UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

IMPLEMENTAÇÃO DISTRIBUÍDA DE ALGORITMOS EVOLUTIVOS

João Pedro Lopes da Silva

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Doutor Pedro Miguel Frazão Fernandes Ferreira

2017

Agradecimentos

Gostaria de agradecer ao meu orientador, Professor Pedro Miguel Frazão Fernandes Ferreira pelo apoio e disponibilidade apresentado durante o decorrer da dissertação. Durante a escrita da dissertação, a sua experiência e espírito critico levaram ao aperfeiçoamento deste trabalho.

Agradeço ao LASIGE pela atribuição de uma bolsa de investigação e pelo espaço de trabalho disponibilizado na faculdade.

Quero agradecer também aos meus amigos e colegas, em especial, ao Pedro Figueiredo e Gonçalo Lima pela disponibilidade e apoio no desenvolvimento deste trabalho. Adicionalmente, quero agradecer ao Ricardo Mendes e ao Tiago Oliveira pelo apoio, disponibilidade e interesse em dúvidas relacionados com o SCFS, *DepSky* e *DepSpace*.

Por fim, quero agradecer à minha família, em especial aos meus pais e irmão, pela motivação e apoio incondicional durante o decorrer deste trabalho.

Resumo

Os sistemas para computação evolutiva, tanto distribuída como não distribuída, têm recebido bastante atenção na ultima década. Isto deve-se principalmente ao aumento da complexidade dos problemas de otimização presentes no mundo real. Embora sejam mais simples de implementar e de perceber, os sistemas não distribuídos revelam-se pouco práticos para problemas complexos e de elevada dimensões. Consequentemente, os sistemas distribuídos tornam-se mais atrativos para resolver problemas mais complexos e multi-dimensionais em tempo aceitável. Contudo, estes sistemas apresentam alguns problemas adicionais relacionados com a sua natureza distribuída, requerendo que os seus utilizadores possuam conhecimentos em sistemas distribuídos.

Neste trabalho apresentamos um estudo dos diferentes modelos distribuídos capazes de implementar algoritmos evolutivos. Apresentamos também o *state-of-the-art* de um dos modelos e dos sistemas de armazenamento, capazes de implementar uma *Pool* de recursos partilhados.

Por fim, apresentamos um sistema distribuído baseado no modelo *pool*, que suporta a execução de vários algoritmos evolutivos ao mesmo tempo, sem impor qualquer restrição no seu tipo. O mesmo garante uma elevada tolerância a faltas, confidencialidade, disponibilidade e desempenho. Finalmente, o sistema abstrai a sua componente distribuída, possibilitando que seja operado por utilizadores sem o conhecimento em sistemas distribuídos.

Palavras-chave: Computação evolutiva distribuída; Sistemas Distribuídos; Sistemas de armazenamento distribuídos; Sistemas distribuídos baseados no modelo *Pool*; Tolerância a faltas distribuída.

Abstract

Evolutionary computation, both distributed and non distributed has received considerable attention over the past decade. This is mainly due to the increasing complexity of real-world optimization problems. Although simpler to implement and to understand, non distributed systems are impractical for complex high dimension problems. Consequently, the distributed systems become more attractive to solve more complex and multi dimension problems in acceptable time. However, they have additional challenges related to their distributed nature, requiring their users to have knowledge in distributed models.

In this thesis we present a study on different distributed models that are able to implement evolutionary algorithms. We also present a state-of-the-art on one of the models and storage systems.

Finally, we present a distributed system based on a Pool, which supports the execution of several evolutionary algorithms at the same time, without imposing any restriction on their type. It guarantees a high fault tolerance, confidentiality, availability and high performance. Lastly, the system abstracts its distributed nature, allowing it to be operated by users without knowledge in distributed systems.

Keywords: Distributed evolutionary computation; Distributed systems; Distributed Storage systems; Pool based distributed systems; Distributed fault tolerance.

Conteúdo

Lista de Figuras **x**

Lista de Tabelas **xi**

1 Introdução **1**
1.1 Motivação 1
1.2 Objetivos 2
1.3 Contribuições 2
1.4 Estrutura do documento 2

2 Trabalho Relacionado **5**
2.1 Como funcionam os AEs 5
2.2 Modelos de Sistemas Distribuídos para Implementação de AEs 7
2.2.1 Modelo *master-slave* 7
2.2.2 Modelo *Island* 8
2.2.3 Modelo *Cellular* 8
2.2.4 Modelo *Hierarchical* 9
2.2.5 Modelo *Coevolution* 10
2.2.6 Modelo *Multi-Agent* 11
2.2.7 Modelo *Pool* 12
2.3 Modelos Pool em detalhe 12
2.3.1 *EvoSpace* 13
2.3.2 *SofEA* 14
2.3.3 *SofEA* alterado 16
2.4 Sistemas de Armazenamento para Implementação da *Pool* 18
2.4.1 *ZooKeeper* 18
2.4.2 Cassandra 20
2.4.3 OpenDHT 21
2.4.4 BFT-SMART 23
2.4.5 DepSky 24
2.4.6 DepSpace 27

2.4.7 Shared Cloud-backed File System (SCFS) 30
2.4.8 CHARON 32

3 Arquitetura do Sistema **35**
3.1 Análise 35
3.2 Arquitetura 38
3.2.1 Sistema de *Locks* 41

4 Implementação **43**
4.1 Plataforma 43
4.1.1 Estrutura em Camadas 44
4.1.2 Arquitetura 45
4.2 *Daemon* 52
4.3 Caso de teste e exemplo de implementação 52
4.3.1 O problema específico 54
4.3.2 Implementação do MOGA no sistema 54
4.4 Tolerância a faltas 55
4.5 Tecnologias 55

5 Resultados **57**
5.1 Ambiente Experimental 57
5.2 Validação da implementação 58
5.3 Validação do desempenho 60
5.4 Tolerância a Faltas 62

6 Conclusão **67**
6.1 Contribuições e Conclusões 67
6.2 Trabalho Futuro 68

Abreviaturas **69**

Bibliografia **71**

Lista de Figuras

2.1 Pseudo-código de um AE genético 6
2.2 Fluxo de execução de um AE genético 6
2.3 Modelo *master-slave* 7
2.4 Modelo *Island* 8
2.5 Modelo *Cellular* 9
2.6 Modelo *Hierarchical* 9
2.7 Modelo *Coevolution* 10
2.8 Modelo *Multi-agent* 11
2.9 Modelo *Pool* 12
2.10 Arquitetura do *EvoSpace* 13
2.11 Exemplo de uma possível estrutura do *Zookeeper* 19
2.12 Estrutura do sistema *OpenDHT* 22
2.13 Modelo de dados do *DepSky* 25
2.14 Arquitetura do *DepSky* 26
2.15 Arquitetura do *DepSpace* 29
2.16 Arquitetura do SCFS 31
2.17 Arquitetura do CHARON 33

3.1 Arquitetura do sistema 39
3.2 Estrutura de ficheiros da *Pool* 40

4.1 Estrutura em três camadas 44
4.2 Arquitetura da plataforma 45
4.3 Classe abstrata *Simulator* 45
4.4 Classe abstrata *Solver* 46
4.5 Classe abstrata *StorageManager* 49
4.6 Classe abstrata *Operator* 50
4.7 Classe abstrata *Problem* 50
4.8 Classes abstratas *Solution* e *SolutionSample* 51
4.9 Classe abstrata *Space* 52

5.1 Comparação de resultados da primeira e última geração do MOGA, com 4 participantes. 58

5.2 Comparação de resultados da última geração do MOGA, com 2 e 4 participantes. 59
5.3 Tempo de execução de cada geração. 60
5.4 Tempo total de execução do algoritmo. 61
5.5 Tempo de avaliação de indivíduos, por geração, com 2 participantes. . . . 62
5.6 Tempo de avaliação de indivíduos, por geração, com 4 participantes. . . . 62
5.7 Tempo de execução de cada geração, com e sem faltas. 63
5.8 Tempo total de execução do algoritmo, com e sem faltas. 64
5.9 Excerto do ficheiro de *log*, revelando falha por *crash* de *workers*. 64
5.10 Excerto do ficheiro de *log*, revelando falha por *crash* do *Master* e *Manager*. 65
5.11 Excerto do ficheiro de *log*, revelando eleição de *Master*. 65

Lista de Tabelas

3.1 Sumário dos modelos 36

4.1 Métodos disponibilizados por camada 53

5.1 Características dos participantes, por configuração (*cluster* S e R respetivamente) 58

Capítulo 1

Introdução

Neste capítulo apresentamos a motivação, os objetivos e a contribuição do desenvolvimento deste trabalho. Por último apresentamos a organização do documento.

1.1 Motivação

A computação de Algoritmos Evolutivos (AEs) tem-se revelado bastante eficiente na resolução de problemas de otimização difíceis, presentes no mundo real. Contudo com o desenvolvimento rápido das tecnologias, o aparecimento de *big data* e o aumento do tamanho e complexidade dos problemas, os custos e tempo de computação revelam-se bastante elevados em modelos não distribuídos, tornando-os pouco práticos. Deste modo, recorre-se à utilização de modelos distribuídos de computação evolutiva em vez de não distribuídos. Estes são capazes de resolver problemas complexos de elevadas dimensões, recorrendo a computação distribuída. Adicionalmente, a componente distribuída destes sistemas permite manter um leque de soluções diferentes, oferecendo diversidade e facilitando a procura multiobjectivo. Contudo estes modelos possuem problemas adicionais, relacionados com a sua componente distribuída, como a distribuição de tarefas evolutivas, a escolha de protocolos de comunicação entre clientes, problemas de confidencialidade, coordenação de participantes, entre outros. Estes problemas requerem que os utilizadores do sistema possuam conhecimento não só em AEs, como também em sistemas distribuídos.

Para além das limitações apresentadas, muitos dos sistemas distribuídos da atualidade ainda apresentam características de sistemas não distribuídos (como controlo centralizado), encontram-se restritos a um tipo de problemas que podem implementar (sem que seja possível decompor em problemas mais simples, por exemplo) e apenas podem executar um Algoritmo Evolutivo (AE) de cada vez.

1.2 Objetivos

Este trabalho tem como objetivo principal a criação de um sistema distribuído capaz de executar AEs. Pretende-se que este:

- seja tolerante a faltas;
- seja flexível à entrada e saída de participantes;
- favoreça cooperação entre organizações;
- permita a implementação de uma vasta gama de AEs geracionais
- permita a execução de vários AEs geracionais ao mesmo tempo
- não apresente um ponto único de falha;

Adicionalmente, pretende-se que este forneça um alto nível de abstração, de modo a ser facilmente operado por utilizadores que não tenham conhecimento em sistemas distribuídos ou em segurança, mas com conhecimento em AEs.

De modo a realizar os objetivos mencionados, temos de resolver vários problemas, como a abstração à diversidade dos vários AEs geracionais, coordenação dos participantes e monitorização *master*/*slave* de uma rede *Peer-to-peer*.

1.3 Contribuições

Com este trabalho conseguimos criar um sistema distribuído capaz de suportar uma vasta gama de AEs, fornecendo uma elevada tolerância a faltas, flexibilidade e disponibilidade. Contudo, este trabalho foca-se em algoritmos evolutivos geracionais. Face aos trabalhos existentes na mesma área, o nosso sistema pode ser operado por utilizadores que não possuam conhecimentos em sistemas distribuídos, não impõe restrições ao tipo de AEs que nele são implementados e não possui características de sistemas centralizados. Apesar de esta dissertação se focar no uso do *Shared Cloud-backed File System* (SCFS) [13] como *Pool*, o nosso sistema permite a utilização de qualquer forma de armazenamento, sem ser necessária alguma alteração ao AE.

1.4 Estrutura do documento

O documento encontra-se organizado da seguinte forma: o Capítulo 2 apresenta a pesquisa do trabalho relacionado. No Capítulo 3 apresentamos as vantagens e desvantagens de cada modelo, expondo as razões para as decisões tomadas. Apresentamos também o desenho e as premissas da solução proposta. O Capítulo 4 contém os detalhes de

implementação do nosso sistema e de um AE utilizado como exemplo. O Capítulo 5 apresenta as experiências efetuadas assim como os seus resultados, quando executado o *Multi-Objective Genetic Algorithm* (MOGA) [20] sobre um caso de estudo. Finalmente, o Capítulo 6 apresenta as conclusões obtidas na realização da dissertação e o trabalho futuro.

Capítulo 2

Trabalho Relacionado

Neste capítulo explicamos o que são AEs, analisamos os modelos existentes de sistemas distribuídos capazes de suportar a implementação e execução de AEs, um estudo mais aprofundado do estado da arte do modelo *pool* e o estado da arte de alguns sistemas de armazenamento distribuídos.

2.1 Como funcionam os AEs

Os AEs genéticos são algoritmos baseados na biologia genética, que são bastante práticos e eficientes na resolução de problemas de otimização complexos. Nestes, considera-se que existe uma população inicial de indivíduos e um mecanismo de avaliação. Adicionalmente, cada indivíduo da população representa uma possível solução do problema. Através das condições mencionadas, surge um fenómeno de seleção natural, onde os indivíduos mais fortes têm maior probabilidade de sobreviver, aumentando assim o nível de aptidão da população. Desta população sobrevivente, alguns indivíduos são escolhidos para servirem de base à formação da população da próxima geração. Aos escolhidos, aplicam-se operadores, como por exemplo, os operadores de recombinação e de mutação, de modo a gerar novos indivíduos. Após a criação da nova população, esta é avaliada e o processo repete-se até ser atingido o critério de paragem.

Por exemplo, o operador de recombinação simula a procriação e utiliza dois ou mais indivíduos existentes (pais) e combina-os de modo a criar um ou mais indivíduos novos (filhos). O operador de mutação simula um evento não planeado que, através de mutações, altera parte da constituição genética ou Genótipo do indivíduo.

Todo o processo repete-se até que existam um ou mais indivíduos que satisfaçam os objetivos definidos ou até que se tenha atingido o limite computacional desejado. Os operadores de recombinação e mutação fornecem um mecanismo de variação de população, permitindo que o AE explore um vasto leque de possíveis soluções. Por outro lado, o operador de seleção fornece um mecanismo de aumento da qualidade das soluções. Em combinação, estes mecanismos levam à otimização dos valores de aptidão da população.

Contudo, é importante realçar que muitos destes mecanismos são estocásticos. Como exemplo, no processo de seleção, os melhores indivíduos não são escolhidos deterministicamente e tecnicamente, até os indivíduos mais fracos têm uma pequena probabilidade de sobreviverem ou até de serem pais.

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Figura 2.1: Pseudo-código de um AE genético [1]

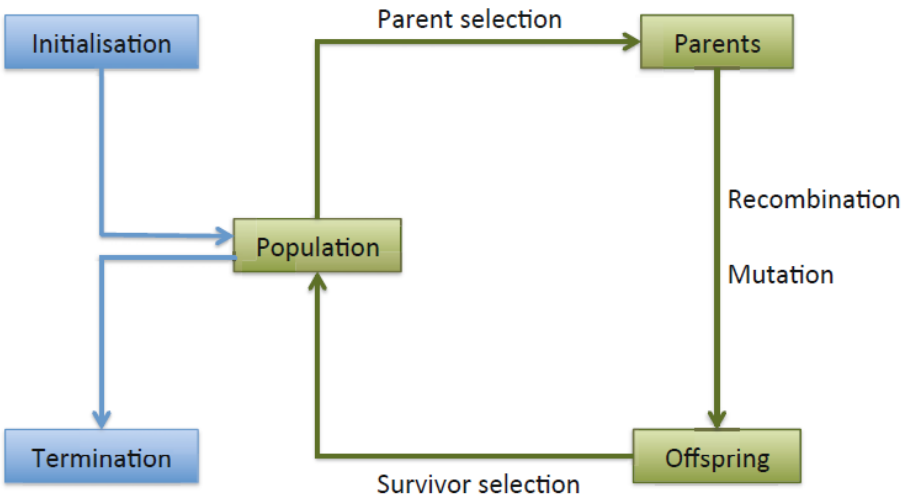

Figura 2.2: Fluxo de execução de um AE genético [2]

A Figura 2.1 apresenta o pseudo-código de um AE genético e a Figura 2.2 apresenta um fluxo de execução de um AE genético. De um modo geral, a principal diferença para outros AEs são os operadores aplicados. Adicionalmente, podemos observar que estes algoritmos geram e testam várias soluções até encontrarem a solução pretendida. O processo de avaliação (ou de aptidão) fornece uma aproximação heurística da qualidade da solução, enquanto que os operadores fornecem um processo de procura.

[1] retirado de [16]
[2] retirado de [16]

2.2 Modelos de Sistemas Distribuídos para Implementação de AEs

De seguida apresentamos os vários modelos de sistemas distribuídos, que atualmente possibilitam a integração de AEs, assim como descrito em [23].

2.2.1 Modelo *master-slave*

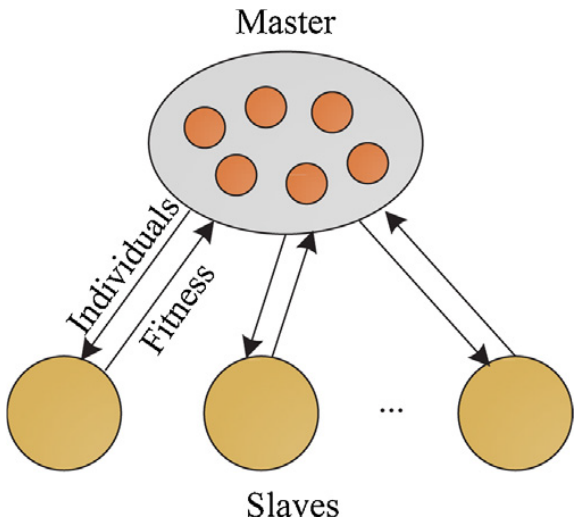

Figura 2.3: Modelo *master-slave* [3]

Representado na Figura 2.3, o modelo *master-slave* é um modelo de distribuição de população em que as operações de cruzamento, mutação e seleção são efetuadas pelo sistema mestre. Este envia a especificação de indivíduos aos seus escravos (ou *slaves*) que os avaliam e devolvem os resultados. Deste modo, o modelo evita comunicações entre os trabalhadores.

Contudo, em problemas que não requerem elevado poder computacional, o preço de comunicação torna o modelo ineficiente.

Ao longo do tempo surgiram algumas variantes que tentam melhorar o desempenho, como por exemplo, os escravos contêm uma parte dos indivíduos, enviando apenas os melhores para o Mestre ([34]).

Nas variantes síncronas, o mestre espera pelos resultados dos trabalhadores até continuar com a próxima geração, enquanto que em variantes assíncronas, o mestre executa os operadores apenas numa fração da população. O *Speedup* e a eficiência estão ligados ao desempenho do mestre, à comunicação entre mestre e trabalhadores e aos custos de computação dos escravos. A tolerância a faltas pode ser atingida através da reposição da população de um *slave* que não respondeu em tempo útil.

[3] retirado de [23]

2.2.2 Modelo *Island*

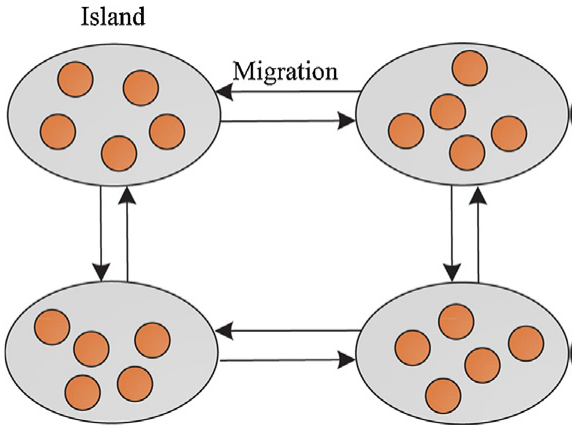

Figura 2.4: Modelo *Island* [4]

No modelo *Island*, representado na Figura 2.4, a população é dividida em vários subconjuntos denominados ilhas. Cada um destes subconjuntos é executado em um processador e as comunicações entre ilhas apenas ocorrem com a migração de um indivíduo.

O processo de migração depende do tipo de sistema. Em sistemas síncronos, durante o decorrer da geração existe um intervalo definido em que cada ilha envia o seu elemento mais bem classificado para as outras ilhas. Num sistema assíncrono, este envio de informações não se encontra definido, permitindo que cada ilha o execute assim que desejar.

Adicionalmente o sistema pode ser homogéneo ou heterogéneo. Num sistema homogéneo, cada ilha usa o mesmo conjunto de operadores, parâmetros de controlo e funções de aptidão. Esta abordagem favorece uma pesquisa mais minuciosa mas desfavorece a exploração de novas soluções. Os modelos de ilha heterogéneos ([31]) possuem parâmetros diferentes, como por exemplo, as ilhas são divididas em três camadas, cada uma responsável por explorar, explorar e aprofundar ou apenas aprofundar. Cada ilha executa os passos do AE localmente e envia o seu melhor indivíduo para outra ilha (migração). O desempenho destes sistemas está ligado ao número de ilhas e pode ser limitado, não aumentando diretamente com a adição de ilhas no sistema. Em outras palavras, em determinada altura é necessário adicionar um elevado número de ilhas para que o desempenho sofra um aumento significativo.

2.2.3 Modelo *Cellular*

Neste tipo de modelo, representado na Figura 2.5, a população encontra-se distribuída em forma de grelha. A comunicação é definida pela topologia da rede e apenas permite a cada

[4] retirado de [23]
[5] retirado de [23]

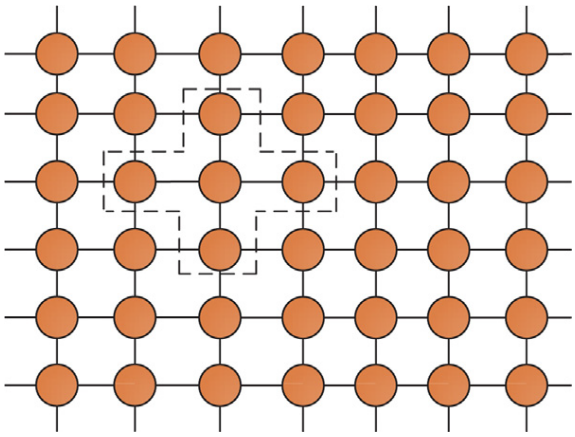

Figura 2.5: Modelo *Cellular* [5]

elemento competir ou acasalar com os seus vizinhos. Idealmente cada célula está a cargo de um processador. Este modelo possibilita que os bons indivíduos se possam propagar por toda a população.

Adicionalmente este modelo possui uma variante síncrona e outra assíncrona. Na variante síncrona, as células são atualizadas em simultâneo, enquanto que na variante assíncrona, as células são atualizadas uma a uma.

Neste modelo é importante tomar em consideração a escolha de topologia de rede, pois esta tem grande influência no desempenho do algoritmo. Esta topologia deve alterar com a complexidade do problema.

2.2.4 Modelo *Hierarchical*

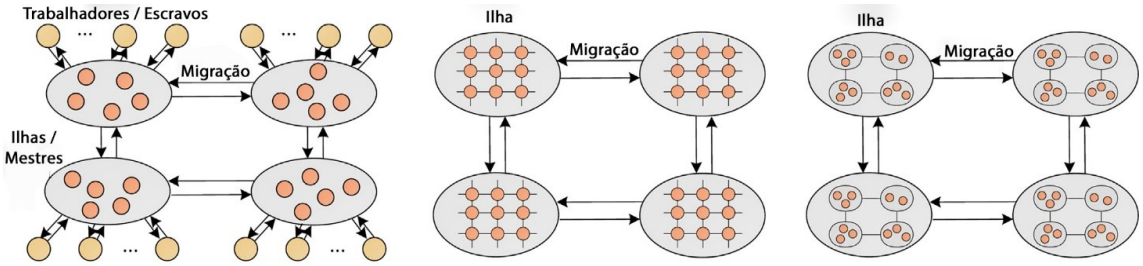

Figura 2.6: Modelo *Hierarchical* com combinação *Island* - *Master-slave*, *Island-cellular* e *Island-Island* respetivamente. [6]

Os modelos hierárquicos ou modelos híbridos, representado na Figura 2.6 , combinam de forma hierárquica dois ou mais modelos distribuídos, tentando tomar partido dos benefícios dos modelos escolhidos.

- Combinação *Island – Master-slave*, a população é dividida em subconjuntos que executam em diferentes *Masters* e comunicam em instantes específicos. Para cada

[6] retirado de [23]

subconjunto, o *Master* envia trabalho para os seus escravos, paralelizando o problema. Este modelo apresenta elevada escalabilidade e reduz a dependência de um único *Master*.

- Combinação *Island* - *Cellular*: A população é dividida em várias ilhas, que localmente contém uma distribuição celular.
- Combinação *Island* – *Island*: A população é dividida em ilhas, e estas são posteriormente divididas em outras ilhas. Este modelo distingue dois tipos de migração, migração local e global. Adicionalmente, este apresenta elevados níveis de desempenho.

2.2.5 Modelo *Coevolution*

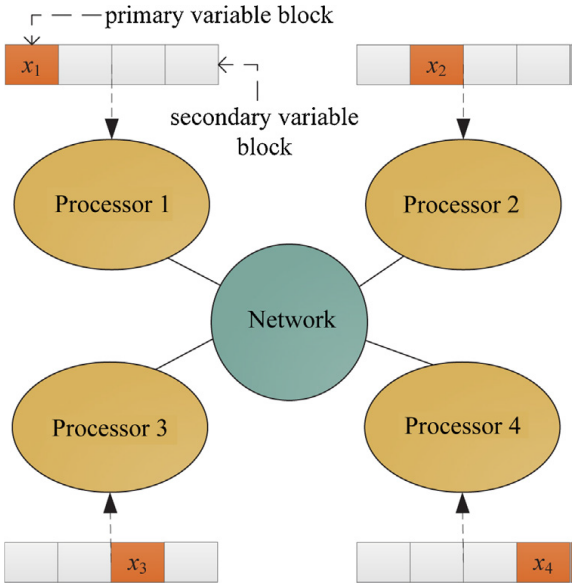

Figura 2.7: Modelo *Coevolution* [7]

O modelo de *Coevolution*, apresentado na Figura 2.7, é um modelo que faz a distribuição segundo a dimensão da representação dos indivíduos (ou *dimension-distributed*). Este divide um problema complexo em vários problemas com menores dimensões, a serem atribuídos a diferentes processos. A ideia central é decompor um problema em subproblemas, resolvê-los (recorrendo a qualquer tipo de comunicação) e obter a ou as soluções após juntar os resultados dos vários processos.

Contudo este modelo tem alguns problemas que precisam de ser resolvidos *a priori*, como a decomposição de problemas. Caso esta decomposição não seja possível, opta-se por formar grupos de problemas que sejam inseparáveis. Após receber uma parte do problema, o participante resolve-o individualmente, devolvendo apenas a solução. Visto que

[7] retirado de [23]

os participantes apenas têm conhecimento do seu espaço de procura e apenas partilham as suas melhores soluções, originam problemas adicionais, como a evolução de componentes independentes e a monitorização de diversidade. Nestes problemas cada participante evolui individualmente e pode fornecer soluções bastante parecidas ao dos seus parceiros, respetivamente.

Neste modelo, a avaliação é feita através da migração do melhor indivíduo de cada processo. Este é inicialmente combinado com o melhor indivíduo de outro processo e posteriormente com um terceiro indivíduo escolhido aleatoriamente, possibilitando assim uma maior diversificação.

2.2.6 Modelo *Multi-Agent*

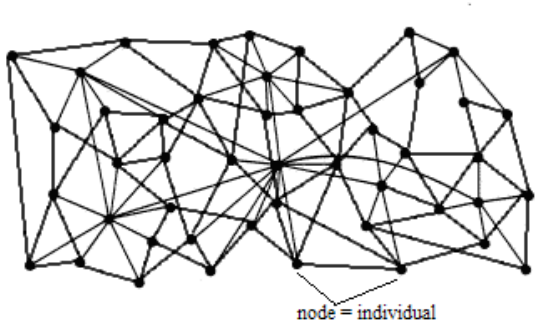

Figura 2.8: Modelo *Multi-agent* [8]

Ao contrário do modelo *Coevolution*, em que cada elemento do sistema trabalha de forma coordenada de modo a atingir um objetivo comum, o modelo *multi-agent*, apresentado na Figura 2.8, não requer qualquer tipo de coordenação para chegar ao objetivo final. O modelo é constituído por um conjunto de nós, ou clientes, que podem comunicar entre si, sem qualquer restrição. Adicionalmente, estes alteram o seu comportamento com base na observação do seu ambiente, de modo a estabelecer um equilíbrio global. O sistema atinge o seu objetivo quando um objetivo local não consegue ser melhorado.

Estes sistemas podem recorrer a protocolos *Gossip-based* (ou *epidemic based*) [33] para comunicar. Este tipo de protocolos é perfeito para trocar informação num sistema distribuído de larga escala, permitindo que os seus participantes entrem e saiam do sistema sem problemas, mesmo que as saídas sejam devido a faltas. Nestes protocolos cada nó periodicamente troca informação com um conjunto de nós. Esta comunicação pode ser na forma de *push* ou *pull*. Na variante *push* um nó verifica se um dos seus vizinhos tem dados e, em caso negativo, este envia os seus dados (faz *push*). Na variante *pull* o nó vai buscar informação aos seus vizinhos.

[8] retirado de [33]

2.2.7 Modelo *Pool*

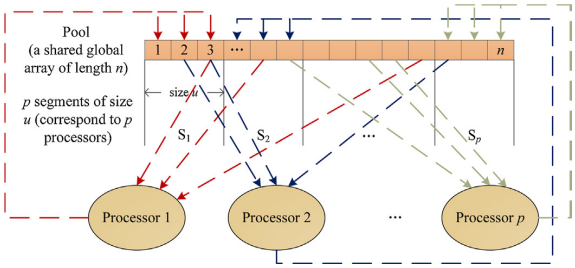

Figura 2.9: Modelo *Pool* [9]

Neste modelo processos autónomos trabalham sobre uma *Pool* [23] de recursos partilhados, de modo a que estes sejam fracamente acoplados (*loosely coupled*), ou seja, sem terem conhecimento da existência de outros. Este modelo permite uma implementação assíncrona, heterogénea ou homogénia. Este permite também que os processos constituintes da *Pool* sejam alterados e que a *Pool* seja replicada, de modo a atingir uma elevada tolerância a faltas. Neste sistema os trabalhadores obtêm dados da *Pool*, aplicam-lhes as operações desejadas e colocam os resultados na *Pool*. Os novos dados colocados substituem os antigos se possuírem valor de aptidão melhor. Contudo, a decisão de implementação da *Pool* é muito importante.

2.3 Modelos Pool em detalhe

Para o nosso sistema, consideramos que o tempo de avaliação de cada indivíduo é significativamente superior ao tempo de comunicação entre participantes. Deste modo e, face à análise das vantagens e desvantagens dos modelos distribuídos capazes de implementar AEs, decidimos focar-nos no modelo *pool*. Em 3.1 apresentamos uma análise detalhada das decisões tomadas. Na secção seguinte apresentamos os tipos de modelos *pool* desenvolvidos ao longo dos anos.

Os modelos baseados em *Pool* são caracterizados por processos autónomos a trabalhar sobre uma *Pool* de recursos partilhados, sem terem conhecimento da existência de outros (são *loosely coupled*). A *Pool* permite que os participantes leiam dados, efetuem algum trabalho sobre eles e de seguida retornem o resultado. O servidor que contém a *Pool* não é responsável pela interação e sincronização de toda a população, este apenas recebe pedidos de participantes isolados. Desta forma, o modelo permite uma colaboração *ad-hoc* de recursos computacionais. Adicionalmente este modelo permite uma implementação assíncrona, heterogénea ou homogénia.

[9] retirado de [23]

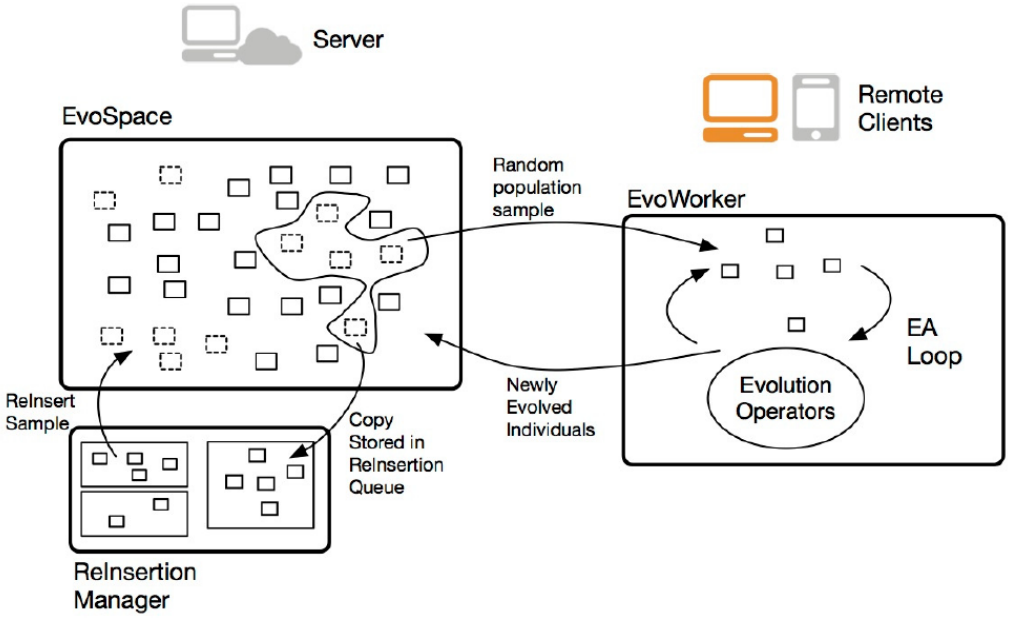

Figura 2.10: Arquitetura do *EvoSpace* [10]

2.3.1 *EvoSpace*

O modelo *EvoSpace* ou ESM apresentado em [21] é construído à volta de um repositório central de população. Como observamos na Figura 2.10, o sistema contém dois componentes, um conjunto de *EvoWorkers* e uma instância da *EvoStore*. A *EvoStore* representa a *Pool* que contém e gere a população. Os *EvoWorkers* representam os trabalhadores, que retiram um conjunto de indivíduos da *EvoStore*, efetuam trabalho sobre este conjunto e colocam novos indivíduos na *Pool*. De modo a evitar que vários trabalhadores operem sobre os mesmos dados, ou seja, de modo a evitar trabalho redundante, o trabalho torna-se inacessível para os outros participantes após ser retirado por um trabalhador. Adicionalmente, o sistema implementa um mecanismo de deteção de *starvation* da *pool*.

EvoStore

A *EvoStore* segue os princípios de espaços de tuplos partilhados, podendo ser descrito como um modelo de memória partilhada distribuída organizada por conjuntos de tuplos. Um tuplo é composto por um ou mais campos e correspondentes valores. Os participantes podem adicionar e remover tuplos, acedendo de modo aleatório ao espaço de tuplos (em vez de acederem a um tuplo em particular, como num espaço de tuplos partilhado normal).

Os indivíduos contidos na *EvoStore* são organizados por dicionários, ou seja, por uma coleção de chaves únicas e valores, e são descritos por:

1. *id*: único em formato string. Representa o indivíduo;

[10] retirado de [21]

2. cromossoma em formato string: especifica a representação usada e depende do AE utilizado;

3. dicionário de aptidão: permite ao sistema guardar mais do que um valor de aptidão para um indivíduo, útil em casos de AEs multiobjectivo;

4. dicionário com o identificador dos seus pais.

Quando a *EvoStore* perde a ligação a um cliente, ou o número de indivíduos se encontra abaixo de um certo limite, esta volta a inserir os elementos que foram removidos ou bloqueados para uso dos clientes.

EvoWorkers

Os *EvoWorkers* são entidades autónomas que executam na máquina do cliente. Estes apenas podem comunicar com a *EvoStore*. Ao obter uma amostra, um participante pode:

1. efetuar uma evolução parcial e posteriormente devolver os novos indivíduos gerados à *EvoStore*, podendo estes sobrepor outros indivíduos.

2. Alterar o estado dos indivíduos e retorná-los à *EvoStore* sem efetuar qualquer operação evolutiva.

A partir deste ponto, os novos indivíduos podem ser utilizados por outros participantes. Adicionalmente os trabalhadores podem também especializar-se numa determinada parte do processo evolutivo (como a seleção, avaliação ou mutação).

Avaliação

Os resultados apresentados em [21] indicam que o *EvoSpace* consegue executar uma procura mais eficiente do que os modelos *Island*, necessitando de apenas metade das avaliações para encontrar uma solução. Concluem também que o sistema pode beneficiar com a adição de clientes e que consegue resistir a falhas nos clientes e na comunicação.

2.3.2 *SofEA*

SofEA, apresentado inicialmente em [27] recorre à base de dados *CouchDB* [10] para servir de *Pool*. Este tem como objetivo separar população do AE, tornando-os independentes e obrigando a que parâmetros e operadores sejam passados como argumentos. O *SofEA* guarda os cromossomas no *CouchDB*, na forma de documento. Este documento contém o nome igual à String do cromossoma, evitando assim repetições de cromossomas. Adicionalmente este documento contém a String do documento, o número de versão, o valor de aptidão e uma constante aleatória. O número de versão encontra-se a 1 quando este é iniciado, passa para 2 depois do cromossoma ser avaliado e posteriormente para 3 caso este se encontre morto.

CouchDB

O *CouchDB* [10] é uma base de dados do tipo *NoSQL*, ou seja, não utiliza a linguagem SQL para manipular os dados. Estes são guardados na forma chave-valor, podendo ser retirados por chave ou por um conjunto de chaves. Adicionalmente, estes podem ter outros atributos, como um número de versão, alterado sempre que o objeto sofre uma alteração. O *CouchDB* utiliza JSON (*JavaScript Object Notation*) para serializar objetos, tornando-o capaz de guardar todo o tipo de estruturas de dados em ficheiros denominados *documents*. Este apresenta uma interface REST (*Representational State Transfer*) simples que pode ser acedida através da linha de comandos ou por várias bibliotecas de clientes. Embora as operações que utilizem chaves sejam mais rápidas, este também possibilita operações de pesquisa por conteúdo do documento (denominadas *views*), embora estas sejam mais lentas.

Algoritmo

Visto que o *CouchDB* está preparado para lidar com um número elevado de pedidos, o algoritmo foi dividido em quatro partes diferentes, que operam independentemente e de modo assíncrono. Estes são:

1. Inicialização: Cria-se um conjunto de cromossomas na versão 1.
2. Avaliação: O cliente atua sobre os cromossomas vivos e que não tenham sido avaliados previamente. Para tal este gera um número aleatório *r*. De seguida este cliente pede *n* cromossomas com a chave *r* à *Pool*. De seguida o *CouchDB* devolve no máximo *n* cromossomas com o valor aleatório (presente no documento) maior que a chave *r*.
3. Reprodução: As operações de *crossover* e mutação são aplicadas em conjuntos de cromossomas que estejam na versão 2, criando novos cromossomas. Estes são colocados na versão 1.
4. Eliminação: clientes denominados *Reapers* obtêm cromossomas do *CouchDB*, ordenados por valores de aptidão. De seguida estes clientes atualizam o número de versão dos piores para um estado morto (versão 3), de modo a que a quantidade de cromossomas na versão 2 (os que já foram avaliados) corresponda a um limite específico.

As últimas três partes podem ser iniciadas de forma assíncrona e podem executar sem qualquer sequência, permitindo que sejam implementados em linguagens, sistemas operativos ou máquinas diferentes, especializadas para cada problema. Cada experiência começa com a eliminação dos documentos existentes e com a criação aleatória da população. Adicionalmente, é criado um documento com uma chave conhecida, de modo a assinalar o fim da experiência ou de modo a obter o número limite de avaliações.

Problemas

Esta abordagem apresenta um problema grave na falta de cromossomas (ou *Starvation*). Visto as operações serem assíncronas, este problema ocorre se a reprodução não for rápida o suficiente (não existem cromossomas para avaliar) ou se o avaliador demorar muito tempo (o reprodutor não tem cromossomas suficientes). Apesar da fase de eliminação não ser muito relevante neste caso, esta também necessita de ser executada de forma regular, caso contrário a população cresce bastante, reduzindo assim o fator de exploração do algoritmo.

Em alternativa, este problema pode ser resolvido se obrigarmos os componentes a esperarem um determinado tempo (um segundo), caso não existirem cromossomas suficientes. Contudo esta abordagem aumenta o número de pedidos desnecessários ao servidor ou servidores. Adicionalmente esta abordagem permite concorrência entre os participantes, ou seja, existe a possibilidade dos participantes escolherem os mesmos cromossomas, efetuando trabalho redundante.

Avaliação

Apesar das desvantagens, esta abordagem garante que o processo de reprodução não gera cromossomas iguais aos existentes (visto que o nome de cada cromossoma é a sua representação).

Os clientes apenas necessitam de saber o endereço da base de dados para começarem a contribuir. Caso um cliente sofra um *crash*, os clientes restantes continuam a trabalhar, tornando o sistema tolerante a faltas. O sistema continua funcional desde que haja pelo menos um reprodutor, um avaliador, um *reaper* e uma *pool*. Toda a informação da base de dados é guardada em disco, permitindo que a informação possa ser recuperada na presença de uma falha. As operações executadas pelos clientes são descentralizadas, com exceção à condição de terminação, que é verificada após cada pedido atendido. O aumento de clientes causa um aumento de desempenho, contudo é necessário encontrar um equilíbrio entre o número de pedidos e o número de indivíduos processados por pedido.

2.3.3 *SofEA* alterado

Face ao algoritmo *SofEA* apresentado em [28] e explicado em 2.3.2, os autores de [27] apresentam uma alteração ao algoritmo. Estes indicam que a falta e sobrecarga de cromossomas na *Pool* pode ser evitada se o cliente que tem o papel de avaliador for também *reaper*, ou seja, se o cliente que trata da avaliação efetuar também a seleção.

Os autores apresentam várias versões do algoritmo, cada uma com alterações ligeiras no sistema. Estas são:

SofEA1

Face aos problemas da versão anterior, decidiu-se juntar as etapas de avaliação e reprodução, num só cliente (em vez de dois clientes). Cada cliente pede um bloco de cromossomas à *Pool*, executa a reprodução e avalia os cromossomas resultantes. De seguida este volta a colocá-los na *Pool*. Este processo é repetido até que a solução seja encontrada. Adicionalmente, esta abordagem mantém o *reaper* independente, ou seja, assíncrono ao reprodutor/avaliador.

Visto que o reprodutor é mais rápido, o sistema está exposto a situações em que a população atingia o dobro do esperado, causando o *reaper* a demorar mais tempo a concluir.

SofEA2

Nesta versão o papel de *reaper* junta-se ao reprodutor e ao avaliador. Esta abordagem permite a um cliente adicionar e retirar elementos da *Pool* em apenas um pedido, removendo a necessidade destes saberem do estado global do problema, pois não é necessária uma contagem de indivíduos disponíveis. De modo a manter o número da população constante, o número de indivíduos adicionados deve ser o mesmo do número de indivíduos eliminados.

Contudo, apesar de obter melhores tempos, esta variante não é escalável, diminuindo o seu desempenho com o aumento de clientes. Esta versão também apresenta problemas quando vários clientes tentam eliminar o mesmo indivíduo, ao mesmo tempo, acabando por quebrar o equilíbrio entre os novos indivíduos inseridos e os indivíduos antigos eliminados. Esta alternativa também não garante que o número de indivíduos inseridos é o mesmo de indivíduos eliminados, resultando no crescimento da população e num elevado número de avaliações por solução.

