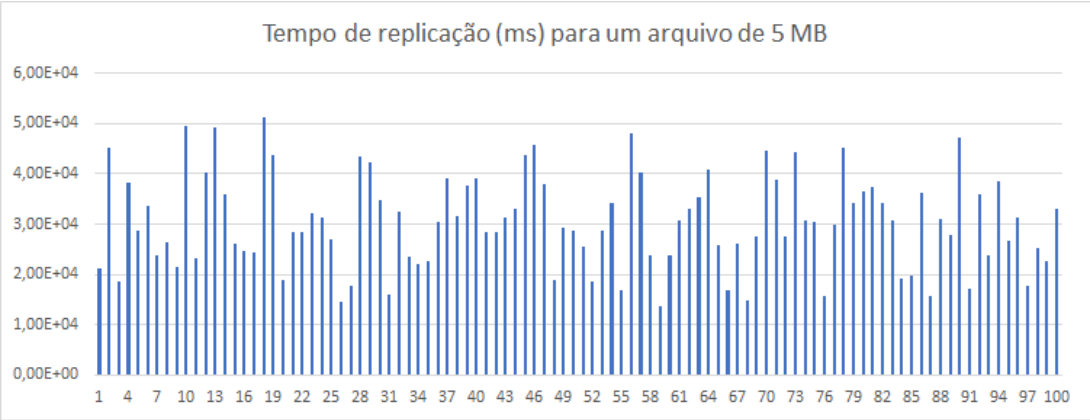

Figura 4.8: Tempo total para a replicação de um arquivo em todos os nós do *Music Filesystem*.

5 Considerações Finais

5.1 Conclusões

Este trabalho se propôs a construção de um *framework* para sistema de arquivos distribuídos que fosse altamente flexível e configurável.

A partir dos estudos relacionados, foi possível entender o que é necessário no desenvolvimento de um *framework* e de um sistema de arquivos distribuído.

Já o estudo de padrões de projeto se mostrou essencial na resolução de problemas presentes durante o projeto deste sistema. Através dos padrões plugin, mediador e proxy, permite-se a criação dos componentes dinâmicos, componentes do tipo *channel* e componentes do tipo *proxy*, que são a chave para que os objetivos deste trabalho fossem alcançados.

Como visto nos trabalhos relacionados, é esperado que um *framework* tenha um tempo de desenvolvimento mais longo do que um software comum, necessitando de um tempo de maturação até a sua estabilidade. Desta forma, é esperado que com o tempo, caso o desenvolvimento deste *framework* continue, as interfaces disponíveis se tornem estáveis, permitindo o desenvolvimento de novos componentes e, com o seu reúso, se tornem cada vez mais testados e de melhor qualidade.

Por falta de tempo, algumas partes não receberam a devida atenção, principalmente, a parte de segurança relacionada aos componentes dinâmicos. Sendo necessário que no futuro seja dada uma maior atenção a esta importante área.

Por fim, as provas de conceito evidenciaram as hipóteses iniciais deste trabalho, já que demonstrou-se a implementação de dois sistemas de arquivos distribuídos, desenvolvidos para cenários de uso muito diferentes entre si. Foi possível reutilizar muitos dos componentes que foram implementados como componentes padrões no *framework* e, a partir dos experimentos realizados, foi possível a análise do comportamento destes sistemas, considerando as operações mais importantes de cada um. Não é possível afirmar que os sistemas possuem um bom desempenho. Afinal, os testes realizados não foram

focados na comparação com outros sistemas de arquivos distribuídos. O objetivo foi a verificação do funcionamento das especializações, sendo estas construídas a partir de componentes presentes no *framework* NGFS. Desta forma, os resultados obtidos mostraram o comportamento que era esperado para as provas de conceito *Storage Filesystem* e *Music Filesystem*.

Portanto, o *framework* NGFS se mostrou altamente flexível, devido as possibilidades de configuração e extensibilidade através do desenvolvimento de componentes. Além disso, devido aos Utilitários que constituem a base fixa do *framework*, é possível agilizar consideravelmente o desenvolvimento destes componentes. Assim, é possível afirmar que o *framework* NGFS possibilita que sistemas de arquivos distribuídos sejam construídos em um tempo menor do que a construção de um sistema do tipo partindo-se do zero.

5.2 Trabalhos futuros

Devido a grande quantidade de interfaces que foram definidas no *framework*, existem vastas possibilidades de implementações de componentes que irão estender as suas funcionalidades. Portanto, no futuro, pode-se expandir as implementações que estão previamente disponíveis no *framework*.

Uma grande preocupação para o futuro é com a segurança dos componentes dinâmicos. Atualmente, não há nenhum mecanismo de segurança para impedir componentes maliciosos. Apesar da validação dos pacotes de componentes, isto não é suficiente para proteger de todos os problemas de segurança possíveis. Desta maneira, pode-se pensar no uso de *sandboxes*, análise de comportamento, entre outros recursos que poderiam atuar ativamente na detecção, isolamento e bloqueio de componentes maliciosos.

Os *proxies* possuem um grande potencial para diversos novos tipos de funcionalidade. Pode-se imaginar, por exemplo, na implementação de cache de resultado de chamadas ou ainda incluir controle de acesso aos serviços. Por fim, foi pensado na possibilidade da criação de *proxies* para componentes *filesystem* no cenário de recursos de hardware limitados, como para IoT. Neste caso, pode-se ter nada mais do que a aplicação e um *proxy* para o sistema de arquivos. Desta forma, todas as chamadas ao sistema de arquivos que a aplicação fizer, serão redirecionadas para um nó remoto que contém a

lógica do sistema de arquivos que, por sua vez, pode ter suas chamadas redirecionadas para outros nós da rede que possuem os serviços dos quais precisa para funcionar.

A parte de utilitários ocupa um determinado espaço em disco. Em muitos cenários, uma ou outra funcionalidade deste conjunto de utilitários que será usada. Desta forma, tem-se um desperdício de espaço de armazenamento. Sendo assim, foi pensada a possibilidade da criação de um repositório de "funções", onde a aplicação poderia buscar e obter através da rede, somente as funcionalidades que necessita usar.

Assim, os trabalhos futuros envolvendo o *framework* NGFS[8] devem focar, principalmente, no desenvolvimento de mecanismos de segurança e redução nos requisitos de hardware para que seja possível utilizá-lo em diferentes tipos de dispositivos, levando a ideia de sistemas de arquivos distribuídos, por exemplo, para o cenário IoT.

[8]Código-fonte disponível em: https://repesq.ufjf.br:10800/ltoscano/ngfs-tcc

Bibliografia

AGRAWAL, N. et al. A five-year study of file-system metadata. In: *ACM Transactions on Storage (TOS)*. [S.l.: s.n.], 2007. v. 3, p. 9.

BASAGNI, S. et al. Mobile ad hoc networking. In: . [S.l.]: IEEE Press, 2004. v. 1.

BOOCH, G. Designing an application framework. In: *Dr. Dobb's Journal*. [S.l.: s.n.], 1994. v. 19.

BORTHAKUR, D. The hadoop distributed file system: Architecture and design. In: *Hadoop Project Website*. [S.l.: s.n.], 2007.

CARNS, P. H. et al. Pvfs: A parallel file system for linux clusters. In: *Proceedings of the 4th annual Linux showcase and conference*. [S.l.: s.n.], 2000. v. 2, p. 391–430.

CHOU, W. Inside ssl: the secure sockets layer protocol. In: *IT professional*. [S.l.: s.n.], 2002. v. 4, p. 47–52.

FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. In: *Communications of the ACM*. [S.l.: s.n.], 1997. v. 40, p. 32–38.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S. The google file system. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. NY: [s.n.], 2003. p. 20–43.

KELLER, L.; LE, A.; CICI, B. Microcast: Cooperative video streaming on smartphones. *Proceedings of the 10th international conference on Mobile systems, applications, and services*, p. 57–70, 2012.

KOUYOUMDJIEVA, S. T.; KARLSSON, G. Device-to-device mobile data offloading for music streaming. *2016 IFIP networking conference (IFIP Networking) and workshops*, p. 377–385, 2016.

LANDIN, N.; NIKLASSON, A. Development of object-oriented frameworks. In: *Department of Communication Systems, Lund Institute of Technology*. Sweden: [s.n.], 1998.

MATHUR, A. et al. The new ext4 filesystem: current status and future plans. In: *Proceedings of the Linux symposium*. [S.l.: s.n.], 2007. v. 2, p. 21–33.

MICROSOFT. How ntfs works. In: *https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc781134(v=ws.10)*. [S.l.: s.n.], 2009.

NAGLE, D.; SERENYI, D.; MATTHEWS, A. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. [S.l.]: IEEE Computer Society, 2004. p. 53.

OVSIANNIKOV, M. et al. The quantcast file system. In: *Proceedings of the VLDB Endowment*. [S.l.: s.n.], 2013. v. 6, p. 1092–1101.

SANDBERG, R. et al. Design and implementation of the sun network filesystem. In: *Proceedings of the Summer USENIX conference*. [S.l.: s.n.], 1985. p. 119–130.

SATYANARAYANAN, M. A survey of distributed file systems. In: *Annual Review of Computer Science*. [S.l.: s.n.], 1990. v. 4, p. 73–104.

SATYANARAYANAN, M. et al. Coda: A highly available file system for a distributed workstation environment. In: *Proceedings of the Second Workshop on Workstation Operating Systems*. [S.l.]: IEEE, 1989. p. 114–116.

SCHWAN, P. Lustre: Building a file system for 1000-node clusters. In: *Proceedings of the 2003 Linux symposium*. [S.l.: s.n.], 2003. p. 380–386.

SOLTIS, S. R.; OKEEFE, T. M. R. M. T. The global file system. In: *Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*. [S.l.: s.n.], 1996. p. 319–342.

SOMMERVILLE, I. Engenharia de software. In: . [S.l.]: Pearson, 2005. v. 8.

STALLINGS, W. Criptografia e segurança de redes: Princípios e práticas. In: . [S.l.]: Pearson, 2006. v. 6.

TANENBAUM, A. S. Redes de computadores. In: . [S.l.]: Editora Campus, 2003.

TANENBAUM, A. S.; BOS, H. Modern operating systems. In: . [S.l.]: Pearson, 2015.

TANENBAUM, A. S.; STEEN, M. V. Distributed systems: principles and paradigms. In: . [S.l.]: Prentice-Hall, 2007.

THANH, T. D. et al. A taxonomy and survey on distributed file systems. In: *2008 Fourth International Conference on Networked Computing and Advanced Information Management*. [S.l.]: IEEE, 2008. v. 3, p. 144–149.