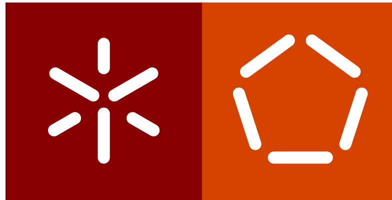


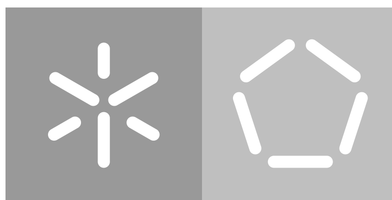
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Manuel Gonçalves Ribeiro

PlaCoR

Plataforma para a Computação orientada ao Recurso

Abril 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Manuel Gonçalves Ribeiro

PlaCoR

Plataforma para a Computação orientada ao Recurso

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação orientada por
António Manuel Silva Pina
Nuno Filipe da Silva Fernandes de Castro

Abril 2019

AGRADECIMENTOS

Concluída esta etapa, resta agradecer a todas as pessoas que, de uma forma ou outra, me ajudaram a atingir este objetivo.

Ao meu orientador, António Pina, pela orientação exemplar, incentivo, disponibilidade e dedicação ao longo do desenvolvimento deste projeto, bem como pela total confiança nas decisões tomadas no percurso do projeto.

Ao meu coorientador, Nuno Castro, pela oportunidade que me concedeu de fazer parte do LIP Minho.

Aos meus amigos, pelo apoio e paciência para com os desabafos sobre o trabalho, bem como por todos os momentos e histórias surgidas ao longo desta caminhada.

À minha família, em especial aos meus pais e irmão, e também à Maria Martins, pelo apoio incondicional nos bons e maus momentos, pelas palavras de incentivo e a confiança depositada nas minhas capacidades.

Este trabalho foi financiado pelo LIP, Laboratório de Instrumentação e Física Experimental de Partículas, através da bolsa de investigação com a referência LIP/BI-16/2018 FCT, COMPETE2020-Portugal2020, FEDER, POCI-01-0145-FEDER-007334, no âmbito do projeto BigData POCI/01-0145-FEDER-029147 PTDC/FIS-PAR/29147/2017, financiado por fundos OE/FCT, Lisboa2020, Compete2020, Portugal 2020, FEDER.

ABSTRACT

The Resource-Oriented Computing Platform (PlaCoR) was designed as an application programming and execution environment based on the resource-oriented computing (CoR) model, specified in CoRes, fully written in Modern C++.
The choice of C++ brought enormous advantages, in support to: i) object-oriented programming through multiple inheritance (in the construction of resources); ii) generic programming (allowing to abstract in the API the different classes of resources); iii) concurrent programming (to take advantage of run-time threads and synchronization structures native to C++).
The platform has facilities for: i) inter-domain communication, ii) message passing between communicating resources, iii) distributed shared memory (DSM), iv) remote activation of execution threads (RPC), v) creation and management of resources and vi) consistency management among all replicas of a resource.
Currently, the design of CoR applications relies on the resources domain, group, closure, agent, proto-agent, data, barrier, guard and read/write guard. The domains establish the first level of concurrency/parallelism, whether created at the beginning of the application or dynamically launched. Agents, for their part, are associated with the fine grain of parallelism and communication by message passing.
The domain, the group and the closure are structured resources that provide join/leave operations of resources; the first two are dynamic while the closure is static, the later meaning that the join/leave operations are collective and the total number of members is fixed initially - characteristics necessary for the parallel start of SPMD-type applications and intra-closure message passing.
The guard is used to create distributed mutual exclusion zones (read/write), the barrier for agent synchronization, while the data comprises distributed shared memory mechanisms, used to make the user's data available in a distributed domain environment.
The evaluation of the platform took as an application example the reading and processing of events registered in TTree, recurrently used in the ATLAS experience. The various versions developed justified the creation of a specific module, the Pool unit, which performs the fork-join model.
The experiment confirmed the feasibility of resource orientation as a hybrid programming paradigm that integrates multiple threads of execution and distributed synchronization, with fine grain communication facilities for message passing and communication in secure contexts, remote access to memory and remote activation of agents.

RESUMO

A Plataforma para a Computação orientada ao Recurso (PlaCoR) foi desenhada como um ambiente de programação e execução de aplicações baseadas no modelo da computação orientada ao recurso (CoR), especificado em CoRes, integralmente escrito em C++ Moderno. A escolha do C++ trouxe enormes vantagens, no suporte à: i) programação orientada aos objetos, através da herança múltipla (na construção dos recursos); ii) programação genérica (permitindo abstrair na API as diferentes classes de recursos); iii) programação concorrente (para tirar partido de fios de execução e estruturas de sincronização nativas ao C++).
A plataforma possui facilidades para: i) comunicação inter-domínios, ii) passagem de mensagens entre recursos comunicantes, iii) memória partilhada distribuída (DSM), iv) ativação remota de fios de execução (RPC), v) criação e gestão de recursos e vi) gestão da consistência entre todas as réplicas de um recurso.
Atualmente, o desenho de aplicações CoR assenta nos recursos domínio, grupo, clausura, agente, proto-agente, dado, barreira, guarda e guarda para leituras/escritas. Os domínios estabelecem o primeiro nível de concorrência/paralelismo, quer sejam criados no início da aplicação ou lançados dinamicamente. Os agentes, pelo seu lado, estão associados ao grão fino de paralelismo e de comunicação por passagem de mensagens.
O domínio, o grupo e a clausura são recursos estruturados que disponibilizam operações de adesão/saída de recursos; distingue-os o facto dos dois primeiros serem dinâmicos enquanto a clausura é estática, na medida em que as operações de adesão/saída são coletivas e o número total de membros é fixado inicialmente - características necessárias para o arranque paralelo de aplicações do tipo SPMD e a passagem de mensagens intra-clausura.
A guarda é usada para a criação de zonas de exclusão mútua distribuídas (leituras/escritas), a barreira para a sincronização entre agentes, enquanto o dado contempla os mecanismos de memória partilhada distribuída, usado para disponibilizar os dados do utilizador num ambiente de domínios distribuídos.
A avaliação da plataforma tomou como exemplo de aplicação a leitura e processamento de eventos registados em TTree, recorrentemente usados na experiência ATLAS. As várias versões desenvolvidas justificaram a criação de um módulo específico, a unidade Pool, que realiza o modelo *fork-join*.
O experimento confirmou a viabilidade da orientação ao recurso como paradigma de programação híbrido que integra múltiplos fios de execução e sincronização distribuída, com facilidades de comunicação de grão fino para a passagem de mensagens e de comunicação em contextos seguros, o acesso remoto a memória e a ativação remota de agentes.

ÍNDICE

1 INTRODUÇÃO 1
1.1 Motivação 2
1.2 Objetivos 3
1.3 Estrutura da dissertação 3
2 COMPUTAÇÃO ORIENTADA AO RECURSO 4
2.1 Entidade Recurso 4
2.2 Classes base de recursos 5
2.3 Recursos físicos e recursos lógicos 5
2.4 Grupos e domínios 6
2.5 Anatomia do recurso CoR 6
2.5.1 Elementos 7
2.5.2 Definição 8
2.5.3 Generalização 9
2.6 Identificação de recursos 9
2.7 Instanciação de Recursos 11
2.7.1 Modelos de Representação 12
2.7.2 Elaboração 13
2.8 Desenho de aplicações CoR 13
2.8.1 Estrutura de uma aplicação CoR 14
2.8.2 Composição de aplicações CoR 15
3 PLATAFORMA PARA A COMPUTAÇÃO ORIENTADA AO RECURSO 16
3.1 Suporte à execução de aplicações CoR 16
3.1.1 Atribuição de identificadores 16
3.1.2 Interação entre domínios 17
3.1.3 Instanciação de recursos 18
3.1.4 Bibliotecas dinâmicas 19
3.2 Modelo e política de representação 19
3.2.1 Política centralizada 20
3.2.2 Política distribuída 20
3.3 Modelo de consistência 21
3.3.1 Objeto de consistência 21
3.3.2 Testemunho de escrita 22
3.3.3 Protocolo de invalidação e de atualização 23

3.4 Comunicação por passagem de mensagens 24
3.5 Programação em memória partilhada 24
4 PROTÓTIPO PLACOR 26
4.1 Apresentação do protótipo PlaCoR 27
4.1.1 Recursos e elementos 27
4.1.2 Identificação de recursos 28
4.1.3 Árvore de dependências 28
4.1.4 Gestão dos recursos 29
4.1.5 Escalonamento de domínios 29
4.1.6 Infraestrutura de comunicação 30
4.2 Arquitetura do sistema 31
4.3 Ferramentas de desenvolvimento 33
4.3.1 Spread 34
4.3.2 cereal 35
4.3.3 libssh 35
4.3.4 libevent 35
4.3.5 czrpc 35
4.4 Arranque e execução de aplicações 36
4.4.1 Arranque básico 37
4.4.2 Criação e interação de recursos 38
4.4.3 Arranque paralelo 42
4.4.4 Lançamento dinâmico de domínios 44
5 INTERFACE DE PROGRAMAÇÃO E FERRAMENTAS 46
5.1 Arquitetura geral 46
5.2 Utilização do C++ 47
5.3 Elementos 48
5.3.1 Contentor 48
5.3.2 Organizador 51
5.3.3 Executor 53
5.3.4 Caixa-Postal 54
5.3.5 Valor 56
5.3.6 Sincronizador 57
5.4 Recursos 58
5.4.1 Criação de recursos 58
5.4.2 Domínio 59
5.4.3 Grupo 60
5.4.4 Clausura 60
5.4.5 Agente 61

5.4.6 Proto-Agente 61
5.4.7 Dado 62
5.4.8 Barreira 62
5.4.9 Guarda 63
5.5 Funções globais 64
5.6 Ferramentas de sistema 64
5.6.1 corx 64
5.6.2 corun 65
6 PROGRAMAÇÃO AVANÇADA 67
6.1 Leitura e processamento de TChain 67
6.2 Unidade Pool 68
6.3 Orientação ao recurso 70
6.3.1 Processamento paralelo com a unidade Pool 70
6.3.2 Processamento distribuído com passagem de mensagens 71
6.3.3 Comunicação por contexto 73
6.3.4 Processamento distribuído com sincronização e memória partilhada distribuída 73
6.3.5 Refinamento ao processamento distribuído com criação coletiva de recursos dado 75
6.3.6 Clausuras e sincronização 76
6.3.7 Ativação remota de agentes 78
6.3.8 Mecanismos RPC 80
7 DISCUSSÃO E PERSPETIVAS FUTURAS 83
7.1 Conclusões 85
7.2 Trabalho Futuro 86

A EXPLORAÇÃO DO SISTEMA PLACOR 89
A.1 Instalação do ambiente 89
A.2 Compilação de aplicações 90
A.3 Configuração do ambiente de execução 90
A.4 Execução de aplicações 91
B UNIDADES PLACOR 92
B.1 Unidade Pool 92
B.1.1 Ficheiro pool.hpp 92
B.1.2 Ficheiro pool.cpp 93
B.2 Unidade Utils 96
C CÓDIGO CLASSE AUXILIAR READCHAIN 98

LISTA DE FIGURAS

Figura 1	O recurso como um objeto composto.	5
Figura 2	Anatomia do recurso CoR.	10
Figura 3	Árvore hierárquica de dependências de uma aplicação CoR.	15
Figura 4	Arquitetura geral do protótipo PlaCoR.	31
Figura 5	Ferramentas de desenvolvimento utilizadas no protótipo PlaCoR.	33
Figura 6	Legenda dos recursos PlaCoR.	37
Figura 7	Árvore de dependências de uma aplicação simples básica.	38
Figura 8	Árvore de dependências do exemplo `basic_perations.cpp`.	40
Figura 9	Árvore de dependências de um arranque paralelo.	43
Figura 10	Árvore de dependências de um arranque dinâmico de domínios.	45
Figura 11	Arquitetura da API.	46

LISTA DE EXCERTOS DE CÓDIGO

4.1 Módulo de arranque de uma aplicação básica. 37
4.2 Exemplo `basic_operations.cpp`. 39
4.3 Código do exemplo `callable_module.cpp`. 41
4.4 Exemplo `parallel.cpp`. 42
4.5 Exemplo `spawn.cpp`. 45

6.1 Exemplo para o processamento de uma TChain. 68
6.2 Cálculo do PI através do método MonteCarlo. 69
6.3 Processamento paralelo de uma TChain utilizando a unidade Pool. 71
6.4 Processamento distribuído de uma TChain com passagem de mensagens. . . 72
6.5 Processamento distribuído de uma TChain com sincronização e memória partilhada distribuída. 74
6.6 Processamento distribuído de uma TChain com criação coletiva de recursos dado. 75
6.7 Exemplo de criação de um novo contexto fechado com base em duas clausuras. 77
6.8 Exemplo de um cliente (`client_rpc`) para a ativação remota de agentes. . . . 80
6.9 Exemplo de um servidor (`server_rpc`) para a ativação remota de agentes. . 81
6.10 Exemplo para o registo de recursos para criação remota e de funções executadas por agentes para ativação remota. 82

ACRÓNIMOS

A

API *Application Programming Interface*. 46

C

CoR Computação orientada ao Recurso. 1–9, 11–17, 19, 21, 22, 24–26, 34, 39, 40, 70, 84

D

DSM *Distributed Shared Memory*. 20, 22, 75

E

ELF *Executable and Linking Format*. 19

M

MPI *Message Passing Interface*. 24, 43, 44, 73, 76, 85

P

PlaCoR Plataforma para a Computação orientada ao Recurso. 3, 4, 26–31, 35–37, 41, 42, 44, 47, 51, 58, 64, 67, 73, 76, 78, 83

R

RPC *Remote Procedure Call*. 12, 33, 78, 80, 84

1

INTRODUÇÃO

A Computação orientada ao Recurso (CoR) é um paradigma de computação e de programação que recorre à definição do recurso como meio para explorar a computação paralela e distribuída de elevado desempenho. A particularidade deste modelo prende-se com a forma como este combina um conjunto de mecanismos de estruturação com os princípios da programação com múltiplos fios de execução, da memória partilhada distribuída e da comunicação por passagem de mensagens, procurando atenuar as fronteiras que separam os componentes físicos de *hardware* dos componentes lógicos de *software*.

Em CoR, as principais características presentes no desenho de aplicações paralelas e distribuídas, tais como estruturação, computação, comunicação, memória e sincronização, são representadas por um conjunto de diferentes recursos, que se designam por classes base do modelo. Estes recursos são instanciados através de construtores próprios do modelo, e compreendem um ou mais elementos embutidos que traduzem as funcionalidades inerentes às classes base do modelo.

Em termos de localização dos dados e de balanceamento de carga, o programador tem uma grande flexibilidade na construção do esquema de representação de recursos que mais se adequa aos requisitos da aplicação, como a cópia única, replicação ou partição. Cada instância de um recurso possui um conjunto geral de atributos, estando estes relacionados com as suas propriedades lógicas e modos de representação no sistema de computação. Assim, é então possível intervir nos aspetos mais relevantes da computação paralela e distribuída, nomeadamente em questões da localidade, paralelismo, comunicação e coordenação, através da interrogação e alteração do valor desses atributos.

As aplicações construídas sobre o paradigma CoR assentam na gestão de um sistema de domínios distribuídos que identificam e representam os diversos recursos, recursos estes que representam entidades e conceitos externos ao modelo, tais como processos, fios de execução, interfaces de comunicação ou nós de computação. Assim, o programador tem ao seu dispor um modelo de programação para o desenvolvimento de aplicações paralelas e distribuídas de fácil utilização, focado no compromisso entre performance e esforço de desenvolvimento, e de fácil ajuste ao sistema de computação adjacente.

1.1 MOTIVAÇÃO

A necessidade de correr aplicações mais eficientemente e até mesmo de acelerar o processamento de grandes volumes de dados, levou a um recurso cada vez mais acentuado à computação paralela e distribuída como forma de reduzir o tempo necessário para a execução das aplicações, e de resolver problemas mais complexos e de maior dimensão. Assim, existe atualmente uma tendência acentuada para a construção e utilização de sistemas de computação que recorrem, simultaneamente, a modelos de memória partilhada e de memória distribuída, tais como *clusters* de multiprocessadores de memória partilhada interligados por redes de alta velocidade.

Foi neste contexto que se optou por reavivar e explorar os conceitos propostos pelo modelo CoR, procurando desenvolver um protótipo de uma plataforma como um ambiente de programação e execução de aplicações paralelas e distribuídas baseadas neste modelo híbrido. Este protótipo surge como um ambiente onde se combinam os princípios da programação com múltiplos fios de execução, da memória partilhada distribuída e da comunicação por passagem de mensagens, como meio para explorar a capacidade computacional dos sistema de computação atuais.

Com este ambiente de programação e execução, não se procura a competição direta com outros ambientes atuais que exploram o paralelismo e a distribuição dos dados, mas sim uma alternativa viável que utiliza o conceito de recurso como uma abstração para os conceitos mais relevantes da computação paralela e distribuída.

O desenvolvimento de aplicações para este ambiente apresenta inúmeras vantagens, das quais se destacam a facilidade de utilização e facilidade no ajuste das aplicações à capacidade computacional dos sistemas de computação. É com base na facilidade de configuração que este ambiente se demonstra focado no compromisso entre performance e esforço de desenvolvimento, pois, com base na alteração do valor dos atributos dos vários recursos da aplicação, consegue-se ajustar o grão de paralelismo e a distribuição dos dados da mesma, adequando assim a aplicação à capacidade computacional dos sistemas de computação adjacente.

No entanto, o modelo CoR foi pensado numa altura anterior à era do *multi-core*, pelo que ainda não contempla conceitos atuais nucleares da programação por memória partilhada. Neste sentido, é necessário desenhar e incluir novos recursos no modelo que contemplem esses mesmos conceitos. Da mesma forma, a plataforma terá de contemplar esses mesmo conceitos, permitindo assim que o paradigma seja capaz de tirar partido das arquiteturas de computação atuais.

1.2 OBJETIVOS

Em termos gerais, o objetivo deste projeto é criar um ambiente de suporte à criação e execução de aplicações distribuídas, com um elevado número de atividades paralelas e um grão fino de paralelismo e de comunicação.

Um outro objetivo a alcançar com o protótipo Plataforma para a Computação orientada ao Recurso (PlaCoR), é a construção de uma infraestrutura autónoma desenvolvida em C++ Moderno, multiplataforma - Linux, Solaris e MacOS -, com facilidades para: comunicação inter domínios, passagem de mensagem entre recursos comunicantes, memória partilhada distribuída, a ativação remota de agentes, criação e gestão de recursos, e capacidade de manter a consistência entre todas as réplicas de um recurso.

O objetivo final é a validação da plataforma, através da criação e execução de um número significativo de aplicações, num ambiente de domínios distribuídos. E, ao mesmo tempo, avaliar o poder expressivo do paradigma da orientação ao recurso, na escrita de programas concorrentes.

1.3 ESTRUTURA DA DISSERTAÇÃO

O capítulo 2, *Computação orientada ao Recurso*, apresenta o modelo de computação orientada ao recurso. Começa-se por explicitar o conceito de recurso, passando-se pela definição da sua anatomia, até se apresentar de uma forma geral as aplicações CoR.

O capítulo 3, *Plataforma para a Computação orientada ao Recurso*, apresenta um conjunto de pontos relevantes a ter em conta para a construção de um protótipo de uma plataforma que permita explorar o modelo CoR.

O capítulo 4, *Protótipo PlaCoR*, apresenta o protótipo desenvolvido, com especial foco na arquitetura e nas ferramentas utilizadas para a implementação deste. São ainda demonstrados alguns exemplos simples de aplicações construídas com base na plataforma, de forma a explicitar conceitos.

O capítulo 5, *Interface de Programação e Ferramentas*, apresenta a interface de programação do protótipo desenvolvido, bem como os utilitários utilizados para correr aplicações. É neste capítulo que se encontra a informação necessária para construir aplicações para explorar o modelo.

O capítulo 6, *Programação Avançada*, apresenta um conjunto de exemplos de aplicações paralelas e distribuídas. Estes explicitam um conjunto mais aprofundado de conceitos do modelo, explorando as facilidades da plataforma e o estilo de programação veiculado pela API.

O capítulo 7, *Discussão e Perspetivas Futuras*, apresenta uma visão crítica geral do projeto realizado, bem como propõe linhas de desenvolvimento para trabalho futuro.

2

COMPUTAÇÃO ORIENTADA AO RECURSO

A Computação orientada ao Recurso (CoR) surgiu com o objetivo desenvolver, enriquecer e flexibilizar os conceitos propostos em PlaCC [1, capítulo 10]. Em PlaCC foi proposto o conceito de recurso, tendo como base a formalização dos conceitos de célula, operão, gene e variável fisiológica propostos no Modelo de Computação Celular [1].

CoRes [2] foi o primeiro esforço na especificação do modelo CoR, um modelo de computação orientado ao recurso, que assenta na definição de uma metáfora genérica que incorpora diretamente a noção de estado, concorrência, localidade e distribuição. É com base no CoRes que surge o protótipo PlaCoR, como um ambiente de desenvolvimento e execução de aplicações, construídas sobre o paradigma da computação orientada ao recurso.

Ao longo deste capítulo vão ser apresentados os conceitos do modelo CoR proposto em CoRes, desde o conceito de recurso e a sua anatomia, até à estrutura de uma aplicação.

2.1 ENTIDADE RECURSO

A entidade recurso, ilustrada na figura 1, foi definida em PlaCC como sendo constituída pelos seguintes elementos:

- ascendente - uma referência usada para ligar um recurso, no momento da sua criação, a um outro recurso seu ascendente;
- identificador - um índice e um nome (opcional) que identificam univocamente o recurso no contexto do recurso ascendente;
- consistência - uma referência ao objeto usado para controlar os modos de acesso a um recurso e manter a coerência de estado entre todas as suas representações.
- atributo - uma referência a um objeto que regista o estado inicial do recurso, no momento da sua criação, e pode ser usado para inquirir ou alterar o seu comportamento e propriedades atuais;
- instância - uma referência ao objeto de computação que é a realização concreta do recurso no sistema hospedeiro;

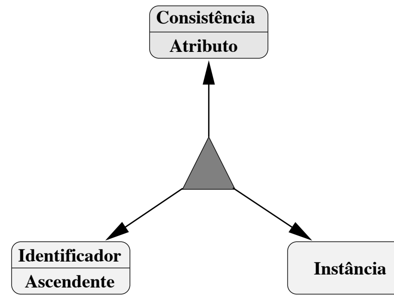


Figura 1.: O recurso como um objeto composto, de CoRes [2, pag.23].

Em CoR, a instância do recurso deu lugar a uma representação que compreende um ou mais elementos embutidos - *contentor*, *organizador*, *executor*, *valor*, *caixa-postal*, *porto* e *sincronizador*. É com base na combinação destes elementos que se traduzem as funcionalidades inerentes às classes base de recursos do modelo - *domínio*, *grupo*, *tarefa*, *dado*, *guarda* e *barreira*.

2.2 CLASSES BASE DE RECURSOS

O *domínio* é um recurso que delimita uma região do espaço de endereçamento onde enquadra um espaço de representação, de computação e de comunicação, na qual os recursos têm existência e interagem entre si.

O *grupo* é um recurso capaz de integrar a atividade de múltiplos fios de execução e outros elementos de computação, incapazes de por si só darem resposta às exigências de estruturação e de cooperação de uma aplicação paralela.

A *tarefa* é um recurso executável, escalonado como um ou mais fios de execução, com capacidade para executar uma determinada função.

O *dado* é uma classe geral de recursos que possibilitam a manipulação de uma grande variedade de blocos de informação, estruturados ou não estruturados, simples ou complexos, globalmente partilhados num ambiente de computação distribuído.

A *guarda* e a *barreira* são instrumentos de sincronização distribuídos, usados para definir zonas de exclusão mútua e para coordenar a execução de atividades paralelas.

2.3 RECURSOS FÍSICOS E RECURSOS LÓGICOS

Com o modelo CoR surgiram dois novos conceitos que classificam os recursos do modelo em relação à sua representação no sistema de computação: o de recurso físico e o de recurso lógico.

O conceito de recurso físico está intrinsecamente relacionado com as noções de localidade, de concorrência e de distribuição, partilhadas por domínios e tarefas. Este tipo de recurso surgiu como forma de dar resposta às crescentes necessidades de eficiência das aplicações, bem como para tirar partido das arquiteturas dos sistemas de computação paralelos.

Já o conceito de recurso lógico, partilhado pela maior parte dos recursos do modelo, surgiu para dar suporte ao desenho de aplicações complexas, modulares e de grande dimensão. Estes recursos criam as condições de execução, os modos de interação e de coordenação, e as estruturas de controlo necessárias para o normal funcionamento das aplicações.

2.4 GRUPOS E DOMÍNIOS

Em CoR, os grupos e os domínios são recursos logicamente equivalentes, na medida em que ambos organizam os recursos que se constituiram como membros destes. No entanto, a diferença entre ambos está na forma como os recursos membros são representados.

O domínio alberga no seu seio outros recursos, incluindo também recursos da classe grupo, o que o leva a comportar-se como um grupo. No entanto, sendo o domínio um recurso que delimita espaços de endereçamento onde têm lugar as interações entre recursos, os recursos são representados fisicamente pelo domínio no sistema de computação. Já o grupo apenas representa logicamente os recursos nele presentes, sendo que a representação física dos mesmos apenas existe no contexto de um domínio pré-existente, no qual o grupo foi construído.

Assim, visto que as propriedades atribuíveis aos grupos lhe são extensíveis, o domínio pode ser considerado simultaneamente recurso físico e recurso lógico. Por outro lado, é possível também atribuir dupla personalidade ao grupo, se considerarmos a equivalência entre recurso físico (para o domínio) e recurso físico virtual (para o grupo).

2.5 ANATOMIA DO RECURSO COR

A materialização da instância de um recurso como uma composição de elementos que expressam funções específicas, surgiu como uma evolução concetual da entidade recurso em relação a PlaCC. Esta nova abordagem torna o modelo mais flexível, na medida em que oferece ao programador a possibilidade de construir os seus próprios recursos tendo como base os elementos propostos em CoR, bem como a adição de novas funcionalidades.

2.5.1 *Elementos*

Contentor

O *contentor* é um elemento básico criado para dar suporte aos requisitos de memória e aos meios de comunicação e computação necessários à interação entre recursos. É no seio deste elemento que os recursos estão representados, e também onde ocorrem as interações entre si.

Este elemento é uma peça fulcral na gestão de todo o sistema de domínios distribuídos, na qual os diferentes elementos contentor instanciados pelo sistema interagem entre si e coordenam em conjunto as atividades de gestão do sistema. Assim, um recurso que contenha um elemento contentor é considerado um recurso físico, surgindo assim uma nova entidade ativa.

Executor

O elemento *executor* é o agente responsável pelo suporte à atividade computacional, pelas transformações de estado e pela interação entre recursos. Este elemento básico possui a capacidade de se associar a um qualquer recurso, estaticamente no momento de criação do recurso ou dinamicamente, munindo o recurso com um fio de execução capaz de processar a tarefa computacional correspondente ao contexto do recurso na origem da associação.

Os executores são lançados no domínio onde é criado o recurso associado, e assumem a sua identidade para poderem agir sobre a região demarcada do recurso e interagir com os demais recursos da aplicação.

Organizador

O *organizador* é um elemento básico que fornece um mecanismo de estruturação para as aplicações baseado na fixação das afinidades lógicas entre recursos. Este elemento é constituído por uma estrutura que contém toda a informação necessária à gestão dos elementos que agrupa, ou seja, a identificação do recurso do qual é elemento e a identificação dos recursos agrupados.

Valor

O elemento *valor* é uma região contígua de memória com um tipo associado. Este elemento surgiu para suprir a necessidade do conceito de estado em CoR, estando na maior parte dos modelos de programação associado à definição de variáveis.

Com este elemento é possível definir vetores, estruturas e classes, ficando a consistência destes, num ambiente de memória partilhada distribuída, ao encargo do sistema. O sistema

possui, assim, um conjunto de modelos, orientados ao objeto, para garantir a consistência destes elementos.