SofEA3

Esta variante foca-se em manter a população constante, tentando que o número de indivíduos adicionados seja o mesmo de indivíduos removidos. Para tal recorre ao sistema de *ranking*, criando duas listas: uma lista contendo os novos indivíduos ordenados dos melhores para os piores; outra lista contendo indivíduos antigos, ordenados dos piores para os melhores. De seguida os novos indivíduos são inseridos e os antigos indivíduos apagados, pela ordem de *ranking*. Este processo termina quando um indivíduo antigo é melhor que o novo indivíduo, ou não restem mais indivíduos para inserir.

Com esta abordagem o desempenho aumenta significativamente e o número de avaliações diminui com o número de clientes. Contudo o problema de concorrência de dois clientes a apagar o mesmo indivíduo continua por resolver.

SofEA4

Esta versão resolve os conflitos de remoção de indivíduos recorrendo à avaliação de respostas do *CouchDB*. Em mais detalhe, quando um cliente pede a alteração de um conjunto de *b* indivíduos, o *CouchDB* envia a resposta contendo o identificador dos indivíduos que causaram conflitos. Através desta resposta os clientes podem verificar se os indivíduos em conflito fazem parte do conjunto de indivíduos que deviam de ter sido eliminados. No próximo ciclo o cliente pede o conjunto de $b + c$ indivíduos (sendo c o número de conflitos causados anteriormente). De todos os indivíduos recebidos, os c piores são automaticamente eliminados.

Contudo esta versão não toma em conta conflitos de inserção de novos indivíduos, podendo levar a um decrescimento da população.

Avaliação

Os autores concluem que embora o desempenho do sistema aumente com a adição de clientes, existe um limite dependente do número de indivíduos vivos (aptos para a reprodução). Concluem também que é necessário encontrar um equilíbrio entre o tamanho da população e o número de indivíduos que cada cliente escolhe.

2.4 Sistemas de Armazenamento para Implementação da *Pool*

Após o estudo do estado da arte dos modelos *Pool* existentes, concluímos que a escolha do sistema para implementação da *pool* é muito importante. Deste modo, decidimos utilizar um sistema de armazenamento distribuído como implementação. De seguida apresentamos o estudo de sistemas de armazenamento tolerantes a faltas que possam servir para implementar a *pool* de recursos partilhados.

2.4.1 *ZooKeeper*

Embora seja um sistema de coordenação, maioritariamente utilizado para guardar metadados, o *ZooKeeper* fornece algumas garantias de tolerância a faltas que podem ser relevantes. Em mais detalhe, descrito em [24], o *ZooKeeper* é um serviço de coordenação de processos distribuídos. Este serviço incorpora elementos tais como registos partilhados e distribuição de carga pelos diferentes servidores, garantindo um elevado desempenho. Garante também ordem *FIFO* (*First In First Out*) em operações realizadas para cada cliente, ou seja, a primeira mensagem a ser enviada por um cliente será sempre entregue antes de qualquer outra que seja enviada posteriormente pelo mesmo cliente.

O *ZooKeeper* apresenta uma API simples e permite aos seus clientes utilizarem as suas próprias primitivas sem que o serviço seja alterado (tais como *group membership*,

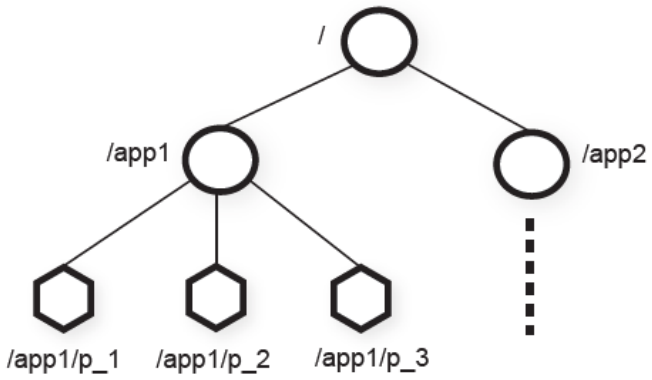

Figura 2.11: Exemplo de uma possível estrutura do *Zookeeper* [11]

rendezvous de clientes, entre outras).

A API do sistema fornece as operações *create*, *delete*, *exists*, *getData*, *setData*, *getChildren* e *synch*, com variantes síncronas e assíncronas. Adicionalmente, o *ZooKeeper* apresenta uma arquitetura em árvore, em que cada nó é denominado por *znode*. Estes *znodes* têm armazenamento interno, podem ter filhos (a não ser que seja um nó temporário, denominado como *Ephemeral*) e são referenciados através da notação Unix comum (*/app1/modulo1* como exemplo para referenciar o módulo1 na aplicação1).

Esta API foi implementada recorrendo a objetos não bloqueantes (descrito como *wait-free objects*) de modo a evitar mecanismos de bloqueio (como *locks*). Deste modo evita que um objeto execute uma série de ações e outros ficam à espera, garantindo que o bloqueio de um objeto não compromete outros. De modo a atingir uma elevada disponibilidade, o *ZooKeeper* utiliza replicação e implementa uma arquitetura simples de *pipeline*, que permite um número elevado de transporte de pedidos, atingindo latências muito reduzidas. Esta arquitetura permite que cada cliente envie pedidos de forma assíncrona, e que estes sejam executados na ordem *FIFO*, atingindo um elevado desempenho. Adicionalmente, o *ZooKeeper* também garante *linearizability* (é possível criar uma linha temporal de eventos, denominada *history*), através do protocolo *Zab*, um protocolo de *atomic broadcast* (algoritmo de *broadcast* que satisfaz ordem total). O sistema também fornece um mecanismo de relógio que, a pedido de cada cliente, o notifica após o surgimento de uma atualização, possibilitando que os clientes utilizem e mantenham as suas *caches* atualizadas. Este mecanismo evita pedidos contínuos de informação e, consequentemente, possibilita um elevado desempenho.

O modelo de dados do *ZooKeeper* consiste num mapa chave-valor e representando sistema de ficheiros hierárquico simples. Estes dados estão acompanhados de um *timestamp* e de contadores de versão. Estes são úteis para o processo de iniciação de um servidor da aplicação cliente ou, em casos de falha, possibilitando assim que estes obtenham informação sobre os nós do sistema, assim como do líder. Adicionalmente, as

[11] retirado de [24]

réplicas guardam em memória uma base de dados contendo toda a estrutura da árvore de dados. Após uma falha, o servidor necessita de recuperar esta base de dados, recorrendo a uma *snapshot* da base de dados. Após esta recuperação, o servidor necessita apenas de executar os comandos executados após essa *snapshot* ter sido tirada. Contudo o *ZooKeeper* não bloqueia os seus serviços durante o processo de tirar a *snapshot*, podendo levar a situações em que comandos são executados enquanto esta está a ser obtida (tornando-se numa fuzzy *snapshot*). Porém este acontecimento não apresenta problemas, pois os comandos são idempotentes (podem ser executadas múltiplas vezes, logo que seja na ordem certa). Adicionalmente quando um servidor recupera o protocolo *Zab* entrega as mensagens pela ordem a que elas foram executadas.

O *ZooKeeper* garante:

- *Linearizable writes*: todas as escritas respeitam uma sequência;
- *FIFO Client Order*: todos os pedidos de um certo cliente são executados pela ordem que são enviados.

2.4.2 Cassandra

O Cassandra é apresentado em [25] no âmbito de implementar o *Facebook Inbox Search*. Este é um sistema de armazenamento distribuído, tolerante a falhas, altamente escalável e confiável. Cassandra possibilita a gestão de grandes quantidades de dados, espalhados por vários servidores, fornecendo um sistema de elevada disponibilidade e sem um ponto único de falha.

O seu modelo de dados não segue o modelo relacional, normalmente presente nas bases de dados tradicionais, mas sim um esquema mais simples que possibilita aos seus clientes um controlo dinâmico sobre o formato e organização dos dados. Este fornece três operações: *insert*, *get* e *delete*.

Adicionalmente, cada nó pode ser visto como uma divisão em três módulos:

- partição;
- monitorização de membros e deteção de falhas;
- armazenamento.

Operações de escrita de dados são enviadas para todas as réplicas, e apenas são aceites se obtiverem resposta de um quórum. Operações de leitura de dados com origem em um cliente são entregues à réplica mais próxima ou a um conjunto de réplicas. Por fim, operações de leitura ou escrita de uma chave (que indexa um mapa/tabela) podem ser enviadas para qualquer nó, ficando este responsável por determinar quais as réplicas responsáveis pela chave.

A partição de dados é implementada através de funções de *hash* consistentes, onde o resultado é mapeado na forma de anel, ou seja, o valor seguinte ao maior valor de *hash* é o menor resultado desta função. Deste modo a cada nó é atribuído um intervalo de resultados da função de *hash*, ao qual fica responsável, indicando assim a sua posição no anel. Estes nós são denominados de coordenadores.

Contudo existem alguns inconvenientes à utilização destas funções, como por exemplo, o não reconhecimento da quantidade de carga ou do desempenho de um determinado nó. O Cassandra resolve esse problema analisando a carga do anel e levando nós menos sobrecarregados a moverem-se ao longo do anel para aliviar carga aos nós parceiros. Através da replicação, o Cassandra garante uma elevada disponibilidade assim como uma elevada tolerância a falhas.

Por fim, todas as operações de escrita e leitura são sequênciais, evitando assim a necessidade de abordar casos de concorrência, ou de implementar qualquer mecanismo de exclusão mútua. Após existir um número elevado de ficheiros de dados, é efetuado um processo de compactação, em que os vários ficheiros são combinados em um só. Contudo este processo é bastante intensivo e pode sobrecarregar o disco, podendo ser otimizado no futuro.

2.4.3 OpenDHT

O *OpenDHT*, apresentado em [29] e disponível em [7], é um sistema de *Distributed Hash Table* (*DHT*) implementado em C++11, baseado em [2]. Este sistema fornece:

- suporte a operações simples de *put* e *get*;
- elevada capacidade e disponibilidade;
- repartição de dados por um conjunto de elementos cooperativos;
- replicação de dados guardados;
- Biblioteca *DHT* C++11 Kademlia simples e fácil de utilizar;
- Repositório distribuído e partilhado de dados na forma de chave-valor;
- API simples e poderosa com a capacidade de guardar dados de forma arbitrária;
- Proteção criptográfica recorrendo a chaves públicas, possibilitando assinatura e cifra de dados (recorrendo a *GnuTLS*);
- Suporte para IPv4 e IPv6;
- Possibilidade de ligação com a linguagem *Python*.

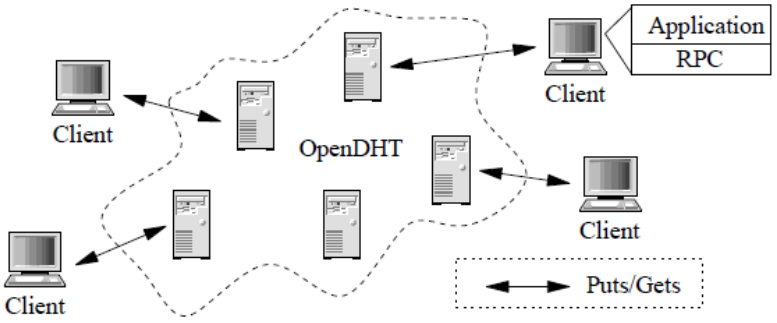

Figura 2.12: Estrutura do sistema *OpenDHT* [12]

O sistema é constituído por nós exteriores à infraestrutura, que representam os clientes, e nós interiores à infraestrutura que representam os servidores. Os servidores executam o código do *OpenDHT* enquanto que os clientes executam o código específico da aplicação. Estes clientes invocam o serviço do *OpenDHT* recorrendo a pedidos *Remote procedure call* (*RPC*). além de guardarem dados, os servidores aceitam pedidos de clientes. Esta abordagem permite que os clientes se abstraiam do sistema *DHT* e focarem-se no código da aplicação.

O sistema foi desenvolvido a pensar na facilidade de utilização (ou simplicidade) e na generalização. Este foi dividido em três interfaces diferentes, sendo que cada uma contém diferentes níveis de generalidade e simplicidade.

O *OpenDHT* pode ser visto como um repositório de dados público e, como tal, se oferecer as semânticas de sistemas de ficheiros comuns, acabará por ficar cheio, especialmente através de dados sem dono. Uma possível solução para este problema, recorreria à implementação de um sistema de recolha de lixo (*Garbage Collection*) contudo, implementar um que seja eficiente é uma tarefa difícil. Para tal o *OpenDHT* utiliza uma modificação da solução proposta em [30], em que se o disco se encontrar cheio, os novos dados sobrepõem os mais antigos. Na sua implementação, o *OpenDHT* difere em:

1. Apresenta um tempo de validação dos dados *TTL* bem definido. Se este tempo chegar ao final, os dados tornam-se inválidos. Desta forma, os clientes apenas têm de conter temporizadores para saberem quando necessitam de atualizar os dados.

2. Os clientes têm os mesmos direitos, tanto no acesso ao servidor como no limite máximo de dados que podem ter. O primeiro indica que mesmo que o servidor se encontre sobrecarregado, todos os clientes possuem a mesma chance de aceder ao sistema. O segundo indica que cada cliente pode ter um número ou tamanho de dados razoável, sem que exista nenhuma restrição. Contudo estas restrições surgem quando o sistema se encontra sobrecarregado.

[12] retirado de [29]

2.4.4 BFT-SMART

O BFT-SMART, apresentado em [14], é uma biblioteca de *Machine State Replication*, que garante as seguintes propriedades:

- Tolerância a faltas bizantinas, se estas não forem causadas por agentes maliciosos. Este tolera atrasos, perdas e corrupções de mensagens e faltas de *crash* ou de corrupção de estado por parte dos clientes;
- Simplicidade, de implementação em vez de alternativas por vezes mais otimizadas, mas mais complexas de gerir e de implementar;
- Modularidade, implementando o protocolo *Mod-SMaRt* que utiliza um modelo bem definido de consenso no seu núcleo;
- API Simples e extensível, permitindo que os programadores implementarem serviços determinísticos de maneira simples;
- Multi-Core, permite que o BFT-SMART tire o máximo partido da máquina em que se encontra, aumentando o seu desempenho.

O BFT-SMART divide os processos em dois grupos: réplicas e clientes e considera que cada processo tem um identificador único. Adicionalmente, durante uma execução, o sistema utiliza uma sequência de *views* para identificar a configuração atual (visto que as réplicas podem entrar e sair livremente).

Este requer que o modelo em que se insere seja eventualmente síncrono, de modo a garantir *liveness*, e requer também que os canais de comunicação entre máquinas (*point-to-point*) sejam confiáveis e autenticados. Para concretizar este último, o BFT-SMART recorre aos *Message Authentication Codes* (*MACs*) sobre o protocolo TCP/IP. O uso de *MACs* requer que os interlocutores possuam um par de chaves pública/privada. Adicionalmente, as suas funcionalidades núcleo incluem:

- *Total Order Multicast,* para garantir um consenso bizantino;
- *State Transfer*, para reparar réplicas que estejam com o seu estado corrupto ou atrasado, sem danificar o sistema;
- *Reconfiguration*, permitindo que réplicas possam ser adicionadas e removidas em tempo de execução, levando o sistema a crescer ou decrescer e, deste modo torna-lo dinâmico.

O BFT-SMART também permite algumas configurações avançadas que lhe permitem obter tolerância a falhas de *crash* e tolerante a intrusões.

2.4.5 DepSky

Em [12] os autores apresentam o *DepSky*, um sistema que utiliza uma combinação de *clouds* comerciais para guardar dados. O *DepSky* resolve algumas das limitações de sistemas *cloud* da seguinte maneira:

- Perda de disponibilidade: Quando se trata de dados que estão guardados remotamente, é necessário garantir que problemas de disponibilidade causados tanto pela perda de ligação à Internet como por falhas nos sistemas de *cloud* não impossibilitam os utilizadores de aceder aos seus dados. Para lidar com este problema, o *DepSky* recorre à replicação de dados por várias *clouds*, permitindo que num dado momento exista sempre um certo número de *clouds* disponíveis.

- Perda ou corrupção de dados: Existem ocorrências em que os sistemas de *cloud* perderam ou corromperam dados de utilizadores e que, em alguns casos, estes não os conseguiram recuperar ([Naone 2009] referenciado em [12]). Para resolver este problema, o *DepSky* recorre a replicação de dados tolerante a faltas bizantinas em vários serviços de *cloud*. Deste modo permite ao cliente aceder aos seus dados mesmo que estes tenham sido corrompidos ou perdidos em algumas *clouds*.

- Perda de privacidade: Os fornecedores dos serviços de *cloud* têm acesso tanto aos dados guardados na sua *cloud* como à maneira de como estes são guardados. Mesmo que estes fornecedores sejam de confiança, existe sempre a possibilidade de agentes maliciosos internos. Uma possível solução leva os clientes a não guardar os seus dados em claro, mas sim cifrados. Contudo, quando se trata de várias aplicações distribuídas que partilham o mesmo serviço de *cloud*, é necessário executar protocolos de distribuição de chaves, ou seja, o processo cliente que cifra os dados, tem de fornecer a chave de cifra aos outros clientes para que eles possam decifrar os dados. Para tal o *DepSky* utiliza um esquema de segredos partilhados e *erasure codes* para evitar que os dados sejam guardados de forma desprotegida e para evitar também que estes sejam corrompidos.

- *Vendor Lock-in*: Este fenómeno [9] ocorre quando a ligação entre o cliente e o fornecedor é muito forte, tornando o processo de troca de fornecedor pouco ou nada viável. Um exemplo deste problema é o custo associado a uma operação de movimento de dados entre fornecedores. Para resolver este problema o *DepSky*: não depende de uma única *cloud*, permitindo que os acessos aos dados possam ser distribuídos pelos vários fornecedores de modo a respeitar as suas regras; Recorrendo aos *erasure codes*, o *DepSky* permite que os dados possam ser repartidos por várias *clouds*, originando custos reduzidos caso seja preciso mover para outro fornecedor, pois apenas é necessário passar uma fração dos dados.

Contudo, as medidas apresentadas anteriormente trazem com alguns custos adicionais, como tempos de execução mais elevados devido à utilização de várias *clouds*. Um dos principais objetivos do *DepSky* é reduzir estes custos.

Modelo de dados

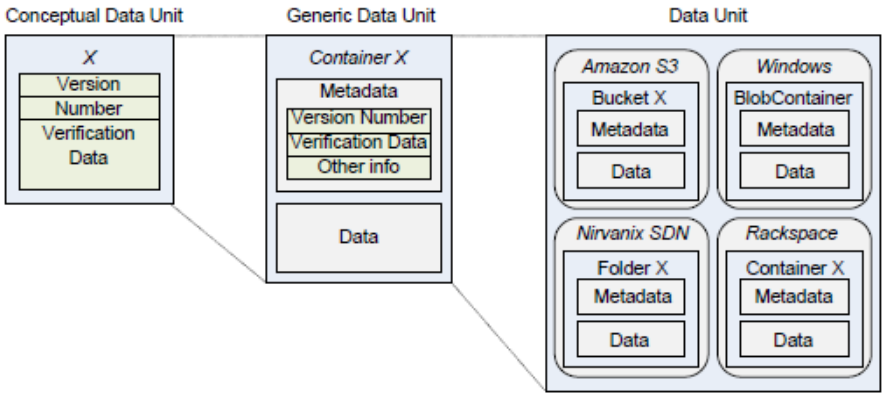

Figura 2.13: Modelo de dados do *DepSky* [13]

Com a utilização de várias *clouds* surge o problema de que cada *cloud* tem de ser acedida pela sua própria interface, sem que esta sofra modificações. Para lidar com a heterogeneidade das interfaces dos diferentes fornecedores de serviços de *cloud*, o *DepSky* recorre ao seu modelo de dados apresentado na Figura 2.13. Este está dividido em três partes, cada uma com o seu nível de abstração, e permite que os utilizadores se abstraiam dos detalhes das diferentes *clouds* quando desenvolvem os seus algoritmos.

Arquitetura

Os autores consideram presente um sistema assíncrono composto por três tipos de participantes: escritores, leitores e fornecedores dos serviços de *cloud*, representados na Figura 2.14. Os clientes fazem o papel de leitores e escritores (não necessariamente em clientes diferentes).

Os leitores podem sofrer falhas arbitrárias, ou seja, além de poderem sofrer um *crash* podem apresentar um comportamento arbitrário. Por outro lado assume-se que os escritores podem apenas falhar por *crash*. Todos os escritores possuem uma chave privada, de modo a assinarem os dados que escrevem no sistema, e contêm também as chaves públicas dos outros participantes, de modo a verificarem as suas assinaturas. Estas chaves podem ser distribuídas utilizando o sistemas de *cloud*. O sistema assume também o uso de uma função de *hash* criptográfico resistente a colisões.

[13] retirado de [12]
[14] retirado de [12]

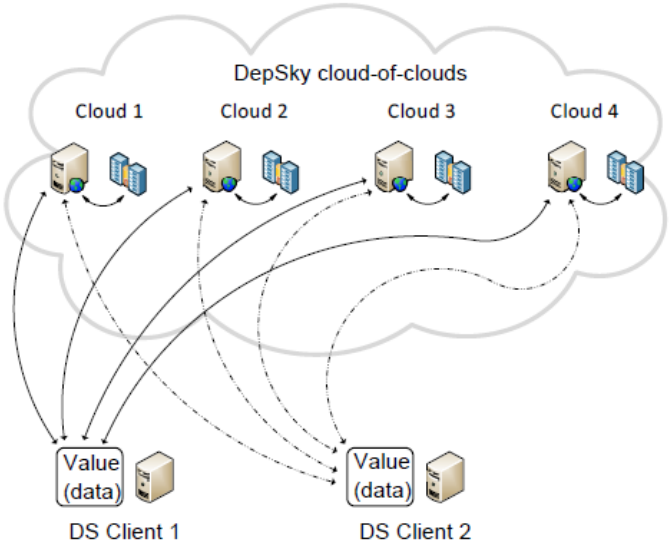

Figura 2.14: Arquitetura do *DepSky* [14]

Cada sistema de *cloud* está modelado para suportar cinco operações, acedidas através de *RPC* (*Remote Procedure Calls*).

