Eduardo Castro Mota Câmara

Um Estudo sobre Atualização Dinâmica de Componentes de Software

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós–graduação em Informática

Rio de Janeiro
Março de 2014

Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Castro Mota Câmara

Um Estudo sobre Atualização Dinâmica de Componentes de Software

Dissertação de Mestrado

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador: Prof. Noemi de La Rocque Rodriguez

Rio de Janeiro
Março de 2014

Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Castro Mota Câmara

Um Estudo sobre Atualização Dinâmica de Componentes de Software

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Noemi de La Rocque Rodriguez
Orientador
Departamento de Informática — PUC–Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática — PUC-Rio

Prof. Roberto Ierusalimschy
Departamento de Informática — PUC-Rio

Prof. Alexandre Sztajnberg
UERJ

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 26 de Março de 2014

Eduardo Castro Mota Câmara

Graduou-se em Engenharia da Computação pela Pontifícia Universidade Católica do Rio de Janeiro.

Ficha Catalográfica

Câmara, Eduardo Castro Mota

Um estudo sobre atualização dinâmica de componentes de software / Eduardo Castro Mota Câmara; orientador: Noemi de La Rocque Rodriguez. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2014.

v., 85 f: il. ; 30,0 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Atualização Dinâmica de Software; Atualização de Software; Componentes de Software; Arquitetura de Software. I. Rodriguez, Noemi de La Rocque. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Primeiramente gostaria de agradecer a Deus, sem ele jamais teria conseguido chegar até aqui. Agradeço pela minha vida, as oportunidades e lições aprendidas.

Gostaria de agradecer também aos meus orientadores Renato Cerqueira e Noemi Rodriguez pelo conhecimento e orientação que me passaram durante toda a realização deste trabalho. Agradeço a PUC-Rio, a CAPES e o TECGRAF pelo corpo docente de extrema qualidade, pelo apoio e pela complementação da minha formação.

A minha esposa, Gabriela, aos meus pais, João e Margarene, e aos meus sogro e sogra, Cláudio e Carla, por todo o amor, carinho, paciência e amparo que tiveram comigo nesta caminhada.

Aos meus amigos pelos incentivos e incansáveis debates sobre a evolução deste trabalho. Em especial ao Pablo, pelos puxões de orelha e incentivos nas horas de deadlock.

Gostaria também de agradecer a todos os professores da PUC-Rio que contribuíram na minha formação acadêmica e profissional.

Muito obrigado!

Resumo

Câmara, Eduardo Castro Mota; Rodriguez, Noemi de La Rocque. **Um Estudo sobre Atualização Dinâmica de Componentes de Software**. Rio de Janeiro, 2014. 85p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O desenvolvimento baseado em sistemas de componentes de software consiste em compor sistemas a partir de unidades de sotfware prontas e reutilizáveis. Muitos sistemas de componentes software em produção, precisam ficar disponíveis durante 24 horas por dia nos 7 dias da semana. Atualizações dinâmicas permitem que os sistemas sejam atualizados sem interromperem a execução dos seus serviços, aplicando a atualização em tempo de execução. Muitas técnicas de atualização dinâmica, na literatura, utilizam aplicações feitas especificamente para cobrir os pontos implementados e poucas utilizam um histórico de necessidades de um sistema real. Este trabalho estuda os principais casos de atualizações que ocorrem em um sistema de componentes de uso extenso, o Openbus, que consiste em uma infraestrutura de integração responsável pela comunicação de diversas aplicações de aquisição, processamento e interpretação de dados. Além deste estudo, implementamos uma solução de atualização dinâmica para acomodar as necessidades deste sistema. Depois, utilizando a solução implementada, apresentamos um teste de sobrecarga e algumas aplicações de atualizações do Openbus.

Palavras–chave

Atualização Dinâmica de Software; Atualização de Software; Componentes de Software; Arquitetura de Software.

Abstract

Câmara, Eduardo Castro Mota; Rodriguez, Noemi de La Rocque (Advisor). **A Study of Dynamic Update for Software Components**. Rio de Janeiro, 2014. 85p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The component-based development of software systems consists on composing systems from ready and reusable sotfware units. Many software component systems on production, need to be available 24 hours a day 7 days a week. Dynamic updates allow systems to be upgraded without interrupting the execution of its services, applying the update at runtime. Many dynamic software update techniques in the literature use applications specifically implemented to cover the presented points and only a few use a historical need of a real system. This work studies the main cases of updates that occur in a system of components with extensive use, the Openbus, which consists of an integration infrastructure responsible for communication of various applications for acquisition, processing and interpretation of data. In addition to this study, we implement a solution of dynamic software update to accommodate the needs of this system. After, using the implemented solution, we present an overhead test and applications of updates on Openbus.

Keywords

Dynamic Software Update; Software Update; Software Components; Software Architecture.

Sumário

Lista de figuras

Lista de tabelas

O sucesso nasce do querer, da determinação e persistência em se chegar a um objetivo. Mesmo não atingindo o alvo, quem busca e vence obstáculos, no mínimo fará coisas admiráveis.

José de Alencar

1
Introdução

Técnicas de desenvolvimento baseadas em componentes de software visam promover o desacoplamento e a modularização das funcionalidades do sistema em componentes bem definidos, de forma a permitir a reutilização desses componentes na construção de outros sistemas com funcionalidades similares. Com interfaces contratualmente especificadas, serviços bem definidos e dependências de contexto explícitas, um componente de software pode ser implantado de forma independente e normalmente é utilizado na composição de sistemas complexos com múltiplos componentes.

Apesar da evolução em engenharia de software e do auxílio de ferramentas no processo de desenvolvimento, os sistemas de componentes de software em uso ainda demandam atualizações constantes. Atualizações de software podem ocorrer de maneira estática ou dinâmica. A atualização estática é o método tradicional de atualização: a execução do software é interrompida, a aplicação da atualização é feita e depois a execução do software é reiniciada. Atualizações dinâmicas permitem que os sistemas sejam atualizados sem interromperem a execução dos seus serviços, aplicando a atualização em tempo de execução.

Muitos sistemas de componentes software em produção precisam ficar disponíveis durante 24 horas por dia nos 7 dias da semana, por diversos motivos: missão crítica, perda de receita para os negócios, inconveniência para o usuário, etc. Junto da necessidade de sempre estar disponível, alguns sistemas de componentes software estão em constante atualização seja para otimizações, corrigir bugs ou introduzir novas funcionalidades. Os sistemas que estão em constante evolução e mesmo assim precisam ficar disponíveis o tempo todo inspiram pesquisas de técnicas e mecanismos de atualização dinâmica de software (do inglês *Dynamic Software Updating*) ou atualização online (do inglês *Live Updating*).

Quando um sistema é atualizado dinamicamente, vários detalhes específicos de sua execução precisam ser abordados no processo [1]. Escolher o melhor momento para atualizar não é tarefa fácil. Uma atualização, dependendo da granularidade, pode alterar um determinado método F, porém, é difícil prever o comportamento esperado, caso F esteja em execução durante a

atualização. Alguns autores [2] sugerem a procura por um estado quiescente, um estado calmo, em que a parte que vai ser atualizada não está sendo executada. Outros autores [3] definem *safe-points* que são marcações internas onde o sistema indica em seu código fonte qual seriam os pontos seguros para se aplicar uma atualização.

A atualização nem sempre é instantânea e pode vir a demorar alguns milissegundos para ser aplicada. Durante esses milissegundos é necessário decidir se o sistema ficará inoperante ou se irá funcionar parcialmente. No caