Desenvolvimento de Controlador de Experimentos OMF em Céu

Carlos Mattoso

Projeto Final de Graduação

Centro Técnico Científico
Departamento de Informática
Curso de Bachalerado em Engenharia de Computação

Orientadora: Prof.ª Noemi Rodriguez

Rio de Janeiro
Dezembro de 2016

Carlos Mattoso

Desenvolvimento de Controlador de Experimentos OMF em Céu

Relatório de Projeto Final, apresentado ao programa Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do Bacharel em Engenharia de Computação.

Prof.ª Noemi Rodriguez
Orientadora
Departamento de Informática — PUC–Rio

Rio de Janeiro, 22 de Dezembro de 2016

Agradecimentos

À minha orientadora Noemi Rodriguez e praticamente co-orientador Francisco Sant'Anna, pela oportunidade, apoio e compreensão em um ano que foi muito corrido e conturbado.

À PUC–Rio, pela sólida formação acadêmica e pelas portas que abriu internacionalmente.

Aos meus pais, sem os quais eu não teria atingido tantas conquistas.

Finalmente, aos meus colegas da PUC–Rio, da UC-Irvine e da Amazon, que fizeram dos últimos anos uma experiência inesquecível.

Resumo

Mattoso, Carlos; Rodriguez, Noemi. **Desenvolvimento de Controlador de Experimentos OMF em Céu**. Rio de Janeiro, 2016. 50p. Relatório de Projeto Final — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Neste trabalho foi implementado um controlador de experimentos OMF na linguagem de programação Céu que possibilita o uso dos construtos nativos da linguagem para tratamento de eventos na descrição de experimentos destinados a ambientes de experimentação em rede. Outro artefato deste trabalho é um *binding* em Céu para uma biblioteca de AMQP escrita na linguagem de programação C, que concilie as propriedades e métodos desta biblioteca ao paradigma de programação reativa e estruturada de Céu.

Palavras–chave

Ambientes de experimentação em rede. Programação orientada a eventos. Passagem de mensagens por filas.

Abstract

Mattoso, Carlos; Rodriguez, Noemi. **Development of OMF Experiment Controller in Céu**. Rio de Janeiro, 2016. 50p. Capstone Project Report — Department of Informatics, Pontifical Catholic University of Rio de Janeiro.

In this project an OMF experiment controller was implemented in the Céu programming language to enable the use of the languages's native event handling constructs for the description of experiments targeting networking testbeds. Another product of this work is a Céu binding to an AMQP library written in C that combines the properties and methods of such library with the structured reactive programming paradigm of Céu.

Keywords

Networking Testbeds. Event-oriented Programming. Message Queuing.

Sumário

Lista de figuras

1
Introdução

A emergência de ambientes de experimentação, conhecidos como *testbeds*, para o amparo ao desenvolvimento de novos protocolos e técnicas em redes, cria a necessidade por protocolos e ferramentas que possibilitem a descrição dos recursos, da configuração e dos processos que definem completamente um experimento reproduzível. Neste contexto, desenvolveu-se o *OMF* (1), um arcabouço para gerência de *testbeds* e controle de experimentos.

O objetivo do OMF é assegurar o nível apropriado de abstração, tanto do ponto de vista do operador de *testbeds* quanto do pesquisador (2). O operador tem acesso a um conjunto de serviços que facilitam a gerência do *testbed*. Por outro lado, o arcabouço oferece suporte a *scripts*, escritos pelo pesquisador em linguagem de alto nível, que descrevem o experimento a ser realizado, automatizando sua execução e facilitando sua reprodução.

O LabLua desenvolveu, nos últimos anos, pesquisas sobre redes de sensores sem fio (RSSF) e participou do desenvolvimento do *testbed* CeuNa-Terra (3), implementado com o apoio da Rede Nacional de Ensino e Pesquisa. Também desenvolveu um conjunto de tecnologias que podem ser empregadas em aplicações destinadas a RSSF, como a linguagem de programação estruturada síncrona e reativa Céu (4) e o sistema de programação de RSSF Terra (5).

Neste contexto, surgiu a ideia de se expandir os casos de uso da linguagem Céu através do desenvolvimento nesta de uma implementação alternativa do arcabouço OMF. Isto se deve a experimentos OMF serem descritos em torno de eventos externos e internos aos agentes sob teste e a existência de construtos nativos na linguagem Céu para definição e tratamento de eventos como entidades de primeira classe.

Como parte deste projeto, implementou-se também uma biblioteca de *RabbitMQ* para suporte ao desenvolvimento de um arcabouço OMF. *RabbitMQ* é uma implementação do *Advanced Message Queueing Protocol* (A), um protocolo de troca de mensagens por meio de filas. Por si só, esta já é uma contribuição ao ecossistema de Céu que se distingui de bibliotecas em outras linguagens por permitir a realização de seus métodos de forma reativa, paralela

e não bloqueante.

Por fim, também como base ao desenvolvimento do arcabouço OMF, produziu-se código que implementa a especificação do *Federated Resource Control Protocol (FRCP)* (9). Este protocolo rege a comunicação entre dispositivos independentes e é empregado pelo OMF na orquestração de um experimento. Baseia-se na biblioteca de *RabbitMQ* para a comunicação entre recursos, servindo assim de exemplo de uso da biblioteca Céu de AMQP.

Estes três artefatos exploram as propriedades da linguagem Céu. São pensados segundo paradigmas de reatividade, visto que baseiam-se na emissão e reação a eventos, e de programação estruturada, de modo que o controle do ciclo de vida de entidades é regido por seus escopos.

2
Situação Atual

2.1
OMF

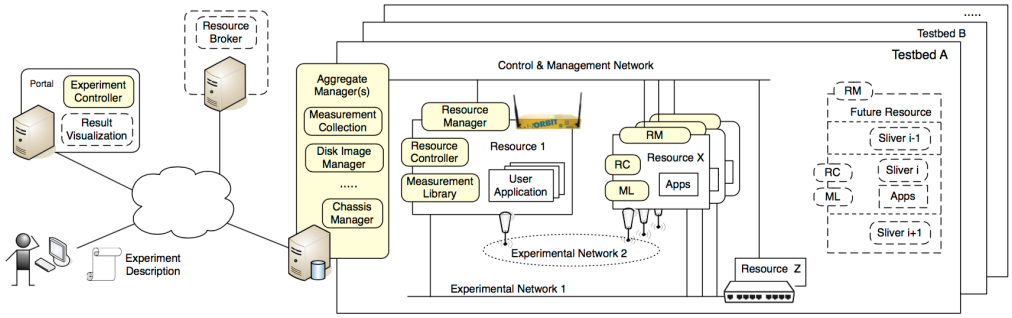

Figura 2.1: Arquitetura OMF (1)

O arcabouço OMF pode ser discretizado em três planos: controle, medição e gerenciamento (1). O primeiro engloba as ferramentas e diretivas que possibilitam ao pesquisador definir e executar um experimento; aqui destaca-se o *controlador de experimentos (CE)*. O segundo é constituído das tecnologias que possibilitam ao pesquisador coletar métricas dos recursos sob teste para avaliação posterior; a coleta de métricas é definida através da *OMF Measurement Library (OML)*, não abordada neste trabalho. Finalmente, a camada de gerenciamento diz respeito à infraestrutura e *testbed* nos quais o experimento é executado, destacando-se neste plano os recursos e seus controladores.

Os recursos na camada de gerenciamento são os alvos do experimento. Para possibilitar o desenvolvimento de tais recursos como módulos coesos e independentes, o arcabouço OMF define entidades denominadas *controladores de recursos (CRs)*, que são responsáveis por intermediar a comunicação entre o CE e os recursos sob seu controle. A distribuição oficial do OMF disponibiliza uma biblioteca em *Ruby*, chamada de *omf_rc* (8), que permite a descrição e uso de controladores de recursos. Contudo, controladores de recursos podem ser implementados sem depender desta biblioteca, desde que lidem corretamente com os protocolos de comunicação.

O CE na camada de controle é o aplicativo que executa o experimento, sendo sua implementação oficial chamada de *omf_ec* (7). Para executar um experimento, o pesquisador deve fornecer como entrada para o CE um arquivo denominado *descrição de experimento (DE)* que especifique completamente os recursos sob teste, sua configuração inicial e as ações a serem executadas sobre os mesmos durante o experimento. Com base neste arquivo, o CE envia pedidos aos CRs dos recursos sob teste e coleta as respostas enviadas por estes, exibindo-as ao usuário caso este especifique tal comportamento. Na distribuição oficial do OMF, tais arquivos devem ser elaborados na linguagem de programação *OMF Experiment Description Language (OEDL)* (6). Esta linguagem é, na verdade, uma extensão de *Ruby* que provê, além de toda funcionalidade suportada por Ruby, uma série de comandos específicos ao domínio de definição e orquestração de experimentos. Utilizando esta linguagem um pesquisador poderia, por exemplo, dar partida em aplicações de sensoriamento em uma rede de sensores distribuídos e, com o passar do tempo, variar parâmetros das aplicações para avaliar diferentes estratégias de disseminação de mensagens e assim determinar qual produz melhores resultados.

A comunicação entre o CE e os CRs não se dá de forma direta. O arcabouço OMF exige o emprego de um servidor que suporte o paradigma de troca de mensagens *pub/sub*. Utilizando-se este servidor, ambas as partes trocam mensagens através de tópicos definidos ao longo da execução do experimento, segundo o protocolo FRCP (9). Este protocolo especifica o formato do *payload* das mensagens intercambiadas e uma série de regras sobre como o intercâmbio destas deve ocorrer, segundo seu tipo e parâmetros. A implementação oficial de OMF suporta dois tipos de protocolos de troca de mensagens: *Extensible Messaging and Presence Protocol (XMPP)* e *Advanced Message Queuing Protocol (AMQP)*, sendo este último mais recomendado.

O ciclo de vida de um experimento inicia-se com a elaboração de uma descrição de experimento que é então passada como entrada a um controlador de experimentos. Este, por sua vez, inicia uma sequência de troca de mensagens com os controladores de recursos especificados na DE. Esta troca de mensagens ocorre através de um *broker* AMQP, sendo a definição dos tópicos, empacotamento das mensagens e roteamento destas realizados segundo as regras estipuladas pelo FRCP. Tais processos são executados por trás dos panos pelo CE, sem a necessidade de conhecimento ou intervenção do pesquisador sobre os mesmos. Ao fim do experimento, o estado no servidor de mensagens é limpo e possíveis métricas são armazenadas automaticamente.