Caixa-postal

O elemento *caixa-postal* está intimamente relacionado com a passagem de mensagens, geralmente presente no desenvolvimento de aplicações paralelas e distribuídas. Em CoR, a passagem de mensagens é o mecanismo de suporte à interação entre recursos, sendo que um recurso que possua este elemento tem capacidade para enviar e receber mensagens de outros recursos, locais ou remotos.

Porto

O *porto* permite selecionar o protocolo e o meio de comunicação que melhor se adequa à transferência de informação entre dois domínios da aplicação. Assim, este constitui-se como um elemento passível de ser acrescentado ao domínio, de carácter opcional.

Sincronizador

O elemento *sincronizador* é utilizado como base para a construção de recursos de sincronização, tais como a guarda ou a barreira. Este foi introduzido em CoR para dar resposta às necessidades de coordenação e sincronização na construção de aplicações paralelas e distribuídas.

2.5.2 *Definição*

O *grupo* e o *domínio* partilham a característica de organizadores de outros recursos. No entanto, o domínio distingue-se do grupo por ter representação física no sistema de computação, delimitando uma região do espaço de endereçamento onde ocorrem as interações entre os recursos que representa, ou seja, é constituído por um elemento contentor. Assim, o domínio e o grupo definem-se da seguinte forma:

```
domínio :: contentor . organizador . (porto)?
grupo :: organizador
```

A *tarefa* é um recurso computacional constituído por fios de execução autónomos, os executores, que executam uma determinada função. Sendo a sobreposição de computação e comunicação um dos métodos mais utilizados para refinar o grão de paralelismo de uma aplicação, é importante poder ter vários executores associados à mesma tarefa. Uma tarefa pode também estar associada a dados necessários para efetuar a computação, pelo que pode também ter associado um elemento valor. Por fim, para que a tarefa tenha capacidade

para enviar e receber mensagens, é necessário esta conter uma caixa-postal na sua definição. Assim, uma tarefa é definida como:

```
tarefa :: (executor)+ . (caixa-postal)? . (valor)?
```

O *dado* compreende uma zona de memória contígua que contém os valores requeridos pelo programador, ou seja, é constituído apenas por um elemento valor, definindo-se da seguinte forma:

```
dado :: valor
```

Uma *guarda* ou uma *barreira* são recursos com funções de sequenciação e de sincronização de atividades concorrentes, constituídos apenas por um elemento sincronizador, sendo definidos da seguinte forma:

```
guarda | barreira :: sincronizador
```

2.5.3 *Generalização*

Com base nas definições apresentadas acima, é possível generalizar a definição do recurso CoR. No entanto, é necessário acrescentar dois elementos fundamentais para a sua representação: a identificação e as propriedades (atributos) que o qualificam.

O organizador, a caixa-postal, o valor e o sincronizador, como se materializam em estruturas de dados e construtores que sobre elas atuam, foram agregados num componente chamado corpo do recurso. É importante denotar que o corpo do recurso só pode contemplar apenas um elemento de cada tipo.

Os restantes elementos estão relacionados com os meios físicos de computação e comunicação. Os executores que no decorrer da aplicação assumem a entidade do recurso, são reunidos no componente operão. Já o contentor é usado sempre em conjunto com um organizador, permitindo assim construir e distinguir um domínio físico de um domínio lógico.

Assim, a definição genérica da entidade recurso, ilustrada na figura 2, é a seguinte:

```
recurso :: identificação . propriedades . (contentor)? .  (corpo)? . (operão)? . (porto)?
corpo :: (organizador)? . (caixa-postal)? . (valor)? . (sincronizador)?
operão :: (executores)*
```

2.6 IDENTIFICAÇÃO DE RECURSOS

Um dos aspetos principais deste modelo é a identificação dos recursos, pois é com base nos identificadores que os recursos promovem interações entre si. Assim, foram definidos dois níveis de identificação: identificação local e identificação global à aplicação.

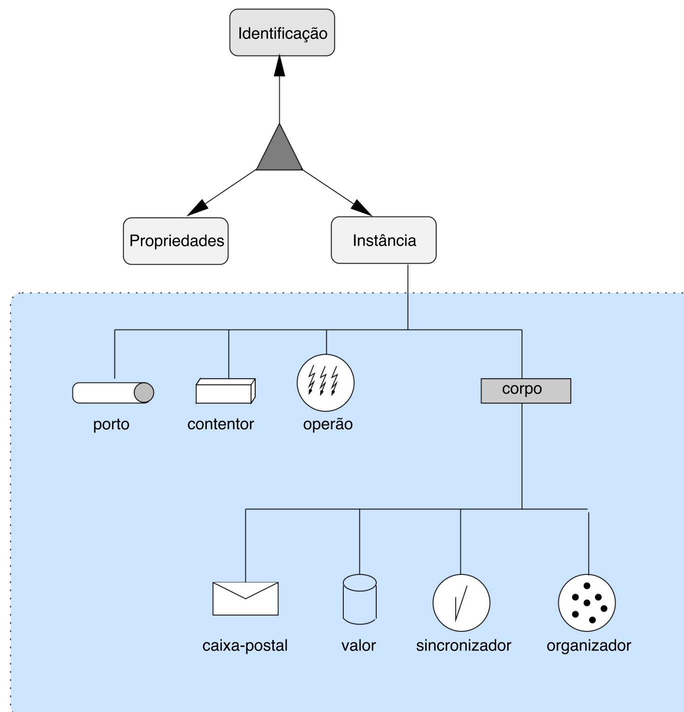


Figura 2.: Anatomia do recurso CoR, de CoRes [2, pag.30].

A primeira forma de identificação, também designada por identificação no contexto do grupo ascendente, recorre à utilização de dois identificadores, sendo eles o nome e o identificador de membro (idm), para os quais é necessário garantir que são únicos no contexto do grupo. O nome é a designação atribuída pelo programador, e o identificador de membro corresponde a um índice de membro, gerado automaticamente aquando da operação de adesão ao grupo.

Desta forma, tanto o nome como o idm terão de ser utilizados sempre em conjunto com uma referência ao seu ascendente, formando assim um par (<identificador local>, <ascendente>), que identifica o recurso no espaço global da aplicação. Da mesma forma, o seu ascendente terá de ser necessariamente identificado por um par (<identificador local>, <ascendente>), definindo assim uma identificação recursiva que representa os recursos da aplicação numa árvore de dependências.

Já a identificação global baseia-se na atribuição de identificadores individuais para os recursos, sendo estes únicos no contexto da aplicação - identificador principal (idp). Estes identificadores surgiram da necessidade de os domínios e os grupos terem uma identificação direta ao objeto que os representa, tendo sido depois generalizados para os restantes recursos do modelo.

Estas identificações são geradas assim que um recurso é criado ou adere a um recurso ascendente (grupo ou domínio). A qualquer momento, um recurso pode tornar-se membro de um outro ascendente, criando-se assim um pseudónimo do recurso original. Este pseudónimo é identificado por um novo idp e um novo idm no novo contexto, e estabelece um outro caminho na árvore de dependências.

O primeiro pseudónimo do recurso, ou seja, a identificação da primeira instância do recurso, é produzido através de uma operação de criação de um recurso, o qual é denominado por membro fundador do grupo (ou domínio) seu ascendente. Os restantes pseudónimos do mesmo recurso, instanciados por operações de adesão a outros grupos (ou domínios), são chamados membros aderentes. Estes possuem, em geral, uma representação simbólica (ou parcial) no grupo a que pertencem, enquanto que os membros fundadores possuem uma representação rígida (ou completa) no contexto do grupo a que pertencem.

De um modo geral, a operação de adesão é uma forma de criação virtual de recursos, na qual, através do conceito de pseudónimo, se estabelecem múltiplas visões sobre o mesmo recurso em contextos diferenciados (estes relacionados com os ascendentes dos grupos de que são membros).

Estes conceitos mostram-se vitais para o modelo CoR, pois sendo este um ambiente de programação e computação distribuído, a localidade de referências é fundamental para a eficiência do sistema. Assim, faz sentido que um recurso que necessita de cooperar em atividades fora do seu domínio de origem o consiga fazer, preservando a sua identidade original através da criação de uma nova identidade sobre o novo contexto em que está inserido.

2.7 INSTANCIAÇÃO DE RECURSOS

A instanciação de recursos através da composição de elementos básicos, oferece ao programador a possibilidade de construir recursos com diferentes funcionalidades, adaptadas às exigências das aplicações. No entanto, esta composição de elementos pode introduzir ambiguidade na classificação de um determinado recurso em relação à sua classe.

A classificação de recursos mostra-se então fundamental, não só para assegurar os conceitos do modelo, como o conceito da abstração "ascendente", restrito a grupos e domínios, mas também para fornecer ao programador um conjunto de classes pré-definidas de recursos que se adequam às necessidades das aplicações.

Em CoR foi definido um conjunto de regras que determinam a classe dos recursos em relação aos elementos que os constituem, apresentadas de seguida:

- um recurso composto por um elemento organizador pertence à classe grupo, independentemente de conter, ou não, outros elementos básicos;

- um recurso que contenha simultaneamente os elementos contentor e organizador é necessariamente da classe domínio;
- um recurso que não considera qualquer um dos elementos anteriormente referenciados, mas seja instanciado com pelo menos um executor, é da classe tarefa;
- um recurso que contemple a existência isolada dos restantes elementos do corpo de um recurso resulta nas seguintes classificações:
  - um elemento valor corresponde à classe dado;
  - um elemento sincronizador corresponde às classes guarda e barreira;
  - um elemento caixa-postal corresponde a recursos comunicantes.

2.7.1 *Modelos de Representação*

A representação de um recurso está relacionada com a forma como o recurso é representado e gerido no sistema de domínios distribuídos, em relação a todos os seus pseudónimos. É importante denotar que, apesar de poderem existir vários pseudónimos do mesmo recurso, em todo o sistema apenas existe uma única representação do corpo do recurso.

O modelo CoR permite a escolha do esquema de representação dos recursos que melhor se adequa aos requisitos de localidade, concorrência e paralelismo das aplicações. Os esquemas possíveis são: cópia única, replicação e partição.

O esquema mais simples é o de cópia única, que implica que sempre que um pseudónimo, que não o original, necessitar de aceder ao corpo do recurso para executar uma determinada funcionalidade, é feita uma *Remote Procedure Call* (RPC) [3] ao domínio onde o objeto se encontra instanciado.

No entanto, num sistema de domínios distribuídos, a necessidade de redução da sobrecarga das comunicações pode determinar a existência de múltiplas réplicas do corpo de um recurso, p. ex. quando é criado um pseudónimo de um recurso num domínio remoto, ou quando uma operação envolve recursos não representados no domínio local. Assim, estamos perante um esquema de replicação, em que o sistema é responsável por gerir automaticamente as réplicas locais e remotas dos recursos, assegurando a consistência das mesmas de acordo com a política definida (pelo programador ou por omissão).

Por fim, foi proposta uma alternativa mais eficiente comparativamente à replicação, na qual um recurso é dividido pelos vários pseudónimos, sendo atribuído a cada um uma parte de acesso individual. Este mecanismo potencia o aumento de concorrência e paralelismo, tanto ao nível dos dados como ao nível da tarefa, reduzindo simultaneamente a sobrecarga inerente à manutenção da consistência das múltiplas réplicas do corpo de um recurso partilhado e distribuído.

Podemos também classificar estes diferentes modelos de representação quanto à sua política de representação, centralizada ou distribuída. A existência de um recurso com uma única representação no sistema (cópia única) leva a que o modo de acesso ao seu corpo seja centralizado. Já a existência de múltiplas réplicas ou de pseudónimos com partes do corpo do recurso, disseminadas pelo sistema de domínio distribuídos, é compatível com acessos paralelos e distribuídos, constituindo assim uma política distribuída.

2.7.2 *Elaboração*

A elaboração é uma propriedade aplicada a cada um dos diferentes elementos do corpo de um recurso - organizador, caixa-postal, valor e sincronizador -, que define o comportamento de cada elemento básico na criação de novos pseudónimos deste. O grau de elaboração de cada um dos componentes do recurso é determinado, pelo utilizador, na criação do pseudónimo original. Assim, é o pseudónimo original que determina a estrutura dos restantes pseudónimos, mantendo-se a coerência entre os mesmos.

Um recurso é trivial (uno) se todos os seus pseudónimos são uma referência para o mesmo corpo. Pelo contrário, um recurso é elaborado (particionado) se cada um dos seus pseudónimos contém uma parte independente, ou seja, se cada um possui um corpo independente do original.

No entanto, é necessário denotar que os modelos de representação de recursos e o seu grau de elaboração correspondem a abordagens diferentes. Enquanto que o primeiro está relacionado com o modelo escolhido para a representação e gestão de recursos, o segundo está relacionado com a criação de novos pseudónimos de um recurso, e com a visão que cada novo pseudónimo tem sobre o corpo de um recurso.

2.8 DESENHO DE APLICAÇÕES COR

A orientação ao recurso facilita a idealização e o desenvolvimento de aplicações paralelas, fornecendo ao programador um conjunto de recursos que abstraem os conceitos de estado, concorrência, paralelismo, localidade e distribuição. Assim, o programador possui um modelo que se foca na agilização das relações entre as fases de estruturação e computação, tipicamente envolvidas no desenvolvimento de aplicações.

Por estruturação entende-se a definição das estruturas físicas e lógicas, e a nomeação das entidades presentes na aplicação. A computação relaciona-se com o processo contínuo de transformação do estado das entidades da aplicação, através da execução das instruções de um programa paralelo e distribuído com múltiplos fios de execução.

O desenho de uma aplicação em CoR é, numa primeira fase, orientado à definição das entidades identificáveis no problema, à avaliação das suas afinidades lógicas e físicas, e

à identificação dos modos de interação entre as entidades, mapeando depois todas estas características a partir das classes de recursos fornecidas pelo modelo. No entanto, existe ainda a possibilidade de combinar os elementos básicos pertencentes à anatomia do recurso CoR para construir novas entidades, ou então estender novas funcionalidades às entidades pré-existentes para responder à necessidade de incluir novas funcionalidades.

Numa segunda fase, as entidades definidas na primeira fase são ajustadas às condições de execução, atendendo à capacidade de armazenamento, de concorrência, paralelismo e distribuição, da infraestrutura de computação e comunicação, para cada arquitetura alvo.

2.8.1 *Estrutura de uma aplicação CoR*

As aplicações em CoR são uma construção lógica formada por um sistema de domínios distribuídos, na qual os recursos são organizados numa árvore hierárquica de dependências, que tem como raiz o primeiro domínio da aplicação - meta-domínio. Este domínio, também designado por domínio principal, é o ascendente primordial do grupo meta-domínio, e, simultaneamente, o seu primeiro membro fundador, assumindo assim na estrutura da aplicação a posição de raiz na árvore de dependências. Os nodos da árvore correspondem a recursos estruturados (domínios e grupos), e as folhas a recursos simples (tarefas, dados, guardas e barreiras).

Na árvore de dependências de uma aplicação, como por exemplo a ilustrada na figura 3, cada recurso é definido como sendo descendente de um domínio ou grupo, seu ascendente, e instanciado no domínio da sub-árvore de dependências onde está representado o seu ascendente. Assim, é possível definir relações de afinidade entre as entidades envolvidas na computação, como por exemplo a localidade dos recursos, facilitando a estruturação da aplicação e a identificação das entidades envolvidas na computação. A árvore de dependências é construída dinamicamente, em tempo de execução, à medida que os recursos são criados e removidos.

A criação de domínios numa aplicação é uma operação traduzida pelo sistema como uma operação de adesão ao grupo inicialmente nomeado como seu ascendente, ao mesmo tempo em que o domínio é efetivamente criado como membro fundador do meta-domínio da aplicação. Todos os domínios são funcionalmente equivalentes ao domínio principal da aplicação, como sendo as raízes das sub-árvores de dependência da aplicação. A diferença entre estes reside no facto de o domínio principal da aplicação ser responsável por todo o arranque do sistema de domínios distribuídos.

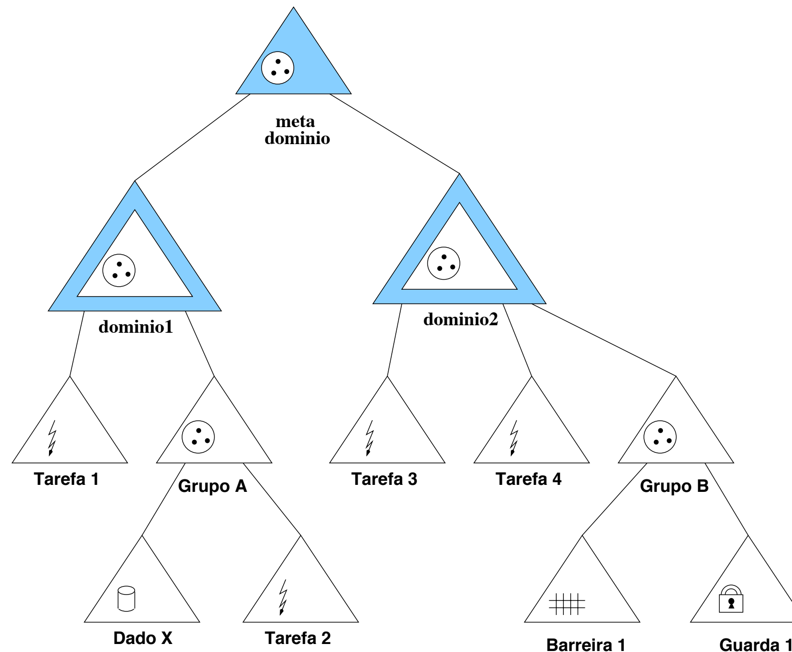


Figura 3.: Árvore hierárquica de dependências de uma aplicação CoR, de CoRes [2, pag.26].

2.8.2 *Composição de aplicações CoR*

A noção de aplicação como um conjunto de recursos hierarquicamente organizados numa árvore dinâmica de dependências, levou ao aparecimento do conceito de composição de aplicações. Este conceito estendeu as características do modelo CoR, permitindo que sistemas de elevado grau de complexidade e níveis de organização possam ser construídos, não só pela composição de vários módulos e funções, mas também por composição e integração de diferentes aplicações.

Em CoR, é através da utilização de recursos estruturados que combinam uma variedade de recursos simples, que uma aplicação realiza uma tarefa computacional que, sem estes mecanismos de estruturação, não era exequível. Este é considerado o primeiro nível de composição, equivalente a um programa construído a partir da conjugação de vários módulos e funções.

No entanto, também foi previsto para este modelo um nível de composição elaborado, que permite a interação entre diferentes aplicações. Esta é conseguida através da possibilidade de um domínio, membro fundador do respetivo meta-domínio, aderir a um outro meta-domínio de uma outra aplicação, surgindo assim o conceito de meta-aplicação. O grupo meta-aplicação é assim considerado um recurso de nível superior, que permite a integração de diferentes aplicações.

3

PLATAFORMA PARA A COMPUTAÇÃO ORIENTADA AO RECURSO

Ao longo deste capítulo vão ser levantados aspetos a ter em conta para a construção de uma plataforma, que permita explorar o modelo CoR. Estes vão desde os detalhes técnicos para permitir o suporte à execução de aplicações, até às diferentes formas para representar e gerir a consistência dos recursos, num ambiente de memória partilhada distribuída.

3.1 SUPORTE À EXECUÇÃO DE APLICAÇÕES COR

No modelo CoR, o domínio é proposto como um recurso intimamente ligado aos meios físicos de computação, delimitando uma região do espaço de endereçamento na qual são representados os recursos que existem no seu seio, bem como onde estes interagem entre si. Assim, o domínio possui um conjunto de serviços e estruturas de suporte que lhe permitem gerir e controlar as entidades que lhe estão confinadas.

Nos pontos seguintes serão abordadas questões relacionados com a execução de aplicações CoR, nomeadamente serviços e estruturas de suporte necessárias ao nível do domínio para garantir um ambiente de execução de aplicações.

3.1.1 *Atribuição de identificadores*

Tal como anteriormente abordado, existem dois tipos de identificadores: identificadores locais e identificadores principais.

Os primeiros identificadores são estritamente locais ao nível dos grupos, sendo que a sua atribuição baseia-se na geração de um identificador único no contexto do grupo, sempre que um novo recurso faz uma adesão, que pode ou não vir a ser reaproveitado quando este terminar as suas funções.

Relativamente aos identificadores principais, estes são atribuídos em duas fases. Numa primeira fase, o meta-domínio é responsável por criar e gerir endereços únicos (globais) no contexto da aplicação, que atribui em lotes aos seus membros da classe domínio. A

qualquer momento, os domínios podem requisitar novos lotes de endereços sempre que se esgotarem os endereços dos lotes anteriores.

Numa segunda fase, aquando da criação de novos recursos no contexto local do domínio, identificadores principais serão atribuídos aos recursos, gerados a partir do lote de endereços local ao domínio. Desta forma, é possível estabelecer uma correspondência entre a identidade do recurso e o domínio na raíz da sub-árvore de dependências do seu ascendente.

Identificador principal

O identificador principal de um recurso é composto por dois campos: endereço e pseudónimo. Quando um recurso é criado, é-lhe atribuído um bloco de endereços com o tamanho determinado pelo número de bits definidos para o campo pseudónimo. O primeiro endereço do bloco é sempre atribuído ao pseudónimo original do recurso, pelo que os restantes endereços são reservados para os seus pseudónimos, criados por operações de adesão a outros grupos ou domínios, locais ou remotos.

Com esta política de atribuição, a geração de endereços para novos pseudónimos do recurso é uma operação local ao domínio do pseudónimo original. O campo endereço é fixado na criação do pseudónimo original e partilhado com os restantes pseudónimos do recurso. O campo pseudónimo é diferente para os diferentes pseudónimos do recurso, sendo atribuído pela ordem de criação dos mesmos.

3.1.2 *Interação entre domínios*

Uma aplicação CoR por natureza paralela e distribuída, pressupõe a existência de vários domínios, pelo menos um por cada nó de computação utilizado na execução da aplicação. Assim, é necessário munir o sistema de domínios distribuídos com a capacidade de os diferentes domínios poderem interagir entre si e cooperarem nas tarefas de gestão do sistema.

Neste sentido, é necessário que todos os domínios instanciados pela aplicação tenham conhecimento geral sobre todos os domínios da aplicação, bem como sejam notificados sempre que um domínio é criado ou removido, sendo ainda necessário garantir a entrega das mensagens. Este aspeto está frequentemente associado à implementação de sistemas distribuídos, na qual a construção de sistemas altamente fiáveis é um ponto crítico.

Para endereçar este problema, é comum recorrer-se a bibliotecas que implementam mecanismos de comunicação por grupo [4]. Os sistemas que implementam estes mecanismos são projetados para que as mensagens enviadas para um grupo cheguem corretamente a todos os membros do mesmo, ou então não cheguem a nenhum deles. Esta propriedade

denomina-se por atomicidade, garantindo a confiabilidade das comunicações dentro de um grupo.

Outra propriedade importante inerente a este mecanismo prende-se com a ordenação das mensagens. Esta está relacionada com um mecanismo de entrega de mensagens que possa assegurar a completa sequenciação das mensagens, isto é, garantir que a sequência temporal de entrega de mensagens seja equivalente à sequência temporal de envio. Ainda assim, é também possível agilizar o processo de envio, não garantindo a ordenação das mesmas.

3.1.3 *Instanciação de recursos*

Sendo um domínio constituído por um espaço de representação para os recursos inseridos no seu contexto, é necessário que o mesmo contenha mecanismos para gerir a instanciação e representação dos mesmos. Para tal, cada domínio de uma aplicação contém uma estrutura de dados na qual identifica todos os recursos existentes no seu seio.

Esta estrutura é constituída por um conjunto de entradas, sendo que cada uma representa um recurso ou pseudónimo. Cada entrada é constituída por informação que caracteriza globalmente o recurso (identificação e propriedades), e também por campos auxiliares que representam a instância do recurso, ou seja, o conjunto dos elementos que o constituem.

A representação de um recurso pode ser: completa, sendo constituída pela instância total do recurso; parcial, contendo apenas alguns elementos da instância; nula, possuindo apenas a identificação do recurso.

O espaço de representação de entradas num domínio é dividido em três áreas distintas: recursos fundadores e para-fundadores, recursos aderentes e recursos de sistema.

Para os recursos fundadores, que correspondem aos recursos que fazem parte da sub-árvore de dependências da qual o domínio é raíz, é obrigatória a existência de uma representação completa.

Os recursos para-fundadores correspondem aos recursos criados pelo domínio local no contexto de um recurso ascendente localizado num domínio que não o original. Estes possuem uma entrada permanente no domínio original, podendo conter ou não uma representação da instância do recurso. A sua instância está permanentemente representada no domínio que hospeda o seu ascendente.

Já para os recursos aderentes, que correspondem aos pseudónimos dos recursos que aderiram a grupos locais aos domínios onde foram instanciados, a sua representação pode ser completa ou parcial, mediante a natureza elaborada ou trivial dos seus elementos.

Os recursos de sistema são constituídos pelos recursos que necessitaram de ser replicados para interagirem com outros recursos fora do domínio do pseudónimo original. A

representação destes recursos pode ser parcial ou completa mediante as propriedades dos seus elementos.

Foi com base neste último grupo de recursos que surgiu o conceito de persistência. Este conceito está relacionado com a persistência da entrada de um recurso nas estruturas de um domínio. Os recursos de sistema não são persistentes, sendo que apenas estão representados enquanto forem necessários. Pelo contrário, os restantes tipos de recursos são persistentes, pelo que as suas representações não podem ser desalocadas, a menos que os recursos sejam explicitamente eliminados pelo utilizador.

3.1.4 *Bibliotecas dinâmicas*

Os domínios delimitam uma região do espaço de computação e comunicação onde coabitam o código que suporta o interface de programação, as primitivas do sistema e as funções definidas nos programas do utilizador. Quando um domínio é criado, é-lhe associado o código que o programador solicitou, sob a forma de bibliotecas dinâmicas. Da mesma forma, os grupos, enquanto domínios virtuais, podem acrescentar, em tempo de execução, novos módulos de código ao domínio da sub-árvore de dependências a que pertencem.

Os executores são as entidades ativas responsáveis pela execução das funções definidas no código dos programas, escalonados como fios de execução pelo sistema. É através do nome da função que lhes foi atribuída que estes procedem à sua atividade computacional, tendo assim de existir um mecanismo que traduza o nome da função ao correspondente apontador que a representa.

Assim, a utilização de bibliotecas dinâmicas mostra-se de elevada utilidade, uma vez que o seu interface permite efetuar, em tempo de execução, a conversão direta entre os símbolos do programa e os respetivos apontadores. Diferentes módulos ou programas da aplicação podem, assim, ser compilados e carregados dinamicamente à medida que vão sendo necessários. Estas bibliotecas também permitem partilhar o mesmo código por vários programas, reduzindo o espaço de memória ocupado pelo código em tempo de execução.

As bibliotecas dinâmicas seguem o formato *Executable and Linking Format* (ELF) [5], que é um formato de ficheiro *standard* adotado em sistemas Unix, e também por alguns sistemas não Unix. Este formato caracteriza-se por ser flexível, extensível e multi-plataforma, permitindo assim que aplicações CoR se adaptem facilmente a ambientes de computação e sistemas operativos heterogéneos.

3.2 MODELO E POLÍTICA DE REPRESENTAÇÃO

Em CoR, o programador tem a possibilidade de selecionar o modelo de representação que melhor se adequa aos requisitos de execução, paralelismo e distribuição da sua aplicação,

entre os quais: cópia única, replicação e partição. Com a decomposição do recurso em elementos básicos, introduziu-se maior flexibilidade no modelo, já que um recurso pode ser particionado em relação a um elemento, e uno em relação a outro.

Outra possibilidade oferecida ao programador prende-se com a escolha da política de representação mais adequada a cada elemento do corpo de um recurso. De entre as políticas constam-se a política centralizada, que dita a existência de apenas uma única representação em todo o sistema (cópia única), e a gestão distribuída, que considera a existência de múltiplas representações (réplicas) disseminadas pelo sistema. De notar que para um modelo de representação particionado, cada uma das políticas é aplicada em relação a cada uma das partes.

3.2.1 *Política centralizada*

A política de representação centralizada caracteriza-se pela existência de apenas uma representação de um determinado recurso em todo o sistema de domínios distribuídos. Esta política tem as vantagens de utilizar menos memória para a representação do recurso no sistema, e também de os acessos ao recurso serem feitos no domínio onde este está representado, facilitando assim o controlo dos acessos concorrentes.

Num sistema de domínios distribuídos, uma única cópia dos dados implica a existência de um único ponto de acesso. Assim, todos os acessos para leitura e escrita, locais ou remotos, são solicitados ao domínio onde o recurso está representado, podendo provocar uma sobrecarga extra nesse domínio e, consequentemente, afetar o desempenho global da aplicação.

No entanto, em certas situações esta política pode ser útil para as aplicações, nomeadamente quando a quantidade de interações entre domínios não é significativa. Outra situação vantajosa para esta política está relacionada com a escassez de memória física nos nós da infraestrutura de computação.

3.2.2 *Política distribuída*

A existência de múltiplas réplicas de recursos, disseminadas pelo sistema de domínios distribuídos, está relacionada com uma política de representação distribuída. A criação de réplicas ocorre através de uma operação de adesão do recurso a grupos não pertencentes ao domínio local, ou então aquando de uma referência a um recurso não representado no domínio local.

A construção de réplicas é uma facilidade que permite, aos executores, a manipulação direta para leitura e escrita de dados disseminados num ambiente distribuído. Esta facilidade introduz as questões relevantes da *Distributed Shared Memory* (DSM) [6], como a garantia

da coerência das múltiplas representações e os protocolos de invalidação e atualização dos dados.

A possibilidade de criação de réplicas locais nos domínios onde o mesmo recurso é utilizado, pode trazer benefícios a nível de desempenho, principalmente se os acessos forem maioritariamente para operações de leitura.

3.3 MODELO DE CONSISTÊNCIA

Estando o CoR assente num ambiente de recursos partilhados globalmente, é fundamental definir um modelo de consistência para definir o comportamento do sistema em relação a operações de escrita sobre os recursos. Os modelos de consistência são fundamentais não só para especificar a semântica das operações e o funcionamento interno do sistema, mas também para auxiliar a escrita de programas consistentes e coerentes.

É importante evidenciar que o CoR oferece um sistema de memória orientado ao objeto, na medida em que cada elemento da anatomia do recurso é um objeto partilhado, mais concretamente um objeto de memória, sobre o qual atuam os mesmos modelos e os mesmos protocolos de gestão.

3.3.1 *Objeto de consistência*

Em CoR, aquando da instanciação de recursos, é-lhes associado um objeto de consistência, que possui um conjunto de mecanismos para garantir a sua consistência e a integridade. A cada elemento do corpo de um recurso, uma vez que correspondem a entidades autónomas, é-lhes associado um objeto de consistência derivado.

O objeto de consistência principal atua como um mediador do acesso ao recurso, e também aos seus elementos constituintes. No entanto, o acesso aos elementos pode ser tratado diretamente pelo objeto de consistência derivado, sendo que o acesso ao objeto de consistência principal apenas é necessário quando ocorre uma alteração a algum valor que afete a globalidade do recurso.

Este objeto tem de garantir a consistência dos diferentes pseudónimos de um recurso, tanto ao nível dos acessos concorrentes locais ao domínio onde reside a representação do objeto, como também nos acessos distribuídos ao recurso com múltiplos pseudónimos representados em diferentes domínios, eventualmente em diferentes nós de computação.

Assim sendo, a funcionalidade do objeto de consistência terá de ser alargada, de modo a contemplar mecanismos que garantem a coerência global do recurso, tanto ao nível do domínio local como nas várias representações distribuídas por diferentes domínios. Esta garantia é exequível através da inclusão de um objeto de consistência em cada representação

do recurso, responsável pela sequenciação dos acessos locais e pela sincronização com os demais objetos de consistência dos pseudónimos do recurso.

3.3.2 *Testemunho de escrita*

A técnica de passagem de testemunho é uma técnica muito utilizada em sistemas DSM, que consiste na obtenção prévia do direito de escrita, sob a forma de um testemunho, a fim de realizar uma operação de alteração de estado. A conclusão da operação de escrita despoleta um pedido de invalidação ou atualização das restantes réplicas. É desta forma que o modelo garante que, após uma operação de escrita, todas as operações de leitura posteriores têm acesso aos valores atualizados.

A ocorrência de uma escrita suspende as leituras sobre a cópia local e, enquanto a escrita não terminar e os dados não forem invalidados, as operações de leitura sobre as outras réplicas do sistema podem ser processadas. Assim, este modelo adotado em CoR é um modelo de consistência relaxado, sendo disponibilizadas as políticas um escritor/múltiplos leitores e um escritor/um leitor.

Direitos sobre o testemunho

Mesmo tendo em posse o testemunho, uma operação de escrita só pode prosseguir se não existirem, no momento, operações de leitura locais sobre o mesmo objeto de memória, obrigando assim a uma operação de aquisição do direito de escrita. Após a conclusão da operação é necessário proceder à libertação do direito de escrita, despoletando um processo de invalidação ou atualização das restantes réplicas.

De forma a que o sistema controle as operações de leitura sobre os objetos de memória, é necessário que as operações de leitura sejam precedidas de pedidos de aquisição, tal como acontece para as operações de escrita. Da mesma forma, assim que já não seja necessário o acesso para leitura ao objeto, é feita a respetiva libertação do direito de leitura.

Controlo do testemunho

A primeira localização do testemunho de escrita corresponde ao domínio original do recurso ou da parte, na qual foi instanciado. No entanto, com o desenrolar da aplicação e da execução de operações de escrita sobre o recurso, o testemunho pode estar localizado em qualquer domínio que contenha réplicas do recurso, ou então mantido no domínio original.

No primeiro caso, denominado por testemunho errante, o testemunho é deslocado para os domínios na origem de pedidos de escrita sobre qualquer uma das réplicas existentes, sendo que a última localização do testemunho coincide com o domínio na origem da última operação de escrita. No segundo caso, denominado por testemunho fixo, o testemunho está sempre presente no domínio na origem da criação do recurso, sendo cedido a outros

domínios através de operações de aquisição do direito de escrita, retornando novamente ao domínio de origem assim que é libertado o direito de escrita.

O comportamento errante tem a vantagem de poderem ser executadas operações de escrita consecutivas no domínio que possui o testemunho, sem necessitar de ser solicitado novamente. No entanto, como o testemunho não está fixo a um domínio, a obtenção do testemunho é precedida pelo envio de um pedido a todo o sistema de domínios distribuídos, a fim de obter a localização do testemunho. O mesmo acontece para determinar, a qualquer momento, quem detém a cópia atualizada para garantir a consistência das várias réplicas.

Já a política de controlo por testemunho fixo tem a vantagem de o testemunho se encontrar sempre no domínio do pseudónimo original. Além do mais, no processo de devolução do testemunho ao domínio original, os dados atualizados podem também ser enviados e atualizados ao mesmo tempo, facilitando todo o processo de atualização de réplicas inválidas. No entanto, a devolução obrigatória do testemunho ao domínio original pode conduzir à sobrecarga do sistema, considerando, por exemplo, operações consecutivas de escrita evocadas no contexto do mesmo domínio, que requerem aquisição prévia do testemunho de escrita.

3.3.3 *Protocolo de invalidação e de atualização*

A libertação do direito de escrita resulta em uma de duas alternativas: invalidação ou atualização dos dados. No primeiro caso, são enviadas mensagens aos detentores das restantes réplicas para procederem à invalidação das mesmas, sendo só atualizadas mediante uma operação de leitura/escrita posterior. No segundo caso, são despoletadas mensagens para procederem à invalidação das réplicas, contendo também os dados atualizados para proceder à atualização das réplicas.

Aquando da atualização de uma réplica, é despoletado um pedido de escrita sobre a réplica local com prioridade superior aos restantes pedidos. No caso do protocolo de invalidação, o objeto de memória é apenas marcado como inválido, sendo que a próxima operação sobre a réplica vai gerar um pedido de atualização.

Após a invalidação ou atualização de uma réplica local, é enviada uma mensagem ao domínio que gerou o pedido a confirmar a concretização da operação solicitada com sucesso. Da mesma forma, assim que o domínio que gerou o pedido receber a confirmação por parte dos restantes domínios que contêm réplicas, informa o escritor atual que o processo de invalidação ou atualização está concluído.

Ambas as técnicas têm vantagens e desvantagens, pelo que a escolha da melhor técnica para o problema em questão é fundamental. Quando os dados dos recursos são lidos ou escritos com muita frequência, a partir de domínios distintos, ou o tamanho da informação a transferir é reduzido, a atualização imediata das réplicas pode favorecer o desempenho

global do sistema. Pelo contrário, quando o acesso aos recursos é esporádico, ou quando a informação a transferir é volumosa, o protocolo de invalidação tende a ser mais vantajoso.

3.4 COMUNICAÇÃO POR PASSAGEM DE MENSAGENS

O modelo de comunicação CoR assume também que qualquer recurso pode enviar mensagens para outro qualquer recurso, localizado num qualquer domínio do sistema. No entanto, a receção de mensagens apenas pode ser efetuada se o recurso destino possuir, no seu corpo, um elemento caixa-postal. Caso o recurso não possua uma caixa-postal associada, as mensagens serão descartadas.

Em CoR, é possível construir aplicações num ambiente de comunicação por passagem de mensagens, sendo este garantido através da utilização de uma biblioteca que implemente um padrão de comunicações de dados, como por exemplo o *Message Passing Interface* (MPI) [7]. Assim, é disponibilizado ao programador a possibilidade de a emissão/receção de mensagens poder ser efetuada em modo bloqueante, esperando até que a operação seja concluída, ou então em modo imediato, retornando logo que a operação é iniciada, podendo ser inquirido o estado da operação a qualquer momento.

3.5 PROGRAMAÇÃO EM MEMÓRIA PARTILHADA

O modelo CoR foi proposto como um modelo de programação híbrido que combina os paradigmas da programação por passagem de mensagens com a programação com múltiplos fios de execução. O paralelismo de baixa escala é explorado, em CoR, através do aumento de concorrência dentro da mesma unidade de processamento, procurando assim benefícios ao nível do desempenho global do sistema. Como o modelo foi proposto numa altura em que o *multi-core* ainda estava na sua alvorada, este ainda não contempla os conceitos atuais da programação por memória partilhada.

Ao nível da programação em memória partilhada, os primeiros conceitos remontam à construção de uma *thread-centric pool*, na qual a cada fio de execução é feito corresponder estritamente uma tarefa, sendo possível saber, a cada momento, o que qualquer fio de execução do grupo de trabalho está a executar. Este é o modelo tradicional presente no OpenMP [8], designado por modelo de programação *thread-centric*.

Um outro modelo de programação alternativo foi proposto inicialmente pelo Cilk [9], no qual um qualquer número de tarefas paralelas são executados por um número fixo de fios de execução, sendo que a execução é controlada através de mecanismos implementados por um escalonador de tarefas. Este modelo de programação é denominado por *task-centric*, e é parte integrante das bibliotecas de software Intel TBB [10] e OpenMP. Este modelo é utilizado, sobretudo, para explorar o paralelismo em algoritmos irregulares, recursivos ou não

balanceados, algo que o seu congénere não consegue solucionar ou explorar devidamente, devido ao foco no fio de execução, ao invés da tarefa em si.

Particularmente, um conceito pilar ao modelo *task-centric* é o construtor *taskgroup*, que permite um estilo de programação o qual uma tarefa lança paralelamente um conjunto de tarefas recursivas, continua a sua execução e, no fim, aguarda pelo término da região paralela. É com base neste conceito que alguns algoritmos recursivos, não balanceados e não estruturados, são passíveis de serem paralelizados. Assim, é importante explorar e incluir este conceito no modelo CoR, assim como introduzir o paradigma *thread-centric* e outros conceitos como afinidade de fios de execução e memória, associados à programação em memória partilhada. Esta introdução de novos conceitos leva, necessariamente, à inclusão de novos elementos e, consequentemente, novos recursos no modelo.

4

PROTÓTIPO PLACOR

Este trabalho foi essencialmente dirigido para a construção do protótipo PlaCoR, como um ambiente de programação e execução de aplicações baseadas no modelo CoR. O trabalho atual não é a primeira tentativa de criar esta plataforma, tendo já havido dois esforços anteriores baseados no mesmo modelo, nomeadamente o pCoR [11] e o pyCoR [12].

O primeiro protótipo, pCoR, que acompanhou a evolução do projeto de conceção do modelo, foi construído como uma biblioteca de funções C, suportado pelo PVM [13] e o DoTS [14] ao nível da infraestrutura. O PVM, máquina paralela virtual, disponibiliza facilidades para a gestão de múltiplos nós de computação heterogéneos, enquanto o DoTS é uma camada lógica que combina a programação com múltiplos fios de execução, memória partilhada e passagem de mensagens, na linha do TPVM [15].

O pCoR é um sistema robusto que permite o desenvolvimento e execução de aplicações paralelas e distribuídas, com um elevado número de tarefas concorrentes e com um grão fino de paralelismo. Este foi descontinuado por duas razões principais: por ser altamente dependente do PVM, cuja evolução foi por sua vez interrompida; e por apenas implementar um conjunto simples de conceitos do modelo CoR.

Um segundo esforço levou à construção do protótipo pyCoR que, tal como o nome indica, foi construído na linguagem Python. É uma abordagem à idealização e construção de uma plataforma, num paradigma orientado ao objeto, com a premissa de apenas ficar dependente de bibliotecas de terceiros em casos muito específicos. Em termos efetivos, apesar de a nível de conceitos e facilidades realizadas não ter ido muito mais longe do que o seu congénere pCoR, inspirou fortemente a construção do protótipo PlaCoR.

Neste trabalho, numa primeira fase, procurou-se instanciar a visão proposta pelo pyCoR, uma vez que os conceitos usados tinham sido validados, sem deixar de ter presente a implementação pCoR. Seguiu-se-lhe, naturalmente, a inclusão de outras propriedades do modelo que até ao momento não tinham sido realizadas.

4.1 APRESENTAÇÃO DO PROTÓTIPO PLACOR

O PlaCoR é um projeto em desenvolvimento especialmente dedicado à exploração do paradigma da computação orientada ao recurso. Apesar de não corresponder a uma implementação completa da especificação CoRes, é um sistema que fornece ao programador um modelo de programação híbrido que combina o paralelismo tradicional associado aos processos (domínios), com o paralelismo de grão fino dos fios de execução (agentes) e a memória partilhada distribuída.

O protótipo foi instanciado como uma biblioteca de classes em C++ Moderno [16], mais propriamente na versão padrão C++17. A escolha desta linguagem trouxe enormes vantagens para o desenvolvimento do protótipo, com destaque para o suporte à: programação orientada aos objetos - usada para a construção dos recursos assente em mecanismos de herança múltipla; programação genérica - por forma a abstrair na API as diferentes classes de recursos; programação concorrente - para tirar partido dos fios de execução e estruturas de sincronização nativas da própria linguagem.

A ampla utilização do C++ e a premissa de apenas recorrer a ferramentas externas à linguagem em casos muito específicos, garante uma maior portabilidade do protótipo para diferentes ambientes. De momento, o protótipo pode ser instalado e utilizado nos ambientes Linux, Solaris e MacOS.

O desenho do PlaCoR, como uma plataforma de domínios distribuídos, está em alta medida dependente das facilidades existentes para a interação entre processos. É por esta razão que, a nível de sistema, assume um relevo especial a utilização da biblioteca `Spread` [17] pelas garantias que oferece. Esta implementa mecanismos robustos de comunicação por grupo, resiliente a falhas em redes internas ou externas, essenciais na construção de aplicações distribuídas escaláveis e confiáveis.

A utilização do `Spread` foi fundamental no desenvolvimento do protótipo, nomeadamente na programação dos componentes que sustentam a interação entre os recursos e a gestão da consistência entre réplicas. Uma outra razão para a utilização do `Spread` é o suporte para múltiplas plataformas Unix.

4.1.1 *Recursos e elementos*

O comportamento dos recursos é integralmente dedutível das propriedades dos elementos (classes) que compõem a sua hierarquia, a saber: contentor, organizador dinâmico e estático, executor, caixa-postal, valor, barreira, guarda e guarda para leituras/escritas. Em relação à especificação da anatomia do recurso em CoRes, foram acrescentados os elementos: organizador estático - cria uma contexto fixo em termos de número de membros - e a guarda para leituras/escritas - uma estrutura de sincronização distribuída usada para criar

regiões de exclusão mútua para leituras e escritas. Assim, cada um dos diferentes tipos de recursos é construído com base na seleção das classes de elementos mais apropriadas, de entre as disponíveis, de forma a obter a funcionalidade requerida. A generalização do conceito de recursos é alcançada incluindo, obrigatoriamente, a classe Recurso como uma das suas classes de base.

Presentemente, a plataforma disponibiliza as seguintes classes de recursos: domínio, grupo, clausura, agente, proto-agente, dado, barreira, guarda e guarda para leituras/escritas. Por comparação com a especificação CoRes, foram acrescentados os recursos clausura, proto-agente e guarda para leituras/escritas. O recurso clausura, intimamente ligado com o elemento organizador estático, foi introduzido com o intuito de criar condições para a existência de grupos estáticos, através de operações coletivas, garantindo a consistência das operações executadas no seu contexto. O recurso tarefa previsto em CoRes deu lugar ao agente em PlaCoR, enquanto que o proto-agente corresponde a um recurso agente sem as facilidades de comunicação.

4.1.2 *Identificação de recursos*

Em PlaCoR, os recursos possuem uma identificação global (idp) e uma identificação contextual, através de um identificador de membro (idm) e, opcionalmente, um nome no contexto do seu recurso ascendente. De referir que, em geral, na API (ver capítulo 5) a assinatura dos métodos usa o idp, sendo a identificação contextual utilizada no caso específico de recursos que incluem um elemento da classe organizador - dinâmico ou estático. Obviamente que o acesso através do idm pressupõe o conhecimento do idp do seu ascendente.

A criação de um recurso implica, naturalmente, a especificação do seu ascendente natural, obrigatoriamente um recurso com um elemento do tipo organizador. A todo o momento, é possível a um recurso já existente tornar-se membro de um outro ascendente, local ou remoto, o que se traduz na geração de um novo idp pseudónimo, e de um novo idm e nome no contexto do seu novo ascendente. No caso do ascendente ser remoto, a API inclui funcionalidades para a criação de réplicas locais, enquanto o sistema assegura automaticamente a consistência entre todas as réplicas, bem como a libertação automática das suas representações locais.

4.1.3 *Árvore de dependências*

O conhecimento da estrutura de uma aplicação, representada pela respetiva árvore de dependências, pode constituir um mecanismo importante para explorar o paradigma da orientação ao recurso. Assim, em tempo de execução, cada aplicação organiza logicamente, numa árvore de dependências, todos os recursos através da associação entre o idp do re-

curso e o idp do respetivo ascendente. A raiz da árvore corresponde ao meta-domínio da aplicação que, por sua vez, é também um domínio cujos membros são todos os domínios da aplicação, incluindo o próprio meta-domínio.

A criação de recursos estruturados, que possuem um elemento organizador na sua hierarquia, introduz novos caminhos na árvore de dependências, onde novos recursos se podem pendurar. Os recursos estruturados correspondem aos nodos da árvore de dependências, enquanto que os recursos simples, ou seja, que não possuem um elemento organizador na sua hierarquia, correspondem às folhas. A operação de junção de um recurso já existente a um outro recurso estruturado, corresponde à criação de uma nova relação de ascendência na árvore de dependências, identificada pelo idp atribuído ao pseudónimo do recurso neste novo contexto.

4.1.4 *Gestão dos recursos*

A plataforma apenas considera a existência de recursos triviais, isto é, que possuem uma representação única do corpo do recurso (ver 3.2). É possível a existência de múltiplas réplicas por recurso, disseminadas pelo sistema de domínios distribuídos, em que cada réplica possui um único objeto de consistência (ver 3.3.1), usado para garantir a consistência entre todas as réplicas. Neste contexto, foi adotado o modelo de consistência relaxado um escritor/múltiplos leitores como política de gestão de réplicas. Para o testemunho de escrita, foi usado o comportamento errante (ver 3.3.2) com o protocolo de invalidação (ver 3.3.3).

4.1.5 *Escalonamento de domínios*

O domínio corresponde à principal entidade presente no modelo, delimitando uma região do espaço de endereçamento onde ocorrem as interações entre recursos. Este tem uma correspondência direta com a plataforma de computação subjacente, já que traduz os módulos do utilizador em processos, estabelecendo o primeiro nível de concorrência/paralelismo. Este, além de conter o código do utilizador, contém um conjunto de serviços que se comprometem com a gestão da execução da aplicação, bem como sustenta a existência dos recursos na sua sub-árvore de dependências.

Os domínios são escalonados estaticamente no arranque das aplicações através do comando apropriado, ou então dinamicamente através de um método que permite lançar novos domínios na aplicação, com o mesmo ou um novo módulo do utilizador. Em PlaCoR, está previsto o arranque de aplicações do tipo SPMD, no qual o utilizador tem de especificar o número total de domínios que pretende arrancar, bem como especificar os nós de computação que pretende utilizar.

Os nós de computação a atribuir à aplicação têm de estar definidos num ficheiro de *hosts*, a atribuir no momento de arranque da aplicação, sendo estes identificados por um endereço IP ou um nome em DNS. A política de escalonamento adotada é de rotatividade pelos nós especificados, no qual é estritamente necessário que estes contenham um *deamon* `Spread` a correr. Estes têm também de pertencer à mesma rede `Spread`, de forma a que seja possível estabelecer comunicação entre as máquinas especificadas.

4.1.6 *Infraestrutura de comunicação*

O protótipo pCoR utilizava o PVM não apenas como infraestrutura para gestão da máquina paralela virtual, mas também para a comunicação por passagem de mensagens entre os processos. Em PlaCoR, o `Spread` é usado para disponibilizar mecanismos equivalentes aos do PVM, sendo a base para a comunicação inter-domínios e a passagem de mensagens entre recursos comunicantes.

Presentemente, o protótipo apenas contempla primitivas de comunicação bloqueantes, entre recursos que possuam as propriedades inerentes ao elemento caixa-postal. Existem dois tipos de comunicação: comunicação global entre recursos através do identificador principal; comunicação por contexto entre membros de recursos que possuem um elemento organizador estático.

Em relação à comunicação global, o envio de mensagens requer a identificação do destinatário através do idp, sendo possível especificar um ou mais destinatários. Para a receção de mensagens é possível especificar o remetente, e caso se prescinda do mesmo é aguardada a chegada de uma qualquer mensagem de um qualquer remetente. Já em relação à comunicação por contexto, esta é realizada através da utilização do idm do destinatário ou remetente, em conjunto com o idp do recurso com organizador estático. A comunicação ocorre então num contexto estático e fechado, entre os recursos membros de clausuras. É possível também difundir mensagens para todos os seus membros, utilizando apenas o idp do recurso clausura.

Uma característica distintiva da passagem de mensagens em PlaCoR, é a possibilidade de as mensagens poderem incluir, além dos tipos de base da linguagem, instâncias de classes definidas pelo programador, desde que sejam suportadas pela biblioteca de serialização `cereal`. Notar que esta biblioteca tem suporte nativo para uma parte significativa dos tipos de dados disponibilizados pela *Standard Library* do C++.

4.2 ARQUITETURA DO SISTEMA

A arquitetura do protótipo PlaCoR, explicitada na figura 4, está centrada no *pod*[1], entidade responsável ao nível do sistema operativo pela memória, os meios de computação e de comunicação, necessários à interação entre os diversos recursos (e consequentemente elementos) do sistema de domínios distribuídos. O *pod*, enquanto núcleo do sistema, vive como um processo no sistema de computação, dotado de um conjunto de serviços que suportam os recursos e a execução de aplicações. Esta entidade está intimamente relacionada com o recurso domínio, pois fornece a este toda a funcionalidade requerida para representar e permitir a interação entre as diversas entidades do modelo.

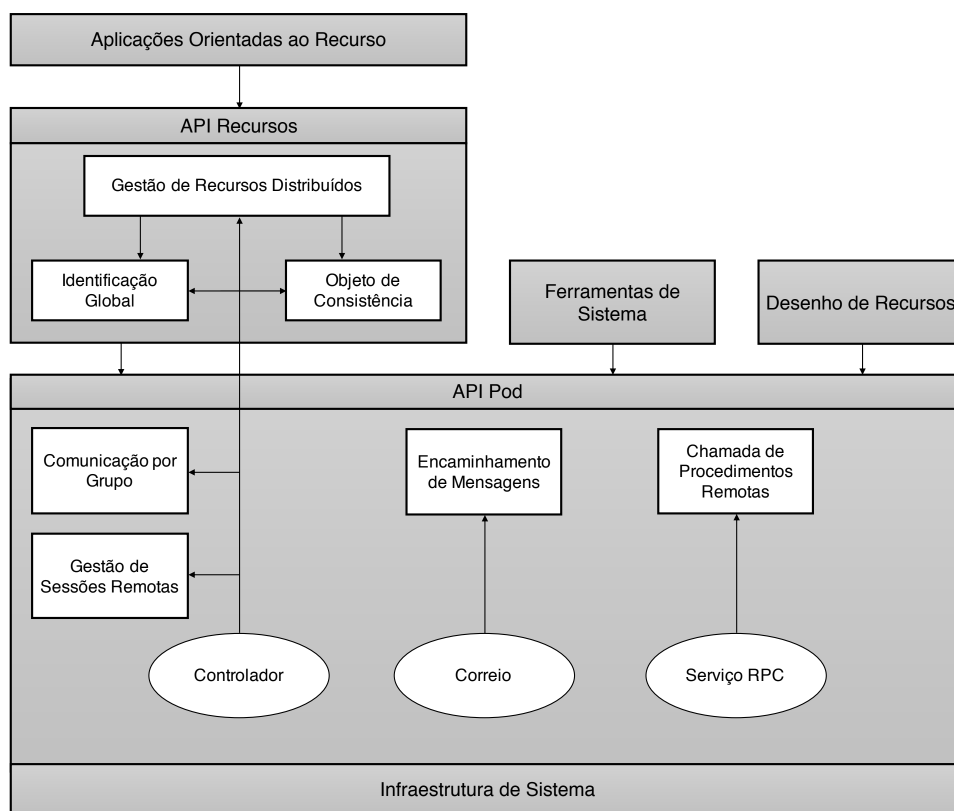


Figura 4.: Arquitetura geral do protótipo PlaCoR.

A plataforma é formada por diversos sub-sistemas independentes, organizados em camadas hierárquicas, cada um dedicado a um serviço específico. Estes níveis vão desde a camada de sistema, na qual estão presentes as várias bibliotecas associadas à construção do protótipo, até ao nível de gestão do sistema e de interface com a aplicação do utilizador.

1 o termo pod, recetáculo em português, tem origem num conceito usado em TPVM com um propósito semelhante.

De um modo geral, as aplicações são construídas com base na interface de programação disponibilizada pelos demais recursos presentes no protótipo, constituindo a API dos recursos. Este disponibiliza também ferramentas para o arranque das aplicações, bem como para desenhar novos elementos e, consequentemente, novos recursos a introduzir na plataforma.

Pod

No *pod* coabitam os fios de execução do utilizador e do sistema. Os fios de execução do utilizador são responsáveis pela execução do código dos programas do utilizador. Já os fios de sistema são responsáveis pela execução das tarefas de gestão do sistema de domínios distribuídos.

No entanto, de forma a tirar partido da partilha do espaço de endereçamento dentro de cada domínio, a maior parte das operações e dos serviços locais podem ser diretamente executados pelos fios de execução do utilizador, evitando a sobrecarga de escalonamento de fios de sistema para responderem a tarefas momentâneas. Os fios de execução de sistema, pela importância que assumem na gestão do sistema de domínios distribuídos, perduram durante todo o tempo de vida de uma aplicação.

Controlador

O fio de execução controlador estabelece a interligação entre os vários domínios presentes na aplicação, tendo ao seu dispor o seguinte conjunto de serviços:

- *serviço de identificação global*, que é responsável pela geração de identificadores para os recursos criados dinamicamente durante a execução da aplicação;
- *serviço de gestão de recursos distribuídos*, que gere a representação dos diferentes recursos no seu espaço de endereçamento, incluindo a gestão de réplicas e a gestão de consistência das múltiplas representações dos diversos recursos e seus elementos constituintes, em todo o sistema de domínios distribuídos, através do objeto de consistência;
- *serviço de gestão de sessões remotas*, que permite o lançamento dinâmico de novos *pods* e, consequentemente, domínios em máquinas locais ou remotas, ou seja, o lançamento dinâmico de novas aplicações.

Correio

Um outro fio de sistema é o correio, que é responsável pela receção e encaminhamento de mensagens, de comunicação ponto-a-ponto prevista pelo CoRes, quer se trate de mensagens entre recursos internos ao domínio ou remotos.

Serviço RPC

Um terceiro fio de sistema corresponde ao serviço RPC, relacionado com a criação de recursos em domínios remotos, bem como com a execução, espera e obtenção do resultado das funções dos recursos ativos remotos (que possuam um elemento executor), essencial para corresponder às funcionalidades impostas pelo modelo.

4.3 FERRAMENTAS DE DESENVOLVIMENTO

A conceção e desenvolvimento do protótipo foi substancialmente influenciada pela necessidade de selecionar um conjunto de pacotes de *software*, de forma a responder a problemas específicos da plataforma. Estes estão explicitados na figura 5.

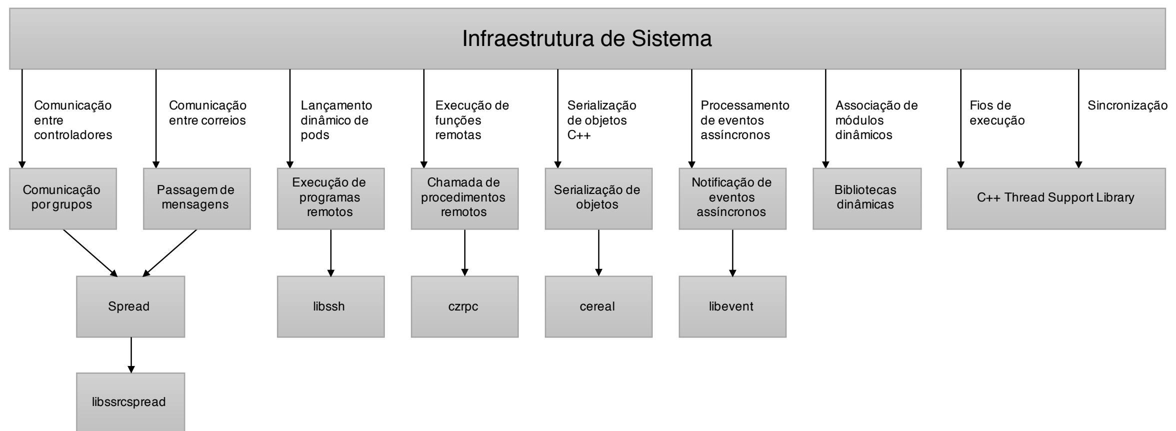


Figura 5.: Ferramentas de desenvolvimento utilizadas no protótipo PlaCoR.

Uma das questões centrais está relacionada com a obrigatoriedade de garantir a comunicação confiável entre os controladores. Dada a complexidade do tema e a dificuldade de desenho e projeto das facilidades necessárias, recorreu-se a uma biblioteca que suporta mecanismos de comunicação por grupo, o `Spread` [18]. Esta biblioteca também dispõe de mecanismos de passagem de mensagens (*unicast*), que são usados para a comunicação explícita por mensagens entre recursos. Com a utilização do `Spread`, foi necessária a utili-

zação de uma biblioteca para o processamento de eventos assíncronos, motivo pelo qual foi utilizada a biblioteca `libevent` [19].

O modelo CoR pressupõe que qualquer recurso possa ser instanciado em qualquer domínio presente no sistema de domínios distribuídos. Para tal, foi necessária a construção de um serviço para chamadas de funções remotas, de forma a permitir a criação de recursos remotamente, no qual foi utilizada a biblioteca `czrpc` [20]. Esta foi também utilizada para permitir executar, esperar e obter os resultados das funções executadas por agentes em domínios remotos.

Outra das problemáticas presentes na plataforma está relacionada com a criação de réplicas de recursos no sistema de domínios distribuídos, ou seja, em diferentes processos. Para tal, é necessário efetuar a serialização dos diversos recursos, representados por classes C++, pelo que se optou pela utilização da biblioteca `cereal` [21] para o efeito pretendido.

Por fim, e de forma a facilitar o lançamento e execução de aplicações em nós de computação distintos, procurou-se utilizar uma biblioteca que implementa-se o protocolo SSH. Assim, é possível criar conexões seguras em máquinas remotas, permitindo executar aplicações. Para este caso, a escolha recaiu na biblioteca `libssh` [22].

4.3.1 *Spread*

O `Spread` é uma ferramenta que implementa os mecanismos de comunicação por grupo para a construção de sistemas distribuídos escaláveis, na qual a robustez e a confiabilidade do sistema são aspetos imprescindíveis. É através da comunicação por grupo que esta ferramenta interliga e sincroniza os diferentes membros de grupos de uma aplicação distribuída, permitindo saber o estado atual dos diferentes componentes, locais ou remotos, possibilitando assim a deteção de falhas.

Esta ferramenta fornece um serviço de mensagens resiliente a falhas, sobre redes internas ou externas, com suporte a comunicação por grupo, e comunicação *unicast* e *multicast*. Este serviço fornece ainda diferentes níveis que controlam o tipo de ordem e de confiabilidade atribuídos às mensagens, que vão desde sem garantias de ordem e de entrega das mensagens, até à garantia de ordem e da entrega das mensagens, mesmo no caso de falhas.

4.3.1.1 *libssrcspread*

A biblioteca `libssrcspread` [23] fornece um conjunto de *bindings* C++ para a biblioteca de comunicação por grupos `Spread`. Esta foi utilizada não apenas para permitir a utilização da biblioteca `Spread`, escrita em C, usando a orientação aos objetos, mas também para facilitar a integração do próprio `Spread` no protótipo. Esta trouxe vantagens significativas no tratamento de erros, na manipulação das estruturas de dados e na gestão de tampões de mensagens.

4.3.2 *cereal*

O `cereal` é uma biblioteca de serialização escrita no padrão C++11, que dispõe de facilidades para a serialização de tipos de dados arbitrários para representação binária, XML ou JSON. A mais valia desta biblioteca, que determinou a sua escolha entre outras opções (como a Boost), é: o suporte a herança e polimorfismo, permitindo ao utilizador serializar as suas próprias classes de dados; a possibilidade de serializar grande parte dos tipos de dados presentes na C++ Standard Library; ser facilmente integrável, pelo facto de ser *header-only* e não possuir dependências externas; poder ser usada em ambientes *multithreaded*.

4.3.3 *libssh*

A criação local e remota de *pods* como suporte para o desenho e execução de aplicações baseadas em sistemas de domínios distribuídos previstos no CoRes, estão na base da necessidade de utilização de facilidades para abrir conexões em máquinas remotas, executar aplicações e obter os resultados produzidos local/remotamente.

A biblioteca `libssh` usa o protocolo SSH, como base para a execução remota de programas e/ou transferência de ficheiros. Possui a vantagem de esconder os detalhes técnicos do protocolo SSH, disponibilizar uma API de fácil utilização, e poder ser usada num ambiente *multithreaded*, como é o caso do protótipo PlaCoR, mediante algumas restrições expostas na documentação.

4.3.4 *libevent*

A biblioteca `libevent` fornece um sistema de notificação de eventos assíncronos, com mecanismos para executar funções *callback* sempre que ocorrem eventos específicos em descritores de ficheiros, que o programador pretende tratar. A sua utilização na plataforma está especialmente concentrada nos mecanismos de envio e receção de mensagens, em íntima relação com biblioteca `Spread`, usada intensa e extensamente pelos serviços de sistema do protótipo.

As vantagens associadas a esta biblioteca estão relacionadas com o facto de possuir uma grande capacidade para o tratamento de eventos, bem como o facto de poder ser utilizada em ambientes *multithreaded*, requisito fundamental na conceção do protótipo.

4.3.5 *czrpc*

A biblioteca czrpc surgiu da necessidade de dar suporte à criação remota de recursos, bem como ao arranque explícito das funções de agentes criados remotamente. De entre as

várias soluções existentes para a chamada de funções remotas em C++, esta destaca-se pelas seguintes razões: não necessita de geração de código adicional para realizar chamadas remotas; utiliza C++ moderno para detetar, em tempo de compilação, chamadas a funções com argumentos inválidos; não possui dependências externas; dispões de uma camada de transporte (entre processos) que pode ser adaptada a bibliotecas de comunicação externas.

O facto de ser possível integrar o `Spread` como camada de transporte foi determinante para a escolha da mesma. No entanto, a sua utilização não foi imediata, uma vez que para além da integração com o `Spread`, foi necessário enriquecê-la para poderem ser usados métodos *templated*, recorrentemente empregues na API do protótipo.

4.4 ARRANQUE E EXECUÇÃO DE APLICAÇÕES

Uma aplicação PlaCoR consiste num ou mais módulos escritos na linguagem C++, compilados sob a forma de bibliotecas dinâmicas e ligados à biblioteca PlaCoR. Um destes módulos é obrigatoriamente um módulo de arranque, que se carateriza por ter de definir uma função de entrada com o nome `Main`, com assinatura semelhante à função padrão `main` da linguagem C/C++. Os demais módulos a serem, eventualmente, incluídos na aplicação poderão ou não conter aquela função.

O protótipo disponibiliza o utilitário `corx` como instrumento base para executar uma aplicação, que atua através da criação de um *pod* local e posterior carregamento do módulo de arranque, que executa a função de entrada `Main`. A sua utilização tem o seguinte formato:

```
corx <app group> <context> <number pods> <parent> <module> <args...>
```

Implícito na execução do comando `corx` para o arranque estático de uma aplicação, está a existência de dois contextos, intimamente ligados a grupos de comunicação `Spread`: o primeiro é o contexto da aplicação (`app group`), um grupo `Spread` que irá incluir todos os *pods* que constituem a infraestrutura da aplicação; o segundo é o contexto de arranque (`context`), um outro grupo `Spread` a que pertencem exclusivamente um ou mais *pods* (tantos quanto `number pods`) que arrancaram simultaneamente. Cada um dos *pods* da infraestrutura irá corresponder a um *domínio* CoRes, pelo que uma aplicação pode ser vista como um sistema de domínios distribuídos. A biblioteca dinâmica e os argumentos a passar à função `Main` são especificados em `module` e `args`. As opções `number pods` e `parent` serão introduzidas quando forem apresentadas as facilidades para o arranque de múltiplos domínios (*pods*).

A figura 6 apresenta a notação utilizada para identificar os recursos em PlaCoR, que será utilizada para ilustrar a árvore de dependências das aplicações descritas nesta secção.

Figura 6.: Legenda dos recursos PlaCoR.

4.4.1 *Arranque básico*

O excerto 4.1 mostra o código `module.cpp` de um módulo básico de arranque de uma aplicação que a seguir descrevemos, tomando como referência os respetivos números de linhas do código. Em (1) é incluída a referência ao interface da biblioteca PlaCoR `"cor/cor.hpp"`, enquanto que em (3) é usada a diretiva `extern "C"`, necessária para permitir carregar a função de entrada do módulo (biblioteca dinâmica).

Em (8) é definida a função de entrada do módulo, `Main`. Em relação à sua definição, em (10) é usada a função `cor::GetDomain` da API para obter um apontador para o *domínio* de arranque, presente na variável `domain`, que é a base para interagir com o sistema. Em (11) é obtido o identificador global do recurso que executou o método `GetActiveResourceIdp`, neste caso o idp do *agente* criado para executar a função de entrada do módulo de arranque.

```
1  #include "cor/cor.hpp"
2
3  extern "C"
4  {
5      void Main(int argc, char *argv[]);
6  }
7
8  void Main(int argc, char *argv[])
9  {
10     auto domain = cor::GetDomain();
11     std::cout << domain->GetActiveResourceIdp() << "\n";
12 }
```

Código 4.1.: Módulo de arranque de uma aplicação básica.

Para o arranque da aplicação foi utilizado o seguinte comando, obtendo como resultado:

```
$ corx app ctx 1 0 placor/examples/libmodule.so
4294967038
```

O comando começa por desencadear o lançamento do *pod* local (processo), que estabelece as condições iniciais para a execução da aplicação. Seguidamente, como é possível ver na figura 7: (a) a aplicação arranca com o estabelecimento do *meta-domínio* que representa a própria aplicação; (b) é feita a criação do primeiro *domínio* (neste caso único) e carregado dinamicamente o módulo de arranque, ficando então acessíveis as funções declaradas na diretiva `extern "C"`.

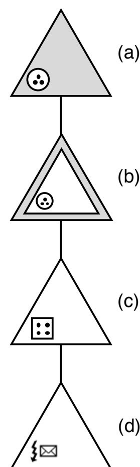


Figura 7.: Árvore de dependências de uma aplicação simples básica.

Após o módulo ser carregado é criada a *clausura* (c) associada ao contexto de arranque da aplicação, e o *agente* (d) que vai executar a função de entrada do módulo, no contexto da *clausura*. O recurso *clausura* está intimamente ligado à criação de um contexto estático (fechado) que inclui apenas os *domínios* que arrancaram simultaneamente, em associação com o *agente* inicial (d). Mais à frente vai ser apresentado um outro exemplo de lançamento de uma aplicação com vários *pods* em simultâneo, na qual cada um cria uma estrutura semelhante por *pod* correspondente aos passos de (b) a (d) da figura. Notar que apenas existe um único *meta-domínio* por aplicação.

4.4.2 *Criação e interação de recursos*

O código apresentado em 4.2 (`basic_operations.cpp`), serve para mostrar as operações básicas recorrentemente usadas para a criação e a interação de recursos e, ao mesmo tempo, apresentar diferentes alternativas para obter os mesmos resultados, tirando partido das

caraterísticas da orientação ao recurso em CoR. Notar que, por simplificação, o código não inclui as linhas (1-7) do exemplo anterior. Convém, desde logo, esclarecer que aqui está implícito a existência de um outro módulo (`callable_module.cpp`), apresentado em 4.3, que irá ser carregado dinamicamente, para posteriormente ser integrado na aplicação original.

```
1  void Main(int argc, char *argv[])
2  {
3      auto domain = cor::GetDomain();
4      auto agent_idp = domain->GetActiveResourceIdp();
5      auto agent = domain->GetLocalResource<cor::Agent<void(int,char**)>>(agent_idp);
6      auto group = domain->CreateLocal<cor::Group>(domain->Idp(), "group", "libcallable_module.so");
7      auto data = domain->CreateLocal<cor::Data<idp_t>>(group->Idp(), "data");
8      auto new_agent = domain->CreateLocal<cor::Agent<idp_t(idp_t)>>(
9          group->Idp(), "agent", group->GetModuleName(), "Test"
10     );
11     new_agent->Run(agent_idp);
12     new_agent->Wait();
13     auto res1 = new_agent->Get();
14
15     auto msg = agent->Receive();
16     auto res2 = msg.Get<idp_t>();
17
18     data->AcquireRead();
19     auto res3 = data->Get();
20     data->ReleaseRead();
21
22     std::cout << res1 << "\t" << res2 << "\t" << *res3 << "\n";
23 }
```

Código 4.2.: Exemplo `basic_operations.cpp`.

Em (3), tal como no exemplo anterior em (10), é iniciada a variável `domain` com um apontador para o *domínio* local. Com este, é obtido em (4) o idp do *agente* local, bem como em (5) um apontador para o objeto que representa o recurso *agente* em execução, através do método `GetLocalResource` com a assinatura da função que este está a executar, neste caso a assinatura da função `Main`. No que segue, é utilizado o método `CreateLocal` para criar os recursos `group` (6), `data` (7) e `new_agent` (8-10), em que o primeiro é criado no contexto do *domínio* local e os restantes no contexto do *grupo* criado.

Para a criação dos recursos é necessário, primeiramente, passar o tipo de recurso a criar (`Group`, `Data<idp_t>` e `Agent<idp_t(idp_t)>`, respetivamente). Em relação aos argumentos, estes incluem: i) idp do recurso ascendente (`domain->Idp()` e `group->Idp()`); ii) nome do recurso no contexto (`"group"`, `"data"` e `"agent"`); iii) argumentos necessários para a criação dos mesmos. O *grupo* necessita de receber o nome do módulo (biblioteca dinâmica) a carregar, neste caso `"libcallable_module.so"`. O *agente* recebe o nome do módulo carregado pelo recurso *grupo*, obtido através do método `GetModuleName`, e o nome da função `"Test"`, que vai carregar dinamicamente e executar. Por fim, o *dado*, como não recebe argumentos para a criação do mesmo, assume um valor por omissão.

Na linha (11) o *agente* ativo executa o método Run do *agente* criado, fazendo com que este execute a função Test com o argumento agent_idp, ficando em (12) à espera pelo término da função, obtendo o valor de retorno da mesma na linha (13) para a variável res1. Em (15) o *agente* ativo fica à espera de receber uma mensagem, via o método Receive, presumivelmente enviada pelo *agente* new_agent, e em (16) usa o método Get para obter o conteúdo da mensagem recebida, neste caso um identificador principal de um recurso guardado na variável res2. Em (18), o *agente* ativo requere o direito de leitura do recurso *dado* criado anteriormente, através do método AcquireRead, obtendo um apontador para o valor através do método Get (16), guardado na variável res3, requisitando posteriormente a libertação do direito de leitura em (20), através do método ReleaseRead. Por fim, em (22), são imprimidos os diferentes valores obtidos pelas três formas diferentes de obter resultados usando o modelo CoR.

A árvore de dependências resultante do código apresentado está representada na figura 8. Por comparação com a árvore correspondente ao módulo de arranque estático, apresentada na figura 7, de referir a junção em (e) do recurso *grupo* (group), no contexto do *domínio* (b) (domain), e a junção em (f) e (g), dos recursos *dado* (data) e *agente* (new_agent), respetivamente, criados no contexto do *grupo*.

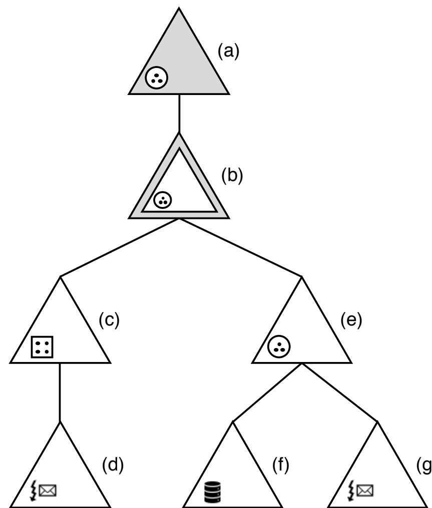


Figura 8.: Árvore de dependências do exemplo basic_perations.cpp.

Em 4.3 é apresentado o código fonte do módulo libcallable_module.so, dinamicamente carregado pelo *grupo* criado no exemplo anterior. Como este módulo pressupõe a disponibilização da função Test, é necessário declarar a mesma na diretiva extern "C" como apresentado nas linhas (1-4).

Em (11), é obtido o idp do recurso ascendente do *agente* ativo que corresponde ao *grupo* criado pelo módulo libbasic_operations.so, através do método GetPredecessorIdp, bem como o objeto que representa o mesmo, em (12). Isto apenas é possível conhecendo a es-

```
1  extern "C"
2  {
3      idp_t Test(idp_t rsc_idp);
4  }
5
6  idp_t Test(idp_t rsc_idp)
7  {
8      auto domain = cor::GetDomain();
9      auto agent_idp = domain->GetActiveResourceIdp();
10
11     auto group_idp = domain->GetPredecessorIdp(agent_idp);
12     auto group = domain->GetLocalResource<cor::Group>(group_idp);
13     auto data_idp = group->GetIdp("data");
14     auto data = domain->GetLocalResource<cor::Data<idp_t>>(data_idp);
15
16     data->AcquireWrite();
17     auto value = data->Get();
18     *value = agent_idp;
19     data->ReleaseWrite();
20
21     cor::Message msg;
22     msg.Add<idp_t>(agent_idp);
23     auto agent = domain->GetLocalResource<cor::Agent<idp_t(idp_t)>>(agent_idp);
24     agent->Send(rsc_idp, msg);
25
26     return agent_idp;
27 }
```

Código 4.3.: Código do exemplo `callable_module.cpp`.

trutura da aplicação, nomeadamente a árvore de dependências da mesma, apresentada na figura 8. O mesmo é válido para obter o apontador para o *dado* criado anteriormente, pois este foi criado no contexto do *grupo*. Utilizando o método `GetIdp` com o nome atribuído a este (`"data"`) na linha (13), é obtido o idp do *dado* para posteriormente obter um apontador para o mesmo, em (14).

Sendo o recurso *dado* um mecanismo para permitir a partilha de dados num sistema de domínios distribuídos, é necessário garantir a consistência dos mesmos. Assim, para proceder a modificações dos dados é necessário garantir a exclusividade no acesso ao mesmo, sendo esta obtida através do método `AcquireWrite`, em (16). Em (17-18) é obtido um apontador para os mesmos, procedendo-se à sua modificação. Por fim, é feita a libertação do direito de escrita em (19), informando o sistema de que foram produzidas modificações ao mesmo, sendo por isso necessário invalidar as possíveis réplicas existentes. É importante referir que a ação sobre os dados ocorre localmente.

Seguidamente, em (21) é criado um objeto `msg`, do tipo `Message`, que representa uma mensagem em PlaCoR. A esta é adicionado conteúdo a enviar, em (22), através do método `Add` com o tipo dos dados que se pretendem adicionar ao campo dos dados da mensagem, neste caso o tipo `idp_t`. Em (23) é obtido um apontador para o *agente* local, para em (24) se proceder ao envio da mensagem para o recurso com identificador `rsc_idp`, recebido como argumento na chamada da função `Test`. Por fim, em (26) é retornado pela função o idp do *agente* local presente na variável `agent_idp`.

O resultado da execução desta aplicação, apresentado de seguida, permite demonstrar que foi obtido o mesmo valor pelos três métodos usados:

```
$ corx app ctx 1 0 libbasic_operations.so
4294967035 4294967035 4294967035
```

4.4.3 *Arranque paralelo*

Nas subsecções anteriores, foi apresentado o mecanismo básico para o lançamento de uma aplicação constituída por um único *domínio*. No que segue, será introduzido o tema do arranque paralelo de um sistema de domínios distribuídos. A este sistema corresponde, naturalmente, a criação de uma infraestrutura concorrente/paralela distribuída formada por múltiplos *pods*.

O exemplo em 4.4 (`parallel.cpp`) expõe os elementos necessários para a compreensão dos mecanismos que, como foi referido anteriormente, assentam no conceito do recurso *clausura*. Esta entidade materializa a noção de grupo fechado, que representa um conjunto de *domínios* e, consequentemente, *agentes* que arrancaram simultaneamente.

```
1  void Main(int argc, char *argv[])
2  {
3    auto domain = cor::GetDomain();
4    auto agent_idp = domain->GetActiveResourceIdp();
5    auto clos_idp = domain->GetPredecessorIdp(agent_idp);
6    auto clos = domain->GetLocalResource<cor::Closure>(clos_idp);
7    auto clos_size = clos->GetTotalMembers();
8    auto rank = clos->GetIdm(agent_idp);
9    auto parent_idp = clos->GetParent();
10
11   std::cout << agent_idp << "\t" << rank << "\t" << clos_idp << "\t"
12     << clos_size << "\t" << parent_idp << "\n";
13 }
```

Código 4.4.: Exemplo `parallel.cpp`.

No código, ignorando as linhas que já foram alvo de anterior explicação, é obtido o idp da *clausura* (`clos_idp`) e um apontador para a mesma presente na variável `clos`, em (5) e (6), respetivamente. Em (7) é obtido o número total de membros (*agentes*) que fazem parte da *clausura*, através da função `GetTotalMembers`, e em (8) é obtido o identificador de membro da entidade ativa (*agente*) com idp `agent_idp` no contexto da *clausura*. Seguidamente, é obtido em (9) o idp do recurso na origem da *clausura*, através do método `GetParent`, sendo feita a impressão de todos os valores obtidos em (11-12), relevantes para a compreensão do arranque paralelo em PlaCoR.

Há semelhança do exemplo inicial apresentado em 4.1, a utilização do mesmo comando para o arranque da aplicação - `corx app ctx 1 0 libparallel.so` - traduzir-se-ia na criação de uma árvore de dependências equivalente à apresentada na figura 7. Caso pretendêssemos desencadear o arranque simultâneo de duas ou mais instâncias do programa, isso

corresponderia a lançar o mesmo número de vezes o comando `corx`, alterando o valor do campo `number pods` para a quantidade de instâncias que se pretende arrancar, neste caso duas, da seguinte forma: `corx app ctx 2 0 libparallel.so`.

A necessidade de lançar várias instâncias do comando `corx` como forma de criar uma aplicação paralela, deu lugar à introdução de um novo comando `corun`. Este novo utilitário permite, de uma forma compacta, o lançamento de múltiplas réplicas do comando `corx` e, ao mesmo tempo, disponibiliza opções apropriadas para a distribuição de domínios, local e remotamente, na infraestrutura do sistema.

Assim, neste caso poderia ser usado o comando `corun --np 2 libparallel.so`, que implicitamente usa a ferramenta `corx` para lançar duas instâncias do mesmo programa em paralelo na máquina local. Este possui ainda uma opção para interpretar um ficheiro de *hosts*, que permite criar um sistema de domínios distribuídos em máquinas locais e/ou remotas, de forma idêntica ao comando `mpirun/mpiexec` disponibilizado pelas várias implementações do *standard* MPI.

A figura 9 apresenta a árvore de dependências do arranque paralelo da aplicação, que resulta da operação coletiva de sincronização no contexto do recurso *clausura*, realizada através da criação dos vários *agentes* locais aos *domínios*.

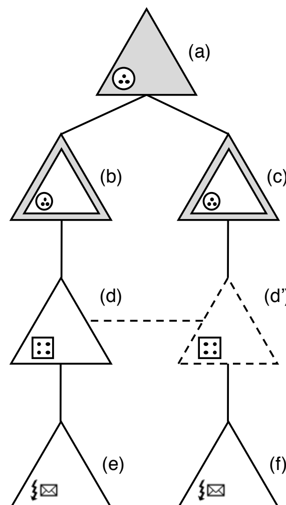


Figura 9.: Árvore de dependências de um arranque paralelo.

A partir do *meta-domínio* (a) existem dois caminhos exatamente iguais, com a particularidade de as duas *clausuras*, (d) e (d'), corresponderem ao mesmo recurso. Isto é, uma aplicação paralela estática constituída por dois ou mais *domínios* partilha o mesmo contexto, representado pela mesma *clausura*. A representação a tracejado da *clausura* (d'), significa que esta é apenas uma representação no *domínio* (c) da *clausura* (d). A confirmar esta representação está o facto de os dois *domínios* também partilharem a mesma origem, o que é demonstrado pelo valor do `parent_idp` corresponder a `0` em ambos os casos. Veremos

num próximo exemplo, como aquele identificador pode mudar no lançamento dinâmico de domínios.