1. *list*: Lista todos os ficheiros de um contentor na *cloud*;
2. *get*: Lê um ficheiro;
3. *create*: Cria um contentor;
4. *put*; Escreve ou modifica um ficheiro no contentor;
5. *remove*: Remove um ficheiro;

Estas continuam a ser invocadas até que os servidores recebam uma resposta ou que a operação seja cancelada. Visto que os sistemas de *cloud* são idempotentes, não existe problema em repetir as operações.

Adicionalmente, considera-se que as *clouds* são um sistema de armazenamento passivo, ou seja, que estas não executam nenhum código do protocolo que não seja das operações em cima referidas. Considera-se também que estas *clouds* tratam do controlo de acessos, ou seja, que os leitores apenas podem executar as funções *list* e *get*. Por último assume-se que as *clouds* podem falhar de forma bizantina [12], ou seja, que os dados podem ser eliminados, corrompidos, criados ou roubados por utilizadores não autorizados. Adicionalmente, de modo a suportar até f faltas bizantinas, o *DepSKy* requer $3f + 1$ serviços de *cloud*.

Protocolos

O *DepSky* implementa os protocolos *DepSky-A* e *DepSky-CA* de modo a obter as condições mencionadas previamente. Em mais detalhe, cada protocolo está responsável por:

- *DepSky-A - Available DepSky*

  O protocolo *DepSky-A* tem como objetivo aumentar a disponibilidade e a integridade dos dados guardados em *cloud*, recorrendo a mecanismos de replicação baseadas em *quorum*, suportando várias escritas para um registo. Adicionalmente, o *DepSky* recorre a um mecanismo de escritor único e vários leitores (*single-writer multi-reader*) para um registo e implementa um mecanismo de bloqueio de dados (*lock/lease*) para a unidade de dados.

- *DepSky-CA - Confidential & Available DepSky*

  O protocolo *DepSky-A* tem como objetivo resolver os problemas de guardar os dados em claro (sem qualquer cifra) e o problema ocupar demasiado tempo e espaço a replicar os dados por *n clouds*. Para resolver estas limitações, o protocolo *DepSky-CA* cifra os dados recorrendo a criptografia simétrica e recorre aos *erasure codes* para repartir os dados em blocos.

Protocolos de Extensão

Além dos protocolos anteriormente apresentados, o *DepSky* permite a utilização de outros protocolos que sejam úteis em casos reais. Estes são:

- Mecanismos de *Lock* utilizando as *clouds*

  Visto que o *DepSky* não suporta vários escritores, recorre o mecanismo *low contention lock mechanism*, que utiliza o sistema como mecanismo de *locks*. Para tal são criados ficheiros que indicam qual o escritor que tem acesso à unidade de dados e por quanto tempo. Contudo esta solução assume que os relógios dos escritores estão sincronizados e que as *clouds* apresentam uma consistência do tipo *regular semantics* [12]

- Operações de gestão

  O *DepSky* permite que os seus participantes executem operações de criação e destruição de unidades de dados, de *garbage collection* e de reconfiguração da *cloud*.

Controlo de acesso

As operações de criação e destruição de unidades de dados apenas podem ser executadas pelo seu criador, enquanto que as restantes operações podem ser executadas por todos os escritores da respetiva unidade de dados.

2.4.6 DepSpace

O *DepSpace* apresentado em [15] é um sistema de coordenação de processos que recorre a um espaço de tuplos partilhado.

O sistema é composto por um número ilimitado de clientes e por um conjunto definido de servidores, e ambos contêm um identificador único. Os servidores implementam um espaço de tuplos confiável [11] (ou *dependable*), ou seja, o sistema de tuplos garante propriedades como:

- *reliability*: As operações no espaço de tuplos têm de estar de acordo com a sua especificação;
- *availability*: O espaço de tuplos tem de estar disponível para executar os pedidos;
- *integrity*: o espaço de tuplos não pode permitir uma alteração indesejada;
- *confidentiality*: Os dados contidos no espaço de tuplos não podem ser revelados a entidades não autorizadas.

O sistema de tuplos partilhado fornece as seguintes operações, sendo *t* um tuplo e $\bar{t}$ um *template* de um tuplo:

- *out(t)*: Insere um tuplo *t* no espaço de tuplos;
- *rdp($\bar{t}$)*: Lê um tuplo que corresponda ao *template* $\bar{t}$ e retorna *true*. Caso não exista nenhuma correspondência, retorna *false*;
- *inp($\bar{t}$)*: Lê e remove um tuplo que corresponda ao *template* $\bar{t}$ e retorna *true*. Caso não exista nenhuma correspondência, retorna *false*;
- *rd($\bar{t}$)*: Lê um tuplo que corresponda ao *template* $\bar{t}$. Bloqueia e espera por alguma correspondência caso contrario;
- *in($\bar{t}$)*: Lê e remove um tuplo que corresponda ao *template* $\bar{t}$. Bloqueia e espera por alguma correspondência caso contrario;
- *cas($\bar{t}$, t)*: Se não existir nenhum tuplo que corresponda ao *template* $\bar{t}$, insere *t* e retorna *true*. Caso contrário retorna *false*.

Arquitetura

Toda a comunicação entre clientes e servidores é feita recorrendo a canais *point-to-point* confiáveis e autenticados (*reliable authenticated point-to-point channels*), tornando-a resistente não só a perdas mas também a corrupção e a atrasos. De modo a que todas as réplicas executem a mesma sequência de operações o *DepSpace* recorre a um protocolo de ordem total baseado no protocolo *paxos* de consensos bizantino [15].

A arquitetura do *DepSpace*, apresentada na Figura 2.15, encontra-se divida em várias partes, cada uma responsável por garantir as propriedades que tornam o sistema confiável.

[15] retirado de [23]

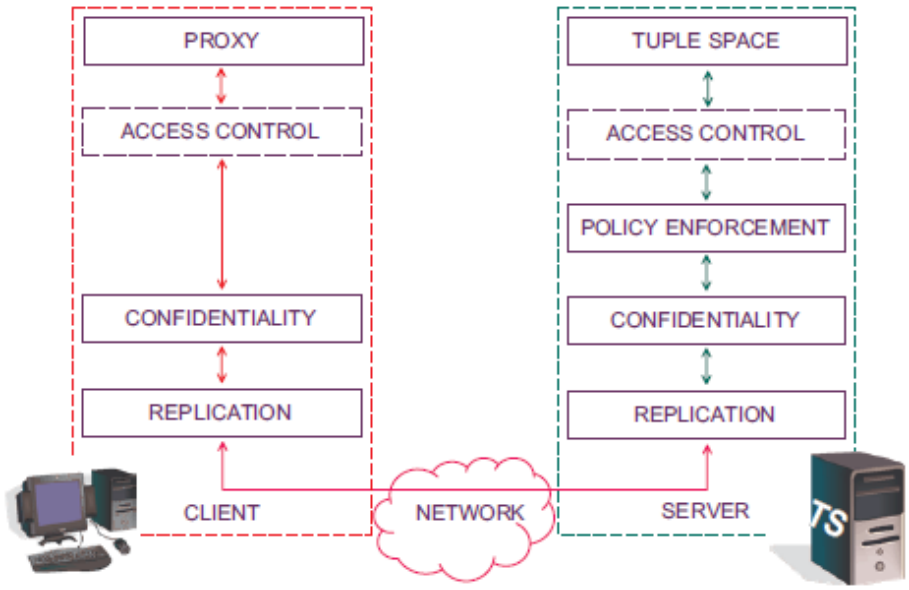

Figura 2.15: Arquitetura do *DepSpace* [15]

Funcionalidades

O sistema recorre à replicação de dados de modo a garantir que o sistema continua disponível (*availability*) e que as operações continuam a funcionar corretamente (*reliability* e *integrity*) na presença de um certo número de falhas. As ultimas apenas são possíveis se o número de réplicas respeitar o fator $3f + 1$, de modo a tolerar até f faltas bizantinas.

De modo a garantir confidencialidade, o sistema cifra os dados do cliente. Contudo, com esta cifra, surge o problema de comparação de tuplos. Para tal um cliente ao inserir um tuplo no sistema, pode escolher entre três modos:

1. *public*: Os campos não são cifrados e podem ser comparados de forma arbitrária. Contudo, visto que os dados não estão cifrados, podem ser vistos por agentes não autorizados.

2. *comparable*: Os campos são cifrados e guardados juntamente com um *hash* calculado a partir dos dados cifrados. Considera-se que a função de *hash* é resistente a colisões.

3. *private*: Os campos são cifrados e guardados sem nenhum valor de *hash*, tornando impossível qualquer tipo de comparação.

Adicionalmente, o controlo de acesso é feito através da camada de controlo de acesso dos servidores. Em mais detalhe, de modo a executar uma operação, os clientes enviam as suas credenciais juntamente com a operação desejada para os servidores. Posteriormente os servidores corretos respondem de acordo com as suas políticas de acesso, enquanto que servidores faltosos respondem de forma arbitrária. As políticas de acesso tomam em

conta o identificador do cliente, a operação desejada, os seus argumentos e os tuplos que estão atualmente no espaço de tuplos. Ao receber *f + 1* respostas negativas, o cliente aceita a rejeição.

2.4.7 Shared Cloud-backed File System (SCFS)

O SCFS é um sistema que permite a entidades partilharem ficheiros de um modo seguro e tolerante a faltas. Este garante uma consistência forte nos acessos a ficheiros e disponibiliza uma interface que permite ligar a vários serviços de *cloud* diferentes.

além das técnicas usuais dos sistemas de ficheiros existentes (S3FS [3], S3QL [1], entre outros), o SCFS implementa algumas técnicas novas:

- *Always write / avoid reading*: Em caso de escrita o SCFS guarda as alterações localmente e envia-as para a *cloud*, enquanto que as Leituras são feitas localmente sempre que possível. Esta abordagem diminui o tempo (*latency*) de leitura e os custos de utilização de *cloud*, visto que os custos de leitura de um ficheiro em *cloud* são maiores que o custo de escrita.

- *Modular coordination*: O SCFS utiliza um sistema de coordenação tolerante a faltas em vez de um mecanismo de *locks* e de gestão de dados embebido. Esta abordagem permite que o sistema de coordenação assista na gestão de consistência e de partilha de ficheiros.

- *Private name spaces*: O sistema utiliza uma nova estrutura de metadados sobre dados privados, que são guardados como um simples objeto na *cloud*. Deste modo evita que o sistema de coordenação mantenha essa informação, aumentando o desempenho do sistema.

- *Consistency anchors*: O SCFS utiliza este mecanismo para atingir consistência forte (*Strong consistency* em vez de uma eventualmente consistente (*Eventualy consistent*). Este mecanismo também fornece a abstração de um sistema de ficheiros que não necessita de modificações dos serviços de *cloud* para funcionar.

- *Multiple redundant cloud backends*: Os administradores do sistema podem optar por integrar um *backplane* de categoria *cloud-of-clouds* (como o *DepSky* [12], apresentado em 2.4.5), tornando o sistema tolerante não só à corrupção de dados mas também à indisponibilidades dos fornecedores dos sistemas de *cloud*. Adicionalmente, todos os dados guardados na *cloud* são cifrados de modo a fornecer confidencialidade e codificados para aumentar a eficiência do sistema.

O SCFS foi concebido com as seguintes ideias centrais:

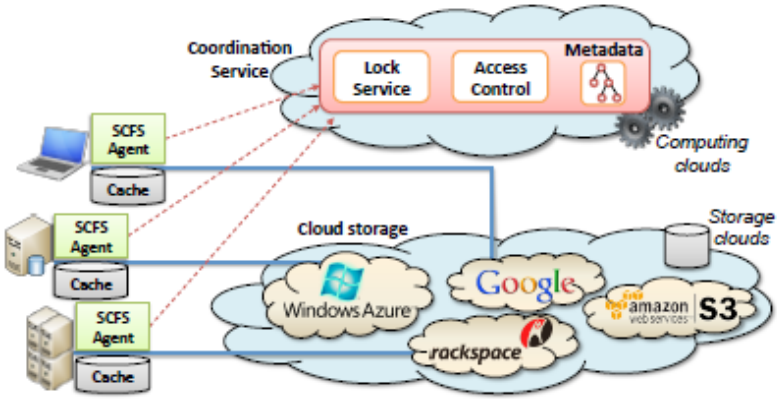

Figura 2.16: Arquitetura do SCFS [16]

1. pagamento por criação (*pay-per-ownership*): indica que os utilizadores do sistema apenas devem pagar pelos ficheiros que criam e não pelos seus acessos;

2. consistência forte (*Strong consistency*): o sistema suporta uma semântica de consistência ao fechar de um ficheiro, ou seja, se um utilizador fechar um ficheiro, as suas alterações serão visíveis por todos os outros utilizadores. Caso esta não seja necessária, o SCFS também suporta consistências mais fracas;

3. reconhecimento de serviços (*service-agnosticism*): o sistema apenas apresenta funcionalidades que são suportadas pelas suas *clouds*;

4. multi-versão (*multi-versioning*):. O SCFS mantém versões de ficheiros antigos e apagados até que sejam removidos definitivamente por um coletor de lixo (*garbage collector*).

Arquitetura

O sistema é constituído por três componentes, apresentados na Figura 2.16: o *backend cloud storage* que mantém os dados (representado na Figura 2.16 por *cloud-of-clouds*, contudo pode ser utilizada uma *single cloud*); o sistema de coordenação (*coordination service*), que gere os metadados e exerce alguma sincronia; o agente SCFS (*SCFS Agent*) que implementa a grande parte das funcionalidades do SCFS e que corresponde ao sistema de ficheiros montado na máquina do cliente

Adicionalmente, o sistema mantém os dados na *cloud* e localmente na forma de *cache*, originando um elevado desempenho, custos reduzidos e uma maior disponibilidade. Todas as alterações efetuadas nos ficheiro locais, são gravadas na *cloud*.

[16] retirado de [23]

Consistency anchor

De modo a fortalecer a consistência do sistema, o SCFS recorre a um sistema externo denominado *consistency anchor* (CA). Este sistema externo está encarregue de gerir os metadados enquanto que o *storage service* (SS) gere os ficheiros.

Ao utilizar um sistema de coordenação externo para gerir os seus metadados, o SCFS torna-se mais flexível, possibilitando que este seja implementado de várias maneiras, dependendo das necessidades do utilizador.

O SCFS *agent* fornece quatro operações de interação com os ficheiros, estas são:

1. Abertura de ficheiros (*open*);
2. Escritas e leituras (*write* e *read*): Operações que apenas interagem com o armazenamento local;
3. Fechar um ficheiro (*close*): Requer que exista uma sincronização dos dados em *cache* local com os dados do sistema de armazenamento *cloud*;

Adicionalmente, o SCFS guarda versões antigas dos ficheiros, ou seja, sempre que um utilizador elimina um ficheiro o SCFS apenas marca-o para eliminação, nos seus respetivos metadados. Este mecanismo possibilita ao cliente recuperar os seus ficheiros. Contudo, com a acumulação de vários ficheiros de versões diferentes, o sistema atinge o limite de armazenamento. Para resolver o problema, a componente possui um mecanismo de coleção de lixo (*garbage collection*), que quando executado (internamente em cada SCFS *agent*) elimina todos os ficheiros indesejados.

2.4.8 CHARON

Apresentado em [26], o CHARON é um sistema de armazenamento baseado em *cloud*, capaz de armazenar e partilhar *big data* com a menor gestão e infraestrutura dedicada possível. O CHARON recorre a um processo de replicação *multi-cloud* (ou *cloud-of-clouds*, como o *DepSky* [12] apresentado em 2.4.5). Deste modo evita que exista um ponto único de falha, pois não tem apenas um fornecedor de serviço de *cloud*, evita que o serviço não funcione corretamente caso alguns dos fornecedores não se encontrarem disponíveis e evita também que o sistema tenha comportamentos inesperados (caso existam *bugs* ou caso existam serviços de *cloud* maliciosos). Adicionalmente, além de permitir que o utilizador armazene os seus dados em várias *clouds*, o CHARON permite também a utilização de apenas uma *cloud* ou de repositórios privados. O sistema aplica um conjunto de algoritmos resistentes a faltas bizantinas, centrados nos dados [8].

De modo a lidar com *big data*, o sistema divide os ficheiros em partes, recorrendo a: *erasure codes* para a recuperação de dados; algoritmos de compressão de dados; mecanismos de receção e envio de dados em segundo plano (*prefetching* e *background uploads*) para disfarçar o tempo de comunicação com as *clouds*. Adicionalmente o sistema

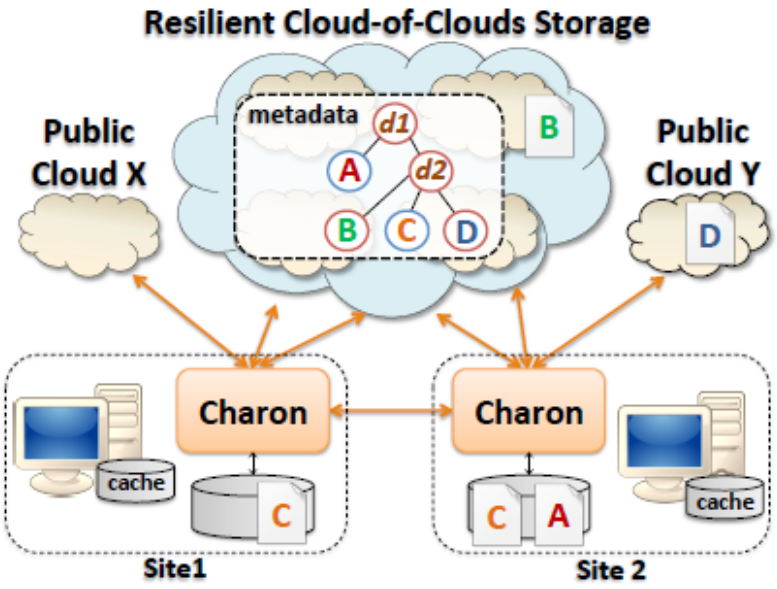

Figura 2.17: Arquitetura do CHARON [17]

apresenta um mecanismo de exclusão mútua inovador, implementado recorrendo aos fornecedores de serviços de *cloud*. Contudo, este requer que *3f + 1* desses serviços estejam disponíveis, de modo a que o mecanismo seja tolerante a f faltas. Adicionalmente esta abordagem facilita a implementação de mecanismos adicionais, como filas de espera.

Arquitetura

O CHARON é composto por um conjunto de clientes com espaço de armazenamento local limitado e por um grupo de fornecedores de serviços de *cloud*. Considera-se que os serviços de *cloud* possuem mecanismos de acesso de controlo, permitindo acesso apenas a clientes autorizados.

O sistema fornece uma interface POSIX de modo a controlar o sistema de serviços de *clouds* e de modo a permitir a transferência de dados entre clientes. Adicionalmente, este guarda os ficheiros na máquina do cliente e envia as atualizações para o local de armazenamento na *cloud*, em segundo plano. Deste modo o sistema continua responsivo apesar do tempo de comunicação com as *clouds*.

O CHARON foi criado com três abstrações de computação distribuída em mente:

- Registos *one-shot*: Todas as escritas de um objeto são finais. Adicionalmente, de modo a garantir confidencialidade, o sistema cifra todos os dados e guarda a chave numa *cloud* segura. Esta aplica registos *single-write multi-reader*;

- Registos *single-writer multi-reader* (SWMR): O sistema recorre a uma versão otimizada do *DepSky* [12] (apresentado também em 2.4.5), pois este oferece uma

[17] retirado de [26]

abstração de registos *single-writer multi-reader* construido em cima de *3f + 1* serviços de *cloud*;

- *Leasing object*: De modo a tolerar concorrência de escritas, o CHARON utiliza um objeto de *leasing* tolerante a faltas bizantinas. Este objeto é responsável por garantir que dois ou mais escritores não acedem ao mesmo ficheiro em simultâneo (exclusão mútua);

Adicionalmente o CHARON separa os ficheiros de dados dos metadados em objetos distintos, guardados em várias *clouds* (*cloud-of-clouds*), em uma única *cloud* ou em uma localização privada. Deste modo possibilita que o utilizador explore cada opção e compare os seus pontos positivos e negativos.

De modo a manter a estrutura do sistema de ficheiros seguro, o CHARON replica a árvore de nomes juntamente com os ficheiros de metadados pelas várias *clouds*. Esta abordagem possibilita também que estes dados sejam mais facilmente acedidos, fornecendo uma alta disponibilidade. Adicionalmente esta abordagem permite também que os clientes sejam libertados da responsabilidade de manter esta informação, ou seja, se um cliente sofrer uma falta por *crash*, os seus dados podem ser reconstruidos através de pedidos às *clouds*, e o sistema continua.

Capítulo 3

Arquitetura do Sistema

Neste capítulo apresentamos a arquitetura e componentes do nosso sistema, assim como as suas premissas. Discutimos também as escolhas da arquitetura e do desenho de algumas funcionalidades.

3.1 Análise

Após o estudo dos vários modelos de sistemas que possibilitam a integração de AEs, construímos a Tabela 3.1 com as vantagens e desvantagens de cada modelo.

Concluímos que a utilização do modelo *Master-Slave* implicaria dificuldades adicionais, uma vez que o *Master* apresenta um ponto único de falha no sistema e condiciona fortemente o desempenho do sistema global [23]. Adicionalmente, este sistema não corresponde aos nossos objetivos de construir uma infraestrutura distribuída, flexível e tolerante a faltas.

optámos por não escolher o modelo *island* pois este apresenta uma escalabilidade questionável que vai contra o nosso objetivo de um sistema altamente escalável. Mais especificamente, o desempenho destes sistemas está ligado ao número de ilhas e, não aumenta diretamente com a adição de novas ilhas ao sistema. Ou seja, a uma certa altura, com a adição de novas organizações, sendo que cada organização é uma ilha, o sistema não sofreria um aumento significativo no desempenho. Adicionalmente, como referido na Tabela 3.1, o número de ilhas e a sua cooperação encontram-se restritos à topologia da rede.

Os modelos *cellular* e *hierarchical* apresentam algumas restrições em relação à topologia da rede em que o sistema é instalado ou restrições ao tipo de problemas a serem resolvidos [23]. Deste modo, optámos por evitar estes modelos, visto que pretendemos desenvolver um sistema genérico que não imponha restrições no tipo de redes nem nos tipos de problemas. Adicionalmente, visto que o sistema *hierarchical* combina modelos, herda não só as suas vantagens como as suas desvantagens.

O modelo *coevolution* requer que o problema seja decomposto em sub-problemas

Modelos	Vantagens	Desvantagens
Master-Slave	- Simples de implementar;	- Controlo centralizado; - Desempenho do sistema condicionado pelo desempenho do mestre;
Island	- Favorece pesquisa global dos AEs;	- Desempenho e escalabilidade limitada; - Elevado número de parâmetros de configuração, que podem não ser iguais entre os participantes (sistema heterogéneo); - Topologia da rede restringe o número de ilhas e a cooperação espontânea entre ilhas; - Parâmetros de configuração muito sensíveis e necessitam de uma informação à priori;
Cellular	- Os bons indivíduos podem-se propagar pelo sistema;	- Desempenho dependente e limitado à topologia da rede escolhida; - Topologia da rede restringe o número de células e a cooperação espontânea entre ilhas;
Hierarchical	- Toma partido das vantagens dos modelos escolhidos;	- Desempenho e tolerância a faltas dependente dos modelos escolhidos, tornando esta escolha bastante importante;
Coevolution	- Os nós efetuam trabalho independentemente, recorrendo pouco à comunicação;	- Decomposição do problema pode ser muito difícil, ou até impossível; - Avaliação de soluções e a monitorização de diversidade têm de ser resolvidos à priori; - Evolução de componentes independentes;
Multi-agent	- Não requer coordenação de agentes para atingir o objetivo; - Participantes são independentes (loosely coupled); - Controlo do sistema descentralizado;	- Difícil criar uma visão do sistema inteiro; - Difícil gestão da população (expansão ou regressão);
Pool	- Participantes são independentes (loosely coupled); - O conjunto de participantes pode ser dinâmico (mesmo que alguns sofram um crash); - *Cost-efficient*, possibilitando que voluntários doem algum poder computacional;	- A escolha de representação da *pool* é bastante importante;

Tabela 3.1: Sumário dos modelos

menores e origina a evolução de componentes independentes. Visto que nem todos os problemas podem ser decompostos, o modelo contrariando o nosso objetivo de suportar todo o tipo de algoritmos evolutivos. Adicionalmente, visto que os seus participantes são independentes, existe dificuldade na monitorização de diversidade, influenciando a qualidade dos resultados do AE implementado.

O modelo *multi-agent* possui algumas características desejadas, como o controlo descentralizado, elevada escalabilidade e elevada flexibilidade na entrada e saída de participantes. Contudo, este dificulta o processo de monitorização e controlo da população, podendo afetar a qualidade dos resultados do AE implementado.

Por fim, optámos por implementar um sistema baseado em *pool* de recursos partilhados. Contudo, existem problemas adicionais que necessitam de ser resolvidos, como a escolha da implementação desta *pool*. Após o estudo do estado da arte de modelos *Pool*, concluímos que as implementações mais frequentes da *Pool* recorrem a um espaço de tuplos [21] partilhado ou a uma base de dados [28] [27] partilhada.

A escolha da implementação da *pool* é muito importante, uma vez que esta pode introduzir problemas graves e representar um ponto de falha único no sistema. Deste modo optámos por recorrer a um sistema de armazenamento distribuído.

Apesar de apresentar uma *API* simples com variantes síncronas e assíncrona, optámos por não recorrer ao *ZooKeeper*, pois este é um sistema de coordenação. Em outras palavras, este guarda principalmente ficheiros de informação (*metadata*) de pequeno tamanho. Este facto contraria a nossa necessidade de guardar informações de uma população de tamanho indeterminado e de guardar um histórico da população em cada geração.

Apesar de apresentar uma arquitetura distribuída, tolerante a faltas e escalável, optámos por não recorrer ao *Cassandra*, pois este não apresenta mecanismos de *lock* de dados. Sem este mecanismo, possivelmente teríamos de recorrer a um sistema de coordenação externo ou a outro método não nativo ao sistema. Este mecanismo é necessário pois evita que vários *workers* trabalhem sobre os mesmos dados, ou seja, evitando a realização de trabalho redundante.

optámos por não recorrer ao *OpenDHT* pois, em semelhança ao *Cassandra*, este também não apresenta um mecanismo de *lock* de dados. Adicionalmente, este apresenta um modelo de consistência eventual (*eventual consistency*). Deste modo, após um participante avaliar um indivíduo, os seus resultados não são imediatamente vizinhos para todos os outros participantes. Em consequência, este modelo possibilita que os *workers* iniciem o processo de avaliação de indivíduo avaliado previamente.

Adicionalmente, atribuímos alguma prioridade aos sistemas restantes, pois estes foram desenvolvidos na nossa faculdade, levando a algumas facilidades no contacto com os seus criadores no caso de surgirem problemas ou dúvidas adicionais. Contudo, optámos por não recorrer ao *DepSpace* pois, em semelhança ao *ZooKeeper*, é um sistema de coordenação usado para o armazenamento de metadados (*metadata*).

optámos por não recorrer ao BFT-Smart pois, apesar deste apresentar uma biblioteca tolerante a faltas bizantinas, apresenta poucas funcionalidades adicionais, demonstrando ter pouca facilidade de utilização. Adicionalmente, visto que os sistemas distribuídos *DepSky*, SCFS e CHARON utilizam o BFT-Smart e oferecem funcionalidades adicionais, não existe razão que justifique a sua utilização.

Apesar do *DepSky* fornecer a possibilidade de utilização de *clouds* como armazenamento, optámos por não o utilizar pois, em semelhança a sistemas mencionados anteriormente, não possui um mecanismo de *lock* de dados robusto. Por outro lado, este é utilizado nos sistemas distribuídos SCFS e CHARON que oferecem funcionalidades e garantias adicionais, não existindo assim razão que justifique a sua utilização.

Por último, face aos sistemas restantes, optámos pelo SCFS pois este supera o CHARON nas seguintes propriedades:

- consistência forte: todas as criações ou alterações efetuadas num indivíduo serão observadas por todos os participantes e mais importante, pelo *Master*, no momento em que são feitas. Considerando o nosso objetivo de reduzir ao máximo o tempo de execução, uma consistência eventual (*eventualy consistent*) fornecida pelo CHARON e pelos outros sistemas não é suficiente;

- apresenta um sistema de *locks* mais favorável: o CHARON apresenta um sistema de *locks* por *namespaces*. Neste sistema um *worker* bloqueia o *namespace* que contém o ficheiro que pretende atualizar, deixando os restantes *workers* que pretendam atualizar outro ficheiro no mesmo *namespace* à espera. Por outro lado, o SCFS apenas bloqueia os ficheiros que estão a ser atualizados, permitindo assim que vários *workers* trabalhem sobre as mesmas pastas, logo que não seja sobre o mesmo indivíduo.

Adicionalmente, o SCFS fornece algumas garantias adicionais:

- Tolerante a faltas bizantinas: não só prevenindo que a *pool* seja um ponto único de falha, mas também prevenindo que esta seja preenchida com indivíduos corrompidos;

- Elevado desempenho, flexibilidade e disponibilidade;

- Permite a utilização de armazenamento privado e de *clouds* públicas;

- Garante confidencialidade dos dados;

3.2 Arquitetura

Ao desenvolver o nosso sistema, assumimos que:

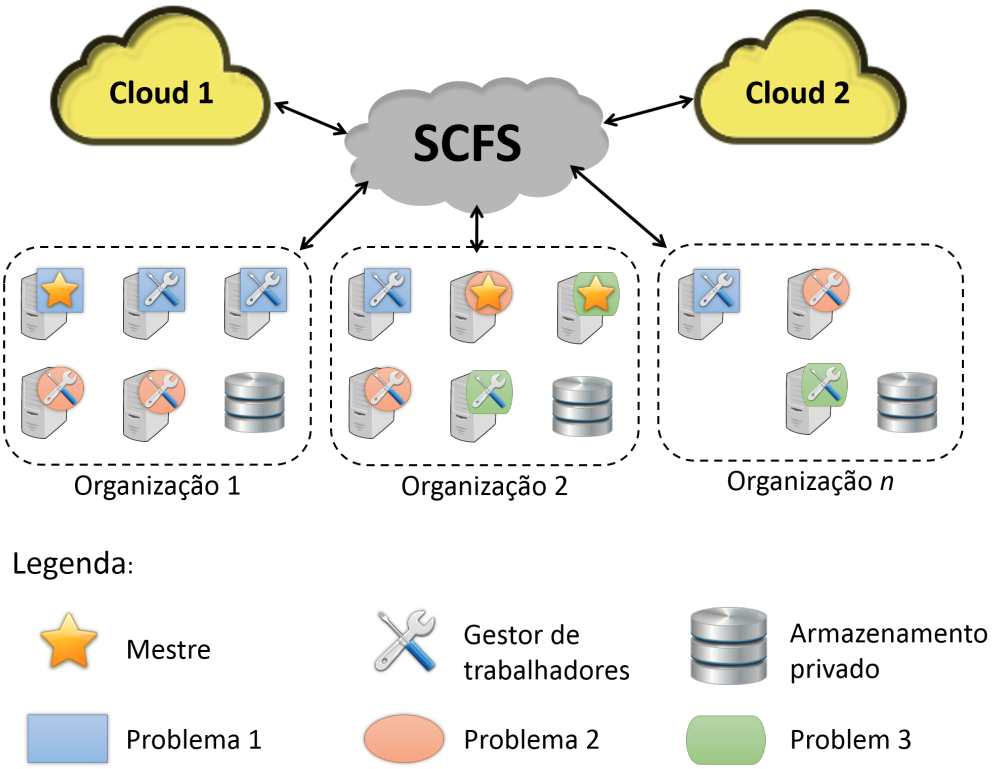

Figura 3.1: Arquitetura do sistema

- Toda a comunicação e distribuição de chaves entre organizações é feita previamente, recorrendo a canais seguros;
- Todos os clientes SCFS são autenticados antes de participarem no sistema;

O nosso sistema, representado na Figura 3.1, é constituído por três componentes principais:

1. armazenamento, representado pelas *clouds* públicas e pelas máquinas de armazenamento privado em cada organização;
2. *pool* de recursos partilhados, representada pelo SCFS;
3. participantes, representados pelas diferentes organizações.

O SCFS permite-nos utilizar não só *clouds* públicas, como também servidores privados para armazenar dados. Através deste, os participantes (organizações) podem cooperar na execução de problemas evolutivos. Cada organização pode destacar várias máquinas para a execução de problemas. Em mais detalhe, as organizações escolhem que máquinas dedicar para a execução de determinado problema. Estas podem também utilizar os seus armazenamentos privados em vez ou em conjunto com *clouds*.

Cada máquina contém um cliente SCFS e um *Daemon*. Cada cliente SCFS pode participar na resolução de vários problemas ao mesmo tempo, enquanto que o *Daemon* verifica se a máquina (ou cliente SCFS) foi registada num problema e, caso necessário, inicia a sua participação neste.

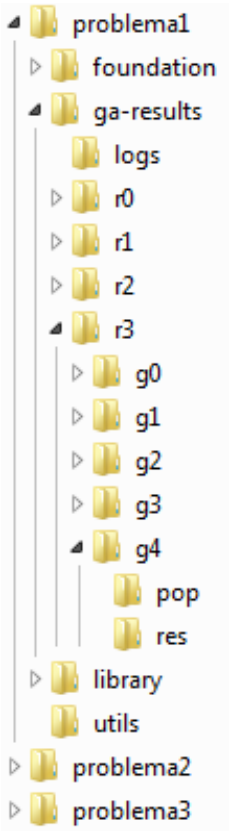

Figura 3.2: Estrutura de ficheiros da *Pool*

A Figura 3.2 apresenta a estrutura dos ficheiros de dados presentes na *Pool*. Podemos observar que cada problema tem a sua pasta própria. Cada pasta contém o código do AE a ser executado pelos participantes (`foundation`, `library` e `utils`) e uma pasta de resultados (`ga-results`). A pasta de resultados contém uma pasta de ficheiros *log* e uma pasta para cada execução (*run*) do problema (`r0`, `r1`, ...). Cada pasta de execução contém várias gerações, separadas por pastas (`g0`, `g1`, ...) e cada geração contém uma pasta para a população (`pop`) e outra pasta para os resultados da avaliação (`res`) Adicionalmente, cada indivíduo ou resultado representa um ficheiro.

Ao participar na resolução de um problema, o participante tem dois papéis: *Master* e *Manager*. O *Master* é responsável por detetar quando todos os indivíduos foram avaliados, preparar a *Pool* para a próxima geração e por fim, dar início à nova geração. O *Manager* obtém informação de um indivíduo da *Pool* e cria um *Worker* para o avaliar. Este processo é repetido após um *Worker* terminar a sua avaliação ou caso um *Worker* sofra uma falha durante o processo de avaliação, desde que existam indivíduos para avaliar na

Pool. Adicionalmente o *Manager* encontra-se à espera caso atinja o número máximo de *Workers* a executar ao mesmo tempo (definido pelo utilizador da máquina). Cada *Worker* encontra-se responsável por avaliar um indivíduo e colocar o resultado na *pool*.

De modo a garantir que apenas pode existir um *Master* ativo, num dado tempo, para cada geração, recorremos ao sistema de *locks* do SCFS. Em mais detalhe, o SCFS garante que apenas um participante obtém o *lock* de escrita de um ficheiro, obrigando os restantes a esperar que este o liberte. Ou seja, o participante que obtém o *lock* de escrita de um determinado ficheiro, elege-se *Master* da geração. Os participantes que não consigam obter o *lock* de escrita do ficheiro, continuam a tentar até que a geração seja completa (o *Master* eleito liberta o *lock* no final da geração) ou até que o *Master* eleito sofra uma falta. Caso um *Worker* sofra uma falta por *crash*, o seu *Manager* deteta a sua falha e lança outro *Worker* para tomar o seu lugar. Adicionalmente, o *Daemon* reinicia o processo de participação do cliente no problema em caso de falha, isto é, se o seu *Master* ou *Manager* sofreu uma falta por *crash*.

Finalmente o SCFS permite-nos fornecer uma camada de abstração aos clientes. Estes apenas têm conhecimento da *Pool*, abstraindo-se do número de parceiros, da sua organização e do tipo de armazenamento que está a ser utilizado na *Pool*.

3.2.1 Sistema de *Locks*

O SCFS fornece um mecanismo de *locks*, garantindo atomicidade nas escritas em ficheiros, ou seja, apenas um cliente consegue direitos de escrita sobre um dado ficheiro. Este *lock* pode ser libertado por escolha do cliente que o possui ou caso este cliente sofra um *crash*.

De modo a tomar partido deste mecanismo e com o objetivo de eliminar problemas de trabalho redundante, decidimos guardar um indivíduo por ficheiro. Deste modo, cada cliente apenas trabalha sobre os indivíduos aos quais conseguiu obter *lock* de escrita. Adicionalmente, o processo de eleição de *Master* pode também tomar partido deste mecanismo. O *Master* que conseguir obter o *lock* de escrita de um determinado ficheiro é eleito o *Master* da geração.

Finalmente, caso o cliente tenha acesso aos ficheiros, o SCFS permite que sejam efetuadas leituras, mesmo que estes estejam a ser atualizados.

Capítulo 4

Implementação

Neste capítulo apresentamos todos os detalhes da implementação da plataforma, assim como uma descrição dos seus componentes. Adicionalmente, apresentamos uma breve explicação do funcionamento do *Deamon* e das suas funcionalidades. Por último apresentamos os detalhes de implementação e as alterações efetuadas de modo a implementar o MOGA na plataforma. Pretendemos que este sirva de exemplo para futuras implementações de AEss.

4.1 Plataforma

Para implementar o nosso sistema recorremos a uma plataforma existente, o *Python Optimisation Framework* (POF) [32]. optámos por esta plataforma, pois esta apresenta objetivos semelhantes aos nossos, nomeadamente o desejo de separação entre definição do problema e a estratégia de procura utilizada para o resolver. Adicionalmente, a plataforma possui um modelo conceptual composto por três tipos de componentes: o avaliador; o espaço de soluções; a estrutura de vizinhança. O espaço de soluções fornece soluções possíveis, a estrutura de vizinhança determina soluções vizinhas e o avaliador avalia ou compara soluções e movimentos. Um movimento consiste numa transformação que foi aplicada a uma solução, de modo a obter-se outra solução diferente. Uma solução vizinha é uma solução criada a partir de outra solução existente, ao qual foi aplicado um conjunto de movimentos.

Contudo, de modo a cumprir os nossos objetivos, efetuamos algumas alterações. Removemos as componentes de vizinhança, os movimentos e as distâncias entre soluções, pois o nosso sistema visa suportar AEs gerativos. Nestes algoritmos, as novas soluções são criadas através da aplicação de operadores às soluções atuais. Adicionalmente, visto tratar-se de um sistema tolerante a faltas, adicionamos comportamentos de escrita e leitura de dados em disco. Estes são utilizados na interação com a *Pool*, por parte do *Master*, do *Manager* e dos *Workers*. Adicionamos também a definição de operadores genéticos.

O nosso sistema suporta o *Hollywood Principle*, ou seja, o principio de *"Don't call*

us. We'll call you". Assim, as classes de topo invocam as classes de baixo nível, permitindo que estas sejam alteradas de forma independente. Adicionalmente a plataforma foi concebida inicialmente para suportar todo o tipo de AEs geracionais. Este segue os seguintes princípios de desenho:

- *Abstract Factory* - Na criação de soluções, pontos e amostras;
- *Iterator* - Fornecendo um mecanismo de enumeração de amostras;
- *Strategy* - Pela articulação das classes;
- *Template method* - Permitindo ao utilizador que implemente os métodos que ache necessários para a execução do seu algoritmo.

4.1.1 Estrutura em Camadas

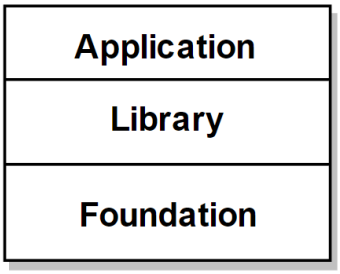

Figura 4.1: Estrutura em três camadas

Após alguma ponderação, decidimos manter a estrutura original do POF, formada por três camadas, representadas na Figura 4.1.

Em mais detalhe, cada camada contém:

- *Application* - Classes definidas pelo utilizador que apenas funcionam para um problema ou algoritmo específico;
- *Library* - Classes definidas pelo utilizador, que podem ser utilizadas em vários problemas ou algoritmos. Estas correspondem a operadores, espaços de representação mais comuns, etc;
- *Foundation* - Classes abstratas que representam o esqueleto da plataforma. Estas classes definem, de um modo geral, o comportamento que cada elemento do sistema deve de ter. Adicionalmente, estas contêm uma classe de *log*. Estas não devem ser alteradas pelos utilizadores.

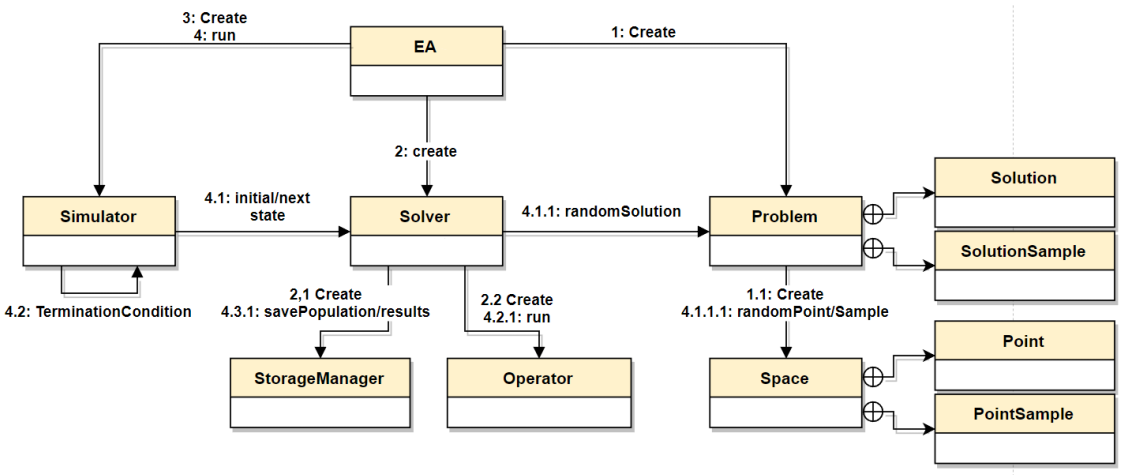

Figura 4.2: Arquitetura da plataforma

4.1.2 Arquitetura

A Figura 4.2 apresenta a arquitetura de classes do sistema desenvolvido. Todas as classes nela representadas, menos a `EA`, representam classes abstratas que fornecem métodos a serem implementados pelo utilizador, de modo a obter o comportamento esperado. A classe `EA`, representa a classe de arranque (ou o *main*) do algoritmo, também ele a ser implementado pelo utilizador. Podemos observar as relações que existem entre as classes, com ênfase no sentido da execução das operações. Em outras palavras, observamos que as classes de topo invocam as classes de baixo nível e nunca no sentido contrário. Por exemplo, o `Simulator` invoca a classe `Solver`, que por sua vez invoca as classes `StorageManager`, `Operator` e `Problem`. Esta relação favorece o *Hollywood Principle*, explicado anteriormente, e possibilita uma maior independência e flexibilidade na alteração dos componentes de baixo nível. De seguida explicamos em mais detalhe cada uma das classes e as suas ligações, e resumimos os métodos disponíveis, por camada, na Tabela 4.1. Não colocamos a camada *Application* pois esta pode conter todo o tipo de métodos que os utilizadores pensem ser úteis.

Simulator

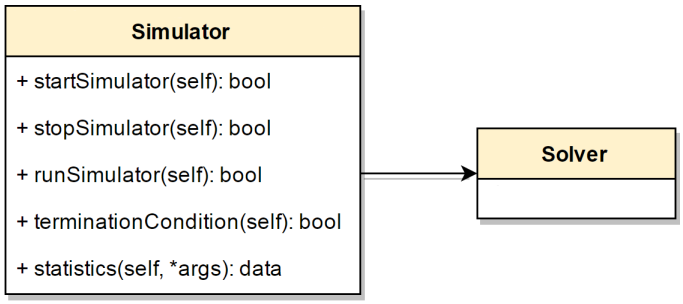

Figura 4.3: Classe abstrata *Simulator*

A classe *simulator*, representado na Figura 4.3, é uma classe abstrata que permite aos utilizadores monitorizem e controlarem o fluxo do seu algoritmo. Este permite uma execução passo a passo do algoritmo que termina quando a condição de paragem for atingida.

Os utilizadores podem definir a sua própria condição de paragem, processo de análise de dados e funções de inicialização e controlo de fluxo do algoritmo.

Solver

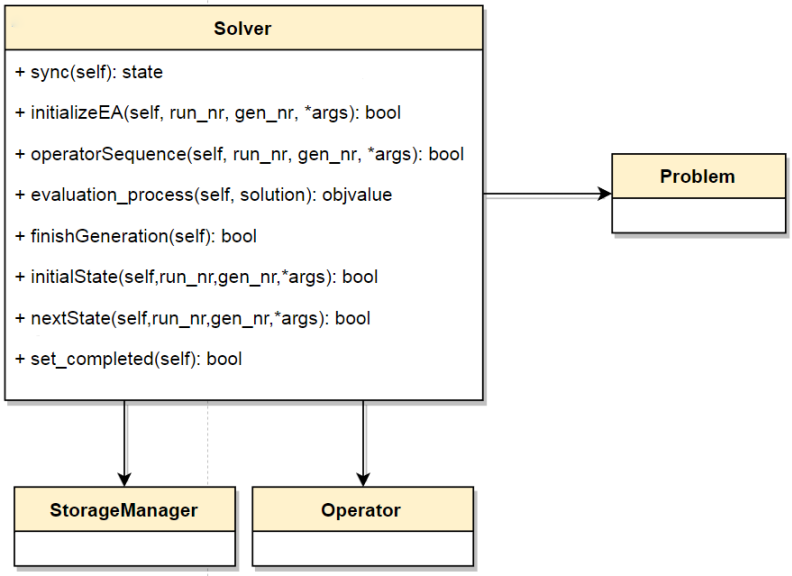

Figura 4.4: Classe abstrata *Solver*

Na Figura 4.4 apresentamos a classe abstrata `Solver` e no algoritmo 1 o seu pseudo-código. Após a sua criação, o *Solver* divide-se em *Master* e *Manager*. Este comportamento ocorre na classe abstrata e é transparente para o utilizador que implementa o seu *solver*. O *Master* gera a população inicial, caso se trate da primeira geração ou obtém os resultados da ultima geração, caso contrário; de seguida aplica a sequência de operadores, partilha a nova população com todos os participantes e espera que o processo de avaliação termine.

O *Manager* está encarregue de criar, gerir e terminar *Workers*. De modo a criar um *Worker*, o *Manager* obtém um indivíduo (ou trabalho) do *StorageManager* e cria um novo processo *Worker* para o avaliar. Este *worker* avalia a solução, guarda o resultado na *Pool* e termina. O *Manager* garante que apenas existe um número máximo de *Workers* ativos ao mesmo tempo. Este valor é definido por cada participante do problema evolutivo e, normalmente, deve ser proporcional ao número de núcleos e *threads* do processador da máquina que o está a executar. Adicionalmente, caso o *Manager* detete que algum dos seus indivíduos a serem avaliados foi avaliado por outro participante, este termina o

Algorithm 1 Algoritmo Solver

```
1:  procedure Master
2:      eleito ← eleição de Master
3:      g_completa ← false
4:      while g_completa ≠ true do
5:          if eleito then
6:              checkpoints ← chamada load_checkpoints
7:              g_completa ← análise de checkpoints
8:              if g_completa = true then
9:                  break
10:             checkpoint ← estado atual do AE
11:             chamada save_checkpoint (checkpoint)
12:             faltam ← chamada has_missing_population
13:             if faltam then
14:                 nr_geração ← avaliação do checkpoint
15:                 if nr_geração = 0 then
16:                     g_completa ← chamada initializaEA
17:                 else
18:                     g_completa ← chamada operatorSequence
19:             while chamada has_work = true do
20:                 sleep
21:             chamada finishGeneration
22:             checkpoint ← estado atual do AE
23:             chamada save_checkpoint (checkpoint)
24:         else
25:             sleep
26: procedure Manager
27:     finished ← false
28:     while finished ≠ true do
29:         workers ← numero de trabalhadores ativos
30:         if workers ≥ número máximo de trabalhadores ativos then
31:             sleep
32:             continue
33:         checkpoints ← chamada load_checkpoints
34:         g_completa ← análise de checkpoints
35:         if chamada has_work = true and g_completa ≠ true then
36:             work ← chamada get_work
37:             worker ← criação do trabalhador com work
38:         else if workers = 0 then
39:             finished ← true
40:         else
41:             sleep
```

Worker encarregue da sua avaliação. Embora raro, este fenómeno ocorre quando existe um atraso entre a libertação do *lock* do indivíduo e o aparecimento do seu ficheiro de resultados em todos os participantes. Deste modo, um participante que ainda não recebeu o ficheiro de resultados, tenta e consegue obter o *lock* de um indivíduo previamente avaliado.

Os métodos `sync()`, `initialState()`, `nextState()` e `setFinished()` permitem ao *simulator* sincronizar, iniciar, avançar e terminar o *Solver*, respetivamente, de modo a controlar o fluxo do algoritmo. Estes permitem-nos manter a gestão dos comportamentos de *Master* e *Manager* transparente para os utilizadores. Na eventualidade de ocorrer uma recuperação ou entrada de um novo cliente a meio da execução do problema, o método `sync()` permite que o seu *Simulator* se coloque no estado atual do problema.

Os métodos `initializeEA()` e `operatorSequence()` têm de ser obrigatoriamente implementados pelo utilizador, enquanto que os métodos `finishGeneration()` e `evaluationProcess()` são opcionais. Os três primeiros vão ser executados pelo *Master* do algoritmo e correspondem ao processo de inicialização da população, ao processo de aplicação de operadores genéticos e ao processo de terminar uma geração, respetivamente. O restante método é executado por cada um dos *workers* e corresponde ao processo de avaliação de uma solução (não confundir com a função de avaliação/aptidão apresentada na classe `Problem`). A execução normal da primeira geração corresponde aos métodos `initializeGA()` e `finishGeneration()`. As restantes gerações correspondem aos métodos `operatorSequence()`, `evaluationProcess()` para todos os indivíduos e terminando com `finishGeneration()`. Adicionalmente, se o *Master* foi eleito por falha de outro *Master*, este consegue perceber qual o estado atual da geração, aplicando apenas os passos necessários para a sua conclusão. Em outras palavras, um *Master* recentemente eleito, após detetar que a população já foi iniciada, apenas espera que processo de avaliação termine e aplica a `finishGeneration()`.

Sempre que necessário o *Solver* aplica o algoritmo de eleição de *Master* da geração. Este algoritmo, quando utilizado sobre o SCFS, consiste no processo de obter o *lock* de escrita do ficheiro de *checkpoints*. Este *lock* é libertado em três ocasiões: no final de cada geração; quando o *Master* de geração sofre uma falta; após um *timeout* definido, iniciado a partir do momento em que o *Master* foi eleito. Finalmente, o *Solver* utiliza o *StorageManager* para ler e escrever dados e para sincronizar os *Masters* e *Worker Manager*.

StorageManager

Representado na Figura 4.5, o `StorageManager` está encarregue da interação do sistema com um sistema de armazenamento. Em mais detalhe, este permite ao solver ler e escrever dados num sistema de armazenamento. Os métodos `has_missing_population()`, `has_work()` e `get_work()` possibilitam ao *Solver* perceber se existe população em falta, se existe trabalho pendente e obter trabalho, respetivamente. Adicionalmente, estas

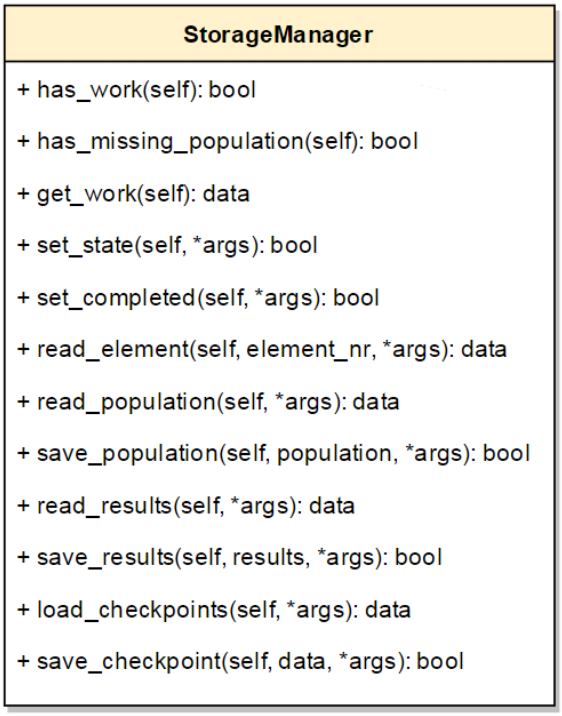

Figura 4.5: Classe abstrata *StorageManager*

permitem ao *Solver* controlar o comportamento do *Master*, do *Worker Manager* e dos *Workers*.

O método `set_state()` permite ao *Solver* atualizar o *StorageManager* para o estado atual do problema evolutivo. Adicionalmente, o método `set_completed()` marca o problema como terminado.

Os métodos `read_element()`, `read_population()`, `save_population()`, `read_results()` e `save_results()` permitem que o *Solver* consiga ler, escrever e partilhar a população com os restantes participantes.

Os restantes métodos `load_checkpoint()` e `save_checkpoint()` permitem ao *Master* de geração ler e guardar o estado atual do problema, ou seja, se a geração foi completa ou ainda está por completar. Caso o *Master* eleito sofra uma falta a meio de uma geração, o *Master* que o substitui avalia o último *checkpoint* e continua o seu trabalho.

Cada utilizador pode implementar o seu *StorageManager* de modo a trabalhar sobre o seu sistema de armazenamento preferido. De modo a evitar trabalho redundante, os métodos de leitura de elementos ou de população devem possibilitar o seu bloqueio, relativamente a outros participantes. A *Library* possui um *StorageManager* implementado para trabalhar sobre o sistema de ficheiros SCFS.

Operator

A classe abstrata `Operator`, representada na Figura 4.6, representa um operador genérico. Este deve ser aplicado pelo *Master* eleito, sobre a população ou sobre uma amostra da

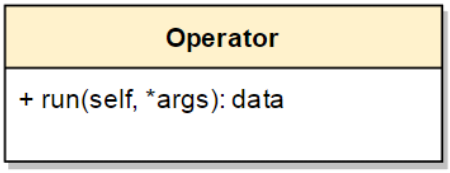

Figura 4.6: Classe abstrata *Operator*

população. Este apenas contém o método `run()`, chamado pelo *Master* da geração quando este se encontra preparado para executar o operador.

Cada utilizador pode implementar os seus próprios operadores. Finalmente a *library* contém operadores de seriação (*ranking*), mutação, seleção e cruzamento que podem servir de exemplo para futuras implementações.

Problem

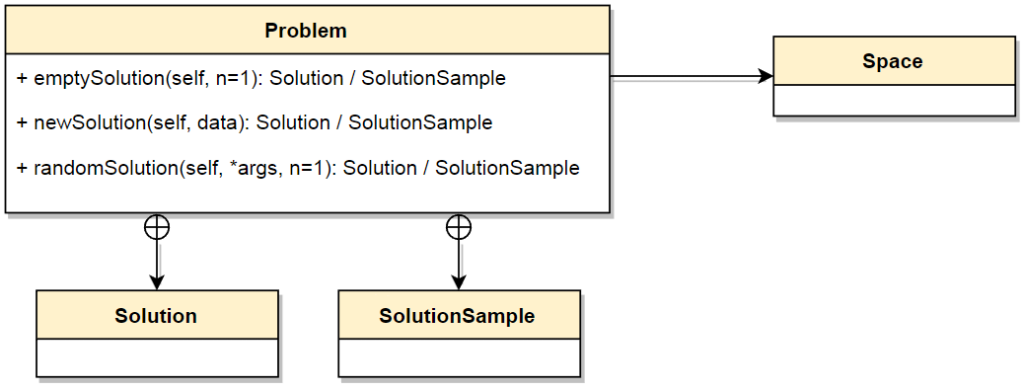

Figura 4.7: Classe abstrata *Problem*

A classe abstrata `Problem` representa um problema específico que será resolvido por um AE implementado no sistema. Esta permite que cada utilizador especifique o seu problema, identificando as variáveis que sejam úteis, como por exemplo o espaço de procura, o número de objetivos, o nome dos objetivos, prioridades e o tamanho máximo de cada indivíduo. Como representado na Figura 4.7, o problema possui duas classes internas, `Solution` e `SolutionSample`, explicadas em 4.1.2

A classe fornece três métodos para a criação de soluções: `emptySolution()`, `newSolution()` e `randomSolution()`, recorrendo a pontos ou amostras obtidas da classe *Space*. Estas criam uma solução vazia, uma solução com base num indivíduo em particular e uma solução aleatória, respetivamente. Através destes métodos, a classe `Problem` fornece ao *Solver* maneiras de criar soluções para o problema, sem que este esteja dependente da sua criação.

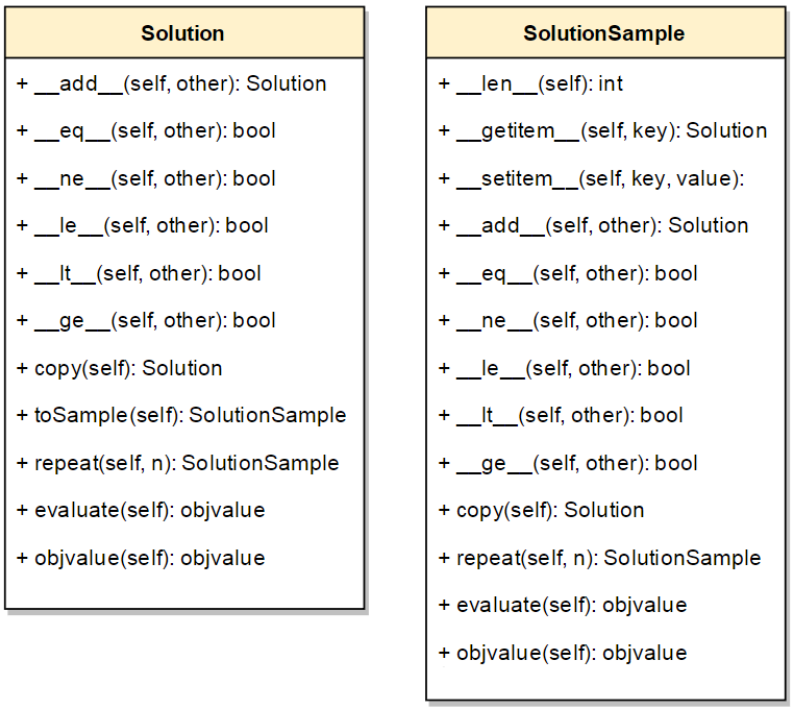

Figura 4.8: Classes abstratas *Solution* e *SolutionSample*

Solution e *SolutionSample*

As classes apresentadas na Figura 4.8, `Solution` e `SolutionSample`, representam uma solução ou uma amostra de soluções. Além dos seus métodos públicos, estas classes possuem mecanismos de comparação de soluções e, caso se trate de uma amostra, de iteração entre soluções. Adicionalmente, estas possuem a função de avaliação, ou de aptidão, implementada pelos utilizadores, a ser aplicada a cada indivíduo. Deste modo cada solução fornece ao sistema uma maneira de se avaliar e de se diferenciar de todas as outras. Além de conterem o cromossoma e o valor de cada objetivo, as soluções contêm um campo adicional que permite aos utilizadores guardar qualquer tipo de informação.

Space

A classe abstrata `Space` representa o espaço de procura do problema evolutivo. Como apresentado na Figura 4.9, esta possui duas classes internas, a `Point` e a `PointSample`, que representam o cromossoma de um indivíduo e uma amostra de cromossomas de vários indivíduos, respetivamente. Adicionalmente esta fornece métodos de criação de pontos ou de amostras de pontos.

Esta é totalmente independente do problema, ou seja, um espaço de procura pode ser utilizado em vários problemas. Cada utilizador pode implementar o seu próprio espaço, indicando como os seus indivíduos são representados e implementando o processo de criação destes. Adicionalmente, a *library* contém uma implementação de um espaço,

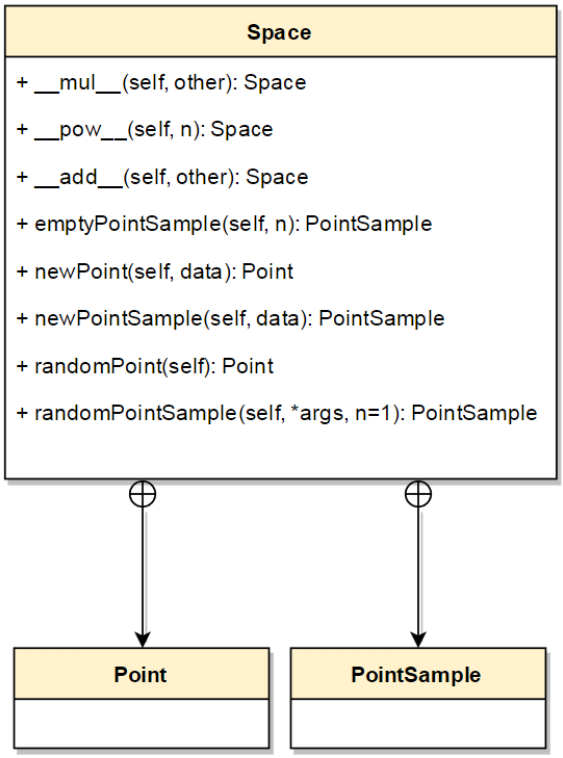

Figura 4.9: Classe abstrata *Space*

utilizado no MOGA, que representa os seus indivíduos na forma de *array* de números inteiros, e pode servir de exemplo para uma futura implementação.

4.2 *Daemon*

O *Deamon* executa na máquina de cada cliente SCFS, e guarda: informações sobre as organizações; os problemas criados e os seus estados; toda a informação dos trabalhadores e em que problemas se encontram registados. Estas informações encontram-se guardadas numa pasta de configuração dentro do SCFS, partilhada entre todos os participantes.

Após a sua máquina ser registada num problema, o *Daemon* executa o processo de inicio do problema, iniciando assim a sua participação no problema. Por sua vez, como explicado anteriormente, este processo acaba por dividir-se em *Master* e *Manager*. Caso algum destes processos sofra um *crash*, o *Deamon* volta a executar o processo inicial do problema.

4.3 Caso de teste e exemplo de implementação

Além dos objetivos principais, os AEs têm de ser capazes de respeitar restrições. Estas podem ser vistas como objetivos mais difíceis e, muitas vezes, o seu cumprimento em conjunto com o cumprimento dos objetivos pode ser uma tarefa bastante complexa. Con-

	Foundation	Library
Simulator		startSimulator stopSimulator runSimulator terminationCondition statistics
Solver	sync initialState nextState	initializeEA operatorSequence evaluation_process finishGeneration set_completed
StorageManager		has_work has_missing_population get_work set_state read_element read_population save_population read_results save_results load_checkpoints save_checkpoints
Operator		run
Problem	emptySolution newSolution randomSolution	
Solution / SolutionSample	copy toSample repeat objvalue	evaluate
Space		emptyPointSample newPoint newPointSample randomPoint randomPointSample

Tabela 4.1: Métodos disponibilizados por camada

tudo, ao longo dos anos têm surgido AEs com a capacidade de conjugar estes fatores. Neste trabalho recorremos ao ***Multi**Objective **G**enetic **A**lgorithm* (MOGA). Este recorre a um processo de decisão complexo que atribui um valor de aptidão e ordena os seus indivíduos com base nos objetivos e nas prioridades do utilizador.

O MOGA pode ser usado, por exemplo, para otimizar a estrutura de uma rede neuronal. Um exemplo neste contexto é a otimização da estrutura de redes de função de base radial [18], onde o número de neurónios e os termos de entrada da rede são otimizados para atingir um conjunto de objetivos.

Este problema foi utilizado como exemplo de utilização e referência de implementação no sistema desenvolvido. A implementação deste problema e do algoritmo MOGA no sistema, seguiu a implementação de referência da linguagem C. Desta forma, o problema e algoritmo foram convertidos de *MATLAB*/C para *Python*/C, e implementados no sistema desenvolvido.

4.3.1 O problema específico

O problema tem como objetivo a otimização de uma rede *Radial Basis Function* (RBF). Em mais detalhe, pretende-se otimizar do número de neurónios de e de parâmetros de entrada da rede RBF, de modo a prever a temperatura do ar em 48 passos (quatro horas) no futuro.

Para tal, recorremos a um conjunto de dados sequenciais de temperatura, recolhidos de cinco em cinco minutos. Estes foram posteriormente divididos em três sub-conjuntos, de modo a servirem de teste e treino de rede neuronal e avaliação da previsão em 48 passos. Recorrendo ao sub-conjunto de avaliação de dados, que não esteve envolvido no processo de treino da rede, esta foi otimizada para simultaneamente minimizar o erro no processo de treino, para minimizar o erro no conjunto de teste e para minimizar o erro de previsão acumulado em 48 passos no futuro.

4.3.2 Implementação do MOGA no sistema

No *Simulator*, definimos a condição de paragem e um mecanismo de estatísticas. A condição de paragem concretiza-se quando o número de gerações de uma execução e o número de execuções atingirem um número máximo. O processo de recolha de estatísticas consiste no cálculo de tempo execução de cada geração e do tempo total do AE.

Na classe *Solver*, foram implementados os procedimentos de criação da população inicial e de aplicação dos operadores usados no MOGA, em adição à criação de um ficheiro de estados, contendo os melhores objetivos e cromossomas até ao momento. O processo de terminação de uma geração consiste em guardar os cromossomas e as suas avaliações correspondentes, de modo a formar uma lista contendo o histórico de avaliações, sem

repetições. No processo de avaliação é verificado se o cromossoma foi avaliado em gerações anteriores e, em caso negativo, as soluções são avaliadas.

Na classe *Operator*, foram implementados os operadores de seriação(*Ranking*), de seleção, cruzamento e de mutação, para serem aplicados no *Solver*.

Na classe *Problem*, foram declaradas algumas variáveis importantes para a execução do problema e especificada a função de avaliação (ou de aptidão) de uma solução.

Na classe *Space*, foi definida a representação dos indivíduos e implementada o procedimento de criação destes. Esta representação segue a especificação apresentada em [19], em que uma parte inicial do cromossoma representa o número de neurónios e o restante cromossoma representa os termos de entrada da rede neuronal.

Finalmente, para assegurar a compatibilidade, convertemos e compilamos algumas funções úteis para a implementação do MOGA. Estas encontravam-se escritas na linguagem de programação C, e integradas no *MATLAB* através de uma *API* própria. Neste caso a implementação foi convertida para uma *API* de integração na linguagem *Python*.

4.4 Tolerância a faltas

Como mencionado anteriormente, o SCFS tolera faltas bizantinas quando se encontra num ambiente que recorra a $3f+1$ (sendo f o número de faltas a tolerar) máquinas privadas ou que recorra a *clouds* públicas.

Caso um *Worker* do problema evolutivo sofra uma falta, o *Manager* volta a iniciar outro para o seu lugar. Deste modo, se o *Manager* continuar a sua execução, os seus *Workers* eventualmente irão terminar o processo de avaliação.

Caso o *Master* ou o *Worker Manager* sofram uma falta, o *Daemon* deteta e volta a iniciar o processo de participação do cliente no problema. Por sua vez este novo processo volta a dividir-se em *Master* e *Manager*. Adicionalmente, estes recorrem a um ficheiro de *checkpoints* para se sincronizarem com os restantes participantes, colocando-se assim na *run* e geração atual.

Por fim, recorrendo a mecanismos do sistema operativo (*CRON*), podemos garantir que desde que este esteja ativo, o *Deamon* também está. Deste modo, tornámos o processo de participação deste cliente em problemas evolutivos tolerante a faltas de *crash*.

4.5 Tecnologias

Todos os componentes do sistema assim como o *Daemon* e o MOGA foram implementados na linguagem orientada a objetos, *Python* versão 3.6+. Recorremos aos módulos *Numpy* para representação e manipulação de *arrays* e matrizes; *Pickle* e *Gzip* para serializar e compactar todos os dados partilhados na *pool*; *subprocess* para a criação e gestão de processos *Workers*.

Capítulo 5

Resultados

Neste capítulo apresentamos os resultados de validação do sistema resultantes da execução do MOGA aplicado ao problema descrito em 4.3.1. Em mais detalhe, apresentamos uma validação da evolução dos resultados da execução do MOGA, uma validação do *speed up* resultante da distribuição em várias máquinas. Apresentamos também um teste à tolerância a faltas por parte dos participantes na execução do AE.

5.1 Ambiente Experimental

O objetivo inicial levar-nos-ia a testar o sistema com diferentes organizações envolvidas numa execução do algoritmo. Contudo, devido a alguns problemas técnicos, e à impossibilidade de coordenação entre as instituições em tempo útil, não conseguimos tornar este objetivo possível. Deste modo, recorremos a máquinas locais, com as características representadas na Tabela 5.1, para executar os testes. Todas as experiências foram executadas recorrendo ao sistema operativo *Ubuntu 14.04* com *Python 3.6+*. Para armazenamento, utilizamos o sistema de ficheiros SCFS [13] [4], configurado em quatro máquinas locais, cada uma a executar um servidor de armazenamento do SCFS, servidor *DepSpace* [15] e cliente SCFS.

Para este fim, preparamos quatro participantes, com as características apresentadas na configuração 1 na Tabela 5.1. Adicionalmente, configuramos o MOGA para escolher o melhor de 10 resultados de treino por cada indivíduo. Definimos também que o número de neurónios estaria entre 2 e 24 e que o cromossoma teria dimensão de 49, a primeira representa o número de neurónios e as restantes 48 representam os termos de entrada da rede neuronal. Finalmente, cada iteração utiliza 100 indivíduos e cada indivíduo é treinado através de 100 iterações do algoritmo de treino. Os parâmetros da rede são escolhidos da iteração em que o erro é mínimo no conjunto de dados de teste (método de paragem *early-stopping*).

	Configuração 1	Configuração 2
Model:	Dell PowerEdge R410	Dell PowerEdge 850
CPU:	2x Intel Xeon E5520 2.27 GHz / 1 MB L2 cache / 8 MB L3 cache	Intel Pentium 4 CPU 2.80GHz 2.8 GHz / 1 MB L2 cache
Memory:	32 GB (8x4GB) / DIMM Synchronous 1066 MHz (0.9 ns)	2 GB (2x1GB) / DIMM Synchronous 533MHz (1.9ns)
Storage:	146 GB / SCSI 3 Gbps / 15000 RPM / Seagate Cheetah ST3146356SS Driver: mptsas	80 GB / SCSI / 7200 RPM / ST380013AS Driver: ata_piix

Tabela 5.1: Características [6] dos participantes, por configuração (*cluster* S e R respetivamente).

5.2 Validação da implementação

Na primeira experiência avaliamos os resultados obtidos na execução do MOGA, sem qualquer introdução de faltas intencionais. Com esta experiência pretendemos avaliar o impacto do número de participantes na qualidade dos resultados obtidos bem como a evolução na qualidade dos resultados obtidos durante a execução do algoritmo.

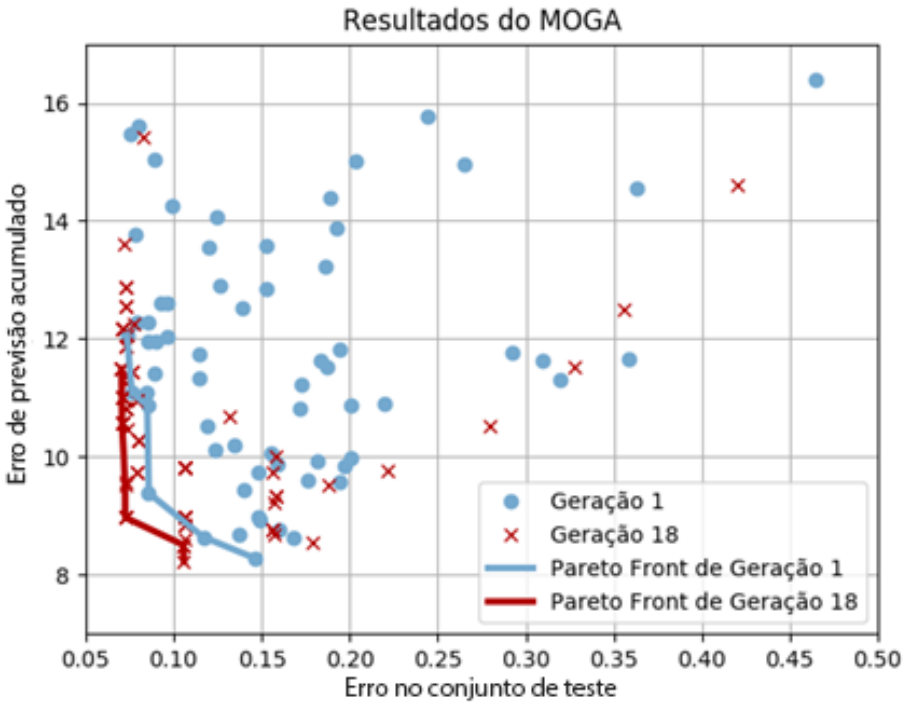

Figura 5.1: Comparação de resultados da primeira e última geração do MOGA, com 4 participantes.

Na Figura 5.1 apresentamos os resultados da primeira geração, em comparação com a última geração executada (a geração 18). Neste gráfico, o eixo das abcissas representa o erro obtido no conjunto de teste, o eixo das ordenadas representa o erro acumulado ao longo dos 48 passos de previsão. Cada ponto (quadrado ou círculo) representa um indivíduo. Estes valores representam dois dos objetivos do problema, e quanto menores forem, melhor a aptidão do o indivíduo. Na figura os indivíduos com maior erro na geração 18 são estrangeiros, isto é, foram criados aleatoriamente, correspondendo a 10% do tamanho da população. Nesta figura, apresentamos também a fronteira de Pareto (ou *Pareto frontier*), que apresenta os indivíduos com melhor compromisso entre objetivos. Como esperado, a geração 18 possui uma fronteira de Pareto melhor que a primeira geração. Este resultado permite concluir que houve evolução dos objetivos ao longo das gerações, no sentido da sua minimização, validando o comportamento esperado da implementação. Ainda assim, uma validação completa a este nível teria de ser efetuada função a função, operador a operador, para se considerar conclusiva. No entanto este procedimento estaria para além do âmbito do trabalho desta tese.

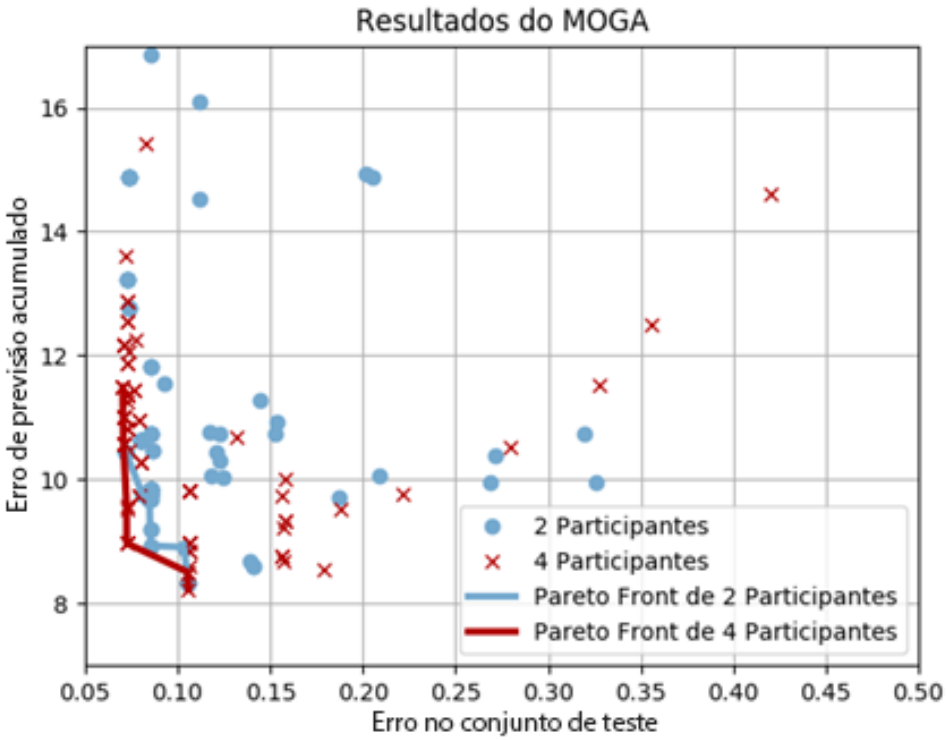

Figura 5.2: Comparação de resultados da última geração do MOGA, com 2 e 4 participantes.

Na Figura 5.2 apresentam-se os resultados da ultima geração de duas experiências independentes, uma com dois e outra com quatro participantes. Podemos observar que, apesar de serem duas experiências independentes, as suas fronteiras de Pareto são seme-

lhantes, apesar da inicialização dos indivíduos na primeira geração ser aleatória. Assim podemos concluir que o número de participantes não influencia a qualidade dos resultados, validando também a implementação no que respeita a armazenamento, distribuição e coordenação.

5.3 Validação do desempenho

Enquanto que a primeira experiência valida aspetos de implementação com base na qualidade dos resultados, a segunda tem como objetivo validar o ganho em termos de tempo de execução do sistema em relação ao número de participantes. À semelhança da experiência anterior, cada participante possui as características da configuração 1 representada na Tabela 5.1, sendo executada a mesma versão do MOGA aplicado ao mesmo problema descrito.

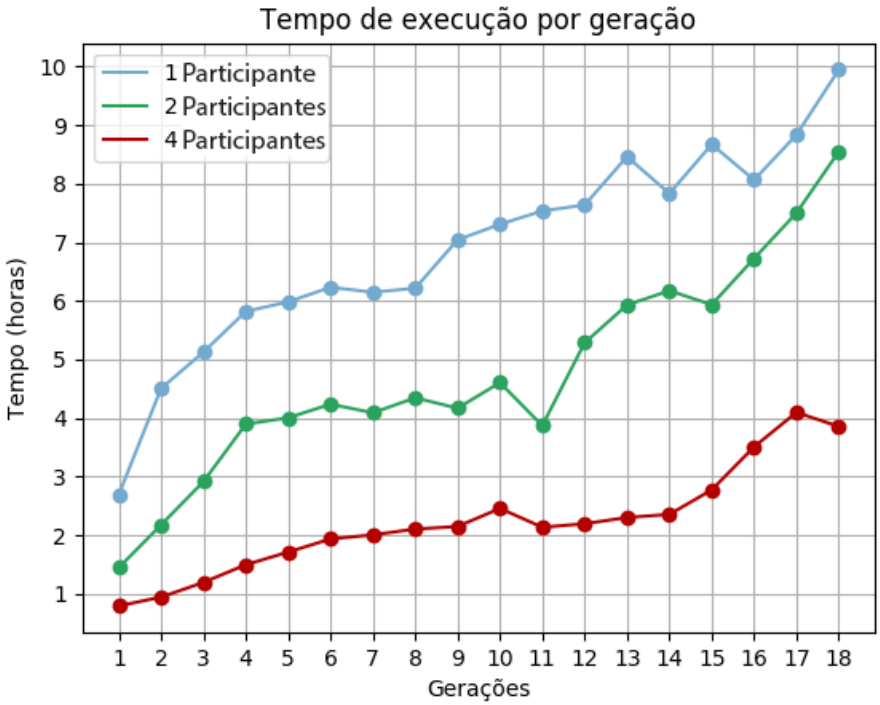

Figura 5.3: Tempo de execução de cada geração.

A Figura 5.3 representa o tempo gasto em cada geração nas experiências com um, dois e quatro participantes. Nesta, o número de gerações encontra-se representado no eixo das abcissas e o tempo, em horas, representado no eixo das ordenadas. Podemos observar que à medida que o número de gerações aumenta, o tempo de execução também aumenta. Este fenómeno deve-se ao facto do AE convergir para indivíduos maiores, ou seja, indivíduos com mais termos de entrada e um número maior de neurónios, que consequentemente tem um tempo de avaliação maior. Podemos observar também que o tempo de execução de

cada geração é significativamente menor com quatro participantes, aproximando-se em muitos casos da metade do tempo com duas máquinas. Estes resultados são confirmados pela Figura 5.4 onde se encontra representado o tempo total de execução do algoritmo com dois e quatro participantes. Este permite-nos verificar que o tempo de execução do algoritmo com quatro participantes é de aproximadamente 1 dia e 16 horas enquanto que o tempo de execução com dois participantes é de aproximadamente 3 dias e 14 horas. Deste modo, com a adição de apenas dois participantes, conseguimos um melhoramento no tempo de execução de aproximadamente 1 dia e 22 horas, em apenas 18 gerações. Assim concluímos que o nosso sistema escala com o número de participantes, favorecendo a cooperação. Fica por avaliar a escalabilidade em casos de várias instituições participantes, pelas razões já apresentadas, bem como o comportamento com maior número de participantes.

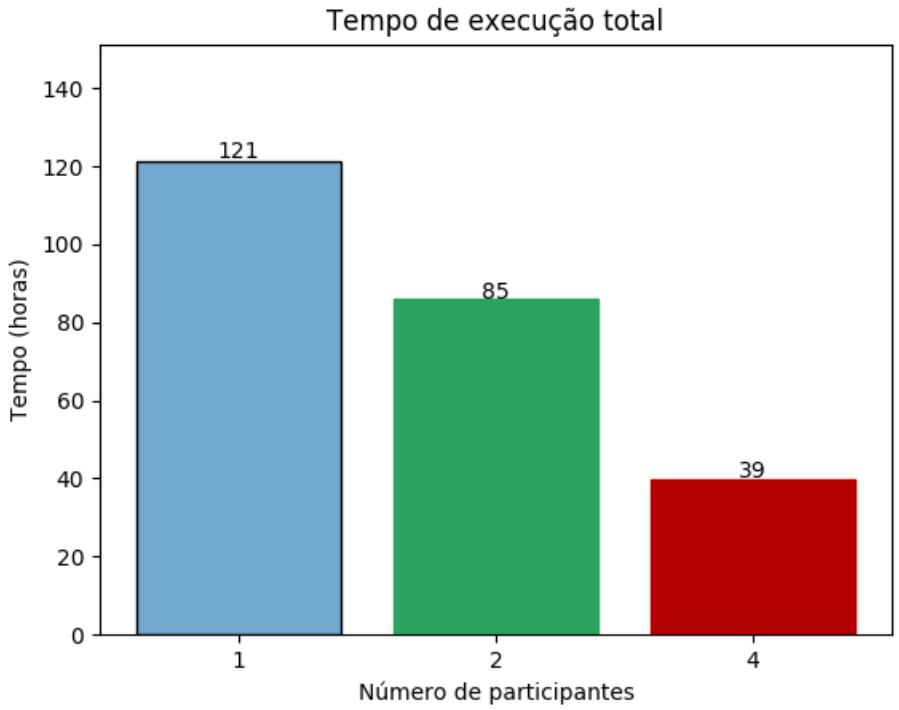

Figura 5.4: Tempo total de execução do algoritmo.

As Figuras 5.5 e 5.6 apresentam informação relacionada com os tempos de avaliação de cada indivíduo. Nesta, o eixo das abcissas representa as gerações e o eixo das ordenadas representa o tempo, em segundos, do processo de avaliação. Como esperado, à semelhança do tempo de execução de cada geração, observado anteriormente, podemos observar que o tempo médio de avaliação dos indivíduos aumenta com as gerações. Contudo, as gerações apresentam valores mínimos bastante semelhantes. Este fenómeno deve-se às seguintes observações: existência de pelo menos um indivíduo que já foi avaliado em gerações anteriores; existência de pelo menos um cromossoma que contenha

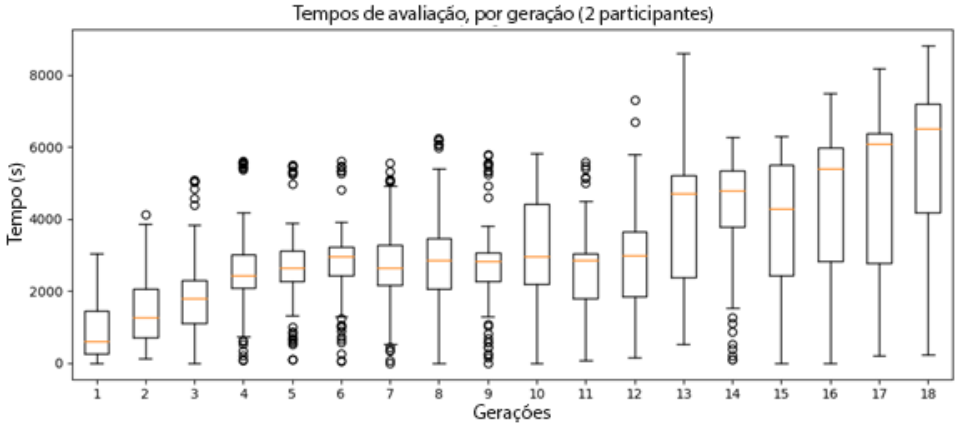

Figura 5.5: Tempo de avaliação de indivíduos, por geração, com 2 participantes.

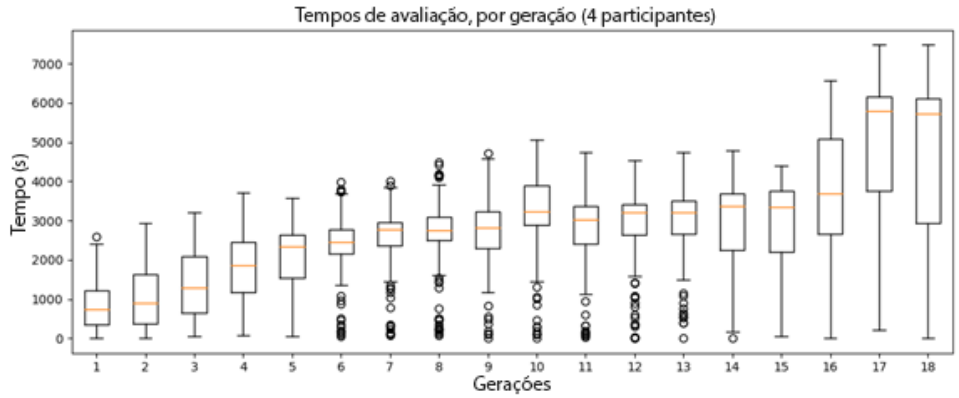

Figura 5.6: Tempo de avaliação de indivíduos, por geração, com 4 participantes.

o número mínimo de parâmetros de entrada e de neurónios, cuja avaliação será rápida quando comparada com indivíduos mais complexos.

Face aos tempos médios de avaliação e aos resultados obtidos, confirmamos que o sistema é viável quando executado com problemas em que o tempo de avaliação é significativamente superior ao tempo de comunicação com a *Pool*.

5.4 Tolerância a Faltas

Por fim, a terceira experiência teve como objetivo avaliar não só a tolerância a faltas do sistema bem como o impacto que esta tem sobre o tempo de execução. Esta experiência foi realizada em máquinas diferentes das anteriores, com as características representadas na configuração 2 da Tabela 5.1. Para obtenção mais rápida dos resultados, foi executada uma versão menos complexa do MOGA, configurada com 30 indivíduos por geração e 10 iterações de treino do modelo. De modo a simular faltas por *crash*, executamos um injetor de faltas que termina processos *Worker*, *Master* ou ambos, com a uma probabili-

dade de 15% a cada 30 segundos. Adicionalmente, nesta experiência não foi avaliada a qualidade dos resultados, pois a presença de *Workers* faltosos não altera a qualidade dos resultados. Em mais detalhe, mesmo que um *Worker* sofra uma falta durante o processo de avaliação, os seus resultados são perdidos e outro *Worker* acabará por o substituir. Na eventualidade de um *Worker* sofrer uma falta durante a escrita do ficheiro de resultados na *Pool*, este ficheiro fica corrupto. Contudo, o ficheiro será eliminado pelo *Master* da geração, e eventualmente outro *Worker* será iniciado para substituir o *Worker* faltoso. Em ambos os casos, o novo *Worker* recomeça o processo de avaliação do indivíduo do início. Com esta configuração observámos que: em média, a recuperação do *Master* e do *Manager* demora 15 segundos (488 amostras, σ = 967 milissegundos); em a média, a substituição de um *Worker* faltoso demora aproximadamente 1 segundo (545 amostras, σ = 393 milissegundos).

Figura 5.7: Tempo de execução de cada geração, com e sem faltas.

Como podemos observar pelas Figuras 5.7 e 5.8, o sistema terminou as 30 gerações previstas. Contudo, o tempo de execução total do algoritmo na versão com faltas demorou aproximadamente 10 horas, em comparação com a versão sem faltas que demorou por volta de 3 horas. Este aumento do tempo de 7 horas deve-se ao facto de, na ocorrência de um *crash*, o *Worker* perde todo o processo de avaliação do indivíduo. Consequentemente, este é libertado, possibilitando que seja avaliado por outro participante, que reinicia o processo de avaliação do indivíduo.

De modo a demonstrar a tolerância a faltas do sistema, apresentamos na Figura 5.9

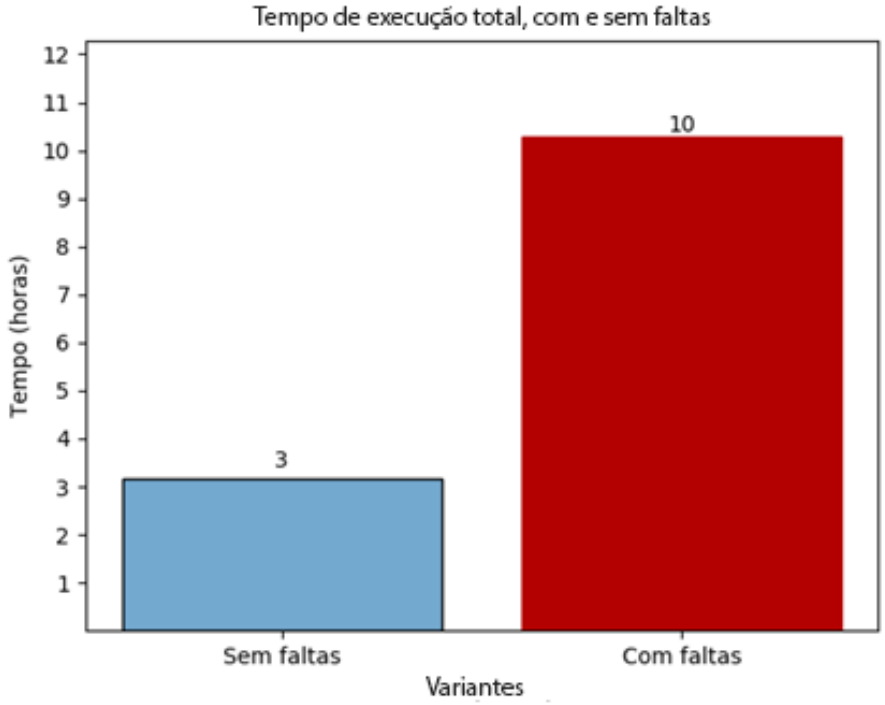

Figura 5.8: Tempo total de execução do algoritmo, com e sem faltas.

```
2017-07-31 12:11:40:939370 :: MANAGER :: Worker died, slot: 7 is now available
   | w:0-i:23 | w:1-i:28 | w:2-i:22 | w:3-i:25 | w:4-i:24 | w:5-i:16 | w:6-i:26 | None | w:8-i:27 | w:9-i:7 |

2017-07-31 12:11:40:940228 :: MANAGER :: Worker died, slot: 8 is now available
   | w:0-i:23 | w:1-i:28 | w:2-i:22 | w:3-i:25 | w:4-i:24 | w:5-i:16 | w:6-i:26 | None | None | w:9-i:7 |

2017-07-31 12:11:40:941021 :: MANAGER :: Worker died, slot: 9 is now available
   | w:0-i:23 | w:1-i:28 | w:2-i:22 | w:3-i:25 | w:4-i:24 | w:5-i:16 | w:6-i:26 | None | None | None |  1

2017-07-31 12:11:40:965689 :: MANAGER :: New worker launched for slot: 7
   | w:0-i:23 | w:1-i:28 | w:2-i:22 | w:3-i:25 | w:4-i:24 | w:5-i:16 | w:6-i:26 | w:7-i:8 | None | None |

2017-07-31 12:11:40:990015 :: MANAGER :: New worker launched for slot: 8
   | w:0-i:23 | w:1-i:28 | w:2-i:22 | w:3-i:25 | w:4-i:24 | w:5-i:16 | w:6-i:26 | w:7-i:8 | w:8-i:29 | None |

2017-07-31 12:11:40:991961 :: WORKER :: Worker: 8 evaluating solution: 29

2017-07-31 12:11:40:994008 :: WORKER :: Worker: 7 evaluating solution: 8

2017-07-31 12:11:41:018678 :: MANAGER :: New worker launched for slot: 9
   | w:0-i:23 | w:1-i:28 | w:2-i:22 | w:3-i:25 | w:4-i:24 | w:5-i:16 | w:6-i:26 | w:7-i:8 | w:8-i:29 | w:9-i:0 |

2017-07-31 12:01:41:026100 :: WORKER :: Worker: 9 evaluating solution: 0
```

Figura 5.9: Excerto do ficheiro de *log*, revelando falha por *crash* de *workers*.

um excerto do ficheiro de *log* do sistema, contendo uma ocorrência de falta por *crash*, em três *workers*. Podemos verificar que às 12 horas e 11 minutos, indicado pelo quadrado 1. da imagem, o *Manager* apercebe-se que os *workers* 7, 8 e 9 sofreram um *crash*. De seguida, o *Manager* inicia novos *Workers* para os substituir, como representado no resto do *log*. Contudo, não há garantias (nem necessidade) que os novos *Workers* avaliem os mesmos indivíduos que os *workers* mortos estavam a avaliar, e caso sejam os mesmos, estes têm de começar o processo de avaliação do inicio.

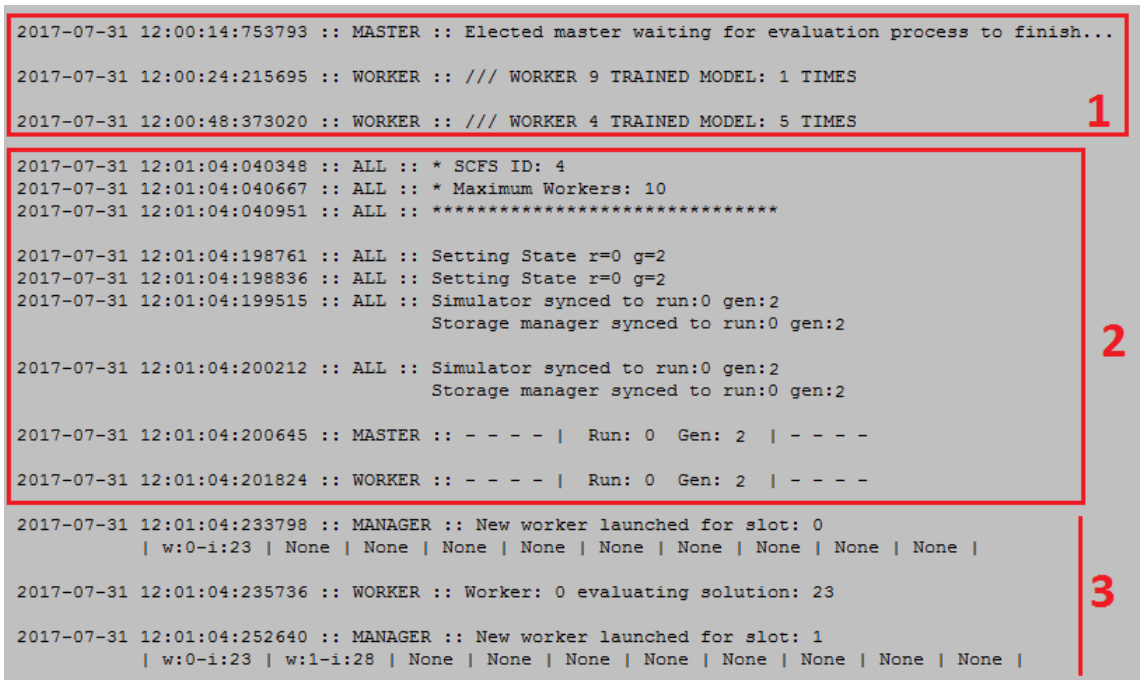

Figura 5.10: Excerto do ficheiro de *log*, revelando falha por *crash* do *Master* e *Manager*.

Na Figura 5.10 apresentamos um excerto do ficheiro de *log*, representando uma situação de falha do *Master* e do *Manager*. Como podemos verificar na área 1. , o *Master* foi eleito como o *Master* da geração e encontrava-se à espera que o processo de avaliação terminasse. Contudo, estes sofreram um *crash* e deixaram de escrever para o ficheiro de *log*, representado no espaço entre as áreas 1. e 2. . De seguida apenas observamos, na área 2., o *Master* e *Manager* a recuperarem e sincronizarem-se para o estado atual do AE, na execução zero e geração dois. Podemos observar também, na zona 3. que o *Manager* inicia o processo de criação de *Workers* para avaliar os indivíduos que conseguiu obter.

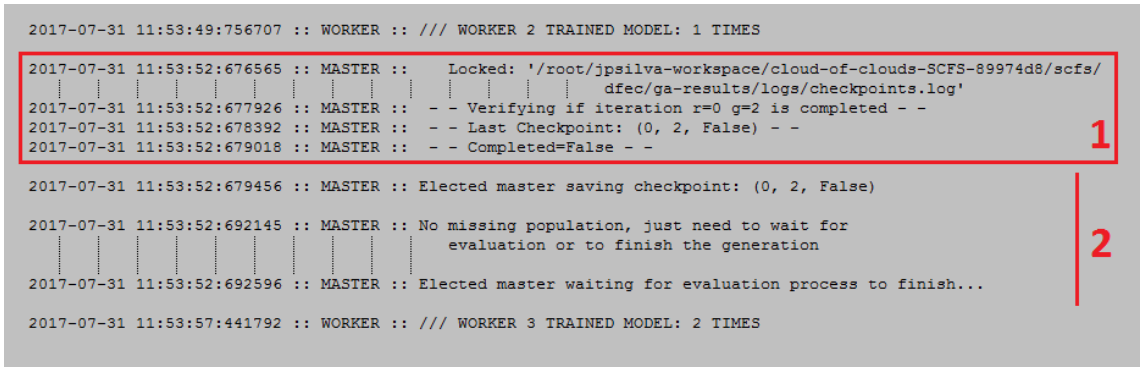

Figura 5.11: Excerto do ficheiro de *log*, revelando eleição de *Master*.

Na Figura 5.11 está representado um excerto do ficheiro de *log* relativamente a um participante do problema. Podemos observar, na área 1., que na ocorrência do *crash* anterior, este participante conseguiu eleger-se *Master* de geração, isto é, obteve o *lock* do ficheiro

de *checkpoints*. Podemos observar também, na área 2., que este *Master* apercebe-se que a população já foi iniciada pelo *Master* anterior e encontra-se à espera que o processo de avaliação termine.

Deste modo, concluímos que o sistema é tolerante a faltas de *crash* por parte dos *Workers*, do *Manager* e do *Master*. Caso a máquina sofra um *crash* total, e consequentemente o *Daemon* termine a sua execução, não existem mecanismos que iniciem a participação da máquina no problema. Contudo, o problema continua a executar recorrendo a todos os outros participantes, podendo ainda ser terminado desde que exista pelo menos um participante e que os servidores *DepSpace* e de armazenamento do SCFS continuem ativos. Para mitigar o impacto de *crash* total, poderíamos colocar o processo de inicialização do SCFS e do *Daemon* juntamente com o arranque das máquinas dos participantes. Uma vez que estes recuperem, o processo de participação no problema torna-se automático. Isto assegura que o sistema tolera o *crash* total da máquina do participante.

Por outro lado, se um problema perde todos os seus participantes, não necessita de ser recomeçado do inicio, uma vez que o seu estado e progresso encontram-se armazenados.

Capítulo 6

Conclusão

Neste capítulo apresentamos as conclusões finais e as contribuições do nosso trabalho. Sugerimos também alguns aspetos a melhorar no nosso sistema como trabalho futuro.

6.1 Contribuições e Conclusões

Neste trabalho desenvolvemos e apresentamos um sistema que cumpre os objetivos propostos. Recorrendo a um sistema baseado em *Pool* com características *peer-to-peer* e *master-salve*, tornamos o sistema tolerante a faltas e flexível à entrada e saída de participantes. Ao escolher um sistema de armazenamento distribuído, como o SCFS, para implementar a *Pool* evitamos que esta se torne num ponto único de falha no sistema. Com a experiência de validação da implementação do MOGA, concluímos que o sistema não interfere com a qualidade dos resultados, possibilitando a implementação de qualquer algoritmo evolutivo geracional. Através dos resultados do tempo de gasto na execução do MOGA, concluímos que o desempenho do sistema aumenta significativamente com o numero de participantes. Contudo esta escalabilidade encontra-se relacionada com o tipo de algoritmo evolutivo implementado. Em geral, quanto maior a razão entre o tempo de avaliação dos indivíduos em relação à média de tempo de comunicação dos participantes com a *Pool*, maior a escalabilidade do sistema. Concluímos também que o numero de participantes não afeta a qualidade dos resultados e, consequentemente, quantos mais participantes um problema tem, menos tempo irá demorar para atingir um resultado. Finalmente, devido à sua estrutura de dados e ao facto de que cada organização pode escolher as máquinas que pretende dedicar a cada problema, o nosso sistema permite a execução de vários AEs ao mesmo tempo.

Contudo o sistema ainda apresenta algumas limitações, em grande parte devido ao SCFS. Nomeadamente alguns ficheiros ficam corruptos e alguns participantes perdem a ligação com o sistema de coordenação *DepSpace*, necessitando de ser reiniciados.

6.2 Trabalho Futuro

Como trabalho futuro, devem ser implementados mais algoritmos evolutivos, de modo a comprovar e a testar o desempenho do sistema. Adicionalmente, devem ser implementados mais componentes do sistema, como diferentes espaços de representação, operadores genéticos e vários tipos diferentes de armazenamentos. Pretende-se que estes recorram a bases de dados tradicionais ou a outros sistemas de ficheiros. Consideramos também que a articulação das classes do sistema possa ser melhorada, possibilitando uma maior independência entre elas. O sistema deve de ser testado com um número elevado de experiências, e com várias organizações, de modo a avaliar o desempenho. Por fim, devem ser implementados mecanismos de *logs* e de *checkpoints* mais eficientes, de modo a fornecerem uma maior tolerância a faltas.

Glossário

AE Algoritmo Evolutivo. 1–3, 5, 6, 8, 14, 37, 40, 47, 50, 54, 57, 60, 65

AEs Algoritmos Evolutivos. 1, 2, 5–7, 12, 14, 35, 36, 43, 44, 52, 54, 67

MOGA *Multi-Objective Genetic Algorithm*. 3, 43, 52, 54, 55, 57, 58, 60, 62, 67

POF *Python Optimisation Framework*. 43, 44

RBF *Radial Basis Function*. 54

SCFS *Shared Cloud-backed File System*. 2, 30–32, 38–41, 48, 49, 52, 55, 57, 66, 67

Bibliografia

[1] Bitbucket: S3ql. https://bitbucket.org/nikratio/s3ql/. [Online; acedido 16-Setembro-2017].

[2] Dht: Github by juliusz chroboczek. https://github.com/jech/dht. [Online; acedido 08-Novembro-2016].

[3] Github: S3fs. https://github.com/s3fs-fuse/s3fs-fuse. [Online; acedido 16-Setembro-2017].

[4] Github: Scfs. http://cloud-of-clouds.github.io/SCFS/. [Online; acedido 16-Janeiro-2017].

[5] Navigators: Depspace. http://www.navigators.di.fc.ul.pt/software/depspace/. [Online; acedido 16-Janeiro-2017].

[6] Navigators quinta cluster r. http://www.navigators.di.fc.ul.pt/wiki/Quinta_Hardware. [Online; acedido 27-Julho-2017].

[7] Opendht: Github. https://github.com/savoirfairelinux/opendht. [Online; acedido 08-Novembro-2016].

[8] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, Apr 2006.

[9] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 229–240, New York, NY, USA, 2010. ACM.

[10] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax*. "O'Reilly Media, Inc.", 2010.

[11] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[12] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *Trans. Storage*, 9(4):12:1–12:33, November 2013.

[13] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. Scfs: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 169–180, Berkeley, CA, USA, 2014. USENIX Association.

[14] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society.

[15] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 163–176, New York, NY, USA, 2008. ACM.

[16] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer Berlin Heidelberg, 2015.

[17] Pedro M Ferreira. Unsupervised entropy-based selection of data sets for improved model fitting. In *International Joint Conference on Neural Networks (IJCNN 2016)*, 2016.

[18] Pedro M. Ferreira and António Ruano. Evolutionary multiobjective neural network models identification: Evolving task-optimised models. In *New Advances in Intelligent Signal Processing*, pages 21–53. Springer Berlin/Heidelberg, 2011.

[19] Pedro M. Ferreira and António E. Ruano. *Evolutionary Multiobjective Neural Network Models Identification: Evolving Task-Optimised Models*, pages 21–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[20] C. M. Fonseca and P. J. Fleming. Multiobjective optimization and multiple constraint handling with evolutionary algorithms. I. a unified formulation. *Trans. Sys. Man Cyber. Part A*, 28(1):26–37, January 1998.

[21] Mario García-Valdez, Leonardo Trujillo, Juan-J Merelo, Francisco Fernández de Vega, and Gustavo Olague. The evospace model for pool-based evolutionary algorithms. *Journal of Grid Computing*, 13(3):329–349, 2015.

[22] Mario García-Valdez, Leonardo Trujillo, Juan Julián Merelo-Guérvos, and Francisco Fernández-de Vega. *Randomized Parameter Settings for Heterogeneous Workers in a Pool-Based Evolutionary Algorithm*, pages 702–710. Springer International Publishing, Cham, 2014.

[23] Yue-Jiao Gong, Wei-Neng Chen, Zhi-Hui Zhan, Jun Zhang, Yun Li, Qingfu Zhang, and Jing-Jing Li. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 34:286 – 300, 2015.

[24] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[25] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[26] Ricardo Mendes, Tiago Oliveira, Vinicius Cogo, Nuno Neves, and Alysson Bessani. Charon: A dependable cloud-backed system for storing and sharing big data. unpublished, February 2016.

[27] J. J. Merelo, A. M. Mora, C. M. Fernandes, and Anna I. Esparcia-Alcázar. Designing and testing a pool-based evolutionary algorithm. *Natural Computing*, 12(2):149–162, 2013.

[28] Juan-Julián Merelo-Guervós, Antonio Mora, J. Albert Cruz, and Anna I. Esparcia. *Pool-Based Distributed Evolutionary Algorithms Using an Object Database*, pages 446–455. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[29] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: A public dht service and its uses. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 73–84, New York, NY, USA, 2005. ACM.

[30] Timothy Roscoe and Steven Hand. Palimpsest: Soft-capacity storage for planetary-scale services. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 22–22, Berkeley, CA, USA, 2003. USENIX Association.

[31] R. Subbu and A. C. Sanderson. Network-based distributed planning using coevolutionary agents: Architecture and evaluation. *Trans. Sys. Man Cyber. Part A*, 34(2):257–269, March 2004.

[32] Ana Cristina do Carmo Cardoso Vieira. *Uma Plataforma para a Avaliaçao Experimental de Meta-heurısticas*. PhD thesis, Faculdade de ciências e tecnologia da Universidade do Algarve, 2009.

[33] W. R. M. U. K. Wickramasinghe, M. van Steen, and A. E. Eiben. Peer-to-peer evolutionary algorithms with adaptive autonomous selection. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1460–1467, New York, NY, USA, 2007. ACM.

[34] Lei Xu and Fengming Zhang. Parallel particle swarm optimization for attribute reduction. *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, 1:770–775, 2007.