Embora OMF seja um arcabouço robusto, uma limitação inerente a *Ruby* é a ausência de eventos como entidades de primeira classe. Isto acarreta

em definições de eventos e diretivas para seu tratamento não tão naturais, constituindo uma barreira para a introdução de pesquisadores a este ambiente. Para facilitar o uso de *testbeds* OMF oferecemos a possibilidade do pesquisador escrever a descrição de seu experimento em Céu, uma linguagem desenvolvida para lidar com sistemas orientados a eventos.

2.2 OEDL

OEDL oferece uma série de comandos em Ruby para facilitar a descrição de um experimento. Toda DE escrita em OEDL é composta basicamente de três fases: definição de aplicações, definição de grupos de recursos e definição do experimento.

A definição de aplicações descreve de forma exaustiva uma representação lógica das aplicações que serão executadas sobre os recursos em teste. Na declaração de uma aplicação devem ser descritos seus atributos, possíveis métricas e os executáveis que as implementam.

```
1  defApplication('ping_oml2') do |app|
2    app.description = 'Simple Definition for the ping-oml2
       application'
3    app.binary_path = '/usr/bin/ping-oml2'
4
5    app.defProperty('target', 'Address to ping', '-a', {:type => :
       string})
6    app.defProperty('count', 'Number of times to ping', '-c', {:type
       => :integer})
7
8    app.defMeasurement('ping') do |m|
9      m.defMetric('dest_addr',:string)
10     m.defMetric('ttl',:uint32)
11     m.defMetric('rtt',:double)
12     m.defMetric('rtt_unit',:string)
13   end
14 end
```

Exemplo de código 2.1: Exemplo de definição de aplicação

A definição de grupos estabelece uma organização de nós sob teste nos quais serão executadas as aplicações previamente definidas. Neste caso, devem ser declaradas as aplicações que serão instanciadas nos recursos que pertencem ao grupo, sendo também já definidos os parâmetros de inicialização de cada aplicação. Além disso, o usuário pode configurar atributos de rede dos recursos pertencentes ao grupo, o que não é exibido neste exemplo.

```
1  defGroup('Sender', 'omf.nicta.node8') do |g|
```

```
  g.addApplication("ping_oml2") do |app|
    app.setProperty('target', 'www.nicta.com.au')
    app.setProperty('count', 3)

    app.measure('ping', :samples => 1)
  end
end
```

Exemplo de código 2.2: Exemplo de definição de grupo

Na fase do experimento são descritos pontos de partida e parada das aplicações que pertencem aos grupos de recursos, segundo o disparo de eventos. Assim, quando da ocorrência de evento específico, uma série de ações do experimento é executada. Um evento é uma ocorrência física decorrente da verificação de condição particular, sendo tratado por uma tarefa específica quando for disparado. Eventos podem ser tanto nativos do arcabouço quanto definidos pelos usuários. No caso de nativos temos, por exemplo, um evento que indica a conclusão do processo de inicialização de um recurso. Eventos definidos pelo usuário atuam sobre o estado corrente do experimento, podendo, por exemplo, ser feito um evento que dispara quando determinada aplicação é encerrada. Visto que o experimento é dirigido por eventos, é natural explorarmos o uso de uma linguagem em que estes sejam primitivas, o que se verifica em Céu.

```
onEvent(:ALL_UP_AND_INSTALLED) do |event|
  # Exibe no terminal esta mensagem
  info "This is my first OMF experiment"

  # Dispara o inicio de todas as aplicacoes
  allGroups.startApplications

  # Espera 5 segundos
  after 5 do
    # Interrompe todas as aplicacoes
    allGroups.stopApplications
    # Encerra o experimento
    Experiment.done
  end
end
```

Exemplo de código 2.3: Exemplo de definição de experimento

Vejamos agora um exemplo de experimento que demonstra algumas limitações de sua descrição ser feita em OEDL, que procuraremos superar em Céu. No exemplo que segue são inicializadas duas aplicações separadamente, três e seis segundos após o início do experimento. Ao se passarem nove segundos, o experimento é explicitamente encerrado pelo pesquisador.

Esta primeira abordagem tem alguns problemas, cujas alternativas que apresentamos posteriormente trazem outras deficiências. Primeiramente, observe que o pesquisador tem que manter, ele próprio, um conceito global de tempo para definir o instante em que as aplicações devem ser iniciadas e o experimento, encerrado. O encerramento explícito do experimento é outra deficiência dessa abordagem, visto que é algo que não deveria ser imposto ao pesquisador.

```
onEvent(:ALL_UP) do |event|
  # 'after' is not blocking. This executes 3 seconds after :ALL_UP
     fired.
  after 3 do
    info "TEST - do ping app"
    group("Actor").startApplication("ping_google")
  end

  # 'after' is not blocking. This executes 6 seconds after :ALL_UP
     fired.
  after 6 do
    info "TEST - do date app"
    group("Actor").startApplication("date_LA")
  end

  # 'after' is not blocking. This executes 9 seconds after :ALL_UP
     fired.
  after 9 do
    Experiment.done
  end
end
```

Exemplo de código 2.4: Experimento com duas aplicações

Uma implementação alternativa, vista abaixo, que supera a necessidade de se manter um conceito global de tempo, aproveita as construções do comando *after* serem meras *callbacks*, definidas entre os blocos *do ... end*. Deste modo, podem-se fazer *callbacks* aninhadas, que executam uma após a outra assim que seus respectivos eventos dispararem. Contudo, isto pode resultar, particularmente em casos mais complexos, no fenômeno denominado *callback hell*, que consiste em um alto grau de aninhamento de código devido ao encadeamento de *callbacks* exibido abaixo.

```
onEvent(:ALL_UP) do |event|
    after 3 do
        info "TEST - do ping app"
        group("Actor").startApplication("ping_google")

        after 3 do
```

```
7               info "TEST - do date app"
8               group("Actor").startApplication("date_LA")
9
10              after 3 do
11                  Experiment.done
12              end
13          end
14      end
15  end
```

Exemplo de código 2.5: Callback hell

Esta última abordagem, que emprega o comando de espera *wait*, possibilita apenas um nível de aninhamento e a eliminação de qualquer *callback*. Contudo, seu uso é problemático: o comando *wait* bloqueia o laço de eventos implementado pelo CE Ruby, que checa a ocorrência de eventos e dispara correspondentes reações. Em testes, isso inclusive levou a problemas na execução do próprio experimento, visto que resultados produzidos e enviados ao CE pelo recurso sob testes eram perdidos.

```
1  onEvent(:ALL_UP) do |event|
2    wait 3
3    info "TEST - do ping app"
4    group("Actor").startApplication("ping_google")
5
6    wait 3
7    info "TEST - do date app"
8    group("Actor").startApplication("date_LA")
9
10   wait 3
11   Experiment.done
12 end
```

Exemplo de código 2.6: Espera bloqueante do laço de eventos

2.3 FRCP

Uma biblioteca de FRCP foi desenvolvida para suporte da implementação do arcabouço OMF. Este protocolo visa o controle e orquestração de recursos distribuídos por uma aplicação central. Basicamente, o protocolo consiste do envio de uma mensagem de um solicitante para um recurso, que pode então aceitá-la e executar as ações a ela associadas. Isto pode acarretar o envio de mensagens subsequentes a uma entidade denominada observador, que no contexto de OMF é o próprio solicitante. A definição dos pedidos e ações

a serem executadas são determinadas através de cinco tipos de mensagem: *inform*, *configure*, *request*, *create* e *release*.

```
{
  // Cabeçalho
  "op" : /* Tipo da mensagem */,
  "mid": /* ID da mensagem   */,
  "src": /* ID do recurso    */,
  "ts" : /* Unix timestamp   */,
  [ "rp": /* Tópico que recebe cópia da mensagem */ ]
  // Corpo
  "props": {
      // ...
  }
}
```

Exemplo de código 2.7: Formato genérico de mensagem

A mensagem de tipo *create* é a que costuma ser o ponto de partida em uma sequência de comunicação, e é através desta que se cria um novo recurso. Note que um novo recurso pode ter outro como pai, e assim é estabelecida uma hierarquia. A fim de obter informações sobre o estado de um recurso, deve-se enviar a este uma mensagem do tipo *request* em que se especificam os atributos de interesse, os quais serão fornecidos em uma correspondente mensagem *inform*. Mensagens do tipo *inform* podem ser também ser enviadas espontaneamente. Elas carregam informações sobre o estado atual do recurso ou sobre o resultado de operação prévia solicitada ao mesmo.Por outro lado, caso se deseje alterar o estado de tal recurso, utilizam-se mensagens do tipo *configure*. Por fim, para finalização de um recurso deve-se enviar uma mensagem do tipo *release*; no caso de uma hierarquia de recursos, uma cadeia de encerramento deve ser executada, iniciada pelas folhas da hierarquia até o recurso que recebeu a mensagem. Confira a seguir um exemplo de fluxo de troca de mensagens regido por este protocolo, entre um solicitante e dois recuros.

Este protocolo exige o uso de algum sistema genérico de comunicação subjacente. Para os fins deste projeto, deu-se suporte apenas ao AMQP em sua implementação pelo RabbitMQ. Em razão da genericidade do protocolo, não se faz uso de propriedades específicas de AMQP. Por exemplo, no caso de uma hierarquia de recursos uma mesma mensagem pode ser recebida por várias entidades, quando apenas um nó é seu destinatário. É de responsabilidade de todo recurso checar um atributo do *payload* da mensagem a fim de determinar se é o destinarário da mesma.