A execução deste programa, através do comando anteriormente explicitado, produz o resultado mostrado abaixo. Deste podemos inferir a estrutura apresentada na árvore de dependências, já que as linhas apresentam agentes que pertencem à mesma *clausura*, evidenciado pelo tamanho da mesma ser 2 e cada um conter um idm único no contexto. O `parent_idp` corresponde a 0 em ambos os casos, o que indica que a *clausura* pertence a um arranque estático da aplicação. Notar que o idp da *clausura* é diferente em ambos, isto porque a *clausura* é criada apenas num dos *domínios*, sendo posteriormente feita uma adesão ao segundo *domínio*, obtendo um novo idp de pseudónimo.

```
$ corun --np 2 libparallel.so
4294966782 1 4294966783 2 0
4294967038 0 4294967039 2 0
```

4.4.4 *Lançamento dinâmico de domínios*

Em PlaCoR, o lançamento dinâmico de domínios acrescenta à criação estática de aplicações paralelas e distribuídas a possibilidade de criar, em tempo de execução, isto é, dinamicamente, múltiplas instâncias simples ou paralelas de módulos de bibliotecas dinâmicas, que fazem parte integrante da aplicação inicial. Para o efeito, a API disponibiliza a primitiva `Spawn`, conceptualmente próxima da congénere oferecida pelo MPI, em estreita relação com a noção de `parent_idp` introduzida na secção anterior, no qual lança novos *pods* e, consequentemente, novos domínios.

No exemplo apresentado em 4.5, o código `spawn.cpp` é em tudo idêntico ao exemplo em 4.4, até à linha (9). No caso paralelo, a aplicação é arrancada paralelamente com duas instâncias do módulo `libparallel.so`. Neste caso de estudo, a aplicação é arrancada com apenas uma única instância do módulo `libspawn.so`.

Em (11), é feito o teste para verificar se o `parent_idp` é igual a `0`. Em caso afirmativo, significa que se trata da instância inicial do módulo, pelo que vão ser arrancados dinamicamente dois novos *domínios* com o mesmo módulo. Estes são arrancados na máquina local (`"localhost"`), num contexto de arranque `"ctx2"`, sem argumentos a serem fornecidos à função de entrada do módulo.

Os dois *domínios* arrancados vão executar o mesmo módulo, mas algumas diferenças vão ser destacadas da execução inicial. Nomeadamente, o valor obtido para o `parent_idp` será diferente de `0`, pois estes foram arrancados dinamicamente. O valor obtido para o mesmo corresponde ao identificador do *agente* que efetuou a chamada ao método `Spawn`.

A árvore de dependências deste caso de estudo, apresentada na figura 10, é constituída por três *domínios*: (a) correspondente ao *domínio* de arranque da aplicação, com a sua *clau-*

```
1  void Main(int argc, char *argv[])
2  {
3    auto domain = cor::GetDomain();
4    auto agent_idp = domain->GetActiveResourceIdp();
5    auto clos_idp = domain->GetPredecessorIdp(agent_idp);
6    auto clos = domain->GetLocalResource<cor::Closure>(clos_idp);
7    auto clos_size = clos->GetTotalMembers();
8    auto rank = clos->GetIdm(agent_idp);
9    auto parent_idp = clos->GetParent();
10
11   if (parent_idp == 0)
12     auto new_clos_idp = domain->Spawn("ctx2", 2, "libspawn.so",
13         {}, { "localhost" });
14
15   std::cout << agent_idp << "\t" << rank << "\t" << clos_idp << "\t"
16         << clos_size << "\t" << parent_idp << "\n";
17 }
```

Código 4.5.: Exemplo `spawn.cpp`.

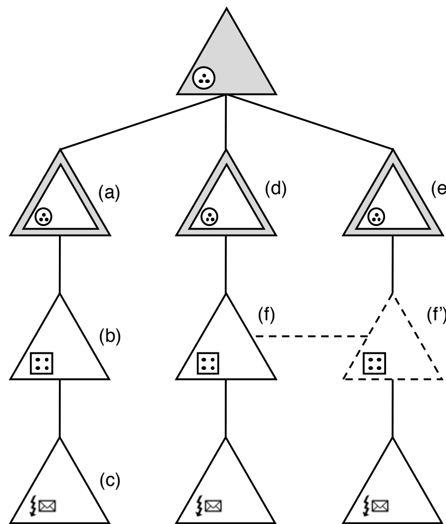


Figura 10.: Árvore de dependências de um arranque dinâmico de domínios.

sura individual (b), cujo `parent_idp` é igual a 0; outros dois *domínios*, (d) e (e), que partilham a mesma *clausura*, (f) e (f'), evidenciado a tracejado, cujo `parent_idp` será igual ao idp do *agente* (c) que pertence à hierarquia do *domínio* inicial da aplicação.

A execução deste programa, através do programa `corun`, produz o resultado abaixo. Deste podemos inferir a estrutura apresentada na árvore de dependências, já que nas duas primeiras linhas são explicitados dois *agentes* que pertencem à mesma *clausura*, evidenciado pelo tamanho da mesma ser 2, e cujo `parent_idp` corresponde ao idp do *agente* inicial.

```
 $ corun --np 1 libspawn.so
4294966526 1 4294966527 2 4294967038
4294966782 0 4294966783 2 4294967038
4294967038 0 4294967039 1 0
```

5

INTERFACE DE PROGRAMAÇÃO E FERRAMENTAS

Este capítulo é especialmente dedicado ao *Application Programming Interface* (API) da plataforma e às ferramentas usadas para o arranque de aplicações. A arquitetura global da plataforma e da respetiva API, foi construída como uma biblioteca de classes desenvolvida em C++ Moderno.

5.1 ARQUITETURA GERAL

A figura 11 apresenta uma visão global, formada por várias camadas, que ilustra como a construção de Aplicações orientadas ao Recurso assenta na existência de múltiplos Recursos (`Resource`), que incluem desde o domínio (`Domain`) até à guarda para leituras/escritas (`RWMutex`). Estes são realizados através de uma grande variedade de Elementos, que incluem desde o contentor (`Container`) até ao sincronizador guarda para leituras/escritas (`SRWMutex`), que são os "tijolos" da plataforma.

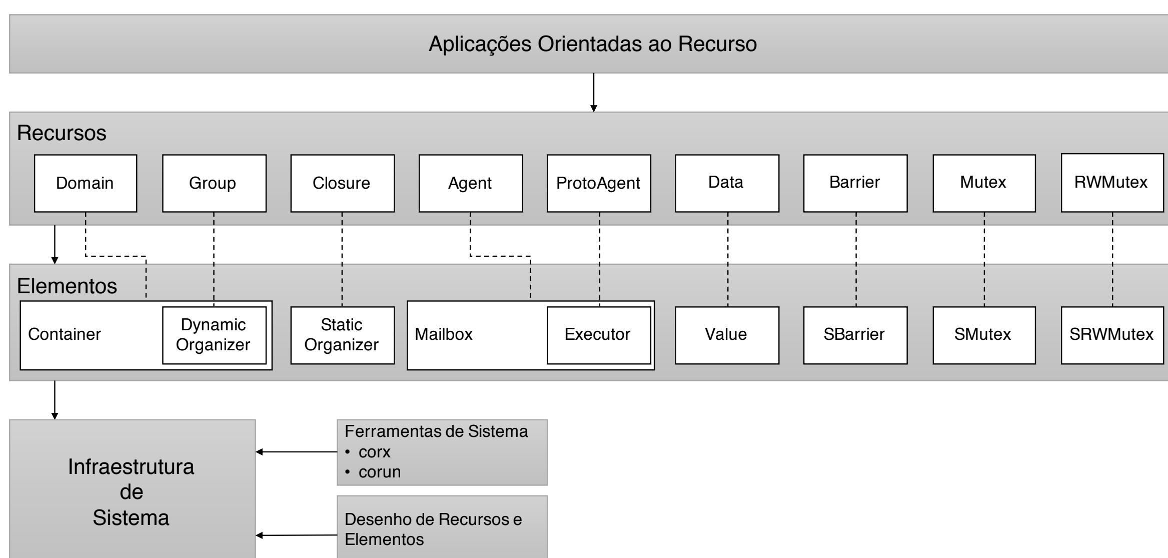


Figura 11.: Arquitetura da API.

Cada tipo de elementos possui funcionalidades e propriedades específicas, que são determinantes na definição das características gerais de cada classe de recursos. Desta forma, a distinção principal entre recursos advém dos seus elementos constituintes, obtidos através de mecanismos de herança múltipla oferecidos pelo C++. Por exemplo, o recurso `Domain` possui as características dos elementos `Container` e `DynamicOrganizer`, sendo que este último também é partilhado pelo recurso grupo (`Group`).

A arquitetura em camadas da plataforma é compatível com a criação de novos elementos com novas propriedades, que podem ser usados para construir novos recursos pela inclusão e combinação de elementos. Por exemplo, um recurso agente (`Agent`) é uma extensão do conceito de proto-agente (`ProtoAgent`), criado com base na inclusão do elemento `Mailbox`. Num futuro próximo, prevê-se a criação de novos elementos para dar resposta à necessidade de funcionalidades a incluir na plataforma, alguns dos quais já previstos em CoRes, tais como o porto, que permitirá novas facilidades para a comunicação entre recursos.

O protótipo inclui ainda Ferramentas de Sistema, que são comandos de linha usados para arrancar e executar aplicações, já exemplificados no capítulo anterior: o `corx` para o lançamento de um domínio local; e o `corun` para lançar aplicações paralelas locais/remotas, com um ou mais domínios (e que, por sua vez, recorre ao `corx` para a instanciação local de cada um dos domínios). A plataforma inclui ainda facilidades para o Desenho de Recursos e Elementos.

5.2 UTILIZAÇÃO DO C++

O desenho da plataforma tira, naturalmente, partido dos vários paradigmas de programação oferecidos pela linguagem C++ moderna (C++11, C++14 e C++17), em particular da programação orientada aos objetos, estando na base para a construção dos elementos e dos recursos, e da programação genérica, recorrendo aos `templates` e `variadic templates`. Estas últimas facilidades são amplamente utilizadas pela API para, por exemplo, a criação dos objetos que representam os diversos recursos presentes na plataforma.

A interface da programação do protótipo é fortemente tipada, pelo que sempre que se pretende criar, obter um apontador ou interagir com um objeto que represente um recurso, é necessário saber o tipo do mesmo e explicitar este nas primitivas de interface do sistema, intimamente ligadas com o elemento contentor. Assim, é possível detetar possíveis erros em tempo de compilação, e eventualmente, em tempo de execução através do lançamento de exceções.

Outra das características do protótipo está relacionada com o facto de este não precisar de saber explicitamente o tipo dos recursos de antemão. Alguns recursos PlaCoR são construídos como classes `templated`, como é o caso do recurso dado; o utilizador pode então definir um dado com o tipo de dados que necessita (até mesmo com classes de dados

implementadas por si, precisando apenas de registar o tipo do dado no seu código, através da macro `COR_REGISTER_TYPE`).

5.3 ELEMENTOS

Ao longo desta secção vão ser apresentadas as diferentes classes de elementos que atualmente constituem a camada dos Elementos, visível na figura 11.

Convém desde logo salientar que a génese de cada recurso é indissociável das características intrínsecas dos seus elementos constituintes, obtidas por herança múltipla. Por sua vez, cada objeto elemento só existe no contexto de uma instância específica de um só objeto recurso. Neste contexto, como veremos a seguir, na assinatura dos construtores de todas as classes de elementos da plataforma estará sempre presente o identificador global do recurso relacionado, associação essa que ocorre na criação de cada recurso e é transparente ao programador.

Outro aspeto a ter em atenção é que alguns métodos abaixo fazem referência à classe `ResourcePtr`. Esta classe, a apresentar mais tarde, é equivalente aos *smart pointers* presentes na linguagem C++, neste caso encapsulando um apontador para um qualquer objeto que representa um recurso. Um outro aspeto ainda está relacionado com os identificadores globais e contextuais dos recursos, que são representados pelos tipos de dados `idp_t` e `idm_t`, respetivamente.

5.3.1 *Contentor*

O elemento contentor estabelece uma relação direta com as facilidades disponibilizadas pela infraestrutura de sistema da plataforma, que inclui métodos para: i) obter os contextos da aplicação e de arranque do *pod* local; ii) obter os identificadores e apontadores dos recursos presentes no domínio local; iii) criar recursos local e/ou remotamente; iv) criar réplicas de recursos remotos; v) lançar dinamicamente novos *pods*/domínios que se vão ligar à aplicação atual num outro contexto de arranque.

`class Container`

- `Container(idp_t idp)`: construtor parametrizado que cria um elemento contentor como parte integrante do recurso com identificador `idp`, fornecendo uma interface para as primitivas de sistema disponibilizadas pela plataforma.

- `std::string const& GetGlobalContext()`: método utilizado para obter o nome do contexto global da aplicação.

- `std::string const& GetLocalContext()`: método utilizado para obter o nome do contexto de arranque do domínio local.
- `unsigned int GetTotalPods()`: método utilizado para obter o número total de *pods* presentes na aplicação.
- `unsigned int GetTotalDomains()`: método utilizado para obter o número total de domínios presentes na aplicação.
- `idp_t GetActiveResourceIdp()`: método utilizado para obter o identificador global do recurso ativo que executou este método.
- `idp_t GetPredecessorIdp(idp_t idp)`: método utilizado para obter o identificador global do recurso ascendente do recurso identificado por `idp`.
- `template <typename T>`
  `ResourcePtr<T> GetLocalResource(idp_t idp)`: método utilizado para obter um apontador para o recurso local do tipo `T` com identificador `idp`. Tal como o nome indica, esta operação é utilizada para obter apontadores para os recursos presentes no domínio local.
- `template <typename T, typename ... Args>`
  `ResourcePtr<T> CreateLocal(idp_t ctx, std::string const& name, Args&&... args)`: método utilizado para criar um recurso do tipo `T` no contexto de um recurso com organizador identificado por `ctx`, presente no domínio local, com nome `name` no contexto do organizador. O recurso é criado com os argumentos passados através de `args`, que terão de corresponder a um dos seus possíveis construtores. Este método devolve um apontador para o recurso criado.
- `template <typename T, typename ... Args>`
  `idp_t CreateRemote(idp_t ctx, std::string const& name, Args&&... args)`: método utilizado para criar um recurso do tipo `T` no contexto de um recurso com organizador identificado por `ctx`, presente num domínio remoto, com nome `name` no contexto do organizador. O recurso é criado com os argumentos passados através de `args`, que terão de corresponder a um dos seus possíveis construtores. Este método devolve o identificador global do recurso criado.
- `template <typename T, typename ... Args>`
  `idp_t Create(idp_t ctx, std::string const& name, Args&&... args)`: método utilizado para criar um recurso do tipo `T` no contexto de um recurso com organizador identificado por `ctx`, localizado no domínio local ou num domínio remoto, com nome `name` no contexto do organizador. O recurso é criado com os argumentos passados através de `args`, que terão de corresponder a um dos seus possíveis construtores. Este método devolve o identificador global do recurso criado.

- `template <typename T, typename ... Args>`
  `ResourcePtr<T> CreateCollective(idp_t ctx, std::string const& name, unsigned int total_members, Args&&...args)`: método semelhante ao `Create`, sendo que a instanciação do recurso é feita de forma coletiva por um total de `total_members` recursos com elemento executor, na qual todos os participantes terão uma cópia local do recurso. Esta cópia local é pendurada no recurso com organizador identificado por `ctx`, especificado em cada chamada do método.

- `template <typename T, typename ... Args>`
  `ResourcePtr<T> CreateCollective(idp_t clos, idp_t ctx, std::string const& name, Args&&... args)`: método semelhante ao método `Create`, sendo que a instanciação do recurso é feita de forma coletiva por todos os executores presentes no elemento organizador estático pertencente ao recurso identificado por `clos`, na qual todos os participantes terão uma cópia local do recurso. Esta cópia local é pendurada no recurso com organizador identificado por `ctx`, especificado em cada chamada do método.

- `template <typename T>`
  `ResourcePtr<T> CreateReference(idp_t idp, idp_t ctx, std::string const& name)`: método utilizado para criar uma réplica, no domínio local, do recurso com tipo `T` identificado por `idp`. Esta réplica é pendurada no contexto do recurso com organizador identificado por `ctx`, com o nome `name` nesse contexto.

- `template <typename T, typename ... Args>`
  `void Run(idp_t idp, Args&&... args)`: método utilizado para correr a função com tipo `T` do elemento executor do recurso identificado por `idp`, local ou remoto, com os respetivos argumentos `args` que têm de corresponder à assinatura da função.

- `template <typename T>`
  `void Wait(idp_t idp)`: método utilizado para esperar pelo término da execução da função com tipo `T` do elemento executor do recurso identificado por `idp`, local ou remoto.

- `template <typename T>`
  `auto Get(idp_t idp)`: método utilizado para obter o resultado da execução da função com tipo `T` do elemento executor do recurso identificado por `idp`, local ou remoto. Este permite também obter uma eventual exceção lançada pela função.

- `idp_t Spawn(std::string const& ctx, unsigned int npods, std::string const& mod, std::vector<std::string> const& args, std::vector<std::string> const& hosts)`: método utilizado para lançar dinamicamente um total de `npods` novos *pods* no contexto de arranque `ctx`, arrancados nas máquinas definidas em `hosts` utilizando o algoritmo *round-robin*. O módulo `mod` vai ser carregado dinamicamente e vai ser executada a função de entrada do módulo com os argumentos presentes em `args`.

5.3.2 *Organizador*

O elemento organizador cria as condições para o estabelecimento de árvores de dependências, que permitem agregar recursos de alguma forma relacionados, atribuindo-lhes identificadores de membro locais (`idm`) e um nome no contexto de um outro recurso, designado por recurso ascendente. Na árvore de dependências, cada um dos recursos que possuem um elemento do tipo organizador dá origem a uma nova sub-árvore, em que os seus membros são, por sua vez, recursos estruturados (que contêm um elemento organizador) ou recursos simples.

Em PlaCoR, o elemento organizador tem duas diferentes realizações, de nome organizador dinâmico (`DynamicOrganizer`) e estático (`StaticOrganizer`), muito semelhantes ao nível do interface. O primeiro permite a adesão/saída dinâmica de recursos a qualquer momento, enquanto que o segundo é criado de uma só assentada, com um conjunto inicial fechado de recursos membros.

A existência de organizadores estáticos está intrinsecamente ligada à necessidade de garantir a estabilidade de um grupo, cuja aridade permanece constante ao longo do tempo. Este último tipo de elemento organizador é fundamental sempre que há lugar à instanciação simultânea de múltiplos domínios, quer no arranque estático como no arranque dinâmico de aplicações paralelas, que necessitam da garantia de consistência de um grupo fechado.

5.3.2.1 *Organizador Dinâmico*

O elemento organizador dinâmico, para além das características gerais inerentes ao organizador, pode ser iniciado com um parâmetro `module`, que é um caminho para uma biblioteca dinâmica. Essa biblioteca, carregada em tempo de execução, disponibiliza um conjunto de funções que podem ser evocadas, no contexto do módulo, com o nome usado no ficheiro do código fonte.

```
class DynamicOrganizer
```

- `DynamicOrganizer(idp_t idp, std::string const& module)`: construtor parametrizado que cria um elemento organizador dinâmico como parte integrante do recurso com identificador `idp`, que introduz o módulo `module` na aplicação.

- `void Join(idp_t idp, std::string const& name)`: método que permite a associação do recurso identificado por `idp` ao organizador dinâmico correspondente, atribuindo o nome `name` e um identificador de membro ao recurso no seu contexto local (identificador este gerado localmente).

- `void Leave(idp_t idp)`: método que desassocia o recurso identificado por `idp` do organizador correspondente.

- std::string const& GetModuleName() const: método que permite obter o nome do módulo do organizador correspondente.

5.3.2.2 *Organizador Estático*

O elemento organizador estático acresce às características do organizador, informação sobre o recurso na origem do próprio elemento, recebida pelo parâmetro parent durante a construção do objeto. Este parâmetro, introduzido no capítulo anterior, terá o valor 0 se o organizador pertencer ao contexto de arranque inicial da aplicação, ou então o idp do agente responsável pela criação do próprio, como é o caso do lançamento dinâmico de novos domínios.

class StaticOrganizer

- StaticOrganizer(idp_t idp, unsigned int members, idp_t parent): construtor parametrizado que cria um elemento organizador estático como parte integrante do recurso com identificador idp, com o total de members membros e com o recurso identificado por parent como origem do mesmo.
- void Join(idp_t idp, std::string const& name): método que permite a associação do recurso ativo identificado por idp ao organizador estático correspondente, atribuindo o nome name ao recurso nesse contexto. Esta operação é coletiva, pelo que deverá ser chamada um total de members vezes por diferentes recursos ativos.
- void Leave(idp_t idp): método que desassocia o recurso identificado por idp do organizador correspondente. Esta operação é coletiva, pelo que deverá ser chamada um total de members vezes por diferentes recursos ativos.
- idp_t GetParent() const: método utilizado para obter o identificador principal do recurso na origem do organizador estático.

5.3.2.3 *Métodos comuns*

De seguida, são apresentados os métodos comuns a ambos os sub-elementos do organizador.

- std::size_t GetTotalMembers() const: método que permite obter o número total de membros do organizador correspondente.
- std::vector<idp_t> GetMemberList() const: método utilizado para obter a lista dos identificadores principais dos recursos que fazem parte do organizador correspondente.

- `idp_t GetIdp(idm_t idm) const`: método que permite obter o identificador principal do recurso com base no identificador de membro `idm` do mesmo no contexto do organizador.

- `idp_t GetIdp(std::string const& name) const`: método que permite obter o identificador principal do recurso com base no nome `name` do mesmo no contexto do organizador.

- `idm_t GetIdm(idp_t idp) const`: método que permite obter o identificador de membro do recurso com base no identificador principal `idp` do mesmo no contexto do organizador.

- `idm_t GetIdm(std::string const& name) const`: método que permite obter o identificador de membro do recurso com base no nome `name` do mesmo no contexto do organizador.

5.3.3 *Executor*

O elemento executor é, fisicamente, um fio de execução a que se associa uma função do programador. Este foi implementado como uma classe `templated`, ao qual é necessário passar o tipo da assinatura da função que vai executar. O tipo de retorno corresponde ao `typename R`, enquanto que o tipo dos parâmetros da função corresponde ao `typename ... P`.

Entre as variantes de construção do elemento, há a possibilidade de especificar o nome do módulo e o nome da função a carregar, ou então passar diretamente a função. Este possui ainda métodos para lançar (`Run`), esperar (`Wait`) e obter (`Get`) o valor de retorno da função, ou capturar uma eventual exceção que ocorra no contexto da função executada.

```
template <typename R, typename ... P>
class Executor<R(P...)>
```

- `Executor(idp_t idp, std::string const& module, std::string const& function)`: construtor parametrizado que cria um elemento executor como parte integrante do recurso com identificador `idp`, que irá carregar e executar uma função com nome `function` presente na biblioteca dinâmica `module`, com assinatura `R(P...)`.

- `Executor(idp_t idp, std::function<R(P...)> const& f)`: construtor parametrizado que cria um elemento executor como parte integrante do recurso com identificador `idp`, que irá executar a função `f` recebida como parâmetro.

- `template <typename ... Args>`
  `void Run(Args&&... args)`: método utilizado para executar a função atribuída ao executor, passando os argumentos `args` que têm de corresponder à assinatura da função. No caso do executor ter sido criado com a informação do módulo e o nome da função, esta vai ser carregada dinamicamente.

- `void Wait()`: método utilizado para esperar pelo término da execução da função do executor.

- `R Get()`: método que retorna o resultado da execução da função do executor, ou então uma possível exceção lançada pela função.

- `void ChangeIdp(idp_t idp)`: método utilizado para trocar a identidade do executor para `idp`.

- `void ResumeIdp()`: método utilizado para o retorno à identidade original do executor.

- `idp_t CurrentIdp() const`: método utilizado para obter o identificador do recurso atual do executor.

- `idp_t OriginalIdp() const`: método utilizado para obter o identificador original do recurso na origem da criação do executor.

5.3.4 *Caixa-Postal*

O elemento caixa-postal é usado para o envio/receção de mensagens entre recursos que possuam um elemento caixa-postal. Este permite o envio de mensagens para um ou mais destinos, desde que os seus identificadores sejam conhecidos, isto em relação à comunicação global. Também possui primitivas de comunicação por contexto, que permitem difundir mensagens para um recurso com um elemento organizador estático, através do idp do mesmo, ou então o envio/receção de mensagens utilizando o idp do recurso clausura e o idm no contexto do mesmo. De momento, as primitivas de comunicação disponibilizadas são bloqueantes.

`class Mailbox`

- `Mailbox(idp_t idp)`: construtor parametrizado para criar um elemento caixa-postal como parte integrante do recurso com identificador `idp`.

- `void Send(idp_t dest, Message& msg) const`: método utilizado para enviar uma mensagem `msg` para o recurso com identificador `dest`.

- `void Send(std::vector<idp_t> const& dests, Message& msg) const`: método utilizado para enviar uma mensagem `msg` para os recursos com identificadores presentes em `dests`.
- `Message Receive() const`: método utilizado para receber uma mensagem enviada para a caixa-postal do recurso correspondente.
- `Message Receive(idp_t source) const`: método utilizado para receber uma mensagem enviada por um recurso identificado por `source` para a caixa-postal do recurso correspondente.
- `void Broadcast(idp_t clos, Message& msg) const`: método utilizado para difundir uma mensagem `msg` para todos os membros de um recurso com elemento organizador estático identificado por `clos`.
- `void Send(idm_t rank, idp_t clos, Message& msg) const`: método utilizado para enviar uma mensagem `msg` para o recurso membro com identificador `rank`, no contexto do recurso com elemento organizador estático identificado por `clos`.
- `Message Receive(idm_t rank, idp_t clos) const`: método utilizado para receber uma mensagem enviada pelo recurso membro com identificador `rank`, no contexto do recurso com elemento organizador estático identificado por `clos`.

5.3.4.1 *Mensagem*

A classe mensagem é uma classe auxiliar do elemento caixa-postal, cujas instâncias incluem um cabeçalho, formado por uma etiqueta e o idp do recurso na origem, e um tampão de memória para os dados. É possível adicionar e obter dados de uma mensagem, desde que os respetivos tipos sejam serializáveis pela biblioteca `cereal`.

`class Message`

- `Message()`: construtor padrão de uma mensagem.
- `std::size_t Size() const`: método utilizado para devolver o tamanho da mensagem.
- `void Clear()`: método utilizado para apagar o conteúdo da mensagem.
- `std::uint16_t Type() const`: método utilizado para obter o tipo da mensagem.
- `void SetType(std::uint16_t type)`: método utilizado para modificar o tipo da mensagem.
- `idp_t Sender() const`: método utilizado para obter a identificação do recurso que enviou a mensagem.

- template <typename T>
  T Get(std::size_t index = 0) const: método que permite obter, do tampão dos dados da mensagem, o conteúdo do tipo T localizado na posição index.

- template <typename T>
  void Add(T const& data): método que permite adicionar ao tampão dos dados da mensagem o conteúdo da variável data do tipo T.

5.3.5 *Valor*

O elemento valor é um recipiente que garante a consistência dos dados que alberga, num sistema de domínios distribuídos, onde podem coexistir múltiplas representações do mesmo valor. É utilizado o protocolo *release-consistency* como modelo de consistência dos dados.

O programador pode criar as suas próprias classes de dados, desde que, tal como no caso da mensagem, os tipos de dados sejam serializáveis. Para tal, novos tipos de dados criados pelo utilizador têm de ser registados pela macro COR_REGISTER_TYPE.

Sendo este elemento especificado sob uma classe *templated*, é necessário especificar o tipo de dados a albergar, correspondente ao typename T. Os dados do utilizador são instanciados por atribuição ou através dos construtores fornecidos pelo tipo de dados albergado.

```
template <typename T>
class Value
```

- Value(idp_t idp, Args&& ... args): construtor parametrizado que cria um elemento valor como parte integrante do recurso com identificador idp, instanciando os dados com base no construtor mais adequado aos argumentos recebidos através de args.

- void AcquireWrite() const: método utilizado para obter o direito de escrita sob os dados do elemento valor.

- void ReleaseWrite() const: método utilizado para libertar o direito de escrita sob os dados do elemento valor.

- void AcquireRead() const: método utilizado para obter o direito de leitura sob os dados do elemento valor.

- void ReleaseRead() const: método utilizado para libertar o direito de leitura sob os dados do elemento valor.

- T* Get(): método utilizado para obter um apontador para os dados do utilizador do tipo T albergado pelo elemento valor. Esta função deve ser utilizada com prévia aquisição de direito de escrita ou leitura.

5.3.6 *Sincronizador*

Na plataforma, o elemento sincronizador, proposto em CoRes, desdobra-se nas seguintes variantes: barreira (`SBarrier`), guarda (`SMutex`) e guarda para leituras/escritas (`SRWMutex`). Todos estes elementos atuam como estruturas de sincronização distribuídas.

5.3.6.1 *Barreira*

O elemento sincronizador barreira, tal como o nome indica, fornece a capacidade de sincronização distribuída do conjunto dos recursos agentes presentes no contexto de um organizador estático.

`class SBarrier`

- `SBarrier(idp_t idp, idp_t clos)`: construtor parametrizado que cria um elemento barreira no contexto de um recurso identificado por `idp`, que servirá de estrutura de sincronização distribuída dos recursos ativos presentes no organizador estático identificado por `clos`.
- `void Synchronize()`: operação coletiva que permite a sincronização de todos os recursos ativos presentes no organizador estático identificado por `clos`.

5.3.6.2 *Guarda*

Ambas as classes de elemento sincronizador guarda servem para criar regiões de exclusão mútua: `SMutex` é uma guarda geral, enquanto que `SRWMutex` é um guarda que permite definir regiões de exclusão mútua para leituras e escritas.

`class SMutex`

- `SMutex(idp_t idp)`: construtor parametrizado que cria um elemento guarda no contexto de um recurso identificado por `idp`.
- `void Acquire()`: método que sinaliza a entrada numa região de exclusão mútua.
- `void Release()`: método que sinaliza a saída de uma região de exclusão mútua.

`class SRWMutex`

- `SRWMutex(idp_t idp)`: construtor parametrizado que cria um elemento guarda no contexto de um recurso identificado por `idp`, tendo este capacidade para fornecer regiões de exclusão mútua para leituras e escritas.

- `void AcquireRead()`: método que sinaliza a entrada numa região de exclusão mútua para leituras.
- `void ReleaseRead()`: método que sinaliza a saída de uma região de exclusão mútua para leituras.
- `void AcquireWrite()`: método que sinaliza a saída de uma região de exclusão mútua para escritas.
- `void ReleaseWrite()`: método que sinaliza a saída de uma região de exclusão mútua para escritas.

5.4 RECURSOS

Em PlaCoR, como referido antes, as características distintivas das diferentes classes de recursos são o resultado das propriedades dos seus elementos constituintes. Numa outra dimensão, para assegurar que todos os recursos se incluam no mesmo paradigma, é necessário, em geral, que todos possuam comportamentos semelhantes. Neste contexto, para generalizar a utilização dos recursos, todas as respetivas classes têm de ser definidas incluindo obrigatoriamente a classe base Recurso (`Resource`) na sua hierarquia.

Esta opção de projeto é consistente com a definição do interface das classes de recursos, possuindo apenas construtores, deixando a caracterização do respetivo comportamento ser conduzido pelas propriedades específicas das suas classes de elementos que compõem a sua hierarquia. Generalizando, todos os recursos através do seu construtor de base definido em `Resource` possuem um identificador global idp, que lhe é atribuído automaticamente na sua génese.

`class Resource`

- `Resource(idp_t idp)`: construtor parametrizado que cria uma instância de um recurso com identificador `idp`.
- `idp_t Idp() const`: método que permite obter o identificador principal do recurso.

5.4.1 *Criação de recursos*

A criação de recursos está intrinsecamente dependente das facilidades disponibilizadas pelo sistema de computação, acessíveis através do elemento contentor, entidade central na constituição do domínio. Com efeito, apesar de todos os recursos terem de ser criados no contexto de um recurso estruturado, em última instância a sua representação física é

sempre concretizada no âmbito do elemento contentor do domínio enquanto recurso físico, através de primitivas específicas na API.

No que diz respeito à libertação de recursos, é a plataforma que, utilizando o conceito de RAII (*Resource Acquisition Is Initialization*), destrói automaticamente os recursos sempre que estes deixam de ser necessários. Para tal, foi construída a classe `ResourcePtr` que oferece uma espécie de *smart pointer* para recursos. Assim que estes deixam de ser referenciados no sistema de domínios distribuídos são automaticamente libertados, garantindo desta forma a não existência de "fugas" de memória.

```
template <typename T>
class ResourcePtr
```

- `ResourcePtr()`: construtor padrão de um apontador para um recurso.
- `T* operator->() const`: operador que permite obter um apontador para o objeto do tipo `T` que representa o recurso.
- `T& operator*() const`: operador que permite obter uma referência para o objeto do tipo `T` que representa o recurso.
- `idp_t Idp() const`: método que permite obter o identificador de pseudónimo do recurso.

5.4.2 *Domínio*

O recurso Domínio possui as funcionalidades dos elementos contentor e organizador dinâmico que, enquanto recurso estruturado, pode ser usado como ascendente para a criação de recursos. A criação de domínios está relacionada com o nível de concorrência/paralelismo de uma aplicação, uma vez que a cada domínio corresponde a criação de um processo no sistema de computação subjacente. De notar que a plataforma permite a criação de mais do que um domínio (virtual) no mesmo processo, o que significa que não faz aumentar, necessariamente, o nível de concorrência/paralelismo da aplicação.

As propriedades herdadas do elemento organizador permitem incluir dinamicamente bibliotecas dinâmicas, e assim aceder a funções que podem ser carregadas e executadas pelos recursos ativos presentes no protótipo. Esta facilidade é utilizada para o carregamento do módulo do utilizador aquando do arranque de uma aplicação, disponibilizando assim a função de entrada do módulo para o recurso agente criado no arranque.

```
Domain :: Container . DynamicOrganizer

class Domain:
```

```
    public Resource,
    public Container,
    public DynamicOrganizer
```

- `Domain(idp_t idp, std::string const& module)`: construtor parametrizado que permite criar um recurso domínio com identificador `idp` e com o módulo dinâmico `module`.

5.4.3 *Grupo*

O recurso Grupo possui as funcionalidades do elemento organizador dinâmico. Tal como o domínio, o grupo é um recurso estruturado, pelo que organiza logicamente outros recursos. Os recursos que são criados tendo como ascendente um determinado grupo residem no domínio da sub-árvore de dependências a que pertence esse grupo. O mesmo é válido para a criação do grupo. Este permite ainda incluir dinamicamente novos módulos na aplicação, disponibilizando as funções dos mesmos para serem carregadas e executadas por agentes.

```
Group :: DynamicOrganizer

class Group:
        public Resource,
        public DynamicOrganizer
```

- `Group(idp_t idp, std::string const& module)`: construtor parametrizado que permite criar um recurso grupo com identificador `idp` e com o módulo dinâmico `module`.

5.4.4 *Clausura*

O recurso Clausura possui as funcionalidades do elemento organizador estático. Tal como o domínio e o grupo, a clausura é um recurso estruturado que permite a junção de recursos ativos ao mesmo, formando um contexto fechado e consistente.

```
Closure :: StaticOrganizer

class Closure:
        public Resource,
        public StaticOrganizer
```

- `Closure(idp_t idp, unsigned int total_members, idp_t parent)`: construtor parametrizado que cria um recurso clausura com identificador `idp`, com `total_members` membros, e terá como origem o recurso identificador por `parent`.

5.4.5 *Agente*

O recurso Agente possui as funcionalidades dos elementos executor e caixa-postal. Este é considerado um recurso ativo com capacidade para comunicar por passagem de mensagens com outros recursos que possuam um elemento caixa-postal, pressupondo o escalonamento de um fio de execução no domínio onde reside o seu ascendente, executando a função requisitada pelo utilizador. O número de agentes em execução no espaço de computação que é circunscrito pelo domínio, define o nível de paralelismo da aplicação no domínio.

Enquanto classe *templated*, é preciso especificar a assinatura da função que vai executar. O tipo de retorno corresponde ao `typename R`, e o tipo dos parâmetros da função corresponde ao `typename ... P`.

```
Agent :: Executor . Mailbox

template <typename R, typename ... P>
class Agent<R(P...)>:
        public Resource,
        public Executor<R(P...)>,
        public Mailbox
```

- `Agent(idp_t idp, std::string const& module, std::string const& f)`: construtor parametrizado que permite criar um recurso agente com identificador `idp`, que irá carregar e executar uma função com nome `f` presente na biblioteca dinâmica `module`, com assinatura `R(P...)`.
- `Agent(idp_t idp, std::function<R(P...)> const& f)`: construtor parametrizado que permite criar um recurso agente com identificador `idp`, que irá executar a função `f` recebida como parâmetro.

5.4.6 *Proto-Agente*

O recurso Proto-Agente possui as funcionalidades do elemento executor. É em tudo semelhante ao recurso Agente, não possuindo as funcionalidades para comunicar por passagem de mensagens, devido a não possuir o elemento caixa-postal na sua hierarquia.

```
ProtoAgent :: Executor

template <typename R, typename ... P>
class ProtoAgent<R(P...)>:
        public Resource,
        public Executor<R(P...)>
```

- `ProtoAgent(idp_t idp, std::string const& module, std::string const& f)`: construtor parametrizado que permite criar um recurso agente simples com identificador `idp`, que irá carregar e executar uma função com nome `f` presente na biblioteca dinâmica `module`, com assinatura `R(P...)`.
- `ProtoAgent(idp_t idp, std::function<R(P...)> const& f)`: construtor parametrizado que permite criar um recurso agente simples com identificador `idp`, que irá executar a função `f` recebida como parâmetro.

5.4.7 *Dado*

O recurso Dado possui as funcionalidades do elemento valor. Este recurso oferece um mecanismo de programação em memória partilhada num ambiente de computação distribuído. Desta forma, os valores associados ao recurso podem ser partilhados por vários agentes em execução em qualquer dos domínios da aplicação, locais ou remotos. É um recurso construído como uma classe *templated*, em que o tipo de dados associado é explicitamente definido pelo programador.

O acesso aos dados é efetuado através de operações explícitas de aquisição do direito de acesso, seja para leitura como para escrita, e a posterior libertação do mesmo. A consistência dos dados é garantida no intervalo entre a aquisição do direito de acesso ao recurso e a operação de libertação do mesmo.

```
Data :: Value

template <typename T>
class Data:
        public Resource,
        public Value<T>
```

- `template <typename ... Args>`
  `Data(idp_t idp, Args&&... args)`: construtor parametrizado que permite criar um recurso dado com identificador `idp`, instanciando os dados com base no construtor mais adequado aos argumentos recebidos através de `args`.

5.4.8 *Barreira*

O recurso Barreira possui as funcionalidades do elemento sincronizador barreira. Este é uma estrutura de sincronização distribuída que permite que um número pré-fixado de agentes, residentes em qualquer domínio da aplicação, fiquem suspensos até que seja atingido o quórum indicado. Este corresponde à totalidade dos agentes presentes no elemento

organizador estático associado ao recurso com o identificador recebido como parâmetro na criação da barreira.

```
Barrier :: SBarrier

class Barrier:
    public Resource,
    public SBarrier
```

- `Barrier(idp_t idp, idp_t clos)`: construtor parametrizado que permite criar um recurso barreira com identificador `idp`, que irá sincronizar os recursos presentes no organizador estático do recurso identificado por `clos`.

5.4.9 *Guarda*

O recurso Guarda possui as funcionalidades do elemento sincronizador guarda. Este é uma estrutura de sincronização distribuída que permite definir regiões de exclusão mútua, através da utilização de primitivas de aquisição e libertação de acesso a essas regiões.

Existem dois tipos de recursos guarda: a guarda e a guarda para leituras/escritas. Em ambas é necessário demarcar as regiões críticas de código, utilizando explicitamente as operações de aquisição e libertação do acesso às mesmas. No caso da guarda para leituras/escritas, é possível ter regiões de exclusão mútua tanto para leituras como para escritas.

Para o recurso guarda normal, assim que a operação de aquisição terminar, é garantido que apenas um fio de controlo (elemento executor) está a executar o código correspondente a essa região. No caso da guarda para leituras/escritas, é garantido que, para uma região de leitura, podem coexistir vários fios de execução em simultâneo, apenas com o intuito de efetuar leituras. Já para uma região de escrita, esta é funcionalmente equivalente ao recurso guarda convencional. A libertação do acesso a zonas de exclusão mútua sinaliza os fios de execução bloqueados, num ambiente distribuído.

```
Mutex :: SMutex

class Mutex:
        public Resource,
        public SMutex
```

- `Mutex(idp_t idp)`: construtor parametrizado que permite criar um recurso guarda com identificador `idp`.

```
RWMutex :: SRWMutex
```

```
class RWMutex:
        public Resource,
        public SRWMutex
```

- `RWMutex(idp_t idp)`: construtor parametrizado que permite criar um recurso guarda com identificador `idp`, tendo este capacidade para fornecer regiões de exclusão mútua para leituras e escritas.

5.5 FUNÇÕES GLOBAIS

A plataforma inclui algumas funções globais, nomeadamente: `Initialize`, `Finalize` e `GetDomain`. As duas primeiras funções são utilizadas pelo utilitário `corx` para a criação do *pod*/domínio local e para a sua eliminação, respetivamente. A última função tem como intuito retornar um apontador para o domínio local.

- `ResourcePtr<Domain> CoR::Initialize(std::string const& app_grp, std::string const& context, unsigned int npods, std::string const& module)`: função utilizada para criar o *pod* local no contexto da aplicação identificada por `app_grp` e no contexto de arranque `context`, ao qual serão instanciados um total de `npods` *pods* locais e/ou remotos, sendo a primitiva coletiva até que todos os *pods* se liguem à aplicação. É também criado o domínio local com o módulo do utilizador `module`, sendo retornado um apontador para o mesmo.
- `void CoR::Finalize()`: função que termina os serviços contidos no *pod* local, sinalizando o término da aplicação.
- `ResourcePtr<Domain> CoR::GetDomain()`: função utilizada para obter um apontador para o domínio local. Apenas deve ser chamada pelos fios de execução dos recursos ativos.

5.6 FERRAMENTAS DE SISTEMA

Nesta secção apresentam-se os dois comandos de linha disponibilizados pela plataforma para o arranque de aplicações.

5.6.1 *corx*

A ferramenta `corx` é utilizada para criar o contexto local de uma aplicação PlaCoR. Este recebe os parâmetros necessários para a criação de um *pod* local, bem como toda a infraestrutura necessária para a execução de uma aplicação. Esta ferramenta é utilizada pela

ferramenta `corun` para a criação de *pods* locais nas máquinas especificadas pelo utilizador, não sendo recomendável o seu uso para o arranque explícito de aplicações.

Esta ferramenta recebe os contextos da aplicação e de arranque, bem como o número de *pods* que vão ser arrancados nesse contexto, esperando até que todos tenham sido criados para construir todo o contexto da aplicação a executar. Assim que todos os *pods* tenham sido arrancados neste contexto, é criado o domínio local com o módulo do utilizador, e executada a sua função de entrada com os argumentos passados.

A ferramenta tem a seguinte sintaxe:

```
$ corx app_group context number_pods parent module [args...]
```

As opções do programa têm o seguinte significado:

- `app_group`: grupo `Spread` da aplicação.
- `context`: grupo `Spread` do contexto de arranque da aplicação.
- `number_pods`: número total de *pods* que vão ser arrancados no respetivo contexto de arranque.
- `parent`: identificador do recurso que esteve na origem do arranque da aplicação caso a aplicação tenho sido lançada dinamicamente através da primitiva `Spawn`. No caso do arranque inicial da aplicação, este tem de ter o valor `0`.
- `module`: módulo do utilizador a ser carregado.
- `args`: argumentos a serem passados à função de entrada do módulo do utilizador.

5.6.2 *corun*

A ferramenta `corun` é usada para arrancar um ou mais *pods*, locais e/ou remotos, que carregam dinamicamente o módulo do utilizador e executam a função de entrada do mesmo com os argumentos passados. Esta utiliza a ferramenta `corx` para a criação dos *pods* e, consequentemente, domínios nos respetivos nós de computação.

É possível passar um ficheiro de *hosts* para explicitar as máquinas onde vão ser criados os respetivos *pods*, no qual é utilizado o algoritmo *round-robin* para a atribuição das máquinas. O ficheiro de *hosts* é constituído por uma lista de máquinas, na qual cada um tem de conter a informação do *hostname* ou IP, e das portas atribuídas ao serviço SSH e `Spread`, da seguinte forma:

```
<hostname ou ip>:<ssh port>:<spread port>
```

A ferramenta tem a seguinte sintaxe:

```
$ corun [options] module [args...]
```

As opções do programa têm o seguinte significado:

- `version`: imprime a versão da biblioteca.
- `help`: imprime uma mensagem de ajuda.
- `debug`: imprime mensagem de debug para verificar o comando corun que vai ser executado.
- `np`: número de *pods* que se pretendem arrancar.
- `appgroup`: grupo `Spread` no qual se pretende arrancar a aplicação.
- `hostfile`: ficheiro com os *hosts* onde serão arrancados os *pods*.
- `module`: módulo do utilizador a ser executado.
- `args`: argumentos a serem passados à função de entrada do módulo do utilizador.

6

PROGRAMAÇÃO AVANÇADA

A apresentação do protótipo PlaCoR, iniciada no capítulo 4, inclui exemplos de código que têm em vista uma primeira abordagem ao desenvolvimento de aplicações orientadas ao recurso. Em particular, foram mostrados exemplos de arranque de aplicações, com um único domínio e múltiplos domínios paralelos, lançados estática ou dinamicamente.

A leitura do código utilizado nos exemplos, permite um primeiro contacto com a realização prática dos conceitos do modelo, as facilidades da plataforma e o estilo de programação veiculado pela API, detalhadamente descrita no capítulo 5.

Neste capítulo, a ênfase é colocada na apresentação de outros exemplos de código real, capazes de demonstrar as potencialidades do paradigma da orientação ao recurso, no desenvolvimento de aplicações paralelas e distribuídas.

6.1 LEITURA E PROCESSAMENTO DE TCHAIN

Em aplicações de física de alta energia (HEP), em particular na experiência ATLAS do CERN, é recorrente a leitura e processamento de eventos independentes entre si, utilizando a *framework* ROOT [24], cujas características são registadas em coleções de ficheiros com formato TTree, organizados numa TChain.

No excerto de código 6.1, de (1-4) é iniciada uma TChain de nome `"CollectionTree"`, em (5-6) são definidas as variáveis que irão resumir o processamento, enquanto que em (7-10) se estabelece a correspondência entre as variáveis `intVar` e `floatVar` com os campos da TChain em processamento. As linhas de código que se seguem lêem e processam iterativamente a totalidade das entradas da TChain.

A análise do código revela um padrão de processamento, cujo desempenho poderia ser naturalmente melhorado se cada uma das TTree fosse processada em paralelo. Estando à partida garantida a independência entre eventos e o acesso partilhado seguro (*thread-safe*) às TTree, disponibilizado pela primitiva `EnableThreadSafety` presente na ROOT, seria possível lançar explicitamente vários fios de execução concorrentes: i) diretamente usando as facilidades do C++ Thread Support Library ou ii) implicitamente através das diretivas

```
TChain chain("CollectionTree");
for(Int_t i = 0; i < 10; ++i) {
    chain.Add(TString::Format("file%i.root", i));
}
Int_t intSum = 0;
Float_t floatSum = 0;
Int_t intVar = 0;
chain.SetBranchAddress("IntVar", &intVar);
Float_t floatVar = 0;
chain.SetBranchAddress("FloatVar", &floatVar);
const Long64_t entries = chain.GetEntries();
for(Long64_t entry = 0; entry < entries; ++entry) {
    chain.GetEntry(entry)
    intSum += intVar;
    floatSum += floatVar;
}
Info("readFiles", "id = %i, intSum = %i, floatSum = %g", id, intSum, floatSum);
```

Código 6.1.: Exemplo para o processamento de uma TChain.

do OpenMP. A primeira das hipóteses é facilmente replicável na orientação ao recurso, através da criação e execução de recursos do tipo *proto-agente*. Para a segunda, é possível escrever uma unidade de código orientado ao recurso, com facilidades equivalentes às da combinação dos `#pragma omp parallel` e `#pragma omp for` da norma OpenMP.

A tomada da segunda opção foi a oportunidade para estender a plataforma com o conceito de unidades de código modulares, que podem ser usadas para realizar padrões simples de paralelismo, tais como o *fork-join*.

6.2 UNIDADE POOL

A unidade (*unit*) Pool foi desenvolvida tomando como referência a abordagem e a implementação da classe SPool, proposta em Alessandrini [25], numa visão *thread-centric* orientada ao recurso, em que uma *pool* de recursos *proto-agente* é lançada explicitamente, ao contrário do OpenMP em que a mesma é lançada implicitamente, através de um `#pragma`. O acesso às facilidades de uma unidade faz-se, naturalmente, importando o respetivo interface, cuja implementação é apresentada no anexo B. Para ilustrar a utilização da unidade Pool, segue-se um exemplo apresentado em 6.2 do cálculo do PI, através do método Monte Carlo, adaptado de Alessandrini.

Em (3) adicionam-se outros utilitários usados na função `McPI` (18) para, por exemplo, obter a identificação de cada recurso *proto-agente* no contexto da Pool (30), em relação ao *grupo* no qual os mesmos são membros, através da função `GetRank`, à qual é necessário fornecer o tipo de recurso *organizador*, neste caso um *grupo*. Notar que o recurso *agente* que executa a função `Main` do módulo não participa na execução do trabalho paralelo. Assim que a Pool é criada (43) e acionada (45), este pode continuar com a sua própria atividade, ou aguardar o término da atividade de todos os fios de execução (46). É utilizada

```
1  #include "cor/cor.hpp"
2  #include "units/pool.hpp"
3  #include "units/utils.hpp"
4  #include "external/CpuTimer.h"
5  #include "external/Rand.h"
6
7  #include <vector>
8  #include <numeric>
9
10 std::vector<long> accepted;
11 long nsamples = 1000000000;
12 Rand R(999);
13
14 extern "C"
15 {
16     void Main(int argc, char *argv[]);
17 }
18
19 void McPi(void *)
20 {
21     unsigned long ct;
22     double x, y;
23     ct = 0;
24     for (auto n = 0; n < nsamples; ++n) {
25         x = R.draw();
26         y = R.draw();
27         if ((x*x+y*y) <= 1.0)
28             ++ct;
29     }
30     auto rank = GetRank<cor::Group>();
31     accepted[rank] = ct;
32 }
33
34 void Main(int argc, char *argv[])
35 {
36     CpuTimer T;
37     double pi, result = 0;
38     std::size_t pool_size = 4;
39     if (argc >= 1) pool_size = std::atoi(argv[0]);
40     if (argc == 2) nsamples = atoi(argv[1]);
41     nsamples /= pool_size;
42     accepted.resize(pool_size, 0L);
43     auto pool = new cor::Pool(pool_size);
44     T.Start();
45     pool->Dispatch(McPi, nullptr);
46     pool->WaitForIdle();
47     result = std::accumulate(accepted.begin(), accepted.end(), result);
48     pi = 4.0 * (double) (result/(pool_size*nsamples));
49     T.Stop();
50     T.Report();
51     std::cout << "Value of PI = " << pi << std::endl;
52     delete pool;
53 }
```

Código 6.2.: Cálculo do PI através do método MonteCarlo.

a classe CpuTimer para fazer a medição do tempo de execução da aplicação, sendo iniciada e terminada a medição em (44) e (49), respetivamente, sendo posteriormente reportado o tempo total em (50).

6.3 ORIENTAÇÃO AO RECURSO

A partir do exemplo de leitura/processamento de TChain acima descrito, vão ser apresentadas várias soluções paralelas utilizando o modelo CoR e a unidade Pool. Com as mesmas, é pretendido explicitar os vários conceitos presentes no modelo, relacionados com a computação paralela e distribuída. Complementarmente, foi desenvolvida uma classe auxiliar ReadChain que estabelece o interface entre as várias aplicações e o processamento de TChain, apresentada no anexo C.

6.3.1 *Processamento paralelo com a unidade Pool*

No excerto de código em 6.3, a função `ReadFiles` começa por criar uma instância da classe ReadChain (21), iniciada com o nome `"CollectionTree"` atribuído à TChain e com o caminho para os ficheiros a processar. Em (22), é feita a fixação do intervalo de ficheiros TTree a considerar, através das variáveis globais `in` e `fin` definidas em (17). De seguida, é feita a iniciação do trabalho (23) a realizar, que corresponde às linhas (2-10) do código em 6.1, e é feito o processamento dos ficheiros em (24). Por fim, é obtido o identificador `rank` (25) do *proto-agente* da Pool, de forma a guardar o resultado no vetor global `vec`, definido em (15), na posição correspondente ao `rank` do *proto-agente*. Notar que os valores de `pool_size`, `in` e `fin` podem ser redefinidos no arranque do módulo em (31-33).

Na função de entrada do módulo, o *agente* inicial cria a *pool* (38) com o tamanho estabelecido por *pool_size*. O processamento dos ficheiros ocorre em (41), ficando o *agente* inicial à espera pela conclusão do mesmo em (42). O valor do somatório é calculado para a variável `result` em (43), sendo imprimido (47) no fim do programa. Convém salientar que no código a utilização da API está totalmente escondida na unidade Pool, e o processamento na classe ReadChain.

Abaixo é explicitado o resultado de uma hipotética execução do módulo usando a linha de comandos, especificando o tamanho da *pool* como 4 (`pool_size=4`) e os ficheiros a processar 5..10 (`in=5` e `fin=10`):

```
$ corun --np 1 libread_files_par.so 4 5 10
Wall time is 0.588563 seconds
Final Value: 252064
```

A linha de comandos `corun` acima especifica `np=1`, isto é, apenas é lançado um único *domínio* que tira partido do paralelismo por memória partilhada por múltiplos fios de execução. Porém, o eventual arranque de vários *domínios* simultaneamente, através de um `np>1`, não acarretaria algum ganho em termos de desempenho, uma vez que a aplicação não está preparada para distribuir o processamento dos ficheiros por vários *domínios* locais/remotos.

```
#include "cor/cor.hpp"
#include "units/pool.hpp"
#include "units/utils.hpp"
#include "read_chain.hpp"
#include "external/CpuTimer.h"

#include <vector>
#include <numeric>

extern "C"
{
    void Main(int argc, char *argv[]);
}

static std::vector<int> vec(2);
static size_t pool_size = 2;
static std::size_t in = 0, fin = 10;

void ReadFiles(void *)
{
    ReadChain rc("CollectionTree", "/Users/brunoribeiro/Desktop/rootfiles");
    rc.SetFileRange(in, fin);
    rc.Initialize();
    rc.ProcessChain();
    auto rank = GetRank<cor::Group>();
    vec[rank] = rc.GetIntSum();
}

void Main(int argc, char *argv[])
{
    if (argc >= 1) { pool_size = std::atoi(argv[0]); vec.resize(pool_size); }
    if (argc >= 2) in = std::atoi(argv[1]);
    if (argc >= 3) fin = std::atoi(argv[2]);

    ROOT::EnableThreadSafety();
    CpuTimer T;
    Int_t result = 0;
    auto pool = new cor::Pool(pool_size);

    T.Start();
    pool->Dispatch(ReadFiles, nullptr);
    pool->WaitForIdle();
    result = std::accumulate(vec.begin(), vec.end(), result);
    T.Stop();

    T.Report();
    std::cout << "Final Value: " << result << std::endl;
}
```

Código 6.3.: Processamento paralelo de uma TChain utilizando a unidade Pool.

6.3.2 *Processamento distribuído com passagem de mensagens*

A 2ª versão do código de leitura/processamento de uma TChain, explicitado em 6.4, usa um modelo híbrido de programação, que combina um grão fino de paralelismo por memória partilhada com comunicação por passagem de mensagens, num ambiente de memória distribuída.

