Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Active Space: Executando Operações no Espaço de Tuplas

Luine Ito Madeira de Ley

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2014

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Wilson Henrique Veneziano

Banca examinadora composta por:

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador) — CIC/UnB
Prof. Dr. Edison Ishikawa — CIC/UnB
Prof. Dr. Flávio de Barros Vidal — CIC/UnB

CIP — Catalogação Internacional na Publicação

Ley, Luine Ito Madeira de.

Active Space: Executando Operações no Espaço de Tuplas / Luine Ito Madeira de Ley. Brasília : UnB, 2014.
123 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2014.

1. sistemas distribuídos, 2. coordenação, 3. middleware, 4. espaço de tuplas, 5. consulta

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Active Space: Executando Operações no Espaço de Tuplas

Luine Ito Madeira de Ley

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. Edison Ishikawa
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
CIC/UnB

Prof. Dr. Wilson Henrique Veneziano
Coordenador do Curso de Computação — Licenciatura

Brasília, 16 de Dezembro de 2014

Dedicatória

Este trabalho é dedicado à minha mãe, Kumi Ito, que sempre me ajudou durante as minhas jornadas. Dedico também à Mel, em memória, e aos meus familiares mais próximos.

Agradecimentos

Agradeço primeiramente aos meus amigos, em especial aos que estão em Fortaleza, e a todas as pessoas que pude conhecer em Brasília. Agradeço ao professor Eduardo Alchieri pelos ensinamentos durante a minha graduação e no período de conclusão do curso. Tenho profundo agradecimento pelas amizades que pude fazer na UnB e pela instituição em si.

Resumo

Sistemas distribuídos baseados em coordenação possuem uma forma única de comunicação através de Espaço de Tuplas. Nesse tipo de comunicação apenas uma tupla pode ser inserida, lida ou removida por primitivas de coordenação que acessam tais espaços. Lidar com um grande número de tuplas pode ser desafiador no sentido de como estas tuplas serão buscadas e recuperadas. Propomos a criação de um sistema de *query* e novas operações de consulta para ler todas as tuplas disponíveis em um Espaço de Tuplas e minimizar a quantidade de dados transferidos pela rede a partir dos servidores. Uma contribuição de Espaços de Tuplas ativos (*Active Spaces*) que suportem estas operações corresponde a proporcionar mais flexibilidade para usuários. Um *Active Space* visa também assegurar uma entrega de tuplas confiáveis e de maneira segura na medida em que buscas complexas são realizadas, e a porção de dados e a rede aumentem.

Palavras-chave: sistemas distribuídos, coordenação, middleware, espaço de tuplas, consulta

Abstract

Distributed coordination-based systems have a unique form of communication through a tuple space. In this type of communication only one tuple can be inserted, read, or removed by coordination primitives that access such spaces. Dealing with a great number of tuples is challenging in the sense of how these tuples will be searched and retrieved. We propose the creation of a query system and new query operations to read all available tuples in the tuple space and minimize the quantity of transferred data from the servers through the network. A contribution of active tuple spaces (Active Spaces) that support these operations corresponds to give more flexibility to users. An Active Space also aims to assure a delivery of trusty tuples and in a secure way to the extent that complex searches are executed, and the amount of data and the network grow.

Keywords: distributed systems, coordination, middleware, tuple space, query

Sumário

1 Introdução **1**
1.1 Justificativa . . . 2
1.2 Objetivos . . . 3
1.2.1 Objetivo Geral . . . 3
1.2.2 Objetivos Específicos . . . 3
1.3 Organização do Texto . . . 4

2 Sistemas Distribuídos **5**
2.1 Segurança de Funcionamento em Sistemas Distribuídos . . . 6
2.1.1 Ameaças . . . 8
2.1.2 Atributos . . . 9
2.1.3 Meios . . . 11
2.2 Garantindo Segurança de Funcionamento em Sistemas Distribuídos através da Replicação . . . 14
2.2.1 Replicação por Máquina de Estados . . . 14

3 Coordenação Baseada em Tuplas **17**
3.1 Espaço de Tuplas . . . 19
3.1.1 Linda . . . 19
3.1.2 TSpaces . . . 20
3.1.3 LIME . . . 21
3.2 Espaços de Tuplas com Segurança de Funcionamento . . . 22
3.2.1 DepSpace . . . 22
3.2.2 WSDS . . . 28

4 Active Space **29**
4.1 Modelo de Sistema . . . 30
4.2 Arquitetura do Active Space . . . 30
4.2.1 Análise sobre a Segurança . . . 31
4.3 Metodologia . . . 32
4.3.1 Implementação . . . 32
4.3.2 Primitivas de Coordenação . . . 33
4.3.3 Query . . . 34
4.4 Aplicações . . . 35
4.4.1 Exemplo 1: consulta do voo pelas escalas . . . 36
4.4.2 Exemplo 2: busca através dos dias de embarque . . . 39
4.4.3 Exemplo 3: procurando a passagem de menor valor . . . 43

4.4.4 Outras aplicações 47
5 Conclusão **49**
Referências **51**

Lista de Figuras

2.1 Taxonomia de Dependabilidade e Segurança. Adaptado de [5]. 7
2.2 Propagação de Falhas, Erros e Defeitos. Adaptado de [4]. 9
2.3 Taxonomia de Tolerância a Falhas. Adaptado de [5]. 12
2.4 Problema dos Generais Bizantinos. Adaptado de [22]. 15
2.5 Solução para o Problema dos Generais Bizantinos. Adaptado de [22]. . . . 16

3.1 Funcionamento das Primitivas de Coordenação. Adaptado de [1]. 19
3.2 Arquitetura do DepSpace. Adaptado de [8]. 24
3.3 Replicação tolerante à Faltas Bizantinas. Adaptado de [20]. 25
3.4 Tuplas confiáveis como retorno. 27
3.5 Arquitetura do WSDS. Adaptado de [1]. 28

4.1 Representação de um TS. Adaptado de [32]. 30
4.2 Espaços Lógicos configuráveis. Adaptado de [7]. 31
4.3 Adicionando a camada de Query ao DepSpace. Adaptado de [8]. 31
4.4 Inicialização dos Servidores. 36
4.5 Exemplo 1. Espaço Lógico criado (Servidor). 36
4.6 Exemplo 1. Tuplas inseridas no Espaço Lógico (Cliente). 37
4.7 Exemplo 1. Output (Cliente). 39
4.8 Exemplo 2. Espaço Lógico criado (Servidor). 40
4.9 Exemplo 2. Tuplas inseridas no Espaço Lógico (Cliente). 41
4.10 Exemplo 2. Output (Cliente). 43
4.11 Exemplo 3. Espaço Lógico criado (Servidor). 44
4.12 Exemplo 3. Tuplas inseridas no Espaço Lógico (Cliente). 45
4.13 Exemplo 3. Output (Cliente). 47
4.14 Exemplo de um serviço de TV ou de Filmes. Adaptado de [36]. 48

Lista de Abreviaturas

1. *ACL - Access Control List* 27

2. *BFT - Byzantine Fault Tolerant* 23

3. *DNS - Domain Name System* 14

4. *DNA - Deoxyribonucleic Acid* 47

5. *DoS - Denial of Service* 32

6. *ID - Identificador Único* 23

7. *IP - Internet Protocol* 13

8. *MTTF - Mean Time to Fail* 10

9. *MTTR - Mean Time to Repair* 10

10. *PVSS - Publicly Verifiable Secret Sharing* 25

11. *TCP - Transmission Control Protocol* 13

12. *TI - Tecnologia da Informação* 2

13. *TS - Tuple Space* 26

14. *WSDS - Web Service Dependable Space* 28

15. *WWW - World Wide Web* 5

Capítulo 1

Introdução

As aplicações distribuídas têm cumprido um papel chave na sociedade no que diz respeito à levar informações aos lugares mais remotos e conectar as pessoas destes lugares. Já os sistemas distribuídos têm assumido uma capacidade de se adaptar e operar diante de diferenças nos tipos de *hardware* presentes e de diferentes *softwares* utilizados.

Devido ao escopo dos ambientes, a especificidade da utilização dos variados tipos de dados, existem diferentes tipos de sistemas distribuídos que visam abstrair a presença de diferentes sistemas operacionais.

Atender conjuntos de pessoas e conjuntos de arquiteturas de computadores de maneira singular traz muitos desafios. Um dos motivos é de que um sistema distribuído tem de minimizar o fato de que seus recursos e processos, executados ou não por pessoas, podem estar distribuídos por 2 ou mais computadores. Essa capacidade dos sistemas distribuídos de se apresentarem para seus usuários e aplicações como se fossem um sistema de um único computador é chamado transparência.

Um sistema distribuído transparente leva em consideração a localização física ou lógica de seus componentes, sejam eles: usuários, computadores, recursos, dispositivos com sistemas embarcados, e outros sistemas.

Além da transparência de distribuição, sistemas distribuídos têm o enfoque de proporcionar acesso a dados de uma forma facilitada. A forma como os recursos são acessados, alocados e compartilhados indica qual é o funcionamento do sistema distribuído, bem como a ação e a coordenação dos usuários e dos outros componentes.

Sistemas distribuídos baseados em coordenação de processos foram criados com o enfoque de coordenar seus componentes, sejam estes: móveis, centralizados, replicados, ou híbridos.

Usuários desses tipos de sistemas distribuídos não se comunicam de forma direta, mas por meio de um espaço que funciona como uma memória compartilhada distribuída. Essa forma de comunicação tem a vantagem de gerar outra comunicação futura, e os usuários não carecem de estar ativos ao mesmo tempo, nem de se conhecerem. O desacoplamento de espaço e de tempo fica evidenciado na linguagem Linda [11] que faz uso de Espaço de Tuplas para o compartilhamento de dados.

Ao mesmo tempo que um usuário possui vantagens em se comunicar somente com o Espaço de Tuplas, não tendo de se preocupar com questões relativas à coordenação ou se outros usuários estão ativos ou não, vem à tona preocupações com o estado do Espaço de Tuplas, e das tuplas (dados), isto é, se estes dados são confiáveis ou não.

Uma implementação de um Espaço de Tuplas com segurança de funcionamento pode ser observado no *middleware* DepSpace [8]. É importante que um Espaço de Tuplas possua segurança de funcionamento, que é a capacidade de um sistema de fornecer um serviço mesmo na presença de falhas (erros de *software* ou intrusões), pois isso permite que o mesmo forneça um serviço segundo sua especificação para aplicações.

Geralmente, as propostas para Espaço de Tuplas consideram apenas as primitivas básicas de leitura e escrita de tuplas definidos no *middleware* pioneiro para Espaço de Tuplas, Linda [17]. Embora muitas aplicações possam ser construídas sobre essas primitivas, as mesmas possuem algumas limitações e podem não apresentar um bom desempenho. Por exemplo, se um usuário (cliente) quiser consultar uma grande quantidade de tuplas (dados) para tomar alguma decisão, todas essas tuplas devem ser lidas do Espaço de Tuplas e enviadas ao cliente, consumindo muitos recursos de rede e, muito provavelmente, apresentando um baixo desempenho.

1.1 Justificativa

A maior motivação para o desenvolvimento de *middleware* ou aplicativos na área de sistemas distribuídos decorre principalmente da demanda cada vez maior, de que o setor de tecnologia satisfaça as necessidades dos mais variados setores, e da população em geral.

A sociedade moderna, assim dita após os primeiros anos do século XXI, tem cada vez mais dependido da tecnologia no seu dia a dia pela facilidade com que os serviços podem ser entregues de forma rápida e segura.

Os investimentos em tecnologia de ponta por meio de institutos, do setor industrial, de empresas do ramo, e do meio científico em geral; são relevantes e causam impactos uns nos outros de forma ramificada. A área de tecnologia promove também um ganho de conhecimento, relativo a comunicação, muito devido a grande quantidade de informações trocadas.

Empresas de *TI* antes mesmo de lançarem produtos, depois de realizados alguns testes, disponibilizam versões betas para alguns usuários com a finalidade de medir o grau de satisfação do usuário e testar se o produto é confiável ou não. Essa medida é adotada por muitos segmentos, pois a opinião dos clientes é valorosa, e se agrava se o contato entre as partes se torna mais presente.

O uso e a compra de dispositivos móveis e aparelhos tecnológicos não cessam de crescer. Áreas relacionadas à saúde ou à aviação, por exemplo, que também usufruem de um aparato avançado, devem ser seguras pois lidam com vidas humanas, o que ressalta a importância da segurança no desenvolvimento de tecnologia.

Áreas midiáticas que englobam emissoras, rádios, indústria cinematográfica, e incluindo a indústria de videogames e jogos têm investido cada vez mais em tecnologia como forma de otimizar seus produtos ou serviços. Os dois primeiros dependem das redes de transmissão, da taxa de transmissão de dados, de *hardware* e de infraestrutura de longa distância para prover seus serviços. Os dois últimos têm se aproximado da área de *software* com grande intensidade, mas dependendo muito também na parte de *hardware* dos processadores gráficos, mais especificamente na área de computação gráfica.

Exemplos como esses mostram quão diversos podem ser os nichos impactados tanto pelo desenvolvimento de *software* quanto pelos sistemas distribuídos. Os usuários de uma aplicação distribuída, em geral, têm demandado um serviço que:

- Seja seguro e confiável com relação ao conteúdo e com quem possa acessá-lo.
- Esteja sempre disponível e de forma contínua.
- Seja rápido com relação ao tempo de resposta.
- Seja flexível, ou seja, que possa ser acessado por dispositivos diferentes e/ou ser acessado em diferentes lugares.
- Permita configurações diferentes dependendo do usuário.

A criação de um Espaço de Tuplas com segurança e que possibilita a utilização de consultas ou *queries* para otimização do desempenho visa justamente atender à essas aspirações e as aspirações correntes da área de sistemas distribuídos e de tecnologia em geral. Nosso trabalho propõe a adição de uma nova funcionalidade a um Espaço de Tuplas, estendendo o conjunto básico de primitivas ou operações que podem ser executadas no Espaço de Tuplas. Esta nova funcionalidade diz respeito a adição de uma operação para envio de consultas a serem processadas no Espaço de Tuplas, isto é, nos servidores que implementam o Espaço de Tuplas. Desta forma, esta abordagem evita a necessidade de que as tuplas sejam enviadas para os clientes ou usuários, isto é, ao invés de todas as tuplas serem enviadas (ou um grande conjunto delas), apenas a resposta da consulta é enviada ao cliente, o que diminui as necessidades de rede. Devido ao fato de consultas serem processadas nos servidores que suportam o Espaço de Tuplas, chamamos nosso Espaço de Tuplas de Active Space.

1.2 Objetivos

Nesta seção serão descritos o objetivo geral e os objetivos específicos do trabalho.

1.2.1 Objetivo Geral

O objetivo geral da pesquisa é implementar um sistema de *query* ou consulta sobre o DepSpace, uma plataforma que suporta que serviços e aplicações sejam construídas sobre o mesmo, levando em consideração questões relativas a segurança e a dependabilidade (funcionamento pleno e sem erros na aplicação).

1.2.2 Objetivos Específicos

Visando atender o objetivo geral, este trabalho possui os seguintes objetivos específicos:

- Estudar os conceitos de segurança de funcionamento e as propostas para Espaço de Tuplas em sistemas distribuídos.
- Estudar o DepSpace, uma implementação de Espaço de Tuplas com segurança de funcionamento.
- Propor e implementar uma forma de integrar uma camada de execução de *queries* nos servidores do DepSpace.
- Analisar a solução proposta através da implementação de aplicações.

1.3 Organização do Texto

O restante deste texto está organizado da seguinte forma:

1. O Capítulo 2 trata de sistemas distribuídos, se aprofundando em como garantir segurança e dependabilidade nestes sistemas.
2. O Capítulo 3 trata da coordenação baseada em tuplas, paradigma este introduzido para lidar com programação concorrente no sentido de coordenar processos por meio de permutação de tuplas presentes num Espaço de Tuplas. Esse mesmo capítulo trata de Espaço de Tuplas com segurança de funcionamento.
3. O Capítulo 4 discute o Active Space, que se baseia nos princípios teóricos e práticos dos capítulos anteriores, e que é uma implementação de um sistema de *query*, tornando o Espaço de Tuplas ativo. A proposta leva em conta as questões relativas à segurança e os impactos que sua utilização pode trazer ao público em geral.
4. O Capítulo 5 conclui o trabalho. Além da conclusão da monografia também é descrito o ferramental e os recursos utilizados para sua elaboração.

Capítulo 2

Sistemas Distribuídos

O segundo capítulo descreve o funcionamento e características de sistemas distribuídos perante aos usuários, e as configurações que permitem alcançar a segurança de funcionamento nesses sistemas.

Um sistema distribuído é geralmente composto por um conjunto de computadores (nós) conectados através de uma rede de computadores (por exemplo, a *Internet*) em que se comunicam através de trocas de mensagens. Cada um dos nós é representado por um computador completo (que, por exemplo, possui sua própria memória) com um conjunto completo de periféricos e seu próprio sistema operacional. Muitas vezes esses sistemas são espalhados em uma grande área geográfica.

O *middleware* é frequentemente colocado no topo do sistema operacional para fornecer uma camada uniforme pela qual as aplicações possam interagir. Os vários tipos de *middleware* incluem os baseados em documentos e em arquivos, em objetos e em coordenação. Alguns exemplos são: Corba [35] baseado em objetos, *WWW* [6] baseado em documentos e arquivos, Linda [11] e Jini [3] baseados em coordenação.

> "Um sistema distribuído é uma coleção de computadores autônomos que aparentam para os seus usuários como se fossem um único e coerente sistema."
>
> Andrew S. Tanenbaum

Aos olhos do usuário essa visão de que se trata de um único sistema denota uma distribuição transparente de todas as outras partes que compõem o sistema, sejam estas as outras máquinas e também os outros usuários.

Um objetivo importante de um sistema distribuído é esconder o fato de que seus processos e seus recursos estão fisicamente distribuídos através de múltiplos computadores, ou seja, existe uma transparência de distribuição. Um sistema distribuído que é capaz de se apresentar para usuários e aplicações como se fosse um sistema de um único computador é dito transparente.

A transparência de acesso trata de esconder a representação dos dados e a maneira como os recursos podem ser acessados pelos usuários. É desejável esconder diferenças na arquitetura das máquinas, mas mais importante é chegar a um acordo de como os dados são representados em máquinas diferentes e diferentes sistemas operacionais.

Um tipo de transparência importante é o da localização de um recurso. Transparência de localização se refere ao fato de usuários não saberem onde um recurso está localizado fisicamente no sistema. Sistemas distribuídos nos quais os recursos podem ser movidos,

sem afetar em como esses recursos são acessados, são capazes de prover transparência de migração.

O mesmo acontece em situações em que os recursos estão sendo realocados enquanto estão sendo acessados sem que os usuários ou a aplicação notem alguma diferença. Um exemplo de transparência de realocação ocorre quando usuários móveis utilizando celulares, por exemplo, continuam com acesso a rede mesmo se deslocando de um lugar para outro longínquo, sem serem temporariamente desconectados ou perderem acesso.

A replicação tem um papel importante em sistemas distribuídos. Recursos podem ser replicados para aumentar a disponibilidade ou melhorar o desempenho colocando cópias próximas de onde são acessadas. Transparência de replicação esconde o fato de que várias cópias de um, ou de que cópias de vários recursos existem.

Um outro importante objetivo dos sistemas distribuídos é permitir o compartilhamento de recursos. Recursos podem ser compartilhados de modo cooperativo ou de modo competitivo. Este último, como no caso de dois usuários independentes que desejam, por exemplo, guardar seus arquivos no mesmo servidor de arquivos. Em qualquer que seja o caso é importante que os usuários não notem que estão fazendo uso do mesmo recurso.

Uma questão importante de um acesso a um recurso compartilhado é de que, mesmo depois de vários acessos (que podem ser concorrentes) de vários usuários, o recurso esteja em um estado consistente. A consistência pode ser alcançada através de mecanismos de trava (mutexes, semáforos, monitores) em que é dado ao usuário acesso exclusivo ao recurso desejado. Um mecanismo mais refinado é fazer uso de transações para atingir a consistência, mas que são ao mesmo tempo mais difíceis de serem implementadas em sistemas distribuídos.

2.1 Segurança de Funcionamento em Sistemas Distribuídos

Uma definição alternativa de um sistema distribuído da citada anteriormente é:

> "Você sabe que está diante de um sistema distribuído quando um computador do qual você nunca ouviu falar para de funcionar e impede você de terminar o seu trabalho."
>
> Leslie Lamport

Se um *crash* (ou parada) de um computador em um sistema distribuído traz situações indesejáveis para outros computadores ou o sistema como um todo, é necessário tratar desse defeito. Existe transparência de defeitos num sistema distribuído quando usuários não percebem que recursos ou outros dispositivos falham ou não funcionam de maneira adequada.

O usuário pode também não notar que o sistema acabou de se recuperar de um defeito. Mascarar defeitos é algo muito desafiador relacionado à sistemas distribuídos, em alguns casos algo impossível de ser implementado, pois a maior dificuldade é distinguir quando um recurso está indisponível ou quando este é de um acesso muitíssimo lento.

A dependência cada vez maior nos sistemas distribuídos tem levado um grande número de pesquisadores e empresas a investigarem por sistemas distribuídos mais confiáveis,

dessa forma foi criado o termo dependabilidade ou segurança de funcionamento para expressar a capacidade de sistema distribuído sobreviver a falhas em uma parte de seus componentes. Estas falhas podem ser de maneira acidental (ex: erros de desenvolvimento) ou maliciosas (ex: intrusões). A Figura 2.1 ilustra uma taxonomia bastante aceita para a dependabilidade, em que:

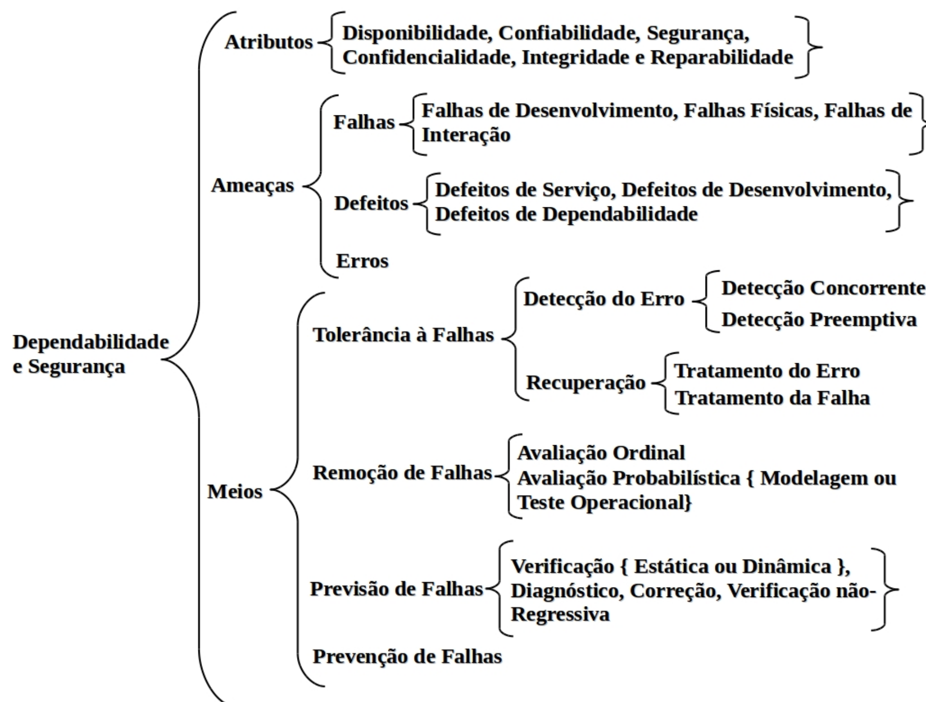

Figura 2.1: Taxonomia de Dependabilidade e Segurança. Adaptado de [5].

- As ameaças à dependabilidade de um sistema são compostas por: falhas, erros, e defeitos.
- Os meios para se obter a dependabilidade são compostos por: prevenção de falhas, tolerância a falhas, remoção de falhas e previsão de falhas.
- Os atributos de dependabilidade são: disponibilidade, confiabilidade, segurança, confidencialidade, integridade e reparabilidade.

A crescente utilização de computadores na sociedade moderna, e em certas atividades com um maior grau de intensidade, tem gerado uma dependência no uso destes equipamentos e seus sistemas.

Atividades que têm apresentado uma melhoria diferencial com a utilização dos sistemas distribuídos são: as transações bancárias, os sistemas integrados de bolsas de valores e mercadorias, o desenvolvimento e o controle automotivo e aeronáutico, o projeto e as inspeções de plataformas marítimas, o projeto e o controle de poços de petróleo e gás, o controle de tráfego aéreo, o controle de reatores nucleares, os sistemas de defesa e os sistemas de suporte à vida.

O mau funcionamento de algum dos componentes do sistema distribuído pode acarretar enormes prejuízos materiais, ou até mesmo a perda de vidas humanas. Para que se

possa expressar o quanto um sistema é confiável, é necessário definir e quantificar termos relacionados à dependabilidade.

A dependabilidade é uma propriedade dos sistemas distribuídos que define a capacidade dos mesmos de prestar um serviço no qual se pode justificadamente confiar. O serviço prestado por um sistema é o seu comportamento, tal qual percebido pelos usuários deste sistema.

O usuário de um serviço pode ser um sistema automatizado ou ser um humano que interage com o primeiro através da *interface* do serviço. A função de um sistema traduz o modo para o qual este foi projetado e é descrita na especificação do sistema. Diz-se que um serviço é correto quando implementa a especificação do sistema.

2.1.1 Ameaças

Existem vários aspectos que podem ameaçar a dependabilidade de um sistema distribuído. Um sistema apresenta defeito quando não é capaz de prestar um serviço correto, ou seja, seu serviço se desvia da especificação do sistema. O defeito é o evento que causa a transição de estado do serviço de um sistema de correto para um serviço incorreto, ou para um estado que não implementa a especificação do sistema.

A restauração do serviço é o evento que faz o serviço de um sistema retornar ao estado de serviço correto. Podemos especificar uma cadeia de ameaças à dependabilidade. O objetivo da especificação de ameaça à dependabilidade é a identificação mais precisa das falhas, dos erros e suas propagações e dos defeitos que podem ocorrer em um sistema distribuído [4].

A falha é o elemento que ocasiona o erro, provocando no sistema uma transição de estado não planejada levando o sistema para um estado de erro. Um sistema pode apresentar uma ou mais falhas e não apresentar erros. Neste caso, a falha é denominada falha latente.

Uma falha que efetivamente produz um erro passa a ser classificada como falha ativa. O tempo entre o surgimento da falha e sua ativação, ou a produção do erro, é chamado de latência de falha. Os tipos de falhas e as suas origens são bastante variados. Existe um sistema de classificação de falhas bastante abrangente, que dizem respeito à:

- Causa fenomenológica: falhas naturais, ou humanas.
- Intenção: falhas acidentais, deliberadas (não maliciosas), ou deliberadamente maliciosas.
- Fase de criação da ocorrência: falhas no desenvolvimento, na produção, ou operacionais.
- Domínio: falhas físicas ou da informação.
- Fronteira do sistema: falhas internas ou externas.
- Persistência: falhas permanentes ou transientes.

A Figura 2.2 mostra a propagação de falhas, erros e defeitos. A ativação de uma ou mais falhas pode ocasionar erros e a propagação de um ou mais erros pode ocasionar

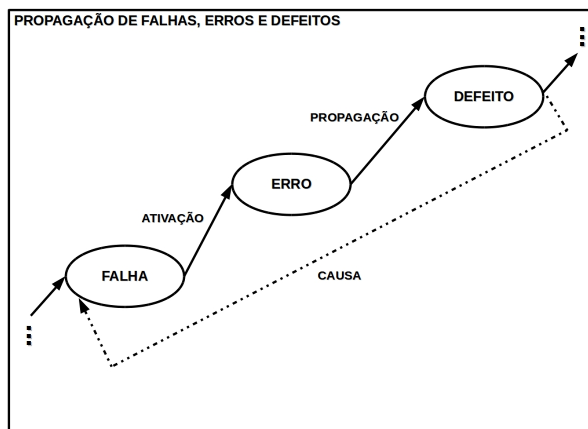

Figura 2.2: Propagação de Falhas, Erros e Defeitos. Adaptado de [4].

defeitos. Os defeitos podem também causar falhas criando um ciclo vicioso entre esses três elementos.

Um defeito pode ocorrer quando um erro existente no sistema alcança a *interface* do serviço e altera o serviço prestado. O erro é um estado indesejado do sistema que pode ou não vir a causar um defeito. Desta forma, um sistema pode ter um ou mais erros e continuar apresentando um serviço correto, sem defeito.

O tempo entre o surgimento de um erro e a manifestação do defeito é chamado de latência de erro, e sua duração pode variar consideravelmente dependendo das circunstâncias e do sistema distribuído.

Os defeitos apresentados podem ser de diferentes tipos e sua classifição baseia-se em características e comportamentos apresentados, que são comumente chamados de modos de defeito (*failure modes*). Os parâmetros de domínio, de percepção pelos usuários e de consequências no ambiente, classificam os defeitos como:

- Domínio: valor ou tempo.
- Percepção pelos usuários: defeitos consistentes ou inconsistentes (defeitos bizantinos).
- Consequências no ambiente: defeitos menores, crescendo até a escala de defeitos catastróficos [15].

2.1.2 Atributos

Nos atributos de dependabilidade é convencional o uso de medidas probabilísticas para enfatizar seus pesos relativos. A razão para tal baseia-se na natureza não determinística das circunstâncias dos atributos (disponibilidade, confiabilidade, segurança, confidencialidade, integridade e reparabilidade). Os atributos podem ter um maior ou menor peso relativo e são descritos como [5]:

- Disponibilidade: prontidão para execução do serviço correto.
- Confiabilidade: continuidade do serviço correto.

- Segurança: ausência de consequências catastróficas sobre o usuário e o ambiente.
- Integridade: ausência de alterações impróprias no sistema.
- Reparabilidade/Manutenibilidade: habilidade de passar por mudanças e reparos.
- Confidencialidade: ausência da revelação não autorizada de informação.

A disponibilidade instantânea é o atributo definido como a probabilidade de um sistema apresentar um serviço correto num determinado instante de tempo, mas também é observado seu comportamento em longos períodos de tempo. É importante saber a alternância entre períodos de serviço correto e os períodos que o sistema estava sob reparo, sendo necessário saber o tempo médio para a ocorrência de defeito ($MTTF$) e o tempo médio para reparo ($MTTR$).

A confiabilidade é quantificada como uma probabilidade, de o sistema não apresentar defeito durante o intervalo de tempo considerado, ou seja, apresentar um serviço correto continuamente [15].

Os modos de defeito, descritos anteriormente, são ordenados em níveis de severidade relacionados com alguns atributos de dependabilidade, em que os critérios são [4]:

- Para disponibilidade: a duração da interrupção do sistema ou parada.
- Para segurança: a possibilidade de vidas humanas estarem em risco.
- Para confidencialidade: o tipo de informação que pode ter sido desnecessariamente revelada.
- Para integridade: o grau de corrompimento dos dados e a habilidade de recuperar os dados corrompidos.

Segurança de funcionamento não diz respeito a um único atributo de dependabilidade pois possui uma noção composta, a saber: a combinação de confidencialidade (a prevenção de serem reveladas informações de maneira não autorizada), integridade (o ato de prevenir a adição ou remoção de informações de forma não autorizada) e disponibilidade (prevenindo de informações estarem retidas de forma não autorizada). Uma definição mais unificada de segurança de funcionamento é: a ausência de acesso ou gerenciamento não autorizado do estado do sistema.

A dependabilidade, composta por todos os atributos, visa evitar defeitos em geral. A segurança de funcionamento (*security*) composta por disponibilidade, confidencialidade e integridade, visa evitar classes mais específicas de defeitos como: defeitos bizantinos, que ocorrem por meio de acesso ou gerenciamento não autorizado de informação. É importante enfatizar a diferença com o atributo específico de segurança (*safety*) que visa evitar a classe de defeitos catastróficos, salvaguardando os seres humanos que possam vir a ser impactados pelo sistema distribuído.

A reparabilidade ou manutenibilidade está relacionada com os meios para se obter dependabilidade, mais especificamente com remoção de falhas. A definição de reparabilidade vai além de manutenção corretiva e manutenção preventiva, que serão vistas mais à frente, e abrange outras formas de manutenibilidade como uma manutenção adaptável e também aumentativa [4].

Alguns métodos de embaralhamento ou para cifrar dados como funções de *hash* podem ser utilizados para se alcançar ou garantir a confidencialidade.

2.1.3 Meios

Os meios de alcançar dependabilidade dizem respeito à: trabalhar na prevenção, remoção, previsão ou tolerância das falhas. Estas políticas têm como objetivo tratar as falhas que podem ser ativadas levando aos erros, que por sua vez ocasionam os defeitos.

Prevenção de Falhas

A abordagem de prevenção de falhas tem como objetivo aumentar a confiabilidade dos sistemas empregando técnicas de controle da qualidade nas etapas de projeto e desenvolvimento dos sistemas. Não há como eliminar todas as falhas possíveis, então assume-se que eventualmente irão ocorrer defeitos no sistema.

Procedimentos manuais de reparo são executados a fim de restaurar o sistema à condição de serviço correto. A prevenção de falhas é insuficiente para alguns sistemas atingirem a dependabilidade [15].

Remoção de Falhas

A técnica de remoção de falhas pode ser aplicada na fase de desenvolvimento ou ao longo da vida operacional do sistema. A remoção de falhas na fase de desenvolvimento é realizada através das etapas de verificação, diagnóstico e correção.

Na fase de verificação ocorre o processo de checagem para se saber se o sistema atende às condições de verificação. Caso seja constatado que as condições não são atendidas, é realizado um diagnóstico das falhas que acarretaram o não atendimento. Finalmente é executada a correção do sistema para eliminação das falhas diagnosticadas.

As técnicas de verificação podem ser classificadas em estáticas e dinâmicas. A verificação estática verifica o sistema sem colocá-lo em execução, já a verificação dinâmica é realizada com ele em funcionamento. A remoção de falhas através de atividades de manutenção pode ocorrer com o sistema também em funcionamento ou parado. As duas atividades de manutenção comumente consideradas são:

- Manutenção corretiva: tem como objetivo remover falhas diagnosticadas no sistema.
- Manutenção preventiva: que tem como meta descobrir e remover falhas latentes no sistema [15].

Previsão de Falhas

A previsão de falhas é uma avaliação do comportamento do sistema com relação à ocorrência e ativação de falhas, e que possui um aspecto qualitativo e outro quantitativo.

No caso qualitativo busca-se identificar, classificar e ordenar por importância as causas de defeito no sistema, enquanto no quantitativo mede-se em termos probabilísticos o atendimento dos atributos de dependabilidade.

Tanto a avaliação qualitativa quanto a quantitativa têm como meta a obtenção de dados que façam a validação das escolhas feitas na constituição da estrutura de dependabilidade do sistema. Estas abordagens podem ainda subsidiar as modificações estruturais para que a sua eficácia ou eficiência seja melhorada [15].

Tolerância a Falhas

A melhor forma de tolerância a falhas, caracterizada pela segurança e operacionalidade garantida, é a que realiza o mascaramento para encobrir ou ocultar falhas. Através do mascaramento o serviço apresentado pelo sistema não deverá apresentar defeito. É a forma mais completa de tolerância a falhas, a mais desejada e também a de maior custo.

A forma conhecida como não tolerância a falhas é considerada extrema e apresenta a solução mais trivial e com o custo mais reduzido, mas é a mais frágil e a mais indesejada. Na ocorrência de falhas o sistema provavelmente apresentará defeito e nada poderá ser afirmado quanto ao estado dele, podendo ingressar em um estado não operacional, ou até mesmo em um estado inseguro.

Como opção em relação às duas abordagens anteriores, encontram-se duas formas intermediárias de tolerância a falhas. São estas:

- Aquela que garante que o sistema irá permanecer em um estado seguro, mas nada diz sobre o seu estado operacional, e por isso é chamada de defeito seguro. De uma forma geral, a opção de defeito seguro é sempre preferível em relação à tolerância a falha sem mascaramento, uma vez que segurança na maioria das ocasiões é muito mais importante que permanecer operacional.
- A segunda forma é aquela que garante que o sistema irá permanecer operacional. Ainda que o mesmo ingresse, por causa da falha, em um estado inseguro. Esta abordagem é denominada de tolerância a falhas sem mascaramento.

Na Figura 2.3 é apresentada uma taxonomia de tolerância a falhas, que subdivide-se em detecção do erro e recuperação:

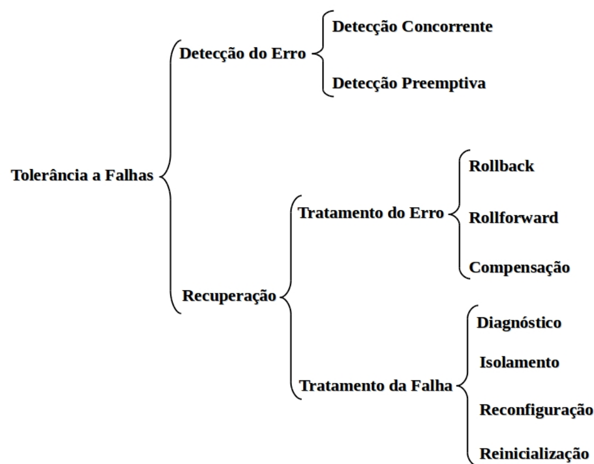

Figura 2.3: Taxonomia de Tolerância a Falhas. Adaptado de [5].

A detecção de erros subdivide-se em detecção concorrente e detecção preemptiva, e corresponde a identificação da presença de um erro. A detecção concorrente ocorre durante a entrega normal do serviço. A detecção preemptiva ocorre quando a entrega normal do serviço está suspensa checando o sistema por erros latentes e falhas dormentes.

Na recuperação, um estado do sistema que contenha um ou mais erros e possíveis falhas é transformado em um estado sem erros detectados e sem falhas que possam ser ativadas de novo. Subdivide-se em: tratamento de erros e tratamento de falhas.

Tratamento de falhas evita as falhas de serem ativadas outra vez, e subdivide-se em:

- Reinicialização: testa, atualiza e registra uma nova configuração, atualizando tabelas e também registros do sistema.
- Reconfiguração: ou troca para componentes reservas ou redistribui tarefas dentre os componentes não defeituosos.
- Isolamento: efetua uma exclusão física ou lógica de componentes defeituosos participarem da entrega do serviço, ou seja, os torna dormentes.
- Diagnóstico: identifica e registra as causas dos erros, em termos tanto de suas localizações quanto seus tipos.

Tratamento de erros elimina erros do estado do sistema, e subdivide-se em:

- Rollforward: um estado sem erros detectados é o novo estado.
- Rollback: leva o sistema para um estado anterior a ocorrência dos erros.
- Compensação: o estado errôneo contém suficiente redundância para permitir que o erro seja mascarado.

Considerando as formas de redundância de *hardware* e *software*, estas se baseiam na replicação de partes componentes de um sistema, podendo também considerar até o sistema como um todo. Réplicas podem ser idênticas em sua constituição ou não, mas sempre possuem a mesma função, a capacidade de permitir a tolerância de falhas permanentes.

Exemplo de redundância na estrutura baseada em *software* é a técnica conhecida como *N-version programming*. Na abordagem é considerada a utilização de duas ou mais versões de um só algoritmo, que é programado de forma independente, comparando suas saídas, das quais é escolhido o resultado correto. A técnica em hardware é conhecida como *N-modular redundancy*, em que utilizam-se cópias de um mesmo módulo de *hardware*.

Os métodos baseados em redundância no tempo são caracterizados pela repetição da mesma atividade uma ou mais vezes. O motivo da repetição se baseia no fato de que a causa do problema é de natureza temporal.

Um exemplo clássico de redundância no tempo é a retransmissão de mensagens que ocorre em protocolos de comunicação. Tomando o protocolo *TCP* (da arquitetura *TCP/IP*) como exemplo, na utilização deste protocolo é efetuada uma retransmissão de segmentos *TCP* todas as vezes que o remetente da mensagem deixa de receber a confirmação do recebimento pelo destinatário [15].

2.2 Garantindo Segurança de Funcionamento em Sistemas Distribuídos através da Replicação

Em tolerância a falhas, o emprego de componentes redundantes em paralelo, operando simultaneamente com o intuito de aumentar a confiabilidade de um sistema, é conhecido como redundância ativa (*active redundancy*) ou replicação por máquina de estados.

Existe também a (*standby redundancy*) que é a de redundância em reserva. Neste método apenas um dos componentes redundantes, denominado de primário, fica operacional e todos os demais permanecem inativos. Quando o primário apresenta defeito, um dos componentes em reserva é ativado de imediato e passa a prestar o serviço no lugar daquele.

2.2.1 Replicação por Máquina de Estados

A replicação de dados é a manutenção de cópias dos dados em vários computadores. A replicação é o segredo da eficácia dos sistemas distribuídos, pois pode fornecer um melhor desempenho, alta disponibilidade e tolerância a falhas.

A replicação é amplamente usada como, por exemplo, no armazenamento de recursos de servidores *web* na *cache* dos navegadores e em servidores *proxies*, pois os dados mantidos nas *caches* e nos *proxies* são réplicas uns dos outros. O serviço de atribuição de nomes *DNS* mantém cópias de mapeamentos entre nomes e atributos dos computadores e conta-se com ele para o acesso diário aos serviços pela *Internet*.

A replicação por máquina de estados exige o determinismo de réplicas: todas as réplicas partem de um mesmo estado inicial e executam a mesma sequência de operações (na mesma ordem), chegando a um mesmo estado final.

A replicação é uma técnica para melhorar serviços. As motivações para a replicação são: melhorar o desempenho de um serviço, aumentar sua disponibilidade ou torná-lo tolerante à falhas.

Melhoria do Desempenho

A colocação dos dados na *cache* em clientes e servidores é uma maneira conhecida de melhorar o desempenho. Navegadores e servidores *proxies* colocam na *cache* cópias de recursos *web* para evitar a latência da busca desses recursos no servidor de origem.

Os dados são replicados, em alguns casos, de forma transparente entre vários servidores de origem no mesmo domínio. A carga de trabalho é compartilhada entre esses servidores por meio da vinculação de seus endereços *IP* ao nome *DNS* do site buscado. Uma pesquisa de *DNS* do site buscado resulta no retorno de um dos vários endereços *IP* de servidores, em um sistema de rodízio.

A replicação de dados imutáveis é simples: ela aumenta o desempenho com pouco custo para o sistema. A replicação de dados mutáveis (dinâmicos), como os da *web*, acarreta sobrecargas nos protocolos para garantir que os clientes recebam dados atualizados. Assim, existem limites para a eficácia da replicação como uma técnica de melhoria de desempenho [13].

Maior Disponibilidade

Os usuários exigem que os serviços sejam de alta disponibilidade, isto é, a proporção do tempo durante a qual o serviço está acessível com o tempo de resposta razoável deve ser próxima a 100%.

Excetuando-se os atrasos decorrentes dos conflitos do controle de concorrência pessimista (bloqueio de dados), os fatores relevantes para a alta disponibilidade são: falhas do servidor ou particionamento da rede e operação desconectada, onde as desconexões da comunicação, que frequentemente não são planejadas são um efeito colateral da mobilidade do usuário.

A replicação é uma técnica para manter automaticamente a disponibilidade dos dados, a despeito das falhas do servidor. Se os dados são replicados em dois ou mais servidores que falham independentemente, então o *software* do cliente pode acessar os dados em um servidor alternativo. A porcentagem do tempo durante a qual o serviço está disponível pode ser melhorada pela replicação dos dados do servidor.

Tolerância a Faltas

Dados de alta disponibilidade não são necessariamente dados rigorosamente corretos. Eles podem estar desatualizados, por exemplo, dois usuários em lados opostos de uma rede que foi particionada podem fazer atualizações conflitantes e que precisem ser resolvidas.

Um serviço tolerante a falhas, por sua vez, sempre garante um comportamento rigorosamente correto, apesar de certo número e tipos de falhas. A correção está relacionada ao caráter atual dos dados fornecidos para o cliente e aos efeitos das operações do cliente sobre os dados.

A correção pode também estar relacionada ao tipo de resposta do serviço como, por exemplo, no caso de um sistema de controle de tráfego aéreo, onde dados corretos são necessários em escalas de tempo curtas.

Na Figura 2.4 é representado o problema dos generais bizantinos, como uma analogia ao que acontece em servidores em sistemas distribuídos, onde o general 2 recebe informações conflitantes e dessa maneira seu comportamento não é seguramente correto:

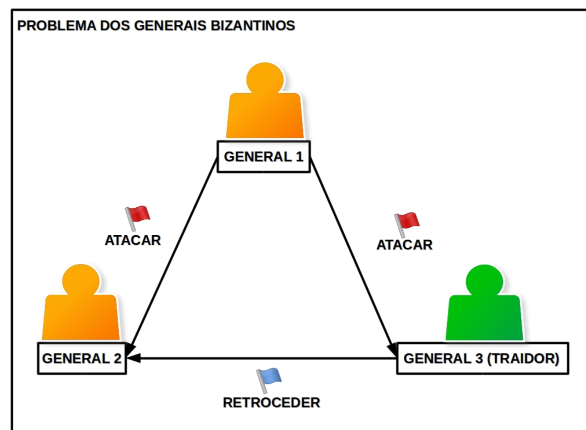

Figura 2.4: Problema dos Generais Bizantinos. Adaptado de [22].

A mesma técnica básica utilizada para alta disponibilidade de replicação de dados e funcionalidade entre computadores pode também ser aplicada para se obter tolerância a faltas. Se até um número x de servidores falham de um total de $x + 1$ servidores, então, em princípio, pelo menos um permanece para fornecer o serviço.

E se até x servidores podem apresentar falhas bizantinas, então, em princípio, um grupo de $2x + 1$ servidores pode fornecer um serviço correto, fazendo-se com que os servidores corretos vençam por voto os servidores falhos (que podem fornecer valores incorretos).

Na Figura 2.5 é apresentada a solução do problema dos generais bizantinos, onde se até x comportam-se de maneira incorreta pelo menos temos de ter $2x + 1$ corretos, de forma análoga aos servidores para propagar informações e dados corretos.

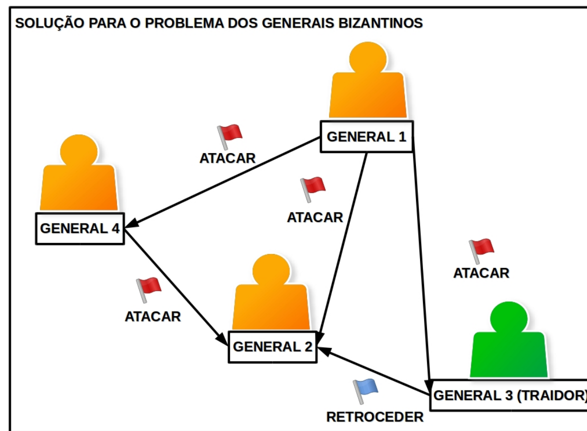

Figura 2.5: Solução para o Problema dos Generais Bizantinos. Adaptado de [22].

A tolerância a faltas é mais sutil do que essa descrição simples faz parecer. O sistema precisa gerenciar a coordenação de seus componentes de forma precisa para manter as garantias de correção diante de falhas, as quais podem ocorrer a qualquer momento.

Um requisito comum quando dados são replicados é a transparência da replicação, isto é, normalmente, os clientes não devem saber que existem várias cópias físicas dos dados.

Um outro requisito geral para dados replicados que pode variar com os níveis de rigor exigidos pelos aplicativos é a consistência. Isso diz repeito ao fato das operações executadas sobre um conjunto de objetos replicados produzirem resultados que satisfaçam a especificação da correção desses objetos [13].

Capítulo 3

Coordenação Baseada em Tuplas

O terceiro capítulo descreve o funcionamento de sistemas distribuídos baseados em coordenação, por meio de tuplas e Espaços de Tuplas, e ao final apresenta um Espaço de Tuplas que atende aos requisitos de dependabilidade.

A coordenação baseada em tuplas foi introduzida no final dos anos 1980 para programação concorrente e paralela, e consistia em um conjunto limitado de primitivas, as primitivas de coordenação (*coordination primitives*), para acessar o Espaço de Tuplas (*tuple space*). Pouco tempo depois, nos anos 1990, o modelo ganhou amplo reconhecimento como um paradigma de coordenação de propósito geral para programação distribuída.

Sistemas distribuídos baseados em coordenação assumem que vários componentes de um sistema são herdados de maneira distribuída e que os principais problemas em tais sistemas decorrem em se coordenar as atividades dos diferentes componentes. Em outras palavras, ao invés de se concentrar em uma distribuição transparente dos componentes, a ênfase está em coordenar as atividades entre estes componentes.

A abordagem chave seguida em sistemas distribuídos baseados em coordenação é a clara separação entre computação e coordenação. Se um sistema distribuído for visto como uma coleção de processos, então a parte relativa à computação é formada pelos processos, cada um com uma atividade específica, que em princípio é executada de forma independente das atividades de outros processos.

Em sistemas distribuídos baseados em coordenação a parte relativa à coordenação de um sistema distribuído trata da comunicação e cooperação entre processos. Neste modelo, o foco é em como essa coordenação entre os processos ocorre.

Uma terminologia de como é feita a coordenação diz respeito ao tempo e ao espaço (como um referencial) em um sentido, e ao desacoplamento ou acoplamento em outro, resultando em quatro formas distintas.

Se processos necessitam saber um nome ou identificador de outros processos (precisam "saber" quem são os outros processos) para poderem se comunicar dizemos que são acoplados espacialmente. Se processos que estão se comunicando precisam estar ativos e executando ao mesmo tempo dizemos que são acoplados de forma temporal.

Quando processos estão acoplados em espaço e em tempo dizemos que a coordenação ocorre de maneira direta. Quando existe desacoplamento tanto de tempo quanto de espaço os processos se comportam de forma oposta. Em outras palavras, quando um processo deseja coordenar suas atividades com outros processos, e eles não se conhecem explicita-

mente, existe o conceito de processos temporariamente se agruparem para coordenarem suas atividades.

O modelo de coordenação mais amplamente conhecido é o de combinação de processos desacoplados tanto em tempo quanto espacialmente, exemplificado como comunicação "generativa" apresentada na linguagem Linda [17].

A ideia chave introduzida nesse tipo de comunicação é que uma coleção de processos independentes fazem uso compartilhado de um espaço persistente de dados. Nesse espaço se encontram as tuplas, rotuladas como registros de dados e consistindo de um número de campos tipados. Processos podem colocar qualquer tipo de registro suportado no Espaço de Tuplas compartilhado, isto é, eles geram registros de comunicação, ou para futura comunicação.

Uma característica desses espaços de dados, que representam uma memória compartilhada distribuída, é a implementação de um mecanismo de busca associativo quando se busca por uma tupla contendo um tipo de dado específico.

Um processo interessado em um ou mais valores encontrados em uma tupla em questão, essencialmente especifica alguns desses valores, e uma tupla que corresponda à essa especificação é então extraída/removida, ou apenas lida do Espaço de Tuplas. Se nenhuma tupla for encontrada, o processo pode escolher bloquear ou não a operação momentaneamente até que exista uma ou mais tuplas que correspondam ao que foi utilizado pelo processo como base da busca.

A base de uma busca ocorre por meio de um *template*, que nada mais é do que uma tupla incompleta, no sentido de que em alguns campos seus existe um asterisco, que denota um campo com um valor genérico. Um *template* e uma tupla que possuem a mesma quantidade de campos se correspondem se os seus campos, quando comparados um a um, são equivalentes, e com a ressalva de que campos com asteriscos são equivalentes a qualquer valor.

As operações ou primitivas de coordenação no Espaço de Tuplas são descritas e funcionam da seguinte maneira:

- *OUT*: insere uma tupla com seus respectivos campos e valores no Espaço de Tuplas.
- *RD*: lê uma tupla do Espaço de Tuplas que se corresponda com um *template* de forma bloqueadora, ou seja, se não existir uma tupla correspondente, o processo que chamou a operação é bloqueado até que exista uma tupla correspondente.
- *RDP*: lê uma tupla do Espaço de Tuplas que se corresponda com um *template* de forma não bloqueadora, ou seja, mesmo se não existir uma tupla correspondente, o processo que chamou a operação não é bloqueado.
- *IN*: lê e remove uma tupla do Espaço de Tuplas que se corresponda com um *template* de forma bloqueadora.
- *INP*: funciona como a operação *IN*, mas de forma não bloqueadora.

Um exemplo utilizando as operações pode ser visto na Figura 3.1:

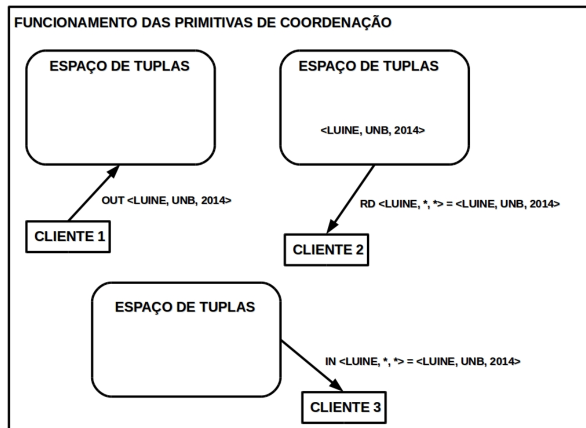

Figura 3.1: Funcionamento das Primitivas de Coordenação. Adaptado de [1].

3.1 Espaço de Tuplas

Esta seção descreve a implementação pioneira do Espaço de Tuplas na Linguagem Linda. E posteriormente uma implementação do *middleware* TSpaces [23] e do *middleware* LIME [26].

3.1.1 Linda

A primeira linguagem de coordenação foi Linda [19]. Linda teve suas origens na computação paralela, e foi desenvolvida como um meio de injetar a capacidade de programação concorrente nas linguagens de programação sequenciais.

Linda originalmente consistia de operações de coordenação (as primitivas de coordenação) e um espaço de dados compartilhados (Espaço de Tuplas), que continham dados (as tuplas). O Espaço de Tuplas é um espaço de dados compartilhado que age como uma memória associativa para um grupo de clientes ou agentes. A linguagem Linda possui as seguintes características de um Espaço de Tuplas, conforme já comentado:

- Há um desacoplamento entre os processos que interagem tanto em referência quanto em tempo. Em outras palavras o produtor e o consumidor de uma tupla não precisam saber o endereço uns dos outros, e nem estarem ativos ao mesmo tempo.
- Linda permite um endereçamento associativo. Isso significa que os dados são acessados em termos dos tipos de dados que são solicitados, e não de que dados específicos são referenciados.
- O suporte para assincronia e concorrência surge como uma parte intrínseca da abstração do Espaço de Tuplas.
- Existe uma separação entre a implementação da coordenação, das características da plataforma e da linguagem de programação.

Linda foi originalmente concebida para permitir processamento paralelo em ambientes de rede próximos e centralizados. Contudo, com a importância crescente de redes

abertas combinada com a necessidade por concorrência existe a necessidade do modelo de coordenação de Linda ser repensado sob estes aspectos.

Numa implementação entre sistemas abertos e sistemas fechados, a diferença chave é o dinamismo que pode ser inferido do primeiro. Em um sistema aberto, um cliente pode participar ou partir de acordo com sua vontade, enquanto em um sistema fechado os clientes participantes são fixados desde o início e não mudam.

É mais complexo otimizar a coordenação entre clientes se o sistema não souber antecipadamente quais os clientes que irão usar o espaço, ou quais características os clientes ativos possuem. Por exemplo, clientes podem não necessariamente concordar antecipadamente nos modelos de compartilhamento e nos tipos de dados que estão nas tuplas.

Pode tornar-se tarefa do *middleware* resolver (parcialmente) as heterogeneidades dos clientes. Para aplicações distribuídas construídas com base na linguagem Linda e em ambientes de Internet, as questões de dinamismo, heterogeneidade e escalabilidade precisam ser observadas.

3.1.2 TSpaces

TSpaces [23] foi um projeto concebido pelo Centro de Pesquisas Almaden da IBM, e consiste em um componente de *middleware* de mensagens que combina mensagens assíncronas com características de um banco de dados em um pacote portátil. Por ter sido escrito em Java possui a habilidade de executar virtualmente em qualquer plataforma.

A capacidade do banco de dados do TSpaces o coloca além de um simples sistema de mensagens puro. Sendo não só um armazenador de dados persistentes, bem como um sistema de entrega de dados.

O TSpaces fornece uma grande liberdade no que diz respeito ao gerenciamento de dados, com filas de mensagens servindo também como armazenadoras do banco de dados, podendo ser consultadas com sofisticadas linguagens de consulta.

O *middleware* promove flexibilidade ao usuário em utilizar um único sistema para comunicação e necessidades de armazenamento de dados básicas, e também tratar os dados permanentes e os dados transitórios da mesma maneira. TSpaces oferece aos usuários um estilo diferente de comunicação por múltiplos pontos.

TSpaces é um descendente direto do sistema Linda, possui uma interface de programação relativamente simples e a estrutura de dado primário é uma tupla. Tuplas ficam em espaços, que são simplesmente coleções de tuplas. Um usuário invoca operações do TSpaces sobre um espaço, lendo ou escrevendo tuplas. Usuários podem criar espaços, dando nomes e definindo propósitos a eles. Por exemplo, um espaço pode ser usado para guardar os trabalhos de impressão, enquanto outro para armazenar verificações de um sensor de temperatura, e outro ainda para suportar transações bancárias.

Cada campo de uma tupla possui um valor, que pode ser de um tipo primitivo (um inteiro, string, ou float) ou de um tipo mais complexo como: um vetor multidimensional (matriz), um objeto Java, uma classe Java. Tuplas podem até conter outras tuplas, criando um aninhamento, o que fornece uma maior liberdade tanto para o usuário quanto para o servidor do TSpaces gerenciar e manipular dados [23].

3.1.3 LIME

LIME [26] (Linda em ambiente móvel) é um modelo e *middleware* para suporte no desenvolvimento de aplicações que exibam mobilidade física de hospedeiros, mobilidade lógica de agentes, ou ambos. LIME adota a perspectiva de coordenação inspirada no trabalho do modelo de Linda.

O contexto para a computação, representado em Linda para acesso global e persistente ao Espaço de Tuplas, é refinado em LIME para o compartilhamento transitório de espaços de tupla nomeados identicamente por unidades móveis. Espaços de Tuplas são estendidos com a noção de localização e se programados possuem a habilidade para reagir a estados específicos.

LIME é uma resposta ao desafio imposto à engenharia de *software* com o advento da mobilidade, e que define uma abordagem original baseada em coordenação para o desenvolvimento de aplicações móveis.

Foi o primeiro modelo de coordenação e *middleware* a se dirigir a necessidade de integrar mobilidade física e lógica dos agentes, nele assume-se um conjunto de hospedeiros agindo como contêineres onde ficam localizados os agentes.

A conectividade entre os hospedeiros é assegurada por pontos de acesso, *links* com ou sem fio, e que pode ser alterada tanto pela mobilidade ou quanto pela explícita conexão ou desconexão. Agentes podem se mover de um hospedeiro para outro que esteja ao alcance de acordo com sua vontade.

LIME preserva em essência a simplicidade do estilo de computação desacoplado, por continuamente entregar todas as ações de coordenação através de uma interface simples que é percebida pelo agente como meramente um Espaço de Tuplas local. O acesso ao Espaço de Tuplas ocorre usando um conjunto estendido de operações projetado para facilitar respostas flexíveis e convenientes a mudanças nos Espaços de Tuplas.

Cada agente pode possuir múltiplos Espaços de Tuplas que podem ser compartilhados com outros agentes dentro do alcance de comunicação. O compartilhamento é manifestado estendendo logicamente os conteúdos de cada Espaço de Tuplas para incluir o que está presente em todos os Espaços de Tuplas participantes.

Os conjuntos de tuplas sendo compartilhadas mudam durante o tempo como resultado do controle local dos agentes com respeito ao compartilhamento e em resposta a mobilidade tanto dos agentes quanto dos hospedeiros.

Quando um hospedeiro entra no alcance de comunicação, o conjunto de Espaço de Tuplas se expande e quando ele sai do alcance, o conjunto se contrai. A rede de resultados é um contexto gerenciado de maneira transparente que expressa a si mesmo em termos de mudanças. O comportamento dos agentes é alterado tanto pela disponibilidade de novos dados quanto por respostas relativas às mudanças contextuais.

O processo de desenvolvimento do LIME envolveu uma integração próxima entre a definição semântica formal, a pragmática de implementação, e a avaliação da aplicação do modelo resultante e do *middleware*. Por fim, o projeto culminou em uma implementação baseada em Java do LIME *middleware*, um projeto com código aberto, e que pode ser um instrumento de engenharia de *software* efetivo no cenário móvel [26].

3.2 Espaços de Tuplas com Segurança de Funcionamento

Nesta seção é descrito o *middleware* DepSpace, uma implementação de Espaços de Tuplas que utiliza, principalmente, a replicação por máquinas de estado para fornecer um serviço com segurança de funcionamento. Também é apresentada uma extensão deste *middleware* para ambientes *web*, o WSDS.

O DepSpace é particularmente interessante pois serviu de base para a nossa proposta e implementação do Active Space, e por empregar tolerância a Falhas Bizantinas, diferentemente dos *middlewares* vistos anteriormente.

3.2.1 DepSpace

No DepSpace, uma tupla t onde todos os campos possuem valores definidos é chamada de entrada. Uma tupla com um ou mais campos indefinidos é chamada *template* ou molde (usualmente denotada por uma barra, isto é, $\bar{t}$). Um campo indefinido é representado por um asterisco ("$*$"). *Templates* são usados para permitir um acesso endereçado ao conteúdo de tuplas no Espaço de Tuplas.

Uma entrada t e um *template* $\bar{t}$ combinam se possuírem a mesma quantidade de campos e se todos os campos definidos de $\bar{t}$ são equivalentes aos correspondentes valores dos campos de t.

O molde $<$ *c1*, *c2*, $*>$ combina com quaisquer tuplas que possuam três campos, onde os valores dos dois primeiros campos são *c1* e *c2* respectivamente; observe que o terceiro campo desse conjunto de tuplas poderia possuir um valor qualquer e ainda assim a correspondência continuaria ocorrendo.

No DepSpace, uma tupla é inserida no Espaço de Tuplas utilizando a operação $out(t)$. A operação $rd(\bar{t})$ é utilizada para ler tuplas do Espaço de Tuplas, e no caso, retorna qualquer tupla oriunda do Espaço de Tuplas que combine com o *template* $\bar{t}$.

Uma tupla é lida e removida do Espaço de Tuplas utilizando a operação $in(\bar{t})$. As operações *rd* e *in* são bloqueadoras, e as versões não bloqueadoras das mesmas são: *rdp* e *inp*, também são implementadas no DepSpace.

Uma outra operação fornecida por este *middleware* é a $cas(\bar{t}, t)$, ou um "*swap*" atômico condicional, na qual: se não existe uma tupla no Espaço de Tuplas que combine com $\bar{t}$, insere t no Espaço de Tuplas e retorna *true*, caso contrário retorna *false*.

Um Espaço de Tuplas Confiável

Um Espaço de Tuplas é confiável se ele satisfaz os atributos de dependabilidade. Atributos relevantes no caso do DepSpace são: confiabilidade (as operações sobre o Espaço de Tuplas devem se comportar conforme suas especificações), disponibilidade (o Espaço de Tuplas deve estar pronto para executar as operações requisitadas), integridade (nenhuma alteração imprópria pode ocorrer no Espaço de Tuplas) e confidencialidade (o conteúdo das tuplas não pode ser revelado aos clientes não autorizados).

A dificuldade de garantir os atributos vem da ocorrência de faltas, tanto vindas de causas acidentais, isto é, um *bug* de *software* que causa um *crash*, o servidor para de funcionar; como vindas de causas maliciosas, isto é, um intruso que modifica ou corrompe tuplas de um servidor.

É difícil modelar o comportamento de um adversário malicioso, de tal forma que sistemas tolerantes à intrusões assumem a classe mais genérica de falhas, as arbitrárias ou Falhas Bizantinas. Neste cenário, um Espaço de Tuplas é construído utilizando uma replicação por máquina de estados *BFT* (Tolerante à Falhas Bizantinas) juntamente com um esquema de controle de acesso que propicia todos os atributos de dependabilidade, exceto confidencialidade que é implementada no DepSpace através de um esquema de distribuição de segredo, conforme descrito mais adiante.

Modelo de Sistema

O sistema é composto por um conjunto ilimitado de clientes que interagem com um conjunto de n servidores. Considera-se que cada cliente e cada servidor possuem um *id*, um identificador único.

Todas as comunicações entre os clientes e os servidores são realizadas sobre um canal de autenticação ponto a ponto confiável. Esses canais podem ser implementados utilizando *sockets TCP* e códigos de autenticação de mensagens com chaves sob a suposição de que a rede pode: perder, corromper, ou atrasar mensagens, mas não pode interromper a comunicação entre processos corretos indefinidamente.

Um Espaço de Tuplas confiável não necessita de qualquer suposição explícita de tempo. Entretanto, quando se utiliza uma primitiva *multicast* de ordem total baseada no protocolo de consenso *Paxos* bizantino para garantir que todas as réplicas executem a mesma sequência de operações, um modelo de sistema parcialmente síncrono é necessário para que algo correto ocorra depois de um certo tempo.

Assume-se que um número arbitrário de clientes e uma quantidade de até no máximo f servidores podem estar sujeitos à Falhas Bizantinas, ou seja, eles podem se desviar do algoritmo aos quais estão especificados para executar e trabalhar em colisão para corromper o comportamento do sistema.

A arquitetura requer uma quantidade n de servidores, em que $n \geq 3f + 1$, são necessários para tolerar os já mencionados f servidores faltosos. Também é assumida independência nas falhas dos servidores, ou seja, que as falhas dos f servidores não são relacionadas.

Arquitetura do DepSpace

A arquitetura do Espaço de Tuplas consiste em uma série de camadas integradas que reforçam cada um dos atributos de dependabilidade. No topo da pilha do lado dos clientes está a camada de aplicação, que proporciona o acesso ao Espaço de Tuplas replicado, enquanto no topo da pilha do lado dos servidores está a implementação do Espaço de Tuplas (Espaço de Tuplas local do servidor).

Na Figura 3.2 estão representadas as pilhas dos clientes e dos servidores, com as respectivas camadas. A comunicação segue um esquema similar aos de chamadas de procedimento remoto. Uma aplicação dos clientes interage com o sistema chamando funções ou métodos com as assinaturas das operações usuais de um Espaço de Tuplas: $out(t)$, $rd(\bar{t})$, $in(\bar{t})$.

Estes métodos podem ser chamados na própria camada de aplicação. A camada abaixo gerencia o controle de acesso às tuplas. Em seguida, existe a camada encarregada da confidencialidade e por último a camada de replicação.

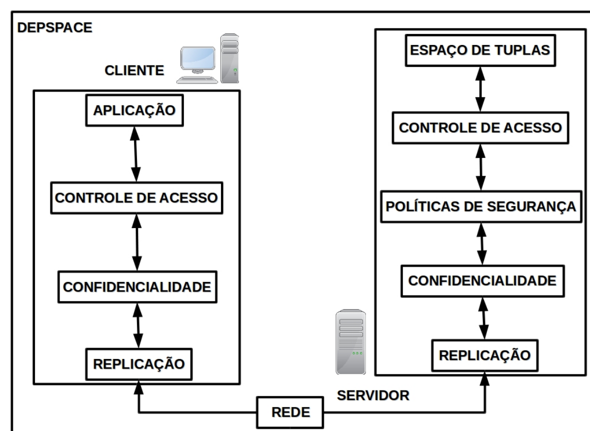

Figura 3.2: Arquitetura do DepSpace. Adaptado de [8].

O lado dos servidores é similar ao dos clientes, mas com o acréscimo de uma nova camada para verificar uma política de acesso para cada operação requisitada.

Vale comentar que nem todas as camadas precisam ser utilizadas ao mesmo tempo em todas as configurações de Espaços de Tuplas. A ideia é de que camadas podem ser removidas ou adicionadas de acordo com a qualidade de serviço desejada para o Espaço de Tuplas.

Camada de Replicação

Um dos mecanismos mais essenciais utilizados no DepSpace é a replicação: o Espaço de Tuplas é mantido em um conjunto de n servidores de maneira que se até f deles falharem não se prejudique a confiabilidade, a disponibilidade e a integridade do sistema.

A ideia é de que se alguns servidores falharem, o Espaço de Tuplas ainda estará disponível (disponibilidade), e as operações funcionarão de maneira correta (confiabilidade e integridade) por que as réplicas corretas conseguem superar (mascarar) à má conduta das faltosas.

Uma abordagem simples para replicação é a replicação por máquina de estados. Esta abordagem garante linearizabilidade, que é uma forma poderosa de consistência na qual todas as réplicas apresentam a mesma sequência de estados. A abordagem por máquina de estados necessita de que todas as réplicas:

1. Iniciem com o mesmo estado.

2. Executem todas as requisições na mesma ordem.

3. Cheguem ao mesmo estado final.

O primeiro item é assegurado iniciando o Espaço de Tuplas sem nenhuma tupla, e o segundo item requer um protocolo *multicast* de ordem total com tolerância a falhas. Já o terceiro item é assegurado pelo fato das operações executadas no Espaço de Tuplas serem determinísticas, isto é, a mesma operação executada no mesmo estado inicial gera o mesmo estado final em todas as réplicas. Isso quer dizer que a leitura (ou remoção) em servidores diferentes, mas com o mesmo estado deve retornar a mesma resposta.

O protocolo para a replicação ocorre da seguinte forma: o cliente envia uma requisição de uma operação utilizando um *multicast* de ordem total e aguarda por $f + 1$ respostas, estas iguais, mas vindas de diferentes servidores.

Como cada servidor recebe o mesmo conjunto de mensagens na mesma ordem, e o Espaço de Tuplas é determinístico, existirá sempre pelo menos uma quantidade $n - f \geq 2f + 1$ de servidores que executam a operação e possuem o mesmo retorno.

No DepSpace são necessários $3f + 1$ servidores para que até f servidores sejam não confiáveis. Dada a existência de 1 servidor não confiável são necessários pelo menos 3 servidores corretos, como pode ser observado na Figura 3.3:

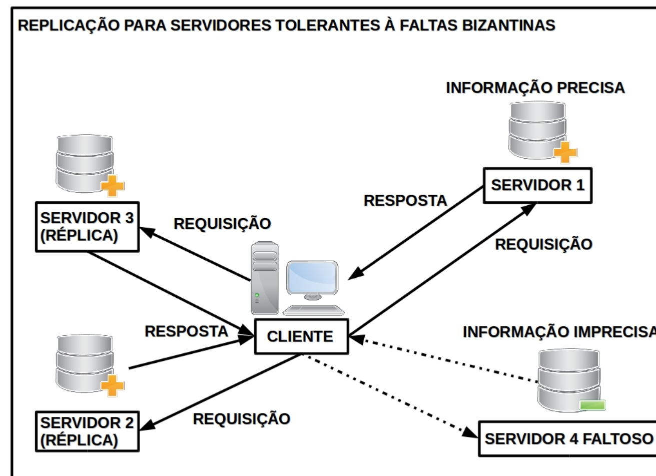

Figura 3.3: Replicação tolerante à Faltas Bizantinas. Adaptado de [20].

Camada de Confidencialidade

A replicação é frequentemente vista não como um auxílio mas como um impedimento para a confidencialidade. A razão para isso vem de: se uma informação secreta ou confidencial é armazenada não apenas em um, mas em vários servidores, fica mais fácil para um intruso consegui-la, não mais difícil.

Uma solução que necessita de um compartilhamento de chaves entre clientes contradiz a propriedade de anonimato do modelo do Espaço de Tuplas, que declara que processos comunicantes não precisam conhecer uns aos outros. Como são assumidas falhas bizantinas, servidores individualmente não podem acessar as tuplas, a solução deve então recair sobre um conjunto de servidores.

A ferramenta utilizada para implementar confidencialidade no DepSpace é um tipo especial de esquema de compartilhamento secreto, baseado no *PVSS*. Em um esquema de compartilhamento secreto, um grupo denominado de negociadores distribui um segredo para n jogadores, mas cada jogador recebe apenas uma parte do segredo. São exigidos que: pelo menos a quantidade $f + 1 \leq n$ de partes diferentes de um segredo para recuperá-lo e nenhuma informação é revelada para f ou menos partes.

Cada servidor i possui uma chave privada x_i e uma chave pública y_i, e os clientes conhecem as chaves públicas de todos os servidores. Os clientes fazem o papel de negociadores, criptografando/codificando a tupla com as chaves públicas de cada servidor e obtendo um conjunto de *shares* (da função *share* do *PVSS*).

Uma tupla pode ser decodificada com $f + 1$ *shares* (utilizando a função *combine*), portanto uma conspiração de servidores maliciosos não podem revelar o conteúdo dos campos de uma tupla confidencial (assumindo que no máximo f servidores são faltosos).

O *PVSS* também fornece funções de verificação, uma para cada servidor verificar a parte que recebeu do negociador/cliente (*verifyD*) e outra para o cliente verificar se a parte coletada a partir do servidor não foi corrompida (*verifyS*).

O esquema de confidencialidade tem de gerenciar o problema de combinar tuplas codificadas com os *templates*. Quando um cliente insere uma tupla no Espaço de Tuplas, é escolhido um dos três tipos seguintes de proteção para cada campo da tupla:

1. Público: o campo não é codificado, assim pode ser comparado arbitrariamente, mas seu conteúdo pode ser revelado se o servidor for faltoso.

2. Comparável: o campo f_i é codificado, mas um *hash* criptográfico do campo obtido com a função $H(f_i)$ também é guardado.

3. Privado: o campo é codificado e nenhum *hash* é guardado, dessa maneira comparações não são possíveis.

O tipo de proteção para cada campo de uma tupla é definido num vetor de proteção de tipo. Dada uma tupla *t*, seu vetor de proteção v_t é uma sequência de tipos protegidos, um para cada campo de *t*.

Os valores para os campos do vetor de proteção são *PU*, *CO* e *PR*, indicando se o correspondente campo da tupla é público, comparável, ou privado, respectivamente. Por exemplo, se uma tupla t = *<408km/h, 2600R$>* possui um v_t = *<PU, PR>* isto indica que o primeiro campo de t é público e o segundo é privado.

Campos comparáveis permitem corresponder tuplas com *templates* sem revelar o conteúdo dos campos. Se um cliente c_1 deseja inserir t no *TS* (Espaço de Tuplas) com um único campo comparável f_1. O cliente c_1 envia t codificada e $H(f_1)$ para os servidores.

Passado algum tempo, um cliente c_2 faz uma requisição $rd(\bar{t})$ e o *TS* precisa verificar se t e $\bar{t}$ combinam. O cliente c_2 calcula $H(\overline{f}_1)$ e o envia para o *TS* que verifica se esse *hash* é igual ao $H(f_1)$. O esquema funciona para igualdades, mas não é tão eficiente para comparações mais complexas.

O esquema de confidencialidade apresenta generalidade mesmo sendo construído em termos de tuplas e *TSs*. O presente esquema pode ser utilizado (com pequenas modificações) em qualquer sistema de armazenamento de dados com *BFT* baseado em conteúdo e construído sobre um protocolo de replicação *BFT*.

Camada de Políticas de Segurança

A ideia de uma camada de políticas de segurança é de que o Espaço de Tuplas é governado por uma política de acesso de granularidade fina. Essas políticas de acesso levam em conta três tipos de parâmetros para decidir se uma operação é aprovada ou negada:

1. *Id* de quem chamou a operação.

2. A operação e seus argumentos.

3. As tuplas presentes no *TS*.

Um Espaço de Tuplas deve possuir apenas uma política de acesso, que deve ser definida durante a configuração do sistema pelo seu administrador. Sempre que uma requisição de operação é recebida em um servidor, existe uma verificação se a operação satisfaz as políticas de acesso ao Espaço na camada de políticas de segurança.

Servidores corretos verificam corretamente os acessos, enquanto servidores faltosos se comportam de maneira arbitrária. A verificação por si só é uma simples avaliação de uma condição lógica expressada nas regras da operação chamada. Quando a operação é negada o servidor retorna um código de erro para quem chamou a operação. O cliente aceita a rejeição se e só se receber $f + 1$ cópias do mesmo código de erro.

Camada de Controle de Acesso

Controle de acesso é um mecanismo de segurança fundamental para *TSs*. O melhor modelo de controle de acesso depende do tipo de aplicação. Por exemplo, listas de controle de acesso ($ACLs$) podem ser utilizadas em sistemas fechados, mas alguns tipos de controle de acesso baseado em cargos são mais apropriados para sistemas abertos.

Para dar espaço a diferentes mecanismos juntamente com uma arquitetura confiável, os mecanismos de controle de acesso são definidos em termos de credenciais: cada *TS* possui um conjunto C^{TS} de credenciais necessárias e cada tupla t possui dois conjuntos ${C^{\mathrm{t}}}_{\mathrm{rd}}$ e ${C^{\mathrm{t}}}_{\mathrm{in}}$ de credenciais necessárias.

Para inserir uma tupla no *TS*, um cliente deve fornecer credenciais que combinem com C^{TS}. Analogamente, para ler ou remover t do *TS*, um cliente deve fornecer credenciais que combinem com ${C^{\mathrm{t}}}_{\mathrm{rd}}$ ou ${C^{\mathrm{t}}}_{\mathrm{in}}$ respectivamente. As credenciais necessárias para inserir a tupla no *TS* são definidas pelo administrador que configura os Espaços de Tuplas.

No lado do cliente as credenciais associadas são anexadas às tuplas, enquanto no lado dos servidores é verificado se as operações podem ser executadas. Se a operação é de inserção, as credenciais do cliente anexadas à tupla devem ser suficientes para inserir t no espaço. Se as operações requisitadas correspondem à leituras ou retiradas, as credenciais associadas com o *template* $\bar{t}$ passado devem ser suficientes para executá-las.

É importante que os servidores verifiquem os acessos e forneçam ao cliente um serviço confiável, ou seja, o mesmo receba tuplas confiáveis como retorno, como visto na Figura 3.4:

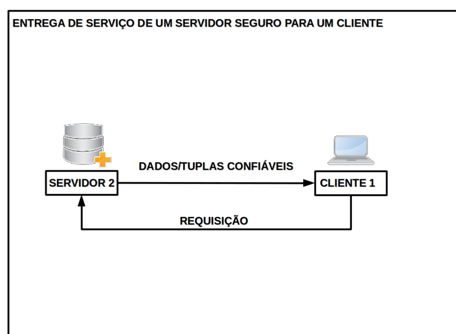

Figura 3.4: Tuplas confiáveis como retorno.

3.2.2 WSDS

A infraestrutura do WSDS (WS-DependableSpace), um Espaço de Tuplas para serviços *web*, é composta pelo DepSpace adicionada de uma "casca" para serviços *web* cooperativos, em que é empregado um serviço de coordenação confiável para compartilhamento de dados e sincronização de ações.

Os principais componentes introduzidos são *gateways* de serviços *web* que funcionam como uma ponte entre os clientes e um *TS* confiável (o DepSpace). Antes de acessar um *TS*, o cliente deve encontrar o endereço de um ou mais *gateways* que provêm o serviço desejado.

O cliente então faz uma requisição a um dos *gateways*, que por sua vez encaminham-na para o DepSpace (*TS*). Mecanismos de segurança são adicionados no sistema para impedir que *gateways* maliciosos quebrem alguma propriedade do sistema.

A Figura 3.5 mostra uma representação da arquitetura do WSDS, em que clientes se comunicam inicialmente com *gateways*, e estes com servidores do DepSpace.

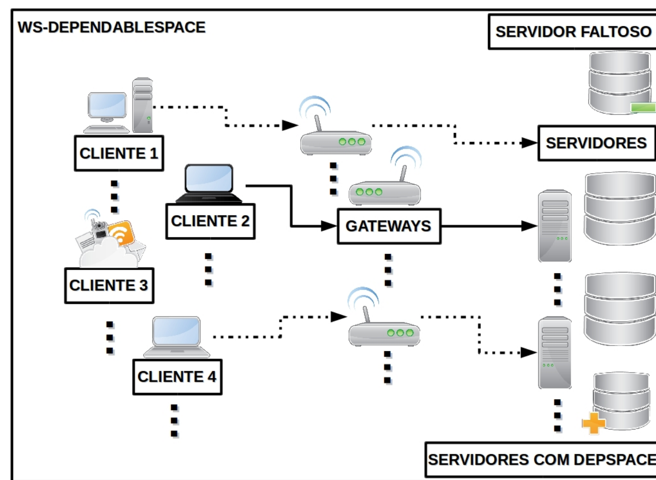

Figura 3.5: Arquitetura do WSDS. Adaptado de [1].

Capítulo 4

Active Space

O quarto capítulo descreve a nossa proposta de funcionamento de um Espaço de Tuplas. Dada a capacidade de suportar a execução de consultas no Espaço de Tuplas, configurou-se como um Espaço de Tuplas ativo, e assim denominado de Active Space.

A solução mais simples para implementar um Espaço de Tuplas ocorre quando existe um servidor central, devido a sua unicidade ele será sempre o destino quando ocorre a busca.

Um servidor central possuiria vantagens no caso de uma busca, que não é facilmente implementada de uma maneira distribuída, e em primitivas de sincronização simples de serem implementadas como no caso de um processo que fica bloqueado até que apareça uma tupla que corresponda ao que ele procura.

O já citado TSpaces [23] e também o JavaSpaces [34] são majoritariamente baseados em servidores centrais. Mas centralizar tudo em um só servidor pode muito bem trazer problemas, um defeito ocorrido nele comprometeria todo o sistema, e poderia ocorrer um problema de *starvation* no caso de operações monopolizadoras de alguns processos ou que levassem muito tempo para serem executadas.

Uma solução alternativa seria disseminar de imediato instâncias das tuplas através de vários Espaços Lógicos, em vários servidores ou máquinas espalhadas. Uma atenção com relação a segurança do sistema pode aumentar, mas com relação à vulnerabilidade, não se ficaria suscetível a um único ponto de falha, ou seja, um único servidor como ponto vulnerável.

Uma implementação distribuída de um sistema que suporte comunicação ou uma comunicação futura, mesmo depois que um processo que utilizou o *TS* deixa de existir, possui dois pontos fundamentais:

- Como realizar buscas, lidar com buscas maciças através de inúmeros *TSs* e servidores.
- Como distribuir as réplicas das tuplas através dessas várias máquinas e localizá-las posteriormente.

Para muitas aplicações, que serão descritas mais adiante, é necessário ler uma quantidade muito grande de dados para processar determinada operação. Várias cópias destes dados precisam trafegar pela rede, dos servidores para os clientes, consumindo recursos de rede e ainda diminuindo o desempenho do sistema.

Desta forma, a ideia fundamental da nossa proposta é executar consultas complexas (*queries*) no Espaço de Tuplas (lado do servidor), tornando o Espaço de Tuplas ativo e eliminando a necessidade de trafegar uma grande quantidade de dados pela rede.

4.1 Modelo de Sistema

O modelo de sistema do Active Space proposto se baseia inteiramente no modelo de sistema do DepSpace. Em virtude da entrega de serviços similares aos encontrados na *web*, como por exemplo, de buscas num site de viagens possuírem limitações, o Active Space têm o intuito de flexibilizar as buscas fornecendo aos usuários alternativas.

Um número ilimitado de clientes com acesso a uma rede composta de *gateways* e servidores implementando o DepSpace interagiriam de forma integrada e coordenada.

Através de um canal seguro as requisições das operações dos clientes respeitaria um limite de tempo para as buscas ou consultas, com o pressuposto de que a *query* não pode monopolizar o sistema, ser muito lenta, ou comprometer outras operações.

Quando for executada uma *query* pelo cliente sobre um conjunto de tuplas no *TS* a finalidade maior é garantir flexibilidade, melhoria do desempenho, bem como uma entrega de serviço confiável.

4.2 Arquitetura do Active Space

A flexibilidade em nível de arquitetura pode ser alcançada se o *TS* for customizado de acordo com o perfil dos clientes, e a entrega de serviço seguro com um sistema que suporte tolerância a falhas e respeite os atributos de dependabilidade, como o DepSpace.

O serviço oferecido pelo DepSpace possui uma característica importante, que é o suporte a múltiplos Espaços de Tuplas Lógicos [18], isto é, o sistema fornece interfaces de administração que permitem a criação de diferentes *TSs* e estes sem relação necessária uns com os outros.

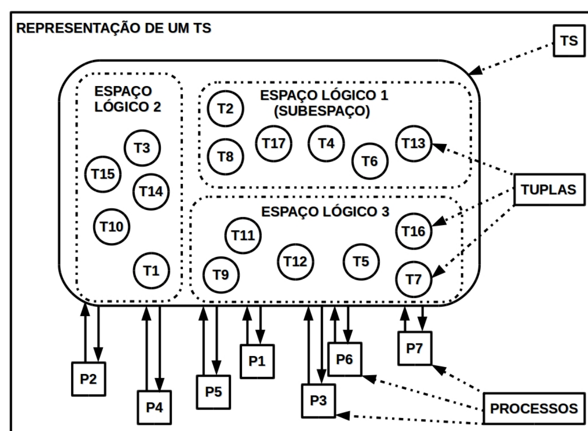

Figura 4.1: Representação de um TS. Adaptado de [32].

Uma representação de um *TS* completo composto por Espaços Lógicos (Subespaços), tuplas e processos pode ser vista na Figura 4.1.

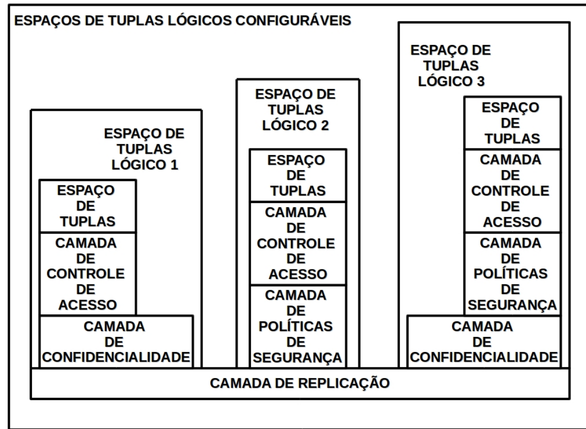

Figura 4.2: Espaços Lógicos configuráveis. Adaptado de [7].

O DepSpace pode então ser configurado de acordo com as necessidades das aplicações, ou seja, pode-se escolher quais as camadas que estarão ativas num determinado *TS*, como mostrado na Figura 4.2. Fazendo uso deste aspecto chave o Active Space foi configurado com a adição da camada de *query*, vide a Figura 4.3.

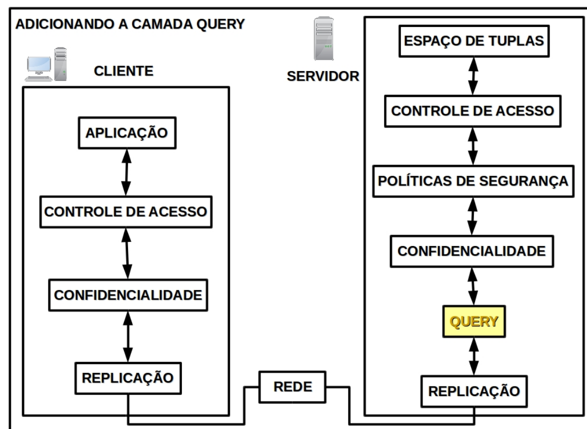

Figura 4.3: Adicionando a camada de Query ao DepSpace. Adaptado de [8].

Quando uma operação de escrita ocorre, uma réplica da tupla é enviada para o subespaço apropriado em cada uma das máquinas. Essa arquitetura é bem direta, mas teria de ser observada com cuidado a questão da escalabilidade (a medida que o sistema cresça juntamente com o número de réplicas das tuplas e com o tamanho da rede, o próprio continue fornecendo um serviço correto e dentro das suas especificações). O sistema também continua fornecendo *interfaces* para as operações de leitura e remoção. Porém, a camada de *Query* estende o *TS* de forma a permitir que consultas complexas sejam realizadas.

4.2.1 Análise sobre a Segurança

A segurança em sistemas baseados em coordenação traz à tona uma questão: por um lado processos devem ser desacoplados em espaço, por outro deve ser garantida a integridade e a confidencialidade das tuplas.

A segurança normalmente é implementada por meio de um canal seguro, que necessita que os processos sejam autenticados uns pelos outros, o único problema é que essa autenticação viola o desacoplamento de espaço (em que os processos não precisam conhecer os outros).

Uma solução para essa questão é que pode ser feito um reconhecimento do processo que deseja executar no *TS*, ou ser verificada uma inspeção do dado, da tupla em si. Para satisfazer esses dois casos e contornar problemas relativos à segurança pode-se proceder da seguinte maneira:

1. Clientes que desejam escrever as tuplas no *TS* devem ser confiáveis, caso de empresas divulgando seus serviços, e que não teriam razões para exibir dados comprometidos.

2. Outros clientes, usuários que estejam interessados em ler tuplas do *TS* devem estar restritos à apenas operações de leitura (*read-only operations*).

Não poderiam ser inseridas tuplas ou dados corrompidos de forma a não modificar o estado dos servidores, das réplicas, e do sistema como um todo. Vale ressaltar também que depois que as tuplas são inseridas, estas não podem ser mais alteradas ou modificadas.

As operações dos clientes no *TS* devem ter um prazo máximo, com um limite de tempo previamente estipulado. Esta questão é bastante relevante no nosso modelo, pois introduzimos a execução de consultas (*queries*) no *TS*. Por exemplo, um cliente mal intencionado poderia enviar uma consulta que nunca termina e "bloquear" o *TS* ou "vários" clientes executarem um ataque *DoS*. Em vista disso, estas execuções devem ser monitoradas e caso um tempo pré-definido seja ultrapassado a execução deve ser interrompida.

Um outro problema com a execução de consultas nos servidores é a possibilidade de um código mal intencionado tentar acessar outras partes do sistema. Em vista disso, pode ser utilizado um *Wrapper* [29] ou envoltório para se empacotar a *query*, fazendo com que fique imutável e não possa ser corrompida. Um acesso total a recursos vitais dos sistema não é desejável, já que, por exemplo, podem ser formatados dados essenciais dos servidores. Para evitar situações como esta, pode ser utilizado um *Security Manager* [28], que será responsável por determinar o não acesso a determinados recursos.

4.3 Metodologia

O caminho trilhado para se desenvolver uma camada de *query* integrada ao *middleware* DepSpace [8] se baseou em conceitos de Engenharia de *Software*. Os conceitos utilizados se originam da programação orientada a objetos, mas especificamente encontrada na linguagem Java [30]. Nesse modelo de orientação a objetos faz-se uso de objetos, classes, herança e interfaces em um pacote, organizado e facilmente gerenciável, que pretendem modelar e relacionar atividades do mundo real.

4.3.1 Implementação

Todas as implementações foram construídas em linguagem Java e adicionadas ao DepSpace.

Material Utilizado

- O sistema operacional utilizado foi o Ubuntu 14.04 LTS.
- O notebook utilizado foi um HP Pavilion g4.
- O ambiente de desenvolvimento foi o NetBeans.

Os códigos fonte estão disponíveis para download nos sites:

- https://github.com/DarthIt0/active-space
- https://drive.google.com/file/d/0BzkB6RLVn1dK0HhvWTNTX0pqYXM/view?usp=sharing

4.3.2 Primitivas de Coordenação

O acesso ao *TS* é realizado através do uso de um conjunto básico de operações de inserção, remoção e leitura, por meio das primitivas *out*, *in* e *rd*. A assinatura dos métodos em Java das operações podem ser vistas no código abaixo:

```
/**
 *  public void out(DepTuple tuple, Context ctx) throws DepSpaceException
 *
 *  public DepTuple rd(DepTuple template, Context ctx) throws DepSpaceException
 *
 *  public DepTuple rdp(DepTuple template, Context ctx) throws DepSpaceException
 *
 *  public DepTuple in(DepTuple template, Context ctx) throws DepSpaceException
 *
 *  public DepTuple inp(DepTuple template, Context ctx) throws DepSpaceException
 *
 */
```

Uma nova primitiva chamada *executeQuery* foi criada para a execução de consultas. Trata-se de uma operação de consulta empacotada dentro de uma tupla. A assinatura deste novo método adicionado, que possui como retorno uma *DepTuple*, ou seja, uma tupla, pode ser observada no seguinte código:

```
/**
 *  public DepTuple executeQuery(DepTuple tuple) throws DepSpaceException
 *
 */
```

Método para leitura de todas as tuplas

Por um lado um conjunto restrito de operações pode trazer benefícios no caso de algumas aplicações, com uma restrição na quantidade de operações. Por outro lado esse conjunto simplificado também faz com que as primitivas paguem por essa simplicidade.

Uma operação de $out(t)$ é benéfica e bem direta, tendo apenas que inserir a tupla t no *TS*. A operação in($\bar{t}$) lê e remove apenas uma tupla que se corresponda com o *template* $\bar{t}$, mas não funcionaria efetivamente no caso de uma remoção de mais de uma tupla, ou de um conjunto de tuplas que não são mais necessárias. Como no caso de deixar um *TS* sem tuplas, ou ilustrado no funcionamento de um *garbage collector* [24], implementado em Ligia [25].

A operação $rd(\bar{t})$ apenas lê uma tupla que se corresponda com o *template* $\bar{t}$, o que não é vantajoso quando se possui um leque de tuplas. Pelo fato de comparações entre uma tupla e um *template* retornarem apenas uma tupla, é necessária a criação de uma operação que, dado um *template*, leia e retorne um conjunto de tuplas correspondentes à ele. A operação *rdpAll*, que lê e retorna uma lista de tuplas, foi criada com essa finalidade. No entanto, para não modificar o conjunto de primitivas definidas na linguagem Linda [17] e, principalmente, não fornecer uma operação que faça com que uma grande quantidade de dados trafeguem pela rede, esta operação está disponível apenas no lado do servidor; para que a camada de *Query* possa ler os dados armazenados (tuplas), na execução de uma consulta.

Uma operação que funciona de forma similar é a *copy-collect(TS1, TS2, template)* [31], copiando todas as tuplas que combinem com o *template* de um *TS1* para um *TS2*, a exceção é que no caso queremos apenas ler uma coleção de tuplas e não passá-las de um *TS* para outro.

O método *rdpAll* pode ser observado abaixo:

```
/**
*
* public List<DepTuple> rdpAll(DepTuple template) throws DepSpaceException {
*
*       return rdpAll(template, defaultContext(-1,template));
*
* }
*
*/
```

4.3.3 Query

A despeito da simplicidade das primitivas, estas são extensíveis, podem ser modificadas, no sentido de que novas operações podem ser criadas a partir das mesmas.

O modelo de Linda pode ser visto como "universal" devido a sua portabilidade através de diferentes linguagens [2]. Um programador pode adicionar novos construtores a uma linguagem escolhida estendendo o modelo de Linda. Devido à obstáculos e restrições em se recuperar tuplas do *TS* é necessário estender funcionalidades, presentes nas primitivas, e/ou criar um processamento de *query* distribuído [9].

O primeiro passo antes do desenvolvimento das aplicações utilizando o sistema de *query* (ou consultas) é criar uma *interface* e uma camada de *query*, mas clientes necessitam apenas de utilizar o tipo de referência *query* configurado pela *interface*.

Interface

Uma *interface* se carateriza por possuir um ou mais métodos não inicializados, com apenas suas assinaturas. A *interface query* possui apenas um método chamado *execute*.

Este método é público (pode ser acessado e visto por classes que implementem a *interface*) e possui como retorno um objeto *DepTuple*, ou seja, retorna uma tupla.

A ideia é que clientes criem suas próprias *queries* (objetos que estendam a *interface Query*) e enviem as mesmas para serem executadas pelos servidores. Nos servidores, o método *execute* é executado de modo que toda a consulta deve ser codificada neste método, cujo retorno será o resultado da operação.

O código da *interface* pode ser observado a seguir:

```
/**
*
* public interface Query extends Serializable {
*
*      public DepTuple execute(DepSpaceServer upperLayer, Context ctx);
*
* }
*
*/
```

Camada de query

Posteriormente, foi definida a camada de *query*, que é uma camada intermediária, chamada *QueryLayer*. Nela existe o método *executeQuery*, que possui como parâmetro uma tupla *t*, no qual é feito um *casting* (transformação) de *t* para o tipo *Query*, e em seguida executado o método *execute* (da *interface Query*).

Para evitar modificações mais profundas e manter a compatibilidade com as implementações do DepSpace, optamos por encapsular os objetos do tipo *Query* dentro de uma tupla (*DepTuple*).

Note que a implementação é muito simples, a *query* é executada e a tupla retornada é enviada ao cliente como resultado da execução da consulta.

```
/**
* public class QueryLayer  implements DepSpaceServer {
*
*  private DepSpaceServer upperLayer;
*
*  public QueryLayer(DepSpaceServer upperLayer) {
*      this.upperLayer = upperLayer;
*  }
*
*  public DepTuple executeQuery(DepTuple t, Context ctx) throws DepSpaceException {
*
*      Query q = (Query) t.getFields()[0];
*
*      return q.execute(upperLayer,ctx);
*
*  }
*
* }
*/
```

4.4 Aplicações

Nesta seção exemplificamos nossa proposta através da descrição e demonstração de algumas aplicações. As aplicações aqui descritas têm como cenário geral o setor de turismo, de viagens, incorporando empresas aéreas, agências de viagens, funcionários e clientes interessados nos serviços dessas áreas.

Serão descritas três aplicações, nelas a apresentação do código fonte seguirá a seguinte ordem:

1. A inserção das tuplas no *TS*, ou seja, como serão populados os dados no *TS*.
2. O funcionamento da *query*, quais são as características da busca.
3. A execução da *query* pelo cliente, e o resultado, o *output* retornado.

4.4.1 Exemplo 1: consulta do voo pelas escalas

No primeiro exemplo, um comissário de uma companha aérea deseja buscar dentre os voos disponíveis, aquele no qual ele irá trabalhar. O comissário de bordo deseja fazer um voo direto, ou seja, um voo sem escalas.

Não sendo possível encontrar um voo com essas características ele passa a considerar um voo com uma conexão e em último caso deseja ver se existe um voo em que entre a cidade de origem e a de destino existem duas conexões.

Antes da inserção das tuplas no *TS*, os servidores devem ser inicializados e o Espaço em que as tuplas ficarão contidas deve ser criado. Devem ser inicializados $3f + 1$ servidores, em que até f sejam faltosos. No cenário dos nossos exemplos utilizamos 4 servidores, que devem ser sempre inicializados. Fica-se subentendido para os próximos exemplos esse mesmo procedimento inicial:

Inicialização dos servidores:

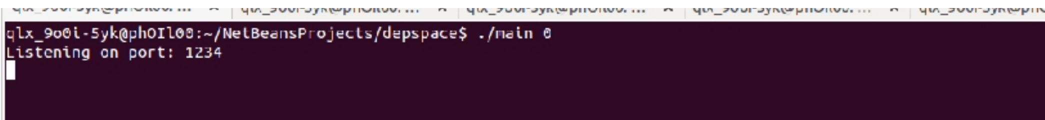

Figura 4.4: Inicialização dos Servidores.

Inserção das tuplas no *TS*

A classe *EscalasCidades* representa a inserção de todas as tuplas relativas aos dados dos voos e das cidades no Espaço de Tuplas. A ideia é que cada empresa aérea acesse o espaço para cadastrar seus voos. Inicialmente é criado o Espaço Lógico com o nome *"Viagens: Escalas"* e em seguida ele é populado com as tuplas.

Espaço Lógico criado:

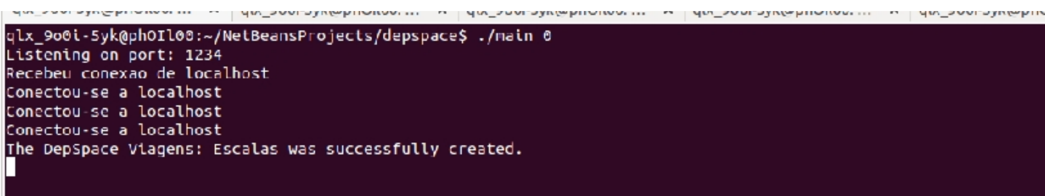

Figura 4.5: Exemplo 1. Espaço Lógico criado (Servidor).

As operações *out* estão acessando, uma por vez, o *TS* e inserindo as tuplas em que: o primeiro campo corresponde a cidade de origem, o último campo corresponde cidade de destino e os campos intermediários são as conexões.

Após a escrita das tuplas no Espaço de Tuplas a mensagem *"Foi populado o TS Logico: Escalas"* será impressa no *terminal.*

```
/**
* public class EscalasCidades {
```

```
*
*  public void run(){
*      try{
*
*          String name = "Viagens: Escalas";
*          Properties prop = new Properties();
*          prop.put(DPS_NAME, name);
*
*          DepSpaceAccessor accessor = new DepSpaceAdmin().createSpace(prop);
*
*          DepTuple t1 = DepTuple.createTuple("Manaus","Belem","Palmas");
*
*          accessor.out(t1);
*
*          DepTuple t2 = DepTuple.createTuple("Manaus","Brasilia","Palmas");
*
*          accessor.out(t2);
*
*          DepTuple t3 = DepTuple.createTuple("Manaus","Belem","Sao Luis","Teresina","Palmas");
*
*          accessor.out(t3);
*
*          DepTuple t4 = DepTuple.createTuple("Manaus","Belem","Teresina","Palmas");
*
*          accessor.out(t4);
*
*          DepTuple t5 = DepTuple.createTuple("Manaus","Palmas");
*
*          accessor.out(t5);
*
*          System.out.println("Foi populado o TS Logico: Escalas");
*
*      }catch(Exception e){
*          e.printStackTrace();
*      }
*  }
*
*  public static void main(String[] args){
*
*      new EscalasCidades().run();
*      System.exit(0);
*
*  }
*
* }
*/
```

Tuplas inseridas no Espaço Lógico:

```
qlx_9o0i-5yk@phOIl08:~/NetBeansProjects/depspace$ ./EscalasCidades
Conectou-se a localhost
Conectou-se a localhost
Conectou-se a localhost
Conectou-se a localhost
Foi populado o TS Logico: Escalas
qlx_9o0i-5yk@phOIl08:~/NetBeansProjects/depspace$
```

Figura 4.6: Exemplo 1. Tuplas inseridas no Espaço Lógico (Cliente).

Funcionamento da *query*

A classe *Escalas* representa a *query* do comissário e dessa maneira deve implementar a *interface Query*. O *template* ou molde do qual o comissário dispõe possui dois campos

inicialmente, representados por asteriscos, pois ele não possui conhecimento do nome das cidades. Em seguida, são criados *templates* com três e quatro campos referentes a um voo com uma ou duas escalas, respectivamente.

Por meio do método *rdp* se uma tupla combinar com algum destes *templates* esta será lida e retornada. Observe que primeiro a consulta tenta retornar um voo direto, seguido pela tentativa de um voo com uma escala e, por último, um com duas escalas.

Essa *query* consiste numa combinação de três métodos *rdp*, que retorna uma tupla com dois campos se ela existir, se não for possível retorna uma tupla com três campos e se ainda não obtiver êxito retorna uma outra com quatro campos.

A vantagem de utilizar essa *query* é não ter de aplicar um *rdp* após outro três vezes mesmo que em sucessivos acessos ao *TS*, e sim executar os três de uma só vez. Desta forma, não é preciso transportar todas as tuplas para o cliente.

```
/**
* public class Escalas implements Query {
*
*  public Escalas() {
*  }
*
*  public DepTuple execute(DepSpaceServer upperLayer, Context ctx) {
*      try{
*
*          DepTuple template = DepTuple.createTuple("*","*");
*          DepTuple ret  = upperLayer.rdp(template, ctx);
*          if(ret != null){
*              return ret;
*          }
*
*          template = DepTuple.createTuple("*","*","*");
*          ret  = upperLayer.rdp(template, ctx);
*          if(ret != null){
*              return ret;
*          }
*
*          template = DepTuple.createTuple("*","*","*","*");
*          ret  = upperLayer.rdp(template, ctx);
*          if(ret != null){
*              return ret;
*          }
*
*      }catch(Exception e){
*          e.printStackTrace();
*      }
*
*      return null;
*
*  }
*
* }
*/
```

Execução da *query*

A execução da *query* diz repeito à classe *Comissário*, nela obrigatoriamente deverá ser chamada uma referência à *Escalas*, isto é, a *query*. É importante notar que o espaço acessado na execução da *query* deve possuir o mesmo nome do que foi populado com as tuplas anteriormente, ou seja, *"Viagens: Escalas"*.

Uma instância do objeto *Query* foi criado a partir do construtor sem argumento do método *Escalas* e que possui como tipo de referência a *interface Query*.

Por último, foi chamada a consulta por meio de *executeQuery* e como resultado foi mostrada a tupla *t5*, *("Manaus", "Palmas")*, com apenas a cidade de origem e a cidade de destino. Um *output* com o retorno dessa consulta será mostrado num *terminal* após o código.

A única desvantagem de executar essa *query* é de que ela poderia retornar outra tupla que apresentasse apenas dois campos. Entretanto, a única informação de que o comissário dispunha era sobre as escalas e suas preferências, não era sobre os nomes das cidades.

```
/**
* public class Comissario {
*
*  public void run(){
*      try{
*
*          String name = "Viagens: Escalas";
*          Properties prop = new Properties();
*          prop.put(DPS_NAME, name);
*
*          DepSpaceAccessor accessor = new DepSpaceAdmin().createSpace(prop);
*
*          Query q = new Escalas();
*
*          DepTuple ret = accessor.executeQuery(q);
*
*          System.out.println("Retorno: "+ret);
*
*          System.out.println("Foi executada a consulta");
*
*      } catch (Exception e){
*          e.printStackTrace();
*      }
*
*  }
*
*  public static void main(String[] args){
*
*      new Comissario().run();
*      System.exit(0);
*
*  }
*
* }
*/
```

Output:

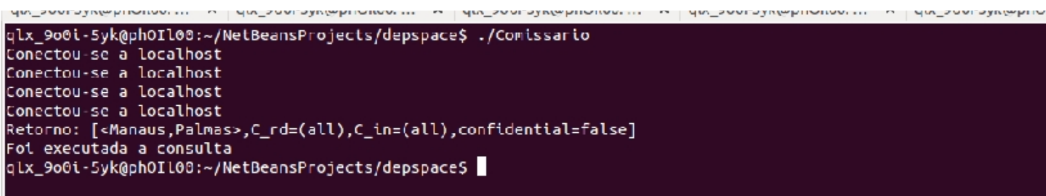

Figura 4.7: Exemplo 1. Output (Cliente).

4.4.2 Exemplo 2: busca através dos dias de embarque

No segundo exemplo, um cliente residente em Manaus deseja encontrar um voo a partir de três dias que serão escolhidos por ele, inicialmente ele apenas dispõe da sua cidade de

origem e da cidade de destino que será Palmas. O cliente deseja fazer sua viagem em qualquer um desses três dias: 18, 25 ou 30 de Outubro.

Inserção das tuplas no *TS*

Inicialmente foi criado o Espaço com o nome *"Viagens: Dias de embarque"*.

Um funcionário de uma empresa aérea inseriu as tuplas de voos com suas informações populando o *TS* e consequentemente o servidor. A ideia é que cada empresa aérea popule o espaço com seus voos.

Os campos das tuplas contêm: cidade de origem, cidade de destino, dia, horário, preço do voo e também a empresa aérea.

Espaço Lógico criado:

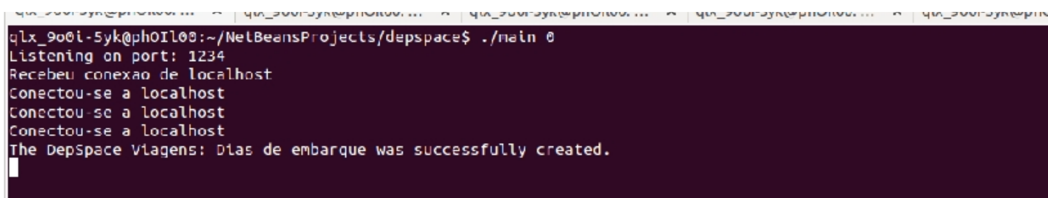

Figura 4.8: Exemplo 2. Espaço Lógico criado (Servidor).

A classe *Companhias* representa a inserção das tuplas no *TS*. Para inserção das tuplas foi criado um vetor de tuplas com dez posições, em seguida, para cada posição foi criada uma tupla com seus repectivos valores. A dezena de tuplas foi inserida no espaço através de um laço de repetição e por meio do método *out*.

Após estas operações é mostrada a mensagem *"Foi populado o TS Logico: Dias de embarque"*.

```
/**
* public class Companhias {
*
*  public void run(){
*      try{
*
*          String name = "Viagens: Dias de embarque";
*          Properties prop = new Properties();
*          prop.put(DPS_NAME, name);
*
*          DepSpaceAccessor accessor2 = new DepSpaceAdmin().createSpace(prop);
*
*          DepTuple[] v;
*
*          v = new DepTuple[10];
*
*          v[0] = DepTuple.createTuple("Manaus","Palmas","10/10/14","03:00","R$ 1.310,00","TAP");
*
*          v[1] = DepTuple.createTuple("Manaus","Palmas","02/10/14","06:30","R$ 1.400,00","PLUNA");
*
*          v[2] = DepTuple.createTuple("Manaus","Palmas","11/10/14","23:45","R$ 1.500,00","PASSAREDO");
*
*          v[3] = DepTuple.createTuple("Manaus","Palmas","23/10/14","01:15","R$ 1.410,00","IBERIA");
*
*          v[4] = DepTuple.createTuple("Manaus","Palmas","30/10/14","13:30","R$ 1.340,00","GOL");
*
*          v[5] = DepTuple.createTuple("Manaus","Palmas","09/10/14","15:00","R$ 1.450,00","TAM");
```

```
*
*           v[6] = DepTuple.createTuple("Manaus","Palmas","18/10/14","12:45","R$ 1.309,00","AVIANCA");
*
*           v[7] = DepTuple.createTuple("Manaus","Palmas","17/10/14","21:00","R$ 1.394,00","WEBJET");
*
*           v[8] = DepTuple.createTuple("Manaus","Palmas","06/10/14","9:30","R$ 1.335,00","AZUL");
*
*           v[9] = DepTuple.createTuple("Manaus","Palmas","25/10/14","18:30","R$ 1.399,00","TRIP");
*
*           int j;
*
*           for(j=0; j<10; j++){
*               accessor2.out(v[j]);
*           }
*
*           System.out.println("Foi populado o TS Logico: Dias de embarque");
*
*       } catch (Exception e){
*           e.printStackTrace();
*       }
*
*   }
*
*   public static void main(String[] args){
*
*       new Companhias().run();
*       System.exit(0);
*
*   }
*
* }
*/
```

Tuplas inseridas no Espaço Lógico:

```
qlx_9o0i-5yk@phOIl08:~/NetBeansProjects/depspace$ ./Companhias
Conectou-se a localhost
Conectou-se a localhost
Conectou-se a localhost
Conectou-se a localhost
Foi populado o TS Logico: Dias de embarque
qlx_9o0i-5yk@phOIl08:~/NetBeansProjects/depspace$
```

Figura 4.9: Exemplo 2. Tuplas inseridas no Espaço Lógico (Cliente).

Funcionamento da *query*

Na *query "DiasEmbarque"* é criado um *template* com os dois primeiros campos fixos relativos às cidades de origem e de destino, no caso do cliente em questão, Manaus e Palmas (ver Execução da *query*, mais adiante).

Fazendo uso do método *rdpAll* são lidas e retornadas uma lista com todas as tuplas que se correspondam com o *template* citado. Primeiramente, é verificada a existência de tuplas no espaço, caso o *TS* esteja vazio retorna *"null"* (nulo ou sem valor).

É criado subsequentemente um comparador. Dentro de um laço *for* é feita uma comparação entre o terceiro campo (correspondente ao dia de embarque) de cada uma das tuplas da lista. Se o valor do campo e o de algum dos dias escolhidos (18, 25 ou 30 de Outubro) forem iguais, o "ret" (retorno) do tipo tupla recebe a tupla e sai do laço.

A diferença dessa *query* para o exemplo derradeiro é de que naquele eram feitas comparações por *quantidade* de campos e o cliente não possuia muitas opções de escolha.

Não era possível o cliente valorar os campos, já neste outro exemplo o cliente pôde ter a flexibilidade de escolher seu dia de embarque.

Neste exemplo a quantidade de campos é fixa, mas utilização da operação *rdpAll* faz com que não seja necessária a realização de 10 operações *rdp* (com uma leitura para cada tupla que combine com o *template*).

Fica evidenciado que a quantidade de acessos a um *TS* ativo por meio de uma *query* é bem menor do que a um Espaço de Tuplas em que se faz uso apenas das primitivas básicas.

```
/**
* public class DiasEmbarque implements Query{
*
*  private String to;
*  private String from;
*
*  public DiasEmbarque(String to, String from){
*      this.to = to;
*      this.from = from;
*  }
*
*  public DepTuple execute(DepSpaceServer upperlayer, Context ctx){
*      try{
*
*          DepTuple template = DepTuple.createTuple(this.to,this.from,"*","*","*","*");
*
*          List<DepTuple> tuplas = upperlayer.rdpAll(template, ctx);
*
*          if(tuplas.isEmpty()){
*              return null;
*          }
*
*          DepTuple ret = tuplas.get(0);
*
*          Collator myCol = Collator.getInstance();
*
*          for(int j=0;j < tuplas.size();j++){
*              if((myCol.compare(tuplas.get(j).getFields()[2], "25/10/14") == 0)
*                  || (myCol.compare(tuplas.get(j).getFields()[2], "30/10/14") ==0 )||
*                          (myCol.compare(tuplas.get(j).getFields()[2], "18/10/14")==0)){
*
*                  ret = tuplas.get(j);
*                  break;
*              }
*          }
*
*          return ret;
*
*      } catch(Exception e){
*          e.printStackTrace();
*      }
*
*      return null;
*  }
*
* }
*/
```

Execução da *query*

A execução da *query* ocorre na classe *"Clientes"* em que é acessado o espaço lógico *"Viagens: Dias de embarque"*. Após o acesso ao *TS* Lógico, o cliente cria uma instância de *"DiasEmbarque"* com dois primeiros campos já predefinidos, a cidade em que reside e

a de destino. O envio da *query* ocorre na parte seguinte do código quando é chamado o método *executeQuery*.

O retorno desta consulta é a tupla *v[4]*, *("Manaus", "Palmas", "30/10/14", "13:30", "R$1.340,00", GOL)*, com o dia de partida igual a 30 de Outubro.

Este foi o dia escolhido ao invés dos outros dois como retorno, por ter sido inserida a tupla *v[4]* primeiro, consequentemente na lista de tuplas se encontrava antes de *v[6]*, *v[9]* que continham os outros dias.

```
/**
* public class Clientes {
*
*  public void run(){
*      try{
*
*          String name = "Viagens: Dias de embarque";
*          Properties prop = new Properties();
*          prop.put(DPS_NAME, name);
*
*          DepSpaceAccessor accessor2 = new DepSpaceAdmin().createSpace(prop);
*
*          Query q = new DiasEmbarque("Manaus","Palmas");
*
*          DepTuple ret = accessor2.executeQuery(q);
*
*          System.out.println("Retorno: "+ret);
*
*          System.out.println("Foi executada a consulta");
*
*      } catch (Exception e){
*          e.printStackTrace();
*      }
*
*  }
*
*  public static void main(String[] args){
*
*      new Clientes().run();
*      System.exit(0);
*
*  }
*
* }
*/
```

Output:

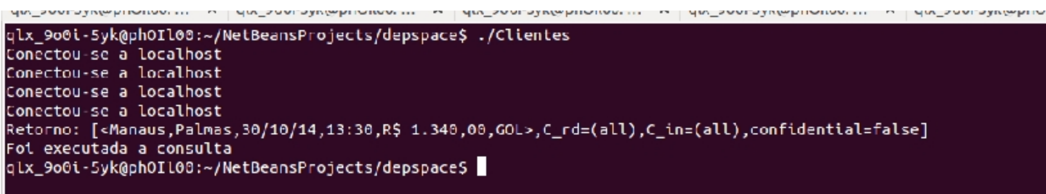

Figura 4.10: Exemplo 2. Output (Cliente).

4.4.3 Exemplo 3: procurando a passagem de menor valor

No terceiro exemplo, um cliente deseja viajar de Manaus até Palmas e seu intuito é encontrar um voo em que a passagem seja a de menor valor.

Inserção das tuplas no *TS*

Como nos exemplos anteriores, primeiramente deve ser criado o *TS*. Foi criado o *TS* *"Viagens: Passagens"*.

Espaço Lógico criado:

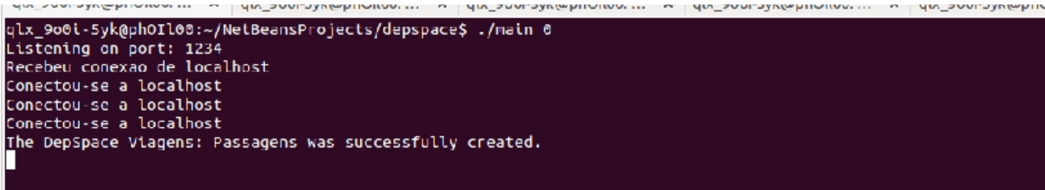

Figura 4.11: Exemplo 3. Espaço Lógico criado (Servidor).

A classe *EmpresaAerea* representa a ação de um funcionário de uma empresa aérea que acessou o *TS* e inseriu as informações do voo da sua companhia aérea. Da mesma forma procederam funcionários de outras companhias, e assim por diante preenchendo o Espaço de Tuplas com os dados.

Para simular isso é criado um vetor de tuplas com dez posições, depois através de sucessivos *"outs"*, as tuplas com os dados dos voos são inseridas no *TS* e em seguida é impressa uma mensagem com êxito.

```
/**
* public class EmpresaAerea {
*
*  public void run(){
*      try{
*
*          String name = "Viagens: Passagens";
*          Properties prop = new Properties();
*          prop.put(DPS_NAME, name);
*
*          DepSpaceAccessor accessor3 = new DepSpaceAdmin().createSpace(prop);
*
*          DepTuple[] v;
*
*          v = new DepTuple[10];
*
*          v[0] = DepTuple.createTuple("Manaus","Palmas","03/10/14","03:00","R$ 1.310,00","TAP");
*
*          v[1] = DepTuple.createTuple("Manaus","Palmas","03/10/14","06:30","R$ 1.400,00","PLUNA");
*
*          v[2] = DepTuple.createTuple("Manaus","Palmas","03/10/14","23:45","R$ 1.500,00","PASSAREDO");
*
*          v[3] = DepTuple.createTuple("Manaus","Palmas","03/10/14","01:15","R$ 1.410,00","IBERIA");
*
*          v[4] = DepTuple.createTuple("Manaus","Palmas","03/10/14","13:30","R$ 1.340,00","GOL");
*
*          v[5] = DepTuple.createTuple("Manaus","Palmas","03/10/14","15:00","R$ 1.450,00","TAM");
*
*          v[6] = DepTuple.createTuple("Manaus","Palmas","03/10/14","12:45","R$ 1.309,00","AVIANCA");
*
*          v[7] = DepTuple.createTuple("Manaus","Palmas","03/10/14","21:00","R$ 1.394,00","WEBJET");
*
*          v[8] = DepTuple.createTuple("Manaus","Palmas","03/10/14","9:30","R$ 1.335,00","AZUL");
*
*          v[9] = DepTuple.createTuple("Manaus","Palmas","03/10/14","18:30","R$ 1.399,00","TRIP");
*
*          int j;
```

```
*
*         for(j=0; j<10; j++){
*             accessor3.out(v[j]);
*         }
*
*         System.out.println("Foi populado o TS Logico: Passagens");
*
*     } catch (Exception e){
*         e.printStackTrace();
*     }
*
*  }
*
*  public static void main(String[] args){
*
*      new EmpresaAerea().run();
*      System.exit(0);
*
*  }
*
* }
*/
```

Tuplas inseridas no Espaço Lógico:

```
qlx_9o0i-Syk@phOIl08:~/NetBeansProjects/depspace$ ./EmpresaAerea
Conectou-se a localhost
Conectou-se a localhost
Conectou-se a localhost
Conectou-se a localhost
Foi populado o TS Logico: Passagens
qlx_9o0i-Syk@phOIl08:~/NetBeansProjects/depspace$
```

Figura 4.12: Exemplo 3. Tuplas inseridas no Espaço Lógico (Cliente).

Funcionamento da *query*

Na *query "PrecoPassagem"* são lidas todas as tuplas do Espaço de Tuplas, cuja origem e destino combinem com os dados especificados. Após a leitura, o quinto campo das tuplas que contém os preços, é comparado entre elas afim de se descobrir o menor valor. O funcionamento desta consulta é descrito a seguir.

A classe *PrecoPassagem* representa o funcionamento da *query* em que é criado um *template* com as cidades do cliente. A operação *rdpAll* lê todas as tuplas que combinem com o *template* e que serão retornadas numa lista. Em seguida há uma verificação se o *TS* está vazio, se estiver, retorna *"null"*, ou seja, as tuplas não foram inseridas, houve um erro de acesso, ou o *TS* pode não estar ativo. A tupla de "retorno" recebe a primeira tupla da lista e em seguida é criado um comparador.

Dentro de um laço *for* existe uma comparação entre o quinto campo (correspondente ao preço) da tupla de retorno e das tuplas da lista. Se o campo da tupla de retorno for maior do que o de alguma outra tupla na sequência da lista, a tupla de retorno passa a ser essa outra tupla. Em outras palavras, sempre que o valor do preço em uma tupla for menor, esta passa a ser a tupla de retorno.

Neste exemplo as únicas comparações realizadas com parâmetros passados do cliente são com as cidades de origem e de destino dele. Não ocorre nenhuma comparação com um valor específico explicitamente passado pelo cliente, pois o próprio simplesmente queria o

menor preço, mas ele poderia pedir uma passagem menor ou igual a 1.300,00 reais, por exemplo, similar ao Exemplo 2.

Seria muito custosa a utilização de muitos *rdp* combinados, e em alguns casos tendo de excluir algumas tuplas da seleção de um conjunto desejado, para satisfazer a configuração das preferências de um cliente como o deste exemplo.

Reforça-se o uso de *queries* num caso de busca complexo, em que primitivas básicas não são suficientes para suprir as necessidades dos clientes.

```
/**
* public class PrecoPassagem implements Query {
*
*  private String to;
*  private String from;
*
*  public PrecoPassagem(String to, String from){
*      this.to = to;
*      this.from = from;
*  }
*
*  public DepTuple execute(DepSpaceServer upperlayer, Context ctx){
*      try{
*
*          DepTuple template = DepTuple.createTuple(this.to,this.from,"*","*","*","*");
*
*          List<DepTuple> tuplas = upperlayer.rdpAll(template, ctx);
*
*          if(tuplas.isEmpty()){
*              return null;
*          }
*
*          DepTuple ret = tuplas.get(0);
*
*          Collator myCol = Collator.getInstance();
*
*          for(int j=1;j < tuplas.size();j++){
*              if(myCol.compare(ret.getFields()[4], tuplas.get(j).getFields()[4]) > 0){
*                  ret = tuplas.get(j);
*              }
*          }
*
*          return ret;
*
*      } catch(Exception e){
*          e.printStackTrace();
*      }
*
*      return null;
*  }
*
* }
*/
```

Execução da *query*

A classe *Cliente* representa um cliente realizando uma consulta. O cliente acessa o *TS* Lógico *"Viagens: Passagens"*, espaço este que pode compor um Espaço de Tuplas maior que abrigue tuplas para serviços correlatos.

É enviada a *query "PrecoPassagem"* com dois campos das cidades de origem e de destino já definidos. Depois do envio da *query* através do método *executeQuery* é realizada a consulta nos servidores.

Após a relização da consulta e feita a comparação, o resultado retornado é a tupla *v[6], ("Manaus", "Palmas", "03/10/14", "12:45", "R$1.309,00", "AVIANCA").*

Uma questão surge se por exemplo existissem duas ou mais tuplas de voos com preço igual a “R$1.309,00”, nestes casos, umas destas seria retornada por meio da *query*.

Esta questão podeira ser resolvida utilizando-se a operação *rdpAll* outra vez, e depois realizando uma outra comparação com, por exemplo, um horário de embarque diferente.

```
/**
* public class Cliente {
*
*  public void run(){
*      try{
*
*          String name = "Viagens: Passagens";
*          Properties prop = new Properties();
*          prop.put(DPS_NAME, name);
*
*          DepSpaceAccessor accessor3 = new DepSpaceAdmin().createSpace(prop);
*
*          Query q = new PrecoPassagem("Manaus","Palmas");
*
*          DepTuple ret = accessor3.executeQuery(q);
*
*          System.out.println("Retorno: "+ret);
*
*          System.out.println("Foi executada a consulta");
*
*      } catch (Exception e){
*          e.printStackTrace();
*      }
*
*  }
*
*  public static void main(String[] args){
*
*      new Cliente().run();
*      System.exit(0);
*
*  }
*
* }
*/
```

Output:

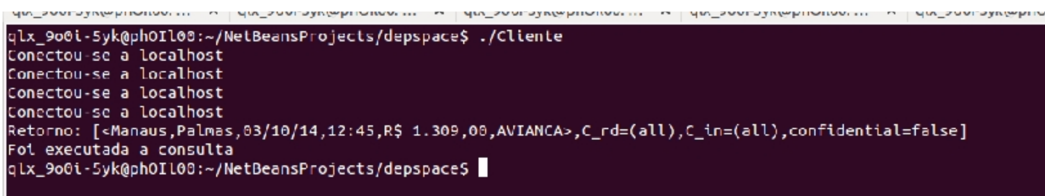

Figura 4.13: Exemplo 3. Output (Cliente).

4.4.4 Outras aplicações

Aplicações com Espaços de Tuplas estão sendo utilizadas para modelar problemas reais, simulando exemplos dos mais variados, como por exemplo:

- Sequenciamento de DNA [10].
- Sistemas de saúde [27].

- Serviço de vídeos por demanda [36].
- Compartilhamento de tuplas transitório em curtos períodos de tempo [12].
- Espaços de Tuplas para *web* [33].
- Utilização de Espaços de Tuplas comparados com banco de dados [16].
- Uso em *workspaces* interativos [21].
- Uso em *workflow* com modelo de eventos baseados em tuplas [14].

Exemplo de um serviço de TV ou de Filmes:

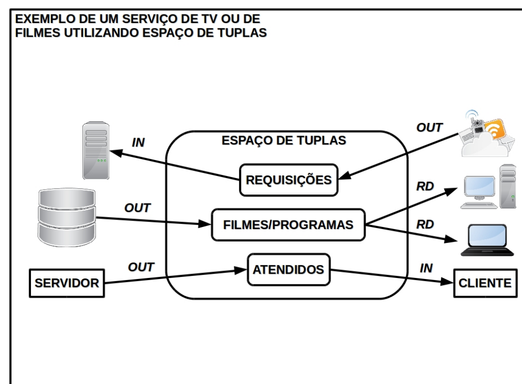

Figura 4.14: Exemplo de um serviço de TV ou de Filmes. Adaptado de [36].

Capítulo 5

Conclusão

Em um Espaço de Tuplas (*TS*) ativo é importante se alcançar segurança de funcionamento como forma de assegurar que o serviço seja entregue conforme sua especificação. Isso pode ser alcançado por meio da utilização da compensação e também sobre restrições acerca das tuplas presentes no *TS* e de quem possa acessá-las.

No *TS* ativo é também de suma importância que haja o suporte para utilização de novas funcionalidades, especialmente as que dizem respeito às buscas maciças. Na realização de buscas destes tipos através de vários servidores e *TSs* é importante visar o conteúdo das tuplas. E como podem estar localizadas, agrupadas ou distribuídas as tuplas nos diversos servidores que implementem o Espaço de Tuplas.

O Active Space possui vantagens sobre implementações de Espaço de Tuplas que recaem somente sobre as primitivas básicas de operação. Uma vantagem é que o cliente pode escolher por meio de uma *query* sobre um vasto conjunto de tuplas o que melhor lhe convém. Outra e mais importante vantagem é que ao serem processadas buscas complexas não é retornada uma grande quantidade de dados dos servidores para os clientes. Isso faz com que a rede não seja sobrecarregada mesmo quando muitas consultas são executadas por vários clientes, um problema recorrente no cenário dos sistemas distribuídos.

A implementação do sistema de *query* sobre o DepSpace, configurando o Active Space, vai de encontro com o objetivo geral da pesquisa. Como forma de sustentar e atender esse objetivo, revisamos os objetivos específicos, que foram alcançados da seguinte forma:

- Os conceitos de dependabilidade e de *middlewares* relevantes utilizando *TSs* foram descritos no Capítulo 2 e no Capítulo 3, respectivamente.
- O *middleware* DepSpace, um *TS* com segurança de funcionamento e uma extensão do próprio para ambientes *web*, o WSDW, foram ambos descritos no Capítulo 3.
- O cerne da proposta, ou seja, a implementação de um sistema de consulta sobre o DepSpace, foi alcançado integrando uma camada de execução de *query* nos servidores do próprio, descrito no Capítulo 4.
- No Capítulo 4, aplicações foram exemplificadas com a finalidade de analisar a nossa proposta, em que ficou evidente que *queries* se sobrepõem às primitivas básicas e predominantemente visam reduzir que uma grande quantidade de dados trafegue na rede.

Acerca de trabalhos futuros, pretendemos executar nossas aplicações, propostas para o Active Space, noutro ambiente de desenvolvimento e com um maior número de computadores e por meio de testes atestar os ganhos com a execução das consultas nos servidores.

Referências

[1] Eduardo Adilio Pelinson Alchieri, Alysson Neves Bessani, and Joni da Silva Fraga. A dependable infrastructure for cooperative web services coordination. In *Proceedings of the 2008 IEEE International Conference on Web Services*, ICWS '08, pages 21–28, Washington, DC, USA, 2008. IEEE Computer Society. vii, 19, 28

[2] Brian Anderson and Dennis Shasha. Persistent linda: Linda + transactions + query processing, 1991. 34

[3] Ken Arnold, Robert Scheifler, Jim Waldo, Bryan O'Sullivan, and Ann Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. 5

[4] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: a taxonomy. In *Building the Information Society*, pages 91–120. Springer, 2004. vii, 8, 9, 10

[5] Algirdas Avižienis, 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. vii, 7, 9, 12

[6] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. *Commun. ACM*, 37(8):76–82, August 1994. 5

[7] Alysson Neves Bessani, Eduardo Adilio Pelinson Alchieri, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Depspace: Um middleware para coordenação em ambientes dinâmicos e não confiáveis. *Salao de Ferramentas do XXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos, SBC*, 2007. vii, 31

[8] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: A byzantine fault-tolerant coordination service. *SIGOPS Oper. Syst. Rev.*, 42(4):163–176, April 2008. vii, 2, 24, 31, 32

[9] Duncan K.G. Campbell. Constraint matching retrieval in linda: extending retrieval functionality and distributing query processing, 1997. 34

[10] Nicholas Carriero and David Gelernter. Applications experience with linda. In *Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems*, PPEALS '88, pages 173–187, New York, NY, USA, 1988. ACM. 47

[11] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, April 1989. 1, 5

[12] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. Teenylime: Transiently shared tuple space middleware for wireless sensor networks. In *Proceedings of the International Workshop on Middleware for Sensor Networks*, MidSens '06, pages 43–48, New York, NY, USA, 2006. ACM. 48

[13] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, USA, 5th edition, 2011. 14, 16

[14] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, September 2001. 48

[15] Mário Dantas. *Computação distribuída de alto desempenho: redes, clusters e grids computacionais.* Axcel Books, 2005. 9, 10, 11, 13

[16] F. Fummi, G. Perbellini, R. Pietrangeli, and D. Quaglia. Interactive presentation: A middleware-centric design flow for networked embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1048–1053, San Jose, CA, USA, 2007. EDA Consortium. 48

[17] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985. 2, 18, 34

[18] David Gelernter. *Multiple tuple spaces in Linda.* Springer, 1989. 30

[19] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, February 1992. 19

[20] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, April 1997. vii, 25

[21] Brad Johanson and Armando Fox. Extending tuplespaces for coordination in interactive workspaces. *J. Syst. Softw.*, 69(3):243–266, January 2004. 48

[22] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. vii, 15, 16

[23] Tobin J Lehman, Alex Cozzi, Yuhong Xiong, Jonathan Gottschalk, Venu Vasudevan, Sean Landis, Pace Davis, Bruce Khavar, and Paul Bowman. Hitting the distributed computing sweet spot with tspaces. *Computer Networks*, 35(4):457–472, 2001. 19, 20, 29

[24] Ronaldo Menezes and Alan Wood. Garbage collection in open distributed tuple space systems. In *In Proc. 15 th Brazilian Computer Networks Symposium — SBRC'97*, pages 525–543, 1997. 33

[25] Ronaldo Menezes and Alan Wood. Ligia: A java based linda-like run-time system with garbage collection of tuple spaces. *REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS*, 1998. 33

[26] Amy L Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):279–328, 2006. 19, 21

[27] Elena Nardini, Andrea Omicini, Mirko Viroli, and Michael I. Schumacher. Coordinating e-health systems with tucson semantic tuple centres. *SIGAPP Appl. Comput. Rev.*, 11(2):43–53, March 2011. 47

[28] Java SE Oracle. The java technotes. *https://docs.oracle.com/javase/jp/8/technotes/guides/security/smPortGuide.html Last Accessed: December 3rd*, 2014. 32

[29] Java SE Oracle. The java tutorials. *https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html Last Accessed: December 1st*, 2014. 32

[30] Java SE Oracle. The java tutorials. *https://docs.oracle.com/javase/tutorial/java/concepts Last Accessed: December 14th*, 2014. 32

[31] Antony Rowstron and Alan Wood. Solving the linda multiple rd problem. In *COORDINATION LANGUAGES AND MODELS, PROCEEDINGS OF COORDINATION '96, VOLUME 1061 OF LECTURE NOTES IN COMPUTER SCIENCE*, pages 357–367. Springer-Verlag, 1996. 34

[32] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. vii, 30

[33] Robert Tolksdorf, Lyndon Nixon, and Elena Simperl. Towards a tuplespace-based middleware for the semantic web. *Web Intelli. and Agent Sys.*, 6(3):235–251, August 2008. 48

[34] Jim Waldo et al. Javaspaces specification 1.0. *Sun Microsystems*, 1998. 29

[35] A. Watson. Omg (object management group) architecture and corba (common object request broker architecture) specification. In *Distributed Object Management, IEE Colloquium on*, page 41, Jan 1994. 5

[36] George C. Wells. New and improved: Linda in java. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, PPPJ '04, pages 67–74. Trinity College Dublin, 2004. vii, 48