Ainda, um conceito abstrato fundamental deste protocolo são tópicos, que são as entidades que de fato recebem as mensagens destinadas a um

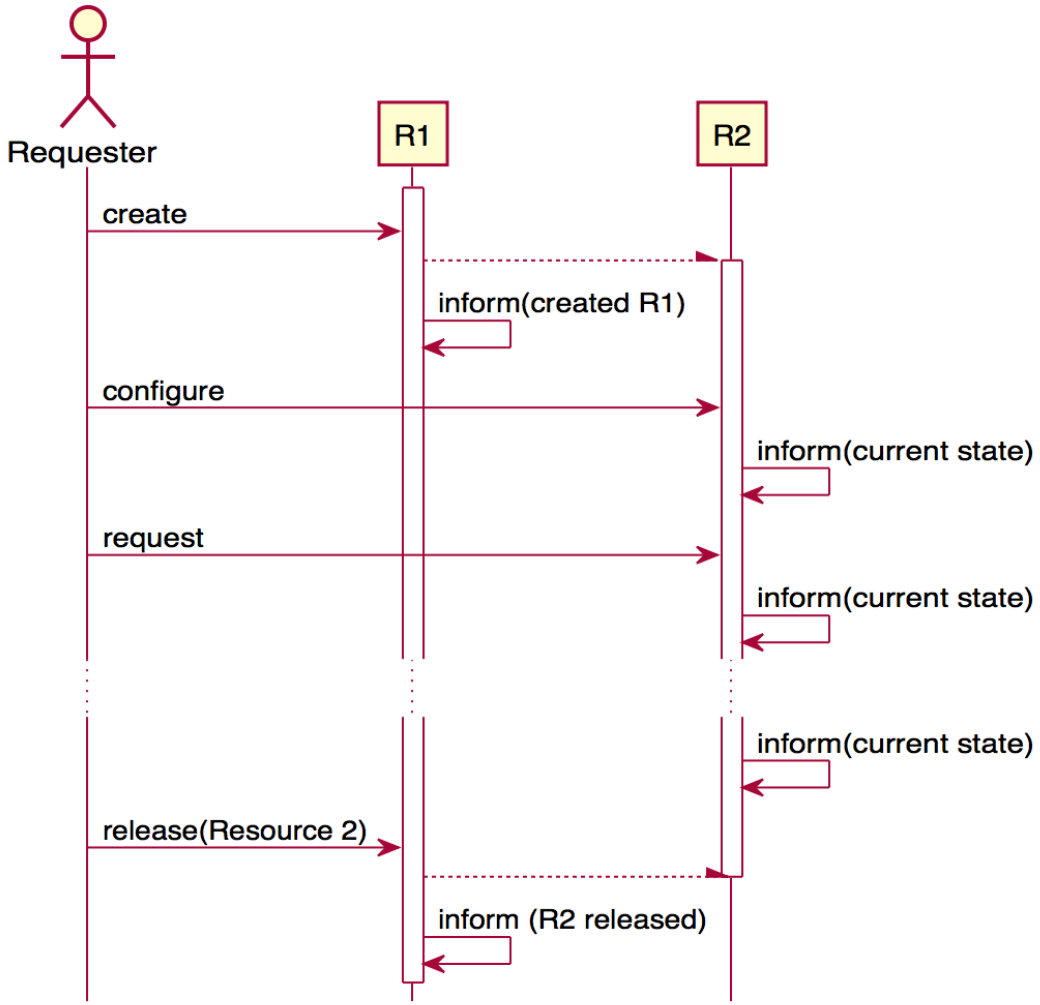

Figura 2.2: Exemplo de fluxo de troca de mensagens regido por FRCP (9)

recurso. Todo recurso deve estar inscrito a pelo menos um tópico, o seu próprio, e no caso de formação de uma hierarquia ou grupo, segundo a aplicação de controle, podem também precisar se inscrever nos tópicos de tais recursos. O controle de inscrições deve ser feito pela aplicação central, que no caso de OMF é o CE. Para isto, ela envia mensagens de tipo *configure* aos recursos alvos especificando em qual novo tópico os recursos-alvo devem se inscrever.

Por fim, no contexto de OMF o prosseguimento de um experimento inicia-se por uma mensagem do tipo *create*, que invoca a criação de recursos. Posteriormente, em geral, uma série de mensagens de tipos *request* ou *configure* são enviadas de modo a testar os recursos, segundo o DE provido ao CE. Os recursos, por sua vez, respondem aos pedidos com *informs*. Ao final do experimento são enviadas mensagens *release* para limpar o ambiente.

2.4

AMQP

O AMQP é um padrão internacional, ISO 19464, que descreve um conjunto de entidades e regras para a troca de mensagens em rede. Seu principal objetivo é promover a interoperabilidade entre aplicações operadas sob a autoridade de diferentes organizações e segundo distintas regras de negócios, ao impor uma separação entre produtores e consumidores de mensagens. O arcabouço OMF faz uso deste protocolo, devendo-se utilizar um servidor que o implemente para a troca de mensagens entre o CE e CRs.

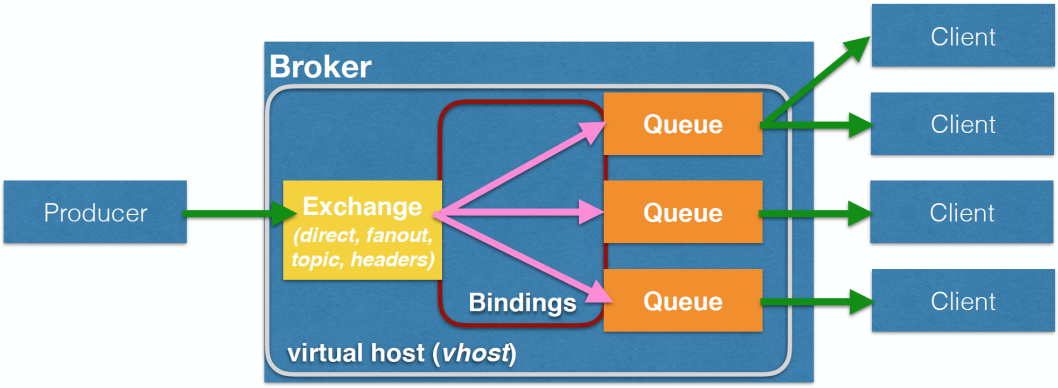

Figura 2.3: Arquitetura AMQP

A imagem acima apresenta uma visão abstrata da arquitetura definida pelo AMQP, destacando as entidades centrais definidas pelo mesmo. Toda implementação de AMQP, das quais RabbitMQ é um exemplo, é composta de um *broker*, *vhosts*, *exchanges*, filas e *bindings*; estas três últimas entidades constituem as primitivas do protocolo. Além disso, as aplicações podem assumir os papéis de consumidoras, produtoras ou, até mesmo, ambos.

O *broker* é um servidor que implementa o protocolo e possibilita a definição de todas as demais entidades, e o envio e recebimento de mensagens por aplicações. Todo *broker* deve permitir a criação de *vhosts*, que são ambientes virtuais que possibilitam a definição das primitivas AMQP e a execução de diferentes grupos de funcionalidade comuns em um mesmo *broker* sem riscos de colisões. Este recurso permite o compartilhamento de um mesmo servidor entre usuários diferentes, associando a cada um ambiente virtual exclusivo; uma funcionalidade especialmente relevante para serviços em nuvem. Todo *broker* pode estender o protocolo. RabbitMQ, por exemplo, provê uma extensão para confirmação do recebimento de mensagens a seus produtores, não apenas ao servidor.

O ponto de entrada de mensagens no servidor são os *exchanges*, encarregados de distribuir uma cópia de toda mensagem por eles recebida para cada fila a qual estão associados. As filas enviam as mensagens nelas arma-

zenadas as suas respectivas aplicações consumidoras segundo uma estratégia *Round-Robin*. Para início do recebimento de mensagens as aplicações devem se inscrever em uma fila e sinalizar ao *broker* sua intenção de consumir mensagens. Caso *acknowledgments* sejam habilitados, as mensagens transmitidas a consumidores são removidas de sua fila origem apenas quando seu tratamento for bem sucedido. Por fim, *bindings* são associações entre filas e *exchanges* definidas por uma chave. O roteamento de mensagens pelos *exchanges* para as filas as quais estão associados depende do tipo do *exchange*, da chave da associação e de uma chave incluída no cabeçalho das mensagens.

Embora este protocolo seja genérico, a única implementação utilizada neste projeto foi o RabbitMQ. Este *broker* oferece alto desempenho e tolerância a falhas, visando a troca confiável de mensagens entre aplicações. Há uma série de bibliotecas que suportam esta implementação, às quais se procurou contrastar a desenvolvida em Céu.

2.5 A Linguagem de Programação Céu

Para entendimento e apreciação do trabalho desenvolvido, apresenta-se uma breve introdução à linguagem de programação Céu. A linguagem de programação Céu oferece construtos nativos para o desenvolvimento de programas de natureza concorrente, mas segura e determinística ao mesmo tempo em que apresente efeitos colaterais.

Uma primitiva da linguagem são as trilhas de execução, sequências de linhas de código definidas através de composições de paralelismo que seguem um regime de tempo global. Trilhas ativas podem esperar por eventos, outra primitiva importante. Quando da ocorrência de um determinado evento, todas as trilhas que o esperam são notificadas. Note que tais primitivas são empregadas em Céu sob um modelo de execução síncrona, isto é, segundo a premissa de que uma cadeia de reação a um evento executa infinitamente mais rápida do que a taxa de ocorrência de eventos externos. Assim, Céu impossibilita a escrita de código bloqueante, visto que este quebraria tal premissa. Além disso, as composições de paralelismo que definem as trilhas representam um paralelismo lógico: na prática a execução das trilhas é feita de maneira intercalada conforme elas cedem controle da execução ao ambiente.

Um resumo do modelo de execução de programas em Céu é o seguinte:

1. O programa inicia a reação de *boot* na primeira linha de código de uma única trilha.

2. As trilhas ativas, em sequência de sua definição, executam até elas

esperarem ou encerrarem. Este passo é denominado uma *cadeia de reação*, e sempre executa em tempo delimitado.

3. O programa fica ocioso e o ambiente obtém controle da execução.

4. Caso ocorra novo evento, o ambiente acorda todas as trilhas que o esperam. Então, retorna-se ao passo 2.

Note que, como dito anteriormente, as trilhas são declaradas através de composições de paralelismo. Estas composições são muito importantes em Céu, e essenciais para que se leve a cabo o paradigma de programação estruturada. Embora não seja possível apresentar neste material um tutorial exaustivo de Céu, observe um exemplo que cobre alguns dos conceitos mencionados.