Esta nova versão da função `Main` pressupõe valores de `np>=1`, isto é, o possível arranque simultâneo de vários *domínios* (locais/remotos), o que significa a criação inicial de um

```
void Main(int argc, char *argv[])
{
    if (argc >= 1) { pool_size = std::atoi(argv[0]); vec.resize(pool_size); }
    if (argc >= 2) in = std::atoi(argv[1]);
    if (argc >= 3) fin = std::atoi(argv[2]);

    ROOT::EnableThreadSafety();

    Int_t partial = 0;
    AgentRange<cor::Closure>(in, fin);
    auto pool = new cor::Pool(pool_size);

    pool->Dispatch(ReadFiles, nullptr);
    pool->WaitForIdle();
    partial = std::accumulate(vec.begin(), vec.end(), partial);

    auto domain = cor::GetDomain();
    auto agent_idp = domain->GetActiveResourceIdp();
    auto agent = domain->GetLocalResource<cor::Agent<void(int,char**)>>(agent_idp);
    auto clos_idp = domain->GetPredecessorIdp(agent_idp);
    auto clos = domain->GetLocalResource<cor::Closure>(clos_idp);
    auto clos_size = clos->GetTotalMembers();
    auto rank = clos->GetIdm(agent_idp);

    if (rank == 0) {
        Int_t final_value = partial;
        for (auto i = 1; i < clos_size; ++i) {
            auto msg = agent->Receive();
            auto res = msg.Get<Int_t>();
            final_value += res;
        }
        std::cout << "Final Value: " << final_value << std::endl;
    } else {
        cor::Message msg;
        msg.Add<Int_t>(partial);
        auto master_idp = clos->GetIdp(0);
        agent->Send(master_idp, msg);
    }

    delete pool;
}
```

Código 6.4.: Processamento distribuído de uma TChain com passagem de mensagens.

recurso *clausura* com um total de `np` *agentes* como membros. Assim, o conjunto inicial de ficheiros em TTree `(in,fin)` a processar é redistribuído em (10) por cada um dos *agentes* iniciais criados nos *domínios* lançados, no que resulta num cálculo parcial efetuado em cada *domínio* para a variável `partial` em (15). Este cálculo é efetuado da mesma forma que o descrito anteriormente, ficando por obter o resultado final agregado dos vários cálculos parciais de cada *domínio*. Porém, esta questão não pode ser tratada sem ter em consideração que o somatório final tem de lidar com concorrência no tempo e no espaço dos múltiplos *domínios* locais/remotos.

Inicialmente, é necessário obter o idp do *agente* na *clausura* (23) formada pelos domínios lançados em paralelo. Seguidamente, nas linhas (26-31) atribui-se ao agente principal (*master*) a responsabilidade pela obtenção dos cálculos parciais e a acumulação dos mesmos para

a variável `final_value`, valor que é impresso em (32). Os outros *agentes* (34-37) enviam os resultados parciais ao *agente* principal.

Para a receção e envio dos dados parciais são usados invólucros `msg` em (28) e (34). No caso do envio, é adicionado o respetivo cálculo `partial` através do método `Add` (35), no qual é preciso explicitar o tipo dos dados. Posteriormente, é obtido o idp do destinatário (36), neste caso o *agente* principal, que corresponde ao *agente* com idm `0` no contexto da *clausura* de arranque inicial, e é enviada a mensagem (37). O *agente* principal recebe em (28) as mensagens de cada um dos demais *agentes* (27), das quais retira os respetivos valores parciais (29), que vai acumulando sucessivamente para a variável `final_value` (30).

O método `Receive` permite, por um lado, a partilha dos valores entre *agentes* locais/remotos para o cálculo do valor final e, por outro lado, assegurar a necessária sincronização. Com efeito, a primitiva de comunicação é bloqueante, pelo que o ciclo em (27) só termina após todos os *agentes* terem processado as respetivas TTree e, subsequentemente, terem enviado os respetivos cálculos parciais. Convém notar que no envio de mensagens (37) foi usado o `master_idp` do *agente* principal, no entanto na receção (28) não foi usada qualquer identificação. Assim, foi utilizada a passagem de mensagens através de uma identificação global e única no sistema de domínios distribuídos.

6.3.3 *Comunicação por contexto*

Em PlaCoR, é introduzida a comunicação por contexto, usada noutros sistemas, para garantir a modularidade do código e a possibilidade de um recurso poder participar em múltiplas *clausuras*, sendo identificado em cada uma delas pelo seu idm, de forma semelhante à usada em MPI para comunicação entre processos pertencentes ao mesmo comunicador. Esta alternativa, no código, corresponderia à substituição da linha (28) para `agent->Receive(i, comm_idp)`, de forma a receber uma mensagem de cada um dos restantes *agentes* presentes na clausura, e da linha (37) para `agent->Send(0, comm_idp, msg)`, para que cada um dos restantes *agentes* envie a mensagem ao *agente* principal da *clausura*. Esta primitiva de receção por contexto mantém as características de comunicação bloqueante, que garantem a sincronização entre produtores/consumidores de mensagens.

6.3.4 *Processamento distribuído com sincronização e memória partilhada distribuída*

A passagem de mensagens é habitualmente usada em sistemas de memória distribuída como forma de sincronização entre entidades ativas. A versão que se segue corresponde a um refinamento do código anterior, que combina a memória partilhada distribuída com uma outra forma de sincronização, assente nas propriedades do recurso *clausura*.

O fragmento de código em 6.5, é uma variante que mantém intactas as linhas referentes ao cálculo parcial do processamento da TChain, e releva o papel dos recursos *dado* e *barreira* no contexto da orientação ao recurso.

```
1   auto domain = cor::GetDomain();
2   auto data = domain->CreateLocal<cor::Data<Int_t>>(domain.Idp(), "data", partial);
3
4   auto meta_domain = domain->GetLocalResource<cor::Domain>(cor::MetaDomain);
5   auto member_list = meta_domain->GetMemberList();
6   member_list.erase(std::remove(member_list.begin(), member_list.end(), cor::MetaDomain),
7       member_list.end());
8
9   auto barrier = domain->CreateCollective<cor::Barrier>(clos_idp, domain.Idp(),
10      "barrier", clos_idp);
11  barrier->Synchronize();
12
13  if (rank == 0) {
14      Int_t final_value = partial;
15      for (auto idp: member_list) {
16          if (idp != domain.Idp()) {
17              auto data_name = TString::Format("data_%i", idp);
18              auto remote_domain = domain->CreateReference<cor::Domain>(idp,
19                  domain.Idp(), "");
20              auto data_idp = remote_domain->GetIdp("data");
21              auto data = domain->CreateReference<cor::Data<Int_t>>(data_idp,
22                  domain.Idp(), data_name.Data());
23              data->AcquireRead();
24              final_value += *data->Get();
25              data->ReleaseRead();
26          }
27      }
28      std::cout << "Final Value: " << final_value << std::endl;
29  }
```

Código 6.5.: Processamento distribuído de uma TChain com sincronização e memória partilhada distribuída.

Em todos os *domínios* intervenientes é criado um *dado* `data` (2) que irá albergar um inteiro, pendurado no contexto do *domínio* local com nome `"data"`, cujo valor inicial corresponde ao resultado do cálculo parcial do processamento das TTree presente na variável `partial`. As linhas (4-5) permitem conhecer a lista de membros do *meta-domínio*, da qual é excluído o próprio em (6-7), o que corresponde na prática ao catálogo de todos os *domínios* presentes na aplicação. No que segue (9-10) há lugar para a chamada a uma operação coletiva que envolve todos os membros da *clausura* inicial, identificada por `clos_idp`, para criar um recurso *barreira* `barrier` com nome `"barrier"`, sendo pendurada uma cópia da mesma em cada um dos *domínios* locais, que irá atuar com os recursos membros da *clausura* identificada por `clos_idp`. Veremos mais adiante a utilização desta primitiva para a criação coletiva de outros tipos de recursos.

A evocação, em (11), do método `Synchronize` tem como resultado uma sincronização entre todos os *agentes* pertencentes à *clausura* associada à *barreira*, terminando apenas com a "chegada" da totalidade dos seus pares. Garantida a sincronização, é o momento para o *agente* principal proceder à obtenção dos dados com os cálculos parciais distribuídos pelos

demais *domínios*. Assim, após a criação de uma referência local (18-19) de cada um dos *domínios* da aplicação, é obtido em (20) o respetivo idp do dado `data_idp`, através do seu nome no contexto do *domínio*, e posteriormente uma referência local (21-22) para o respetivo recurso, presente na variável `data`. Tratando-se de um recurso *dado*, acessível quer local como remotamente, é necessário utilizar devidamente os métodos para adquirir e libertar o direito de leitura sobre o mesmo, em (23) e (25) respetivamente, apesar de no exemplo não estar prevista alguma leitura/escrita concorrente. Finalmente, é usado o método `Get` para obter o cálculo parcial (24), e adicionado o mesmo ao `final_value`.

6.3.5 *Refinamento ao processamento distribuído com criação coletiva de recursos dado*

O recurso a operações coletivas de criação de recursos sugere uma variante à versão apresentada acima, que corresponde à alteração de código visível em 6.6.

```
1     auto domain = cor::GetDomain();
2     auto agent_idp = domain->GetActiveResourceIdp();
3     auto clos_idp = domain->GetPredecessorIdp(agent_idp);
4     auto rank = GetRank<cor::Closure>(agent_idp);
5
6     auto data = domain->CreateCollective<cor::Data<Int_t>>(clos_idp, domain.Idp(), "data", 0);
7     auto barrier = domain->CreateCollective<cor::Barrier>(clos_idp, domain.Idp(), "barrier", clos_idp);
8
9     data->AcquireWrite();
10    *data->Get() += partial;
11    data->ReleaseWrite();
12
13    barrier->Synchronize();
14
15    if (rank == 0) {
16        data->AcquireRead();
17        auto final_value = *data->Get();
18        data->ReleaseRead();
19        std::cout << "Final Value: " << final_value << std::endl;
20    }
```

Código 6.6.: Processamento distribuído de uma TChain com criação coletiva de recursos dado.

Contrariamente à abordagem anterior, na qual em cada *domínio* era criado um recurso *dado* com o cálculo parcial do processamento das TTree, é agora utilizada a operação `CreateCollective` sobre todos os membros da *clausura* inicial, para criar um recurso *dado* `data` (6) com nome `"data"`, pendurado em cada um dos *domínios* locais, iniciado com o valor `0`. Uma vez que o recurso *dado* é acessível e modificado concorrentemente pelos *agentes* presentes em cada um dos *domínios*, é necessário garantir a consistência dos mesmos num ambiente DSM, utilizando as primitivas de aquisição e libertação do direito de escrita sob o mesmo em (9) e (11), respetivamente. O valor do recurso *dado* é atualizado em (10) com o valor parcial calculado em cada *domínio*.

Seguidamente, é necessário recorrer à primitiva coletiva `Synchronize` para garantir a sincronização entre todos os *agentes*, de forma a que o *agente* principal possa obter devida-

mente o valor final acumulado dos cálculos parciais. É então adquirido o direito de leitura (16) sobre o *dado*, de forma a garantir a consistência da leitura do valor, recorrendo à primitiva `Get` (17) para obter o valor final para a variável `final_value`, procedendo à libertação do direito de leitura em (18). Por fim, em (19) é imprimido o valor do resultado final.

6.3.6 *Clausuras e sincronização*

O PlaCoR é um sistema híbrido em que o paralelismo tradicional dos processos e do grão fino dos fios de execução dão lugar, respetivamente, a um sistema de domínios distribuídos e à interação entre *agentes*. O exemplo que segue releva a importância dos recursos *clausura* e *barreira*, enquanto ferramentas incontornáveis nos processos de comunicação e de sincronização, associados à programação concorrente/paralela e distribuída.

Consideremos a biblioteca dinâmica `libinter_closure.so`, que corresponde ao excerto de código em 6.7, e o resultado na execução da linha de comando abaixo:

```
$ corun --np 2 libinter_closure.so 3
Agent:     | Initial Closure: size  rank | Spawned Closure: size  rank | Global Closure: size  rank
4294966782 | 4294966783       2     0    |                             | 4294966780       5     0
4294967038 | 4294966783       2     1    |                             | 4294966780       5     1
4294966526 |                             | 4294966015       3     1    | 4294966780       5     3
4294966014 |                             | 4294966015       3     0    | 4294966780       5     4
4294966270 |                             | 4294966015       3     2    | 4294966780       5     2
```

O código foi desenvolvido para ilustrar a possibilidade de combinar as *clausuras* parciais, correspondentes ao arranque estático *Initial Closure* e ao arranque dinâmico *Spawned Closure*, para, desta forma, criar um novo contexto fechado *Global Closure* que irá incluir todos os *agentes* associados aos contextos parciais. O objetivo é, por exemplo, permitir a comunicação entre *agentes*, identificados por um idm (*rank*) na nova *clausura* - situação análoga à utilizada pela primitiva MPI_Intercomm_merge do MPI para criar um *intracommunicator* à custa de um *intercommunicator*.

O `corun` arranca dois *domínios* com o código do módulo `libinter_closure.so` associado, a que é passado o argumento `3` que estabelece o número de *domínios* que irão ser lançados dinamicamente. O primeiro grupo de *domínios* é automaticamente associado à *Initial Closure*, e o segundo à *Spawned Closure*. No arranque de ambos os grupos, as linhas (8-13) são usadas para estabelecer as condições iniciais do programa, nomeadamente: em (8) o idp do *agente*; em (10) a *clausura* correspondente; em (11) o tamanho da *clausura*; em (12) o `rank` do *agente* na *clausura*; e em (13) o `parent_idp` que permite distinguir os dois grupos de *agentes* e consequentemente *domínios* na condição em (18). A condição em (21) vai permitir selecionar o *agente* principal do primeiro grupo de domínios, que será responsável pelo lançamento dinâmico do segundo grupo de domínios, já que de momento o método `Spawn` não é coletivo no contexto de uma *clausura*.

```
int spawned_domains = 2;

void Main(int argc, char *argv[])
{
    if (argc >=1) { spawned_domains = std::atoi(argv[0]); }

    auto domain = cor::GetDomain();
    auto agent_idp = domain->GetActiveResourceIdp();
    auto clos_idp = domain->GetPredecessorIdp(agent_idp);
    auto clos = domain->GetLocalResource<cor::Closure>(clos_idp);
    auto clos_size = clos->GetTotalMembers();
    auto rank = clos->GetIdm(agent_idp);
    auto parent_idp = clos->GetParent();

    cor::ResourcePtr<cor::Closure> global_clos;
    idp_t master_domain_idp;

    if (parent_idp == 0) {
        auto initial_barrier = domain->CreateCollective<cor::Barrier>(clos_idp,
            domain->Idp(), "Initial Barrier", clos_idp);
        if (rank == 0) {
            global_clos = domain->CreateLocal<cor::Closure>(domain.Idp(),
                "Global Closure", clos_size+spawned_domains, agent_idp);
            auto remote_clos_idp = domain->Spawn("ctx2", spawned_domains,
                "~/placor/cap6/libinter_closure.so", {}, { "localhost" });
        } else {
            auto master_idp = clos->GetIdp(0);
            auto master_clos_idp = domain->GetPredecessorIdp(master_idp);
            master_domain_idp = domain->GetPredecessorIdp(master_clos_idp);
        }
        initial_barrier->Synchronize();
    } else {
        auto master_clos_idp = domain->GetPredecessorIdp(parent_idp);
        master_domain_idp = domain->GetPredecessorIdp(master_clos_idp);
    }

    auto master_domain = (parent_idp == 0 && rank == 0) ? domain :
        domain->CreateReference<cor::Domain>(master_domain_idp, domain.Idp(), "Remote Domain");

    auto global_clos_idp = master_domain->GetIdp("Global Closure");
    if (!(parent_idp == 0 && rank == 0))
        global_clos = domain->CreateReference<cor::Closure>(global_clos_idp,
            domain.Idp(), "Global Closure");

    auto name = std::to_string(agent_idp) + ((parent_idp == 0) ? "" : "_s");
    global_clos->Join(agent_idp, name);

    auto global_barrier = domain->CreateCollective<cor::Barrier>(global_clos_idp,
        domain.Idp(), "Global Barrier", global_clos_idp);
    global_barrier->Synchronize();

    PrintResultTable();
}
```

Código 6.7.: Exemplo de criação de um novo contexto fechado com base em duas clausuras.

A sincronização entre *agentes* pertencentes à *Initial Closure* é estabelecida com base na criação de uma *barreira* em (19) associada àquela *clausura*, e posterior evocação do método Synchronize, chamado em (31). Notar que o *agente* principal é também responsável por criar (22) a *clausura Global Closure*, com capacidade para todos os membros de ambas as *clausuras*, que servirá de base para a construção do contexto fechado constituído pelos

agentes de todos os *domínios* (estáticos e dinâmicos). As linhas (27-29) são executados pelos restantes *agentes* da *clausura* inicial, de forma a obterem o idp do *domínio* no qual a *Global Closure* foi criada. O mesmo vai ser efetuado pelos *agentes* pertencentes ao segundo grupo de domínios, nas linhas (33-34).

Tendo todos os *agentes* da aplicação informação em relação ao idp da *Global Closure*, resta então garantir que todos possuem uma cópia local da mesma. Inicialmente, em (37-38) é obtida uma cópia local do *domínio* que contém a *clausura* global (exceto pelo *agente* criador), que será usada para obter uma referência local para a mesma em (41-43), permitindo que todos os *agentes* adiram (46) sucessivamente àquele contexto. Finalmente, em (48-49) é feita a criação da *barreira GlobalBarrier* de forma coletiva, tendo como base a *clausura* global, para que cada um dos *agentes* obtenha uma cópia local da barreira, para posteriormente ser efetuada a sincronização (50) entre os demais *agentes* pertencentes àquele contexto fechado. A função `PrintResultTable` (52) serve para produzir a saída correspondente à execução do comando `corun` apresentado no início desta subsecção.

De referir que, no exemplo, a criação da primeira *barreira* (19-20) e o seu posicionamento no código são cirúrgicos. O objetivo é assegurar que, no caso de `np>1`, os *agentes* pertencentes ao primeiro grupo de domínios só poderão tentar obter o idp da *clausura* global em (40) depois do recurso ter sido efetivamente criado em (22-23) pelo *agente* principal, o que é alcançado recorrendo à operação de sincronização em (31). No que diz respeito à interação com os *agentes* no outro grupo de domínios, a posição relativa do código que faz o arranque dinâmico em (24-25) em relação à criação da *clausura* global, e o facto do método `Spawn` ser síncrono, garante a não existência de problemas no acesso aquele recurso.

6.3.7 *Ativação remota de agentes*

O PlaCoR, enquanto sistema de domínios distribuídos, oferece não apenas a criação remota de recursos, mas também a criação de cópias locais de recursos. No capítulo 5, foram introduzidos métodos diretamente relacionados com estas facilidades, como é o caso de: `CreateRemote`, usado para criar remotamente um novo recurso no contexto de um recurso estruturado remoto, devolvendo o identificador global do recurso criado; ou `CreateReference`, usado para criar um réplica, no *domínio* local, de um recurso identificado pelo idp.

No caso específico de recursos baseados no elemento *executor*, para correr, esperar e obter os resultados da execução da função que lhes foi associada, em *domínios* remotos, foi necessário introduzir na API um sistema de ativação remota de *agentes* baseado em mecanismos de RPC. Segue-se um exemplo, sustentado por dois módulos cliente-servidor, especialmente desenhados para ilustrar a API disponibilizada pela plataforma. Ambos os códigos, para além da função de entrada do módulo, incluem duas funções auxiliares para ilustrar

a ativação remota de *agentes*, respetivamente `ClientFunction` e `ServerFunction`. Abaixo, é explicitado um possível resultado da utilização do comando `corun` para a execução do exemplo:

```
$ corun --np 1 libclient_rpc.so
From Client: 4294967035
Return From Server: 4294967035
```

6.3.7.1 *Cliente*

O código do cliente, presente em 6.8, para além da já descrita evocação dos métodos da API para conhecer os habituais recursos e identificadores (16-22), inclui o `Spawn` na máquina `"compute-0-1"` de um *domínio* dinâmico (28-29), com o módulo `libserver_rpc.so`, onde será criado um *agente* (35-36) que irá executar no *domínio* remoto a função `ServerFunction` presente no módulo associado. As próximas linhas são usadas para correr a função remota (37) com o argumento `domain_idp`, esperar pela conclusão da execução da função (38), e obter o valor de retorno da função (39) para a variável `res`, que será impressa em (40).

Nos passos seguintes, o agente junta-se à *clausura* global (43), criada em (25-26), para sincronizar os *agentes* principais do cliente e do servidor, ao qual se segue a criação coletiva de uma *barreira* (44-45) que irá ser usada para em (46) sincronizar as atividades dos programas `cliente_rpc` e `server_rpc`. Da mesma forma que a função `Main`, a função `ClientFunction` é declarada na diretiva `extern "C"` em (4), de forma a poder ser chamada remotamente pelo programa `server_rpc` no contexto de uma biblioteca dinâmica. O corpo da função apenas imprime o identificador principal passado como argumento.

6.3.7.2 *Servidor*

O código do servidor, ilustrado em 6.9, tal como o do cliente, inclui (20-24) um conjunto habitual de definições, seguidas da obtenção de uma cópia local do *domínio* (29-30) na origem do arranque dinâmico deste módulo, para obter uma referência local para a *clausura* global (32-33) criada pelo módulo `client_rpc`. Seguidamente, o respetivo *agente* principal junta-se à *clausura* (35), para posterior criação coletiva da *barreira* (36-37) anteriormente explicitada, que irá ser usada para estabelecer a sincronização entre ambos os *agentes* principais (38) de cada módulo.

A função ServerFunction, que irá ser chamada remotamente pelo módulo `client_rpc`, vai lançar remotamente no *domínio* identificado por `domain_idp`, previsivelmente o *domínio* do módulo do cliente, um *agente* (10-11) para executar a função `ClientFunction` presente no módulo `libclient_rpc.so`. Segue-se-lhe a execução, espera e obtenção do valor de retorno da função (12-14), neste caso sem valor de retorno, terminando a função com o retorno do idp do *agente* remoto. A utilização do método `Get`, não contendo valor retorno, é aconselhável para detetar eventuais exceções lançadas pela função executada remotamente.

```
extern "C"
{
    void Main(int argc, char *argv[]);
    void ClientFunction(idp_t idp);
}

void ClientFunction(idp_t idp)
{
    std::cout << "From Client: " << idp << std::endl;
}

const int spawned_domains = 1;

void Main(int argc, char *argv[])
{
    auto domain = cor::GetDomain();

    auto domain_idp = domain->Idp();
    auto agent_idp = domain->GetActiveResourceIdp();
    auto agent = domain->GetLocalResource<cor::Agent<void(int,char**)>>(agent_idp);
    auto clos_idp = domain->GetPredecessorIdp(agent_idp);
    auto clos = domain->GetLocalResource<cor::Closure>(clos_idp);
    auto clos_size = clos->GetTotalMembers();

    auto global_clos = domain->CreateLocal<cor::Closure>(domain->Idp(), "Global Closure",
        clos_size+spawned_domains, agent_idp);
    auto global_clos_idp = global_clos->Idp();
    auto server_clos_idp = domain->Spawn("server", spawned_domains,
        "~/placor/cap6/libserver_rpc.so", {}, { "compute-0-1" });
    auto server_clos = domain->CreateReference<cor::Closure>(server_clos_idp,
        domain->Idp(), "Server Closure");
    auto server_domain_idp = domain->GetPredecessorIdp(server_clos_idp);

    {
        auto rsc_idp = domain->Create<cor::Agent<idp_t(idp_t)>>(server_domain_idp, "",
            "libserver_rpc.so", "ServerFunction");
        domain->Run<idp_t(idp_t)>(rsc_idp, domain_idp);
        domain->Wait<idp_t(idp_t)>(rsc_idp);
        auto res = domain->Get<idp_t(idp_t)>(rsc_idp);
        std::cout << "Return From Server: " << res << "\n";
    }

    global_clos->Join(agent_idp, "client");
    auto barrier = domain->CreateCollective<cor::Barrier>(global_clos_idp, domain->Idp(),
        "Barrier", global_clos_idp);
    barrier->Synchronize();
}
```

Código 6.8.: Exemplo de um cliente (`client_rpc`) para a ativação remota de agentes.

6.3.8 *Mecanismos RPC*

A ativação remota de *agentes* é suportada por uma adaptação da biblioteca `czrpc` (ver capítulo 4). Dentro das facilidades disponibilizadas destacam-se a deteção, em tempo de compilação, de chamadas RPC inválidas, tais como: nomes de funções desconhecidos, número incorreto de parâmetros ou tipos de parâmetros não suportados, e a possibilidade do cliente saber se a chamada provocou alguma exceção. A biblioteca é bidirecional, o que ga-

```
extern "C"
{
    void Main(int argc, char *argv[]);
    idp_t ServerFunction(idp_t idp);
}

idp_t ServerFunction(idp_t domain_idp)
{
    auto domain = cor::GetDomain();
    auto rsc_idp = domain->Create<cor::ProtoAgent<void(idp_t)>>(domain_idp, "",
        "libclient_rpc.so", "ClientFunction");
    domain->Run<void(idp_t)>(rsc_idp, rsc_idp);
    domain->Wait<void(idp_t)>(rsc_idp);
    domain->Get<void(idp_t)>(rsc_idp);
    return rsc_idp;
}

void Main(int argc, char *argv[])
{
    auto domain = cor::GetDomain();
    auto agent_idp = domain->GetActiveResourceIdp();
    auto clos_idp = domain->GetPredecessorIdp(agent_idp);
    auto clos = domain->GetLocalResource<cor::Closure>(clos_idp);
    auto agent = domain->GetLocalResource<cor::Agent<void(int,char**)>>(agent_idp);
    auto parent_idp = clos->GetParent();

    auto client_clos_idp = domain->GetPredecessorIdp(parent_idp);
    auto client_domain_idp= domain->GetPredecessorIdp(client_clos_idp);
    auto client_domain = domain->CreateReference<cor::Domain>(client_domain_idp,
        domain->Idp(), "Client Domain");
    auto global_clos_idp = client_domain->GetIdp("Global Closure");
    auto global_clos = domain->CreateReference<cor::Closure>(global_clos_idp,
        domain->Idp(), "Global Closure");

    global_clos->Join(agent_idp, "server");
    auto barrier = domain->CreateCollective<cor::Barrier>(global_clos_idp,
        domain->Idp(), "Barrier", global_clos_idp);
    barrier->Synchronize();
}
```

Código 6.9.: Exemplo de um servidor (`server_rpc`) para a ativação remota de agentes.

rante a possibilidade de um servidor também poder atuar como um cliente, e assim poder ativar *agentes* remotamente.

A utilização de tipos de dados não previstos pela biblioteca obriga à escrita de código específico, sendo também da responsabilidade do programador registar recursos que irão ser utilizados para criações remotas e a sua respetiva assinatura, assim como as funções que irão ser usadas para efetuar as chamadas. Para o efeito, a plataforma disponibiliza a tabela `rpc_table.hpp`, que deverá ser atualizada de acordo com os requisitos de programação. Em 6.10 destacam-se, nas suas variantes, as assinaturas dos vários recursos com suporte à criação remota, bem como as definições do elemento *executor*, sendo estas utilizadas para fazer a ativação remota dos *agentes*, como demonstrado no exemplo acima de cliente/servidor.

O suporte à criação de recursos remotos é feito através do registo dos mesmos com a macro `CREATE`. Para a execução, espera e obtenção do resultado da execução de funções

remotas, o registo é feito utilizando as macros RUN, WAIT e GET, respetivamente, com o elemento *executor* e a assinatura da função a suportar. Todas estas macros são constituídas por três campos: o primeiro corresponde ao tipo de recurso, no caso da macro CREATE, ou ao elemento executor e assinatura da função, no caso das macros RUN, WAIT e GET; o segundo corresponde a um nome único atribuído à função para chamada remota; o terceiro corresponde à função a suportar.

```
#define RPCTABLE_CLASS cor::RPC
#define RPCTABLE_CONTENTS                                                                              \
    CREATE(Group, group, Create<cor::Group, std::string const&>)                                       \
    CREATE(Data<std::vector<int>>, dataint, Create<cor::Data<std::vector<int>>, std::vector<int>>)    \
    CREATE(Agent<void(idp_t)>, agent1,                                                                 \
        Create<cor::Agent<void(idp_t)>, std::string const&, std::string const&>)                       \
    CREATE(Agent<idp_t(idp_t)>, agent2,                                                                \
        Create<cor::Agent<idp_t(idp_t)>, std::string const&, std::string const&>)                      \
    CREATE(ProtoAgent<void(idp_t)>, protoagent1,                                                       \
        Create<cor::ProtoAgent<void(idp_t)>, std::string const&, std::string const&>)                  \
    CREATE(ProtoAgent<idp_t(idp_t)>, protoagent2,                                                      \
        Create<cor::ProtoAgent<idp_t(idp_t)>, std::string const&, std::string const&>)                 \
    RUN(Executor<void(idp_t)>, run1, Run<void(idp_t), idp_t>)                                          \
    RUN(Executor<idp_t(idp_t)>, run2, Run<idp_t(idp_t), idp_t>)                                        \
    WAIT(Executor<void(idp_t)>, wait1, Wait<void(idp_t)>)                                              \
    WAIT(Executor<idp_t(idp_t)>, wait2, Wait<idp_t(idp_t)>)                                            \
    GET(Executor<void(idp_t)>, get1, Get<void, void(idp_t)>)                                           \
    GET(Executor<idp_t(idp_t)>, get2, Get<idp_t, idp_t(idp_t)>)
```

Código 6.10.: Exemplo para o registo de recursos para criação remota e de funções executadas por agentes para ativação remota.

7

DISCUSSÃO E PERSPETIVAS FUTURAS

Esta dissertação consubstancia-se na criação de uma plataforma baseada no modelo de computação orientado ao recurso que, naturalmente, assenta não apenas na especificação CoRes, mas também nas tentativas prévias, respetivamente, pCoR e pyCoR. O PlaCoR constitui-se, assim, como uma nova abordagem indissociável do legado recebido dos desenvolvimentos anteriores, marcada pela escolha da linguagem C++ Moderno e de um conjunto significativo de outras bibliotecas e ferramentas que tornaram possível a concretização da plataforma.

O ponto de partida para a conceção e desenho é a especificação do recurso em CoRes, como uma entidade que é "uma metáfora genérica" que, ao mesmo tempo, incorpora diretamente a noção de estado, concorrência, localidade e distribuição. Esta perspetiva veio concretizar-se na definição do recurso, como um objeto, cujo comportamento é descrito por uma classe que herda por múltipla herança, as funcionalidades das suas classes de base. Desta forma, a interação entre recursos faz-se sem ter de conhecer à partida os tipos específicos de recursos, bastando conhecer os elementos que os constituem.

Numa outra dimensão, procuramos também simplificar a API, de forma a criar e lidar com os recursos de forma genérica, o que se traduziu na exploração do paradigma da programação genérica, presente na linguagem C++, através dos *templates*. Em termos práticos, assegura-se a possibilidade de utilização de uma grande variedade de tipos de dados e, ao mesmo tempo, favorece-se a deteção de erros em tempo de compilação.

Ao nível da infraestrutura de sistema, existem um conjunto de serviços altamente críticos, relacionados com a geração de identificadores e a gestão de consistência entre as múltiplas réplicas de uma mesma entidade, num sistema de domínios distribuídos. Neste contexto, a escolha e seleção de uma biblioteca de comunicação por grupos foi determinante para assegurar a interligação e sincronização, entre os diferentes membros de grupos, que correspondem às funções dos serviços, garantindo confiabilidade e consistência na aplicação e a deteção de falhas.

A existência de múltiplas réplicas, disseminadas pelos vários domínios locais ou remotos, levanta a questão da serialização de objetos. Neste tema, o requisito é a possibilidade de permitir a serialização de quaisquer tipos de objetos, incluindo os definidos pelo próprio

programador, compatível com um ambiente multiplataforma, em particular Linux, Solaris e MacOS.

A introdução do conceito de arranque estático paralelo de aplicações traduziu-se na necessidade de adaptação da plataforma a esta exigência, que foi posteriormente estendida para contemplar o arranque dinâmico de domínios (Spawn). Na base da solução encontrada, está a utilização do protocolo SSH para a construção de um serviço de gestão de conexões, que permite lançar aplicações em máquinas locais/remotas.

O modelo CoR pressupõe ainda a possibilidade de criar recursos remotamente, o que veio a ser concretizado através da introdução de um serviço RPC, mas apenas na fase final deste trabalho. Uma vez que todo o desenvolvimento anterior estava condicionado pela utilização de programação genérica, foi necessário encontrar alternativas compatíveis com esta exigência. Estas teriam de responder não só à necessidade da criação remota de recursos, como também ao lançamento remoto de agentes, esperando pelo término dos mesmos e a obtenção do valor de retorno previsto na assinatura das funções associadas.

Atualmente, o desenho de aplicações é estruturado com base nas seguintes classes de recursos: domínio, grupo, clausura, agente, proto-agente, dado, barreira, guarda e guarda para leituras/escritas. Os domínios estabelecem o primeiro nível de concorrência/paralelismo, quer sejam lançados estática ou dinamicamente. Os recursos do tipo agente estão associados ao grão fino de paralelismo e de comunicação por passagem de mensagens, correspondendo a fios de execução. Os recursos do tipo guarda disponibilizam facilidade para a criação de zonas de exclusão mútua distribuídas (para leituras/escritas), estando o recurso barreira ligado à necessidade de sincronização entre agentes.

O domínio, grupo e clausura correspondem a recursos estruturados que disponibilizam operações de adesão/saída de recursos. Distingue-os o facto dos dois primeiros serem dinâmicos enquanto a clausura é estática, na medida em que o número total de membros é fixado inicialmente, sendo as operações de adesão/saída coletivas - na base da possibilidade de arranque de aplicações do tipo SPMD e a comunicação por passagem de mensagens intra-clausura. O recurso dado reflete os mecanismos de memória partilhada distribuída, usado para disponibilizar os dados do utilizador num ambiente de domínios distribuídos.

Todo o recurso possui uma identificação global (idp) juntamente com um identificador de membro (idm) e, opcionalmente, um nome no contexto do seu recurso ascendente. Assim, a criação de um qualquer recurso faz-se sempre no contexto de um outro recurso, o que implica a especificação do seu ascendente, obrigatoriamente um recurso estruturado. A identificação global pode ser usada diretamente para a comunicação por passagem de mensagens entre quaisquer dois recursos comunicantes. A utilização do idm para a passagem de mensagens está reservada aos recursos membros de clausuras, através da combinação do idp da clausura e o idm do recurso membro no contexto, nas operações de emissão/receção de mensagens.

Para a avaliação da plataforma, foi usado como exemplo a leitura e processamento de eventos, registados em coleções de TTree organizadas numa TChain, recorrentemente utilizados em física das altas energias (HEP), nomeadamente na experiência ATLAS do CERN. As várias versões desenvolvidas, que exploram um conjunto significativo de métodos da API, foram o pretexto para a criação de unidades (*units*), com é o caso da unidade Pool que realiza o modelo *fork-join*.

A experiência veio confirmar a viabilidade da utilização da orientação ao recurso como um paradigma de programação híbrido, baseado em recursos, que integra múltiplos fios de execução e sincronização distribuída, com facilidades de comunicação de grão fino por passagem de mensagens e de comunicação em contextos seguros, o acesso remoto a memória e a ativação remota de agentes.

7.1 CONCLUSÕES

Como antes foi referido, a especificação do modelo de orientação ao recurso, em CoRes, é extremamente sofisticada e exigente. Assim, em termos práticos, não foi possível abordar e realizar toda a extensão da especificação. No entanto, foram introduzidas novas funcionalidades não previstas originalmente.

A plataforma foi construída integralmente em C++ Moderno, tirando partido dos múltiplos paradigmas que a linguagem oferece, nomeadamente os paradigmas da programação orientada ao objeto e da programação genérica. Foi, assim, possível desenhar uma plataforma em que as entidades do modelo seguem de perto a anatomia do recurso, com uma API orientada aos objetos. De referir, também, a inclusão da técnica RAII, usada para a desalocação automática de recursos.

Os conceitos associados aos elementos sincronizador e organizador foram revistos, dando origem a classes distintas que partilham o mesmo tipo de classe de recurso, mas que têm funcionalidades distintas. O elemento organizador deu origem ao organizador dinâmico e estático. O organizador dinâmico possui as mesmas funcionalidades que o organizador especificado em CoRes. Já o organizador estático foi introduzido com o intuito de criar contextos fechados com uma aridade previamente estabelecida. Este novo tipo de elemento, classe de base do recurso clausura, foi determinante no estabelecimento das condições para o lançamento de aplicações do tipo SPMD. Esta possibilidade, uma das grandes contribuições deste trabalho, equivalente à prevista na norma MPI, permite o lançamento paralelo de aplicações: estático - através da ferramenta `corun` - e dinâmico - através da primitiva `Spawn`.

Foram também definidas novas entidades: os recursos agente, proto-agente e guarda para leituras/escritas. O primeiro é a renomeação do recurso tarefa presente em CoRes, enquanto que o proto-agente é um recurso ativo associado a um fio de execução sem faci-

lidades de comunicação. O último recurso estende a guarda, para contemplar regiões de exclusão mútua para leituras e escritas.

Apesar de não ser uma contribuição deste trabalho, a configuração de toda a plataforma, através da ferramenta CMake, e a utilização de bibliotecas dinâmicas, são uma mais valia deste projeto, uma vez que não apenas facilita a compilação e desenvolvimento de aplicações orientadas ao recurso, como garante a portabilidade da plataforma para diferentes ambientes, nomeadamente Linux, Solaris e MacOS.

7.2 TRABALHO FUTURO

Com base no trabalho desenvolvido, é possível enunciar algumas perspetivas de trabalho futuro. Estas incluem, necessariamente, novas funcionalidades a inserir no protótipo que passam a ser apresentadas:

- desenvolvimento de um sistema de suporte à execução de aplicações mais sofisticado, de forma a melhorar o suporte à criação de *pods* e à gestão dos nós de computação alocados à aplicação;
- estudo e definição de novos elementos/recursos que contemplem os modelos atuais utilizados na programação paralela, tais como os modelos *thread-centric* e *task-centric*, num ambiente de memória partilhada e distribuída;
- extensão da API com novas facilidades para a interação com os recursos, nomeadamente primitivas de comunicação não bloqueantes e coletivas;
- introdução do elemento porto, previsto em CoRes, para permitir tirar partido da eventual existência de transportes de comunicação específicos, de forma a aumentar o desempenho das primitivas de comunicação por passagem de mensagens;
- reformulação do suporte à criação e ativação remota de agentes, para a uniformização da API de serialização.

BIBLIOGRAFIA

[1] António Pina. *MC^2 - Modelo de Computação Celular - Origem e Evolução*. PhD thesis, Universidade do Minho, Braga, Portugal, 1997.

[2] Cecília Moreira. CoRes - Computação orientada ao Recurso - uma especificação. Master's thesis, Universidade do Minho, Braga, Portugal, 2001.

[3] Patrícia Gomes Soares. On remote procedure call. In *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research - Volume 2*, CASCON '92, pages 215–267. IBM Press, 1992. URL http://dl.acm.org/citation.cfm?id=962260.962276.

[4] S. Navaratnam, S. Chanson, and G. Neufeld. Reliable group communication in distributed systems. In *[1988] Proceedings. The 8th International Conference on Distributed*, pages 439–446, June 1988. doi: 10.1109/DCS.1988.12546.

[5] Eric Youngdale. Kernel korner: The elf object file format: Introduction. *Linux J.*, 1995 (12es), April 1995. ISSN 1075-3583.

[6] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, Aug 1991. ISSN 0018-9162. doi: 10.1109/2.84877.

[7] MPI-Forum. MPI: A Message-Passing Interface Standard. Version 3.1. 2015.

[8] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[9] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[10] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. "O'Reilly Media, Inc.", 2007.

[11] A. Pina, V. Oliveira, C. Moreira, and A. Alves. pCoR: a prototype for Resource oriented Computing. In *7th International Conference on Applications of High-Performance Computers in Engineering (HPC '02)*, pages 251–262, Bologna, Italy, September 2002. WIT Press.

[12] António Pina. Comunicação por grupo e orientação ao objeto em cor. Technical report, Universidade Minho, Braga, Portugal, 2012.

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation, MIT Pres, 1994.

[14] António Pina, Cecília Moreira, and Vítor Oliveira. Domains, threads and shared memory in a message passing environment. Technical report, Universidade Minho, Braga, Portugal, 1997.

[15] Adam Ferrari and Vaidy S. Sunderam. Tpvm: Distributed concurrent computing with lighweight processes. pages 211–, 01 1995. doi: 10.1109/HPDC.1995.518712.

[16] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013. ISBN 0321563840, 9780321563842.

[17] Y. Amir and J. Stanton. The spread wide area group communication system. Technical report, 1998.

[18] Yair Amir, C Danilov, M Miskin-amir, J Schultz, and J Stanton. The spread toolkit: Architecture and performance. 01 2004.

[19] libevent – an event notification library, . URL https://libevent.org/.

[20] Rui Figueira. Modern c++ lightweight binary rpc framework without code generation. Technical report, 2016. URL http://www.crazygaze.com/blog/2016/06/06/modern-c-lightweight-binary-rpc-framework-without-code-generation.

[21] W. Shane Grant and Randolph Voorhies. cereal - a c++11 library for serialization. Technical report, 2017. URL http://uscilab.github.io/cereal.

[22] libssh – the ssh library!, . URL https://www.libssh.org/.

[23] libssrcspread c/lua/perl/python/ruby bindings for spread, . URL https://www.savarese.com/software/libssrcspread/.

[24] I Antcheva, Maarten Ballintijn, Bertrand Bellenot, Marek Biskup, Rene Brun, Nenad Buncic, Ph Canal, Diego Casadei, Olivier Couet, Valeri Fine, Leandro Franco, Gerardo Ganis, Andrei Gheata, David Gonzalez Maline, Masaharu Goto, Jan Iwaszkiewicz, Anna Kreshuk, Diego Marcos Segura, Richard Maunder, and Matevz Tadel. Root - a c++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 180:2499–2512, 06 2011. doi: 10.1016/j.cpc.2009.08.005.

[25] Victor Alessandrini. *Shared Memory Application Programming: Concepts and Strategies in Multicore Application Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015. ISBN 012803761X, 9780128037614.

EXPLORAÇÃO DO SISTEMA PLACOR

O PlaCoR pode ser instalado e explorado nos sistemas Linux, Solaris e MacOS. Foi compilado e testado com os compiladores GNU GCC e Clang, com versões que suportam a versão padrão C++17. Pressupõe a instalação prévia das seguintes bibliotecas:

- `Spread` - versão 5 ou posterior
- `libssrcpread` - versão 1.0.15 ou posterior
- `libevent` - versão 2 ou posterior
- `cereal` - versão 1.2.2 ou posterior
- `libssh` - verão 0.8 ou posterior
- `boost` - versão 1.65 ou posterior

Atualmente, de forma a compilar os exemplos apresentados ao longo do capítulo 6, é necessário fazer a instalação prévia do pacote `ROOT`. Como o PlaCoR foi desenvolvido na versão padrão C++17, é necessário que o `ROOT` tenha sido compilado segundo a mesma versão padrão [1].

O PlaCoR foi concebido para dar suporte à execução de aplicações distribuídas em ambientes heterogéneos. Cada nó de computação terá de ter um *deamon* `Spread` a correr para a aplicação se ligar ao mesmo, permitindo a comunicação entre os diferentes processos da aplicação. É necessário criar previamente uma rede `Spread` [2] com todos os nós de computação a serem utilizados pela aplicação, na qual a comunicação ocorre sobre o protocolo TCP/IP.

A.1 INSTALAÇÃO DO AMBIENTE

O pacote PlaCoR encontra-se disponível em `https://github.com/brunoribeiro127/PlaCoR`. Este utiliza a ferramenta CMake para facilitar a configuração, compilação e instalação do

1 `https://root.cern.ch/building-root`
2 `http://www.spread.org/docs/guide/users_guide.pdf`

pacote em diferentes plataformas, sendo necessário uma versão 3.12 ou posterior. Caso as dependências da biblioteca não estejam instaladas nos caminhos padrão do sistema, é possível atribuir o caminho para cada biblioteca utilizando as seguintes variáveis:

- `Spread_ROOT`: caminho para a pasta do pacote `Spread`.
- `SSRCSpread_ROOT`: caminho para a pasta do pacote `libssrcspread`.
- `LibEvent_ROOT`: caminho para a pasta do pacote `libevent`.
- `Cereal_ROOT`: caminho para a pasta do pacote `cereal`.
- `LibSSH_ROOT`: caminho para a pasta do pacote `libssh`.
- `BOOST_ROOT`: caminho para a pasta do pacote `boost`.
- `ROOT_ROOT`: caminho para a pasta do pacote `ROOT`.

A.2 COMPILAÇÃO DE APLICAÇÕES

As aplicações do utilizador podem ser desenvolvidas sob vários módulos, que terão de ser compilados como bibliotecas dinâmicas. É necessário fornecer ao compilador o caminho para os cabeçalhos das bibliotecas `cor` (pacote PlaCoR), `cereal`, `libssrcspread` e `Spread`, bem como compilar os mesmos com a versão padrão C++17. É necessário também ligar os módulos com a biblioteca dinâmica do pacote PlaCoR.

A.3 CONFIGURAÇÃO DO AMBIENTE DE EXECUÇÃO

As aplicações PlaCoR são arrancadas através do programa `corun`. Este utiliza o programa `corx` para criar o *pod* local e executar a função de entrada do módulo do utilizador. Assim, é necessário colocar o caminho para ambos os programas na variável de ambiente `PATH`. É também necessário fornecer os caminhos para as bibliotecas dinâmicas dos pacotes utilizados por estes utilitários, nomeadamente os pacotes `cor`, `Spread`, `libssrcspread`, `libssh` e `boost`, através da variável `LD_LIBRARY_PATH`.

É importante notar que são utilizadas conexões SSH não interativas para lançar os módulos do utilizador nas máquinas correspondentes, locais ou remotas, pelo que é necessário garantir de antemão que tais conexões são possíveis de se fazer. Dependendo do ambiente de execução, é necessário definir devidamente as variáveis de ambiente para que estas forneçam os caminhos para os programas e bibliotecas anteriormente explicitados, de forma a garantir que tais caminhos estão disponíveis nas várias sessões SSH não interativas criadas. Nomeadamente, em ambiente Linux e MacOS, é recomendável definir tais variáveis

no ficheiro `.bashrc`, sendo este interpretado aquando da criação de uma conexão SSH não interativa.

A.4 EXECUÇÃO DE APLICAÇÕES

Para a execução de aplicações, é necessário criar uma rede `Spread`, especificando todos os nós de computação a utilizar pela aplicação. Cada máquina especificada tem de possuir um *deamon* `Spread` a correr na rede especificada anteriormente, sendo necessário verificar se é possível a comunicação entre as várias máquinas.

O arranque das aplicações é feito através do comando `corun`, ao qual é possível especificar um ficheiro de *hosts* com o seguinte formato:

```
<hostname ou ip>:<ssh port>:<spread port>
```

É então possível atribuir um conjunto de máquinas a uma aplicação, especificando o *hostname* ou IP, seguido da porta SSH no qual o serviço está a correr, e da porta do serviço `Spread`. Caso não seja especificado algum dos parâmetros, por padrão assume-se a máquina local `localhost`, o serviço SSH a correr na porta `22`, e o *deamon* `Spread` a correr na porta `4803`.

B

UNIDADES PLACOR

B.1 UNIDADE POOL

B.1.1 *Ficheiro pool.hpp*

```
#ifndef UNITS_POOL_HPP
#define UNITS_POOL_HPP

#include "cor/cor.hpp"

#include <mutex>
#include <condition_variable>

namespace cor
{
    class Pool
    {

    public:
        Pool(std::size_t num_agents);
        ~Pool();

        void Dispatch(void (*fct)(void*), void *arg);
        void WaitForIdle();

    protected:
        void PeerAgent();

    private:
        class Barrier
        {
            public:
                Barrier(std::size_t count);
                ~Barrier();
                void Wait();
                void WaitForIdle();
                void ReleaseThreads();

            private:
                std::mutex _mtx;
```

```
36                  std::condition_variable _qtask;
37                  std::condition_variable _qidle;
38                  std::size_t _nthreads;
39                  std::size_t _counter;
40                  bool _status;
41                  bool _is_idle;
42          };
43
44          void JoinAgents();
45
46          friend void AgentFunc(void *arg);
47
48          std::size_t _num_agents;
49          Barrier *_barrier;
50
51          std::mutex _mtx;
52          void (*_fct)(void*);
53          void *_arg;
54          bool _shut;
55
56          idp_t _group;
57          std::vector<idp_t> _agents;
58      };
59  }
60
61  #endif
```

B.1.2 *Ficheiro pool.cpp*

```
1   #include "pool.hpp"
2
3   namespace cor
4   {
5       void AgentFunc(void *arg)
6       {
7           Pool *p = (Pool*) arg;
8           p->PeerAgent();
9       }
10
11      Pool::Pool(std::size_t num_agents) :
12          _num_agents{num_agents},
13          _barrier{new Barrier(num_agents)},
14          _fct{nullptr},
15          _arg{nullptr},
16          _shut{false}
17      {
18          auto domain = cor::GetDomain();
19          auto agent_idp = domain->GetActiveResourceIdp();
20
21          auto group = domain->CreateLocal<cor::Group>(domain->Idp(), "", "");
22          _group = group->Idp();
23
24          for (std::size_t i = 0; i < num_agents; ++i) {
```

```
25              auto agent = domain->CreateLocal<cor::ProtoAgent<void(void*)>>(group->Idp(), "", AgentFunc);
26              agent->Run((void*)this);
27              _agents.push_back(agent->Idp());
28          }
29      }
30
31      Pool::~Pool()
32      {
33          if (!_shut)
34              JoinAgents();
35      }
36
37      void Pool::Dispatch(void (*fct)(void*), void *arg)
38      {
39          _barrier->WaitForIdle();
40
41          {
42              std::unique_lock<std::mutex> lk(_mtx);
43              _fct = fct;
44              _arg = arg;
45          }
46
47          _barrier->ReleaseThreads();
48      }
49
50      void Pool::WaitForIdle()
51      {
52          _barrier->WaitForIdle();
53      }
54
55      void Pool::PeerAgent()
56      {
57          bool my_shut;
58
59          for(;;)
60          {
61              _barrier->Wait();
62
63              {
64                  std::unique_lock<std::mutex> lk(_mtx);
65                  my_shut = _shut;
66              }
67
68              if (!my_shut)
69                  (*(_fct))(_arg);
70              else
71                  break;
72          }
73      }
74
75      void Pool::JoinAgents()
76      {
77          {
78              std::unique_lock<std::mutex> lk(_mtx);
79              _shut = true;
```

```
80          }
81
82          _barrier->WaitForIdle();
83          _barrier->ReleaseThreads();
84
85          // join agents
86          auto domain = cor::GetDomain();
87
88          for (auto idp: _agents) {
89              auto agent = domain->GetLocalResource<cor::ProtoAgent<void(void*)>>(idp);
90              agent->Wait();
91              agent->Get();
92          }
93      }
94
95      Pool::Barrier::Barrier(std::size_t count) :
96          _nthreads{count},
97          _counter{0},
98          _status{true},
99          _is_idle{false}
100     {}
101
102     Pool::Barrier::~Barrier() = default;
103
104     void Pool::Barrier::Wait()
105     {
106         bool lstatus;
107
108         {
109             std::unique_lock<std::mutex> lk(_mtx);
110
111             lstatus = _status;
112             _counter++;
113
114             if (_counter == _nthreads) {
115                 _counter = 0;
116                 _is_idle = true;
117                 _qidle.notify_all();
118             }
119
120             while (lstatus == _status)
121                 _qtask.wait(lk);
122         }
123     }
124
125     void Pool::Barrier::WaitForIdle()
126     {
127         std::unique_lock<std::mutex> lk(_mtx);
128         while (!_is_idle)
129             _qidle.wait(lk);
130     }
131
132     void Pool::Barrier::ReleaseThreads()
133     {
134         std::unique_lock<std::mutex> lk(_mtx);
```

```
135         _is_idle = false;
136         _status = !_status;
137         _qtask.notify_all();
138     }
139 }
```

B.2 UNIDADE UTILS

```
1   #ifndef UNITS_UTILS_HPP
2   #define UNITS_UTILS_HPP
3
4   #include "cor/cor.hpp"
5
6   template <typename T>
7   idm_t GetRank(idp_t rsc_idp)
8   {
9       auto domain = cor::GetDomain();
10      auto organizer_idp = domain->GetPredecessorIdp(rsc_idp);
11      auto organizer = domain->GetLocalResource<T>(organizer_idp);
12      return organizer->GetIdm(rsc_idp);
13  }
14
15  template <typename T>
16  idm_t GetRank()
17  {
18      auto domain = cor::GetDomain();
19      auto agent_idp = domain->GetActiveResourceIdp();
20      return GetRank<T>(agent_idp);
21  }
22
23  template <typename T>
24  std::size_t GetSize(idp_t organizer_idp)
25  {
26      auto domain = cor::GetDomain();
27      auto organizer = domain->GetLocalResource<T>(organizer_idp);
28      return organizer->GetTotalMembers();
29  }
30
31  template <typename T>
32  std::size_t GetSize()
33  {
34      auto domain = cor::GetDomain();
35      auto agent_idp = domain->GetActiveResourceIdp();
36      auto organizer_idp = domain->GetPredecessorIdp(agent_idp);
37      return GetSize<T>(organizer_idp);
38  }
39
40  template <typename T>
41  void AgentRange(std::size_t& begin, std::size_t& end)
42  {
43      std::size_t l_begin, l_end;
44      std::size_t size, d, r;
45
```

```
46      auto rank = GetRank<T>() + 1;
47      auto num_agents = GetSize<T>();
48
49      size = end - begin;
50      d = size / num_agents;
51      r = size % num_agents;
52
53      l_end = begin;
54      for (std::size_t i = 1; i <= rank; ++i) {
55          l_begin = l_end;
56          l_end = l_begin + d;
57          if (r) {
58              l_end++;
59              r--;
60          }
61      }
62
63      begin = l_begin;
64      end = l_end;
65  }
66
67  #endif
```

C

CÓDIGO CLASSE AUXILIAR READCHAIN

```
#ifndef READ_CHAIN_HPP
#define READ_CHAIN_HPP

#include "TROOT.h"
#include "TChain.h"
#include "TString.h"
#include "TError.h"

#include "units/utils.hpp"

class ReadChain
{

public:
    ReadChain(std::string const& name, std::string const& path) :
        _chain{TChain(name.c_str())},
        _path{path},
        _begFile{0},
        _endFile{0},
        _beg{0},
        _end{0},
        _intVar{0},
        _floatVar{0},
        _intSum{0},
        _floatSum{0}
    {}

    auto GetIntSum() const { return _intSum; }
    auto GetFloatSum() const { return _floatSum; }

    void SetFileRange(int in, int fin)
    {
        _begFile = in;
        _endFile = fin;
    }

    void Initialize()
    {
        for (auto i = _begFile; i < _endFile; ++i) {
            auto file = _path + "/file%i.root";
            _chain.Add(TString::Format(file.c_str(), i));
```

```
42          }
43          auto entries = _chain.GetEntries();
44          _chain.SetBranchAddress("IntVar", &_intVar);
45          _chain.SetBranchAddress("FloatVar", &_floatVar);
46          _end = entries;
47          AgentRange<cor::Group>(_beg, _end);
48      }
49
50      void ProcessChain()
51      {
52          for (Long64_t entry = _beg; entry < _end; ++entry) {
53              if (_chain.GetEntry(entry) <= 0) {
54                  Error("readFiles", "Failed to read entry %lld", entry);
55                  return;
56              }
57              _intSum += _intVar;
58              _floatSum += _floatVar;
59          }
60      }
61
62  private:
63      TChain _chain;
64      std::string _path;
65      std::size_t _begFile, _endFile;
66      std::size_t _beg, _end;
67      Int_t _intVar;
68      Float_t _floatVar;
69      Int_t _intSum;
70      Float_t _floatSum;
71
72  };
73
74  #endif
```