Neste primeiro exemplo são demonstrados vários conceitos importantes de Céu. Primeiramente, temos duas trilhas de execução, definidas pela diretiva *par/or*. Esta composição de paralelismo indica que ao término de qualquer uma das trilhas, a outra também será encerrada, sendo ambas unidas à trilha mais externa na qual elas foram declaradas. A primeira trilha da composição de paralelismo imprime na tela *Hello World!* a cada *1s*. A segunda trilha simplesmente espera pelo pressionamento de alguma tecla, mapeada pelo evento externo de entrada *KEY*. No caso, então, quando do pressionamento da tecla ambas as trilhas morrerão e o programa terminará.

```
1 input int KEY;
2
3 par/or do
4     every 1s do
5         _printf("Hello World!\n");
6     end
7 with
8     await KEY;
9 end
```

Exemplo de código 2.8: Exemplo introdutório de Céu

Note como esta composição propicia de maneira fácil a programação reativa e estruturada: em uma linguagem tradicional como C, teríamos feito um laço de exibição da mensagem que verificaria a cada iteração se a tecla fora pressionada. Isto sem levantar o questionamento de como fazer tal programa de forma não bloqueante e ter acesso a tempo, o que Céu oferece nativamente. Em Céu, por outro lado, definimos dois escopos atrelados que, quando do término do segundo resultante da ocorrência de um evento, decorre o término do primeiro.

Céu oferece integração nativa com os ambientes de C e Lua, o que possibilita o aproveitamento das funcionalidades e ecossistema mais maturos

de ambas, como exemplificado pelo uso da função *printf*. Devido ao paradigma síncrono seguido por Céu é importante que funções de C ou Lua utilizadas em Céu sejam também de caráter não-bloqueante, ou que sejam executadas em uma *thread* caso isto não se verifique. O compilador Céu não tem capacidade de verificar a natureza de funções externas, ficando a cargo do programador lidar com esta situação.

Brevemente, apresenta-se outro conceito de Céu bastante utilizado neste projeto: abstrações de código (*code/await*)e *pools*. Abstrações de código consistem de subprogramas que podem ser invocados em qualquer ponto do programa. *Pools* são estruturas de armazenamento de instâncias de algum tipo primitivo ou de invocações de abstrações de código. O poder de invocar uma abstração de código em uma *pool* decorre da continuidade de sua execução até seu término ou até o encerramento do escopo no qual a *pool* foi declarada. Isto possibilita a instânciação e despacho de múltiplos tratadores do resultado de um evento, o que propicia casos de uso como tratamento de mensagens recebidas.

Neste outro exemplo, observa-se a declaração de estrutura de dados chamada *Msg*, de uma abstração de código, de uma *pool* e de uma variável; o recebimento de uma sequência de eventos de recebimento de mensagens; e, por fim, a instanciação dinâmica de tratadores de mensagens em uma *pool*. Observe que conforme novas mensagens forem recebidas, um novo tratador é despachado para tratá-las. Este código segue uma expressão idiomática própria de Céu, visto que não estamos criando objetos nem especificando uma *callback* para o tratamento das mensagens recebidas.

```
1  input Msg RECV;
2
3  data Msg with
4      // atributos...
5  end
6
7  code/await Msg_Handler (var Msg msg) -> void do
8      // trabalho nao bloqueante ou com threads...
9  end
10
11 pool[] Msg_Handler handlers;
12 var Msg msg;
13
14 every msg in RECV do
15   spawn Msg_Handler(msg) in handlers;
16 end
```

Exemplo de código 2.9: Abstrações de Código em Céu

Os artefatos desenvolvidos procuram expor os métodos e conceitos de OMF, FRCP e RabbitMQ, adaptados ao molde imposto por Céu. A integração com C e Lua propiciou o uso de bibliotecas já existentes e estruturas de dados mais avançadas do que as existentes em Céu, ao mesmo tempo que introduziu desafios quanto ao intercâmbio da semântica destas em contraste a Céu.

3
Objetivos

A complexidade deste trabalho faz juz ao desenvolvimento de dois artefatos de igual importância e contribuição ao ecossistema Céu. Primeiramente, o desenvolvimento de uma biblioteca de AMQP em Céu (C), que mescle os paradigmas desta linguagem aos conceitos e métodos do protocolo. Em segundo lugar, a implementação de um CE alternativo em Céu (D) capaz de processar DEs também escritas em Céu. A escrita de DEs em Céu torna possível a definição e tratamento de eventos como entidades de primeira classe, simplificando a estrutura desses arquivos. Além disso, introduz um novo caso de uso de Céu sob o domínio de definição e orquestração de experimentos em *testbeds*. Por fim, como parte desses esforços, revelou-se vantajoso o desenvolvimento de uma biblioteca própria de FRCP em Céu que abstrai do programador a execução de certas ações decorrentes do recebimento de mensagens de determinado tipo, como envio de respostas em novo tipo e manipulação do cabeçalho.

É necessária a adoção de uma biblioteca de AMQP para que o CE Céu seja capaz de se comunicar com os CRs segundo o processo delineado no capítulo anterior. Sua implementação em Céu como um módulo independente justifica-se por tornar mais conveniente seu emprego em futuras aplicações Céu. Implementamos esta biblioteca com base na biblioteca C *rabbitmq-c* (B), expondo as funcionalidades essenciais do AMQP por esta suportadas. Contudo, devido a natureza bloqueante da implementação de certas funcionalidades na biblioteca C, propriedade esta contrária ao paradigma síncrono de Céu, impomos limitações ao uso de tais funcionalidades para que a biblioteca oferecida seja de mais alto nível em Céu.

As bibliotecas que desenvolvemos adotam de forma o mais rigorosa possível os padrões de Céu, em detrimento aos conceitos dos protocolos AMQP e OMF. Por exemplo, ao invés de se definir uma variável de estado em uma chamada AMQP que destrói uma entidade quando do término de uma conexão, o programador deverá destruir tal entidade através do encerramento do escopo no qual sua representação em Céu fora instanciada. Deste modo, induzimos ao programador o conceito de programação estruturada de Céu: o fluxo de execução do programa se dá segundo reações a eventos, as quais implicam a

ativação ou terminação dos escopos, e por conseguinte a criação e finalização dos recursos neles instanciados.

Para validação dos produtos desenvolvidos, disponibilizamos testes que validam a corretude das funcionalidades da biblioteca AMQP e do CE OMF Céu. Este trabalho não oferece uma integração com OML, mas o código final do CE é o mais claro e extensível possível para que o mesmo seja ampliado e tal integração possa vir a ser adicionada em trabalhos futuros. Além disso, também ficou fora do escopo deste trabalho o desenvolvimento de uma biblioteca em Céu para implementação de controladores de recursos.

4
Atividades Realizadas

4.1
Estudos Preliminares

O aluno tinha prévia experiência com Céu, devido a participação em projeto de pesquisa para desenvolvimento de algoritmos distribuídos destinados a ambiente de redes de sensores sem fio. Contudo, tal pesquisa ocorreu ao longo de 2012 e a linguagem sofreu profundas mudanças desde então. Durante esta pesquisa o aluno também foi exposto ao conceito de *testbeds*, mas não chegou a utilizá-las.

Empreendemos um estudo inicial focado nas partes fundamentais do trabalho. Primeiramente, estudamos AMQP e RabbitMQ (uma implementação popular de *broker* AMQP), focando-se tanto no projeto de arquitetura quanto em sessões práticas segundo tutorais. Posteriormente, estudamos o arcabouço OMF, de modo a também entender seu projeto de arquitetura e como se encaixam, especificamente, o controlador de experimentos, os controladores de recursos e o FRCP.

4.2
Estudos Conceituais e da Tecnologia

Em razão da exposição limitada do aluno aos domínios dos protocolos AMQP e OMF, foi necessário um estudo aprofundado de suas documentações. Neste processo, estudamos também o servidor *RabbitMQ*, uma instância de um *broker* AMQP, e as implementações oficiais em *Ruby* de um controlador de experimentos OMF e de uma biblioteca para desenvolvimento de controladores de recursos OMF.

Através do estudo do material relacionado, familiarizamo-nos com as entidades primitivas de AMQP, aprendemos a forma como são expostas através do *RabbitMQ* e as propriedades deste para ser capaz de ajustar suas configurações a fim de realizar os testes necessários. Por fim, analisamos bibliotecas de AMQP em *JavaScript*, *Python* e *C*, a fim de termos uma boa base para projetar e implementar a versão em Céu.

Em especial, estudamos os exemplos, a implementação e documentação da biblioteca C de AMQP, visto esta ser uma dependência da biblioteca Céu. Desenvolvemos alguns exemplos para aprendizado de sua API e comprovação de sua interoperabilidade com as bibliotecas de outras linguagens. Além disso, este estudo levou a compreensão da natureza bloqueante da biblioteca, revelando a necessidade por soluções em Céu que superassem este problema, visto a premissa de sincronismo desta linguagem.

No contexto de OMF, estudamos a arquitetura de um projeto OMF, a fim de entender bem o encaixe de cada componente da mesma, focando principalmente nos papéis do controlador de experimentos e do servidor de mensagens utilizado para a comunicação entre os controladores de recursos e o controlador de experimentos. Ainda quanto ao processo de comunicação, dominamos os conceitos e regras estipulados pelo FRCP, que dita como a comunicação deve ocorrer.

Por fim, foi necessária a leitura do manual e o estudo dos tutoriais de Céu, para fins de refamiliarização com a linguagem. Dada mudança profunda recente na linguagem, este esforço teve de ser empreendido uma segunda vez ao final do desenvolvimento do projeto. Neste sentido, também foi importante adaptar-se ao estilo de programação reativa e estruturada imposto por Céu, que difere significativamente de linguagens as quais estava mais acostumado, como *C*, *Java* e *JavaScript*.

4.3 Testes e Protótipos para Aprendizado e Demonstração

A princíprio, implementamos uma versão inicial da biblioteca Céu de AMQP, que disponibiliza módulos equivalentes às entidades primitivas de AMQP além de auxiliares para certas operações, como o envio de mensagens. Para cada módulo desenvolvemos testes simples que validam sua funcionalidade. Por fim, implementamos dois exemplos empregando-se todos os módulos, a fim de se realizar um teste de integração completo. Em um exemplo, um processo enviou uma mensagem para um consumidor de mensagens desenvolvido na biblioteca de *Python*. No outro ocorre um processo implementa o caminho contrário: recebe uma mensagem de um publicador escrito em *Python*. Assim, a elaboração desta biblioteca seguindo boas práticas de Céu comprovou a viabilidade da linguagem para sua aplicação neste domínio.

Sob o escopo de OMF, além dos estudos realizados, executou-se um experimento simples sobre um controlador de recursos simulado utilizando-se a implementação em *Ruby* de um controlador de experimentos. Este teste possibilitou ao aluno aprender o funcionamento da ferramenta, observar o fluxo

da execução de um experimento e compreender a troca de mensagens que ocorre entre os dois pontos ao longo deste processo.

4.4 Método

O desenvolvimento de cada parte do projeto, isto é, os componentes AMQP e OMF, se deu em duas etapas: primeramente, um estudo do problema a ser atacado e, posteriormente, a implementação da solução.

O estudo dos conceitos, técnicas e ferramentas dos domínios de AMQP e OMF foi feito de forma bastante aprofundada, tendo sido os pontos mais relevantes documentados e apresentados para a orientadora. Os resumos da documentação foram cuidadosamente registrados para servirem de material de referência durante o desenvolvimento dos códigos do projeto.

O desenvolvimento dos códigos foi um processo bastante iterativo. Conforme os módulos foram desenvolvidos, consultou-se o autor da linguagem Céu para sua análise e crítica, com base na qual os módulos passaram pela refatoração necessária. Deste modo, foi possível o desenvolvimento de módulos fiéis ao paradigma de programação reativa e estruturada seguido pela linguagem e nos quais empregam-se as funcionalidades mais recentes de Céu, servindo assim também de um teste para as mesmas. Estes esforços contribuíram para a descoberta e correção de muitos *bugs* críticos que afetavam a linguagem. Por fim, todos os módulos apresentam testes compreensivos que validam seu funcionamento e servem de material auto-explicativo de seu uso.

4.5 Cronograma

O cronograma que segue eapresenta o desenvolvimento do trabalho ao longo do ano:

- Janeiro e Fevereiro: estudo de Céu, AMQP e OMF.
- Março: estudo de FRCP e de biblioteca em *Ruby* para criação de controladores de recursos OMF.
- Abril e Maio: desenvolvimento da biblioteca de AMQP e estudo da implementação em *Ruby* do controlador de experimentos OMF.
- Junho: documentação dos trabalhos desenvolvidos ao longo do período.
- Setembro a Outubro:
    - Refatoração do procedimento de recebimento de mensagens, extraído do módulo de filas e transferido para nova abstração de código.

  - Introdução e implementação do conceito de tratadores de mensagens.
  - Automação dos testes das biblioteca de AMQP em Céu.
  - Adaptação dos módulos de AMQP para nova versão de Céu.
  - Desenvolvimento de alguns exemplos oficiais do *RabbitMQ*.

- Novembro:

  - Refatoração do recebimento de mensagens para possibilitar a chamada concomitante de outros métodos de AMQP.
  - Desenvolvimento de biblioteca de FRCP, empregada pelo arcabouço OMF.
  - Implementação de módulo base de OMF em Céu.
  - Elaboração de testes simples para validação das implementações de FRCP e OMF.
  - Desenvolvimento dos exemplos oficiais restantes do *RabbitMQ*.
  - Escrita do relatório final de projeto.

- Dezembro: apresentação para a banca.

5
Projeto e Especificação do Sistema

5.1
Biblioteca Céu de AMQP

O sistema consiste de uma série de abstrações de código não-bloqueantes que mapeiam cada uma a um método de AMQP, respeitando o paradigma de programação reativa, síncrona e estruturada de Céu. Como parte destes esforços, tivemos que pensar em como expor a interface da biblioteca C de AMQP da qual dependemos em Céu. Como as chamadas dessa biblioteca são bloqueantes, invocá-las sem tratamento especial quebraria o paradigma síncrono de Céu. Também tivemos que enfrentar a impossibilidade de invocar métodos AMQP enquanto estivesse ativo o consumo de mensagens, problema este ausente em bibliotecas de outras linguagens de alto nível. A fim de superar ambos estes desafios procuramos estratégias próprias de Céu, valorizando as primitivas e construtos que a linguagem nos oferece. Deste modo, limitamos a exposição da biblioteca C de AMQP a algumas estruturas de dados por esta publicadas. Métodos para interação com *brokers*, contudo, são expostos e devem ser invocados apenas em Céu. Deste modo, código cliente desta biblioteca que siga este regime estará de acordo com o paradigma síncrono e não bloqueante Céu. Contudo, uma limitação da dependência não superada é ela não ser *thread-safe* por conexão; isto impede que em uma mesma conexão se façam chamadas em paralelo ao servidor.

O usuário da biblioteca segue uma série de passos similar a das demais, criando as entidades em ordem conforme sua necessidade. Assim, primeiro deve-se instanciar uma conexão que pode ser multiplexada em vários canais. Através de um canal podem-se definir as entidades de uma arquitetura típica de AMQP. Por fim, existe um módulo específico para iniciar o consumo de mensagens em determinado canal. É necessário tratar também a finalização dos recursos gerados. Uma maneira natural de se fazer isto em Céu é usar o próprio escopo. Isto é, ao término deste, finalizam-se todas as variáveis nele declaradas e abstrações de código nele invocadas. Isto possibilita construções interessantes e poderosas. Imagine o caso em que se deseja interromper o

consumo de mensagens quando da ocorrência de determinado evento. Basta utilizar uma composição de paralelismo na qual o consumo de mensagens é iniciado em uma trilha de duração infinita, cujo término decorre de disparo de evento esperado pela outra. Implementar este padrão em outra linguagem que não exponha eventos como cidadãos de primeira classe seria muito mais difícil.

A destruição de entidades no *broker* AMQP quando do encerramento de suas respectivas instâncias em Céu só se aplica àquelas restritas à aplicação, isto é, *bindings*, inscrições em filas e configurações da conexão. Ora, imagine o caso de uma fila com múltiplos consumidores, uma arquitetura comum para a distribuição da carga de processamento de tarefas. Caso um consumidor morra, seria indesejável que a fila fosse deletada em decorrência deste evento. Logo, isto não acontece. Contudo, as associações específicas deste consumidor ao meio são encerradas. Isto é outra propriedade da biblioteca que pode ser aproveitada por seus usuários para construção de composições poderosas. Por exemplo, inscrições em filas podem viver em trilhas em paralelo, sendo estas encerradas e iniciadas segundo sequências de eventos. Este é outro padrão que seria complexo de implementar em linguagens não reativas nem estruturadas.

Outro ponto único desta biblioteca é a garantia de segurança na declaração e acesso a recursos durante o ciclo de vida da aplicação. Isto se deve, novamente, ao uso de propriedades da linguagem Céu. Destacam-se duas funcionalidades: *aliases* e *watching*. As *aliases* são amarrações estáticas entre variáveis de Céu, similares a ponteiros. Contudo, ao contrário de ponteiros cujo acesso é irrestrito e cuja região referenciada pode ser liberada sem conhecimento do ambiente Céu, *aliases* tem garantia de existência enquanto o escopo superior no qual residem as variáveis-alvo de uma amarração ainda existir; como o ambiente Céu tem pleno conhecimento do escopo de declaração de variáveis, garante-se a associação segura de recursos, impedindo o problema que se potencialmente observaria com ponteiros. O construto *watching* provê garantia similar: ele atrela a existência de um escopo a de um recurso específico. Isto é, quando do encerramento do recurso, também se encerrará o escopo associado. Isso assegura que entidades internas a este escopo que dependam do recurso observado não façam uso inválido do mesmo.

Este aspecto de segurança é uma vantagem incidental do uso de Céu para implementação desta biblioteca. Contudo, procuramos oferecer a seu usuário outras facilidades. Em particular, destaca-se a função de *acknowledgment* de recebimento de mensagens. Enquanto nas demais bibliotecas fica a cargo do programador realizar esta operação, a mesma é feita automaticamente pela biblioteca Céu. Isto ocorre apenas caso tal funcionalidade tenha sido ativada e se o tratamento de uma mensagem cujo recebimento deva ser reconhecido

tenha sido bem sucedido. Além disso, a associação de mensagens às filas na qual foram recebidas é também tratada de modo automático, assim como em outras bibliotecas de alto nível, mas não na biblioteca de C.

5.2 FRCP

O ponto chave da implementação em Céu deste protocolo é abstrair ao máximo suas especificidades, de modo que o programador possa trabalhar quase que exclusivamente com o *payload* da mensagem. A única limitação é o atributo *guard*, que contém os parâmetros e valores do estado do recurso a que se destina a mensagem; deste modo, os recursos devem verificar a presença deste atributo e, caso exista, os valores por ele contidos, a fim de determinar se são os destinatários da mensagem.

Esta implementação foi desenvolvida sobre a biblioteca de AMQP, e provê abstrações de código que publicam os cinco tipos de mensagem definidos pelo protocolo. Estas abstrações são necessárias para o desenvolvimento da aplicação central de controle responsável por orquestrar os recursos. A biblioteca exige que aplicações consumidoras implementem tratadores de mensagem para cada um de seus diferentes tipos. A biblioteca implementa um tratador geral no qual inclui pontos de entrada para tais implementações concretas dos tratadores de cada tipo de mensagem. A biblioteca se responsabiliza por enviar à aplicação central o resultado da operação requisitada, uma abstração que permite às aplicações consumidoras se preocuparem apenas com suas regras de negócio e os dados que pretendem transmitir.

Outra vantagem desta biblioteca é que ela lida com a gerência de tópicos e inscrições nos mesmos, abstraindo totalmente este conceito das aplicações consumidoras. Isto é uma funcionalidade muito poderosa que também simplifica a lógica de aplicações consumidoras.

5.3 Controlador de Experimentos OMF em Céu

A versão Céu do CE faz uso basicamente da biblioteca de FRCP discutida acima. Através desta, o CE, a aplicação central de controle, rege a orquestração de todos os recursos sob teste segundo o que for especificado na DE em Céu. Ao contrário de OEDL, em que a descrição de um experimento é feita inteiramente na própria linguagem, optamos por utilizar a integração com Lua para as fases de descrição de aplicações e de grupos. Isto porque, em essência, tais fases são meramente definições de tabelas em Lua, facilitando este procedimento para o pesquisador e a gerência do estado das aplicações em Céu.

A fase de experimento, por outro lado, é completamente escrita em Céu. Nesta, podem-se definir os eventos esperados e as ações subsequentes a serem executadas por aplicações específicas de determinados grupos de recursos. Para invocar a execução de ações, como a inicialização de aplicações, são providas abstrações de código similares às de OEDL. Além disso, todos os construtos de Céu podem ser utilizados; ou seja, podem ser feitas composições de paralelismo, que tornam possível a definição de padrões indisponíveis na versão em Ruby e que por fim conferem mais controle sobre o experimento ao pesquisador.

O registro do estado de aplicações é mantido em Lua, visto que os atributos e parâmetros destas são dinâmicos e definidos pelos usuários. Isto é necessário já que Céu não provê estruturas de dados tão facilmente extensíveis e dinâmcas como tabelas de Lua o são. O estado das aplicações deve ser exposto, de modo que um usuário do arcabouço possa definir novos eventos baseados na verificação dos estados das instâncias de aplicações, funcionalidade esta suportada por OEDL.

6
Implementação e Avaliação

6.1
Desafios e Soluções

6.1.1
Biblioteca de AMQP em Céu

Os principais desafios enfrentados se deram durante o desenvolvimento da biblioteca Céu de AMQP. Destacam-se quatro desafios: adaptação das invocação de métodos bloqueantes da biblioteca C, implementação de estratégia não bloqueante para finalização de recursos, chamadas intercaladas com o laço de consumo de mensagens e, por fim, introdução de estratégia original para tratamento de mensagens recebidas. Estes problemas foram resolvidos com diferentes expressões idiomáticas e funcionalidades de Céu, exploradas a seguir.

Uma limitação da biblioteca C é a natureza bloqueante dos métodos por ela publicados. Mesmo em casos em que um *timeout* possa ser definido, existe a possibilidade do método bloquear enquanto são recebidos pacotes de uma mensagem completa de AMQP. Devido a isto, foi necessário envolver tais chamadas em *threads* de curta duração para que as abstrações de código das entidades AMQP não quebrassem com o paradigma síncrono de Céu. Contudo, como o recurso resultante da operação não estará disponível até o término a chamada e a execução de uma thread cede o controle de execução à trilha invocante, é necessário o uso de um evento que sinaliza quando o recurso estiver disponível.

```
1  code/await Res( /* ... */ ) -> (event& void ok) -> FOREVER do
2    event void ok_;
3    ok = &ok_;
4
5    await async/thread( /* ... */ ) do
6      _aqmp_blocking_method( /*...*/ );
7    end
8    emit ok_;
9
10   // codigo de finalizacao...
```

```
  await FOREVER;
end

event& void ok;
spawn Res( /* ... */ ) -> (&ok)
await ok;
```

Exemplo de código 6.1: Exemplo fictício de entidade AMQP com thread e ok

A finalização de recursos em Céu é feita através de blocos de finalização. Estes podem ser declarados em abstrações de código, sendo utilizados quando estas criam algum recurso e capturam seu estado. Uma exigência de tais blocos é que sua execução seja instantânea, o que impede o uso de estruturas de sincronismo, como *await*. Deste modo, simplesmente envolver a chamada AMQP para finalização de um recurso em uma *thread* não é possível. Para superar este problema, utilizou-se uma *pool* genérica para liberação de recursos, na qual é instanciado uma abstração que encerra o recurso de maneira não bloqueante. Módulos de entidades que precisem liberar recursos devem implementar uma instância concreta de um tipo abstrato e também sua respectiva abstração de código tratadora, a qual pode ser instanciada na *pool* genérica de liberação de recursos graças ao suporte de polimorfismo.

```
data Release_Target.Concreto with
  // propriedades do recurso...
end

code/await/dynamic Release_Entity(var& Release_Target.Concreto
    alvo, /* ... */) -> void do
  // ...
  await async/thread( /* ... */ ) do
    _amqp_release_call(  /* ... */ );
  end
  // ...
end

code/await Res( /* ... */ ) -> (event& void ok) -> FOREVER do
  // codigo de instanciacao...

  var Release_Target.Concreto alvo = /* ... */;
  do finalize with
    spawn/dynamic Release_Entity(&alvo, /* ... */) in outer.
    default_release_pool;
  end

  await FOREVER;
```

```
22 end
```

Exemplo de código 6.2: Estratégia para liberação de recursos

Outra deficiência da biblioteca de AMQP em C é a impossibilidade de serem feitas chamadas para criação de recursos enquanto o canal estiver consumindo mensagens. Nesta situação, a resposta enviada pelo servidor não é capturada pela biblioteca, mas sim dada ao usuário como se fosse uma mensagem recebida de uma fila. Caso não se lide com isso o usuário seria responsável por interromper o consumo de mensagens sempre que quisesse fazer outra chamada. A biblioteca em Céu dispara eventos de controle que interrompem e reiniciam o consumo de mensagens sempre que uma chamada AMQP for feita. Deste modo, chamadas AMQP feitas pelo usuário através de seus respectivos módulos Céu são corretamente intercaladas com o consumo de mensagens, sem sua participação.

```
code/await AMQP_Call (var& Channel ch, /* ... */) -> void do
  // ...
  emit ch.pause_consuming;
  await async/thread( /* ... */ ) do
    _amqp_release_call( /* ... */ );
  end
  emit ch.resume_consuming;
  // ...
end

code/await Consume(var& Channel ch, /* ... */) -> (event& void ok)
    -> FOREVER do
  // ...

  loop do
    par/or do
      await ch.pause_consuming;
    with
      loop do
        // recebe novas mensagens...
      end
    end
    await ch.resume_consuming;
  end
end
```

Exemplo de código 6.3: Intercalação de consumo de mensagens e chamadas AMQP

Por fim, a principal contribuição é a estratégia de tratamento de mensagens. Em outras linguagens a definição do tratador de mensagens é comumente

feita através da especificação de uma *callback* quando da inscrição de um consumidor em uma fila, expressão idiomática esta não existente em Céu. A solução desenvolvida foi a de se exigir do cliente da biblioteca que implemente um tratador geral, no qual internamente através de controles de fluxo lidar-se-á com as mensagens de forma apropriada. A fim de se determinar o tratador responsável por mensagens oriundas de determinada fila, deve-se prover no ato de inscrição a esta fila um identificador do tratador, sendo o mapeamento entre a fila e o tratador mantido internamente através de uma tabela Lua. A biblioteca de AMQP lida com a associação das mensagens recebidas a estes identificadores, sendo ambos passados ao tratador central.

```
code/await Handler (var& Envelope env, /* ... */) -> void do
  // Implementado pelo usuario...
end

code/await LowHandler (var _amqp_message_t msg, /* ... */) -> void
    do
  var int handler_id = /* obtem ID do tratador para `msg` */ ;
  await Handler (Envelope(handler_id, msg));
end

code/await Consume(var& Channel ch, /* ... */) -> (event& void ok)
    -> FOREVER do
  // ...
  loop do
    // recebe novas mensagens...
    spawn LowHandler(msg, /* ... */) in handler_pool;
  end
  // ...
end
```

Exemplo de código 6.4: Tratamento de mensagens recebidas pela biblioteca de AMQP

6.1.2 FRCP

No contexto de FRCP um primeiro desafio que teve de ser superado foi a manipulação de dados JSON. Visto que Céu não apresenta nenhuma biblioteca ou tipo para manipulação nativa de objetos JSON, foi necessário relegar esta tarefa a Lua. Foi possível assim se fazer de maneira fácil a verificação de presença de cabeçalhos específicos e avaliação de seus respectivos valores, assim como a montagem de mensagens de resposta. O código abaixo exibe a tradução de um *payload* JSON para sua respectiva representação em Céu. Note que se atribuem cadeias de caracteres vazias aos membros opcionais do *payload*; isto é

importante tanto para o processamento apropriado de uma mensagem, quanto para a construção correta do valor codificado em JSON posteriormente (i.e. para envio da mensagem). Note, ainda, o uso da integração com Lua para manipulação do objeto JSON.

```
code/tight JSONToPayload(vector&[] byte json_str, var& Payload
    payload) -> void
do
    [[
        frcp_json_tab = JSON:decode(@json_str)
    ]]

    payload.op      = [] .. [[ frcp_json_tab.op ]];
    payload.mid     = [] .. [[ frcp_json_tab.mid ]];
    payload.src     = [] .. [[ frcp_json_tab.src ]];
    payload.ts      = [] .. [[ frcp_json_tab.ts ]];
    payload.props   = [] .. [[ JSON:encode(frcp_json_tab.props) ]];
    payload.rp      = [] .. [[ frcp_json_tab.rp or "" ]];
    payload.it      = [] .. [[ frcp_json_tab.it or "" ]];
    payload.cid     = [] .. [[ frcp_json_tab.cid or "" ]];
    payload.reason  = [] .. [[ frcp_json_tab.reason or "" ]];
    payload.guard   = [] .. [[ JSON:encode(frcp_json_tab.guard) or
    "" ]];
end
```

Exemplo de código 6.5: Tradução de JSON para objeto em Céu

Superado-se este desafio, partimos para o desenvolvimento do tratador de mensagens. Em linhas gerais, o tratador funciona segundo o regimento apresentado abaixo. Primeiramente, verifica-se que o receptor da mensagem não foi seu remetente e que, segundo a propriedade *guard*, a mensagem destina-se a tal nó. Posteriormente, encaixamos os pontos de partida dos tratadores que devem implementados pelo usuário da biblioteca. A triagem de mensagens e qualquer pré-processamento é feita pela própria biblioteca. Observe que, como discutido anteriorment, o tratamento de inscrição em novos tópicos é feito por completo pela biblioteca, completamente invisível do usuário desta. Por fim, é feito um tratamento especial de respostas a mensagens do tipo *configure* que abordaremos a seguir.

```
// Setup inicial e checagem de destinatario...
var int handler_err = FRCP_RET_SUCCESS;
if /* INFORM */ then
    handler_err = await Handle_Inform( /* ... */ );
else
    // Lida com mensagens que requisitam alguma acao do recurso
    if /* CREATE */ then
        spawn Handle_Create(&in_payload.props) -> ( /* ... */ );
```

```
9           // prepara a resposta
10      else/if /* CONFIGURE */ then
11          vector[] byte membership = /* string do membership, caso
        exista */
12          if ($membership > 0) then
13              // Inscricao invisivel ao usuario em novo topico
14          else
15              spawn Handle_Configure(&in_payload.props) -> ( /* ...
        */ );
16              // Codigo especial de tratamento de resultados do `
        configure`
17          end
18      else/if /* REQUEST */ then
19          vector&[] byte new_props;
20          spawn Handle_Request(&in_payload.props) -> ( /* ... */ );
21          // prepara a resposta
22      else/if /* RELEASE */ then
23          spawn Handle_Release(&in_payload.props) -> ( /* ... */ );
24          // prepara a resposta
25      else
26          // controle de algumas variaveis...
27      end
28      //  Automaticamente define insere mensagem de erro
29      if handler_err == FRCP_RET_ERROR then
30          response.reason = [] .. err_reason;
31      end
32
33      // Publica mensagem preenchida em algum trecho acima
34  end
```

Exemplo de código 6.6: Tratador central de mensagens FRCP

O tratamento especial de mensagens do tipo *configure* se deve ao fato do protocolo admitir que em casos de configuração demorada, os recursos enviem mensagens sobre seu estado transiente. Assim, o requerente da configuração tem plena noção em qual estágio desta operação encontra-se o recurso afetado. O protocolo especifica um formato no qual devem ser descritos os atributos que são reportados em estado transiente; esta formatação não é feita pela biblioteca, ficando a cargo do próprio usuário. Contudo, a biblioteca provê uma abstração através da qual o cliente pode despachar o envio de novas mensagens sobre estado transiente.

Esta funcionalidade faz uso de mais uma expressão idiomática em Céu. O tratador de *configure* provido pelo usuário captura um vetor, no qual armazena o JSON do estado a ser enviado, e um evento, que dispara quando deseja transmitir a mensagem. Tal evento carrega um valor que indica se o processo

de configuração do recurso ainda está em andamento ou já se encerrou. Ao se transmitir a mensagem final, o tratamento especial se encerra, e esta é enviada ao final do tratador geral como as demais mensagens. Por fim, observe o uso de uma *pool* para despacho do envio da mensagem, equivalente a *pool* de liberação de recursos apresentadas na seção anterior. O uso desta desta *pool* é necessário pois precisamos esperar imediatamente pelo recebimento da próxima mensagem, a fim de não perder mensagem alguma.

```
1  vector&[] byte new_props;
2  spawn Handle_Configure(&in_payload.props) -> (&handler_yield, &
       err_reason, &new_props);
3
4  // A cada evento emitido pelo cliente:
5  every handler_err in handler_yield do
6    // Prepara o cabeçalhos da mensagem de resposta
7
8    // Lida automaticamente com deteccao de erro
9    if handler_err != FRCP_RET_WIP then
10     if handler_err == FRCP_RET_ERROR then
11       response.it = [] .. FRCP_IT_ERROR;
12     end
13
14     // Em caso de mensagem final, encerra o tratador.
15     break;
16   end
17
18   // Despacha envio de mensagem intermediaria
19   spawn Publish_Payload( /* ... */ ) in outer.
       frcp_publish_dispatch;
20 end
```

Exemplo de código 6.7: Tratador especial de mensagens ‘configure‘

Este código aproveita a integração com Lua para manter um registro de todas as inscrições feitas pelo recurso. Isto é necessário para produzir um JSON que encapsula o estado completo de inscrições de um recurso e que é enviado como resposta a um pedido de inscrição. O requerente, ao receber esta resposta, passa a ter uma visão completa das inscrições do nó. Note nos trechos exibidos abaixo como aproveitamos a integração com Lua para acessar o registro em pontos distintos e, principalmente, para facilitar a manipulação do JSON e produção de cadeia de caracteres a partir deste.

Neste ponto também utilizamos duas *pools* globais providas pela biblioteca para instanciação da inscrição e do tópico de qual ela depende. Por que isto? Porque o tratador de mensagens tem curta duração e, ao seu término, tais entidades seriam finalizadas. Como desejamos que a inscrição permaneça

após o fim do tratador, é necessário instanciá-la em uma *pool* de um escopo superior.

```
if ($membership > 0) then
    // Setup...
    // Note a criacao em uma pool externa.
    spawn New_Topic(&comm, &membership) -> (&memb_topic, &
    memb_topic_ok)
        in outer.frcp_topic_pool;
    await memb_topic_ok;

    // Mais setup...
    // Mesma expressao idiomatica.
    spawn Subscribe_Topic(&comm, &memb_topic) -> (&sub_id, &sub_ok
    )
        in outer.frcp_subscription_pool;
    await sub_ok;

    // Integracao com Lua: Produz a lista completa de inscricoes
    do recurso
    [[
        memb_keyset = {}
        for (k,v) in pairs(frcp_membership_set) do
            table.insert(memb_keyset, k)
        end
        config_props.membership = memb_keyset
        config_props_str = JSON:encode(config_props)
    ]]
    response.props = [] .. [[ config_props_str ]];
    response.it    = [] .. FRCP_IT_STATUS;
else
  // ...
end
```

Exemplo de código 6.8: Inscrição automática em tópicos

É necessário, contudo, prover alguma forma de se finalizar essas invocações de código; isto é tratado quando da invocação do código que estabelece a inscrição. Para isto, são geradas instâncias de estruturas associadas a tais invocações. Cada um desses objetos contém um evento cujo disparo finaliza sua respectiva invocação de código. Tais objetos são mantidos em um vetor central. Assim, pode-se percorrer tal vetor para encontrar o objeto associado à invocação que se deseja encerrar e, então, disparar o evento que a finaliza.

```
code/await Subscribe_Topic(var& Communicator communicator, var&
    Topic topic) -> (var& Subscription sub_id, event& void ok) ->
    FOREVER
do
```

```
3     event void stop;
4     var Subscription sub_id_ = val Subscription(&topic, &stop);
5     sub_id = &sub_id_;
6
7     // Aqui mantem-se registro ao identificador desta invocacao
8     outer.frcp_subscription_ids = outer.frcp_subscription_ids ..
    [&&sub_id_];
9
10    // Configuracao da inscricao...
11    // Finaliza o registro da inscricao em Lua
12    do finalize with
13        [[
14            frcp_membership_set[@topic.name] = nil
15        ]]
16        emit topic.terminate;
17    end
18
19    // Mantem registro da inscricao em Lua
20    [[
21        frcp_membership_set[@topic.name] = true
22    ]]
23
24    emit ok_;
25    await stop;
26 end
27 // Pool que armazena instancias das invocacoes de codigo
28 pool[] Subscribe_Topic frcp_subscription_pool;
```

Exemplo de código 6.9: Inscrição em tópico com registro de identificador

6.1.3 OMF

Procuramos aqui oferecer uma experiência tão simples quanto OEDL para a descrição de experimentos, aproveitando o sistema nativo de tratamento de eventos de Céu. Na seção 2.2 tratamos de três pontos falhos da implementação do CE Ruby: a necessidade de o pesquisador manter, ele próprio, um conceito global de tempo para definir os instantes em que entidades serão iniciadas ou encerradas; o fenômeno de *callback hell*, que torna o código bastante aninhado e ininteligível; e, por fim, a declaração explícita do término de experimentos. Todos estes problemas podem ser superados através de soluções nativas da linguagem Céu, exibidas abaixo para o exemplo de experimento de duas aplicações da seção 2.2.

A definição de aplicações deve ser feita através da declaração de uma tabela global de aplicações, obrigratoriamente chamada *apps*. Nesta tabela é

feita uma associação entre o identificador de cada aplicação e suas respectivas definições. Note que existem alguns atributos padrões que precisam ser definidos pelo usuário, enquanto aqueles específicos da aplicação devem ser especificados no membro *props* da tabela. Visto que a definição da aplicação é meramente uma tabela Lua, a produção destas é até mais simples do que em OEDL.

```
apps = {
    ping = {
        -- Atributos padroes
        description = "Simple Definition for the ping application",
        binary_path = "/usr/bin/ping",
        -- Atributos especificos da aplicacao
        props = {
            target = {
                description = "Address to ping",
                command     = "-a",
            },
            count = {
                description = "Number of times to ping",
                command     = "-c",
            },
        },
    },
    date = {
        description = "Simple Definition for the date application",
        binary_path = "/usr/bin/date",
        props = {
            date = {
                description = "Display time described by STRING, not now",
                command     = "--date",
            },
        },
    },
}
```

Exemplo de código 6.10: Definição de duas aplicações em Céu através de Lua

```
defApplication('ping') do |app|
  app.description = 'Simple Definition for the ping application'
  app.binary_path = '/usr/bin/ping'

  app.defProperty('target', 'Address to ping', '')
  app.defProperty('count', 'Number of times to ping', '-c')
end
```

```
8
9  defApplication('date') do |app|
10   app.description = 'Simple Definition for the date application'
11   app.binary_path = '/usr/bin/date'
12
13   app.defProperty('date', 'display time described by STRING, not
       now', '--date')
14 end
```

Exemplo de código 6.11: Definição de duas aplicações em OEDL

De forma similar, também definem-se grupos através de uma tabela Lua. Assim como para aplicações, esta tabela global, obrigatoriamente chamada *groups*, mantém uma associação entre os identificadores de grupos e suas definições. Na definição dos grupos deve-se indicar os nós que deles fazem parte e também as aplicações a serem instanciadas, com seus respectivos atributos de inicialização. Quando da inicialização do ambiente de experimentação pode se verificar que a definição de uma instância de aplicação respeita sua especificação.

```
1  groups["Actor"] = {
2      nodes = { "omf-rc" },
3      apps = {
4          ping_google = {
5              app_id = "ping",
6              target = "www.google.com",
7              count  = 3,
8          },
9          date_LA = {
10             app_id = "date",
11             date   = 'TZ="America/Los_Angeles" 09:00 next Fri',
12         }
13     },
14 }
```

Exemplo de código 6.12: Definição de grupo em Céu através de Lua

```
1  defGroup('Actor', 'omf-rc') do |g|
2    g.addApplication("ping") do |app|
3      app.name = 'ping_google'
4      app.setProperty('target', 'google.com')
5      app.setProperty('count', 3)
6    end
7
8    g.addApplication("date") do |app|
9      app.name = 'date_LA'
10     app.setProperty('date', 'TZ="America/Los_Angeles" 09:00 next
       Fri')
```

```
11   end
12 end
```

Exemplo de código 6.13: Definição de grupo em OEDL

Por fim, tem-se o experimento. O usuário deve incluir o arquivo Céu que inicializa o ambiente de experimentação e criar e preencher uma abstração de código com a assinatura abaixo. Observe as vantagens deste exemplo frente ao original em OEDL. Primeiramente, o aproveitamento do regime global de tempo oferecido pelo ambiente Céu permite a descrição do experimento de forma mais simples: o pesquisador precisa apenas contar o tempo que deve se passar entre as reações e não quanto tempo se passou ao todo desde o início do experimento. Em segundo lugar, o código apresenta um único nível de aninhamento, aproveitando-se da espera nativa a eventos oferecida por Céu atráves do comando *await*. Isto torna mais fácil a leitura do código frente a abordagem de OEDL previamente apresentada, que superava a deficiência do regime de tempo global do CE Ruby através do uso de *callbacks* aninhadas, mas que, em razão disto, sofria de *callback hell*. Por fim, enquanto em OEDL precisávamos indicar o término de um experimento através da invocação de comando específico, em Céu basta encerrar o escopo do experimento para que todas as entidades (i.e. grupos, aplicações e dependendências) sejam automaticamente liberadas. Deste modo, conseguimos replicar fielmente em Céu o exemplo original descrito em OEDL.

```
1 #include "omf_start.ceu"
2
3 code/await Experiment(var& Communicator c) -> void do
4     await outer.omf_all_up;
5     await 3s;
6
7     vector[] byte group_id = [] .. "Actor";
8
9     // Dispara app de ping
10    vector[] byte ping_app_id   = [] .. "ping_google";
11    spawn Start_Application(&c, &group_id, &ping_app_id);
12    await 3s;
13
14    // Dispara app de ping
15    vector[] byte date_app_id   = [] .. "date_LA";
16    spawn Start_Application(&c, &group_id, &date_app_id);
17
18    // Experimento encerrado automaticamente para o usuario (nao
      precisa chamar finalizacao)
19    await 3s;
```

```
20 end
```

Exemplo de código 6.14: Definição de experimento com duas aplicações em Céu

```
1  onEvent(:ALL_UP) do |event|
2    # 'after' is not blocking. This executes 3 seconds after :ALL_UP
       fired.
3    after 3 do
4      info "TEST - do ping app"
5      group("Actor").startApplication("ping_google")
6    end
7
8    # 'after' is not blocking. This executes 6 seconds after :ALL_UP
       fired.
9    after 6 do
10     info "TEST - do date app"
11     group("Actor").startApplication("date_LA")
12   end
13
14   # 'after' is not blocking. This executes 9 seconds after :ALL_UP
       fired.
15   after 9 do
16     Experiment.done
17   end
18 end
```

Exemplo de código 6.15: Definição de experimento com duas aplicações em OEDL

O exemplo que segue, embora abstrato, demonstra uma vantagem de se escrever uma DE em Céu. Podemos desenvolver sequências de teste em paralelo, de modo que uma trilha possa afetar a outra ao promover indiretamente o disparo de eventos. No exemplo, as operações feitas pela segunda trilha podem levar a primeira a acordar e executar outras modificações sobre os recursos em teste. Esta construção possibilita um controle mais granular sobre a evolução de um experimento. Ainda, podemos facilmente introduzir um tempo limite ao experimento, empregando o padrão de trilha de *timeout*. Em contrapartida, OEDL é rigidamente serial, sendo bastante difícil produzir lá uma construção paralela equivalente.

```
1  // Definicao de experimento
2  code/await Experiment(var& Communicator c) -> void do
3      await outer.omf_all_up;
4
5      // Setup basico
6      spawn Start_Application(/* ... */);
```

```
7     await 3s;
8
9     par/or do
10      par/and do
11        await custom_evt_1;
12        spawn Execute_Command(/* ... */);
13
14        // ...
15      with
16        await 10s;
17        spawn Update_Application(/* ... */);
18
19        await 5s;
20        spawn Execute_Command(/* ... */);
21
22        // ...
23      end
24    with
25      await 5min;
26    end
27    // ...
28 end
```

Exemplo de código 6.16: Experimento com construção paralela

6.2 Testes Funcionais

Embora haja um certo grau de automação, conferido pelo uso de *Makefiles* que ativam a execução de testes quando da compilação do código fonte, este processo ainda é bastante manual e dependente da verificação dos resultados pelo mantenedor da biblioteca.

A biblioteca Céu de AMQP foi a mais completamente testada. Os primeiros testes da biblioteca foram protótipos pequenos visando verificar sua interoperabilidade com o recíproco equivalente implementado em outra linguagem, como Python ou Ruby. Tendo este experimento sido bem sucedido, partia-se para o desenvolvimento de outro módulo.

Posteriormente, para verificação um pouco mais robusta, implementaram-se os exemplos oficiais de RabbitMQ. Estes acabam por servir de uma prova de conceito do funcionamento de todas as primitivas e operações básicas do protocolo na biblioteca Céu. Ainda, como parte deste esforço, os tutoriais foram adaptados ao ambiente Céu, podendo ser assim discutidas em maiores detalhes as vantagens que este ambiente confere e suas peculiaridades.

Ainda, para verificação de seu adequado desempenho, foi feito um teste de estresse em que múltiplos produtores enviavam mensagens simultaneamente a um consumidor, a uma taxa de uma mensagem a cada microsegundo. Constatou-se que a aplicação funcionou sem problemas, indicando que o ambiente de Céu não constitui limitador quando utilizado como suporte a um sistema de AMQP.

Para validação da implementação de FRCP desenvolveram-se alguns testes simples que verificam a corretude da sequência básica de troca de mensagens quando do envio de cada tipo. Visto que esta implementação abstrai para seus usuários muitos detalhes do protocolo, estes testes são ainda mais importantes, já que será complicado para o usuário mitigar falhas do mesmo.

Por fim, a verificação do CE de OMF consistiu na execução de um experimento padrão associado a um controlador de recursos desenvolvido por pesquisadores ligados ao LabLua. Apesar de ser um experimento simples, cobre em detalhes a sequência de passos seguidas por todo experimento, constituindo assim um teste bastante genérico. Deste modo, verifica que os resultados produzidos pelo CE desenvolvido em Céu batem com os do CE original.

7
Considerações Finais

Neste trabalho produzimos três artefatos em Céu que expõem, segundo o paradigma de programação reativa e estruturada da linguagem, as entidades e métodos de três protocolos distintos: OMF, FRCP e AMQP. Os esforços empreendidos visaram validar o uso de Céu para solucionar problemas aos quais a linguagem não foi originalmente pensada. Isto é, sua concepção se deu segundo os problemas enfrentados para programação de redes de sensores sem fio. Ao empregarmos a linguagem neste trabalho, validamos a possibilidade de moldar o código produzido segundo as limitações e imposições dos protocolos implementados e dependência utilizadas, evitando ao máximo delegar operações aos ambientes integrados de C e Lua.

Atingimos resultados satisfatórios nos testes apresentados. No caso da biblioteca de AMQP conseguimos verificar sua interoperabilidade com implementações de outras linguagens, como Python e Ruby. Ainda, fomos bem sucedidos em implementar os exemplos oficiais, utilizando padrões naturais de Céu. Quanto ao controlador de experimentos OMF, validamos sua capacidade de orquestrar um controlador de recursos de maneira equivalente à implementação oficial em Ruby.

Trabalhos futuros poderiam empreender a ampliação das funcionalidades das bibliotecas desenvolvidas ou a integração ao *testbed* CéuNaTerra do controlador alternativo de experimentos OMF implementado em Céu. Devido à complexidade dos protocolos implementados, nos limitamos a oferecer um conjunto de métodos essenciais que possibilite o desenvolvimento em Céu dos casos de uso mais comuns desses protocolos. A integração com o *testbed* seria uma valiosa contribuição, já que este poderia ser então utilizado para a realização de experimentos descritos em Céu.

Referências Bibliográficas

[1] RAKOTOARIVELO, T.; OTT, M.; JOURJON, G. ; SESKAR, I.. **OMF: A Control and Management Framework for Networking Testbeds**. SIGOPS Oper. Syst. Rev., 43(4):54–59, Jan. 2010. 1, 2.1

[2] DWERTMANN, C.; RAKOTOARIVELO, T.. **About OMF - An Executive Summary**. https://omf.mytestbed.net/projects/omf/wiki/Introduction. 1

[3] ROSSETTO, S.; SILVESTRE, B.; RODRIGUEZ, N.; BRANCO, A.; KAPLAN, L.; LOPES, I.; BOING, M.; SANTANA, D.; LIMA, V.; GABRICH, R. ; OTHERS. **CeuNaTerra: testbed para espaços inteligentes**. http://sbrc2015.ufes.br/wp-content/uploads/b-139668_1.pdf, 2015. 1

[4] SANT'ANNA, F.; OTHERS. **Céu: Embedded, Safe, and Reactive Programming**. Technical Report 12/12, PUC-Rio, 2012. 1

[5] BRANCO, A.; SANT'ANNA, F.; IERUSALIMSCHY, R.; RODRIGUEZ, N. ; ROSSETTO, S.. **Terra: Flexibility and Safety in Wireless Sensor Networks**. ACM Trans. Sen. Netw., 11(4):59:1–59:27, Sept. 2015. 1

[6] RAKOTOARIVELO, T.; DWERTMANN, C. ; HONG, J.. **The OMF Experiment Description Language (OEDL)**. https://mytestbed.net/projects/omf6/wiki/OEDLOMF6. 2.1

[7] **Experiment Controller (EC)**. http://mytestbed.net/doc/omf/file.experiment_controller.html. 2.1

[8] **Resource Controller (RC)**. http://mytestbed.net/doc/omf/file.resource_controller.html. 2.1

[9] RAKOTOARIVELO, T.. **Federated Resource Control Protocol (FRCP)**. https://github.com/mytestbed/specification/blob/master/FRCP.md. 1, 2.1, 2.2

[A] OASIS. **Advanced Message Queuing Protocol (AMQP) Version 1.0**. http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf, 2012. 1

[B] ANTONUK, A.. **RabbitMQ C AMQP client library**. `https://github.com/alanxz/rabbitmq-c`, 2016. 3

[C] MATTOSO, C.. **Céu RabbitMQ - Céu API for RabbitMQ**. `https://github.com/calmattoso/ceu-rabbitmq`, 2016. 3

[D] MATTOSO, C.. **Céu OMF Experiment Controller**. `https://github.com/calmattoso/ceu-omf-ec`, 2016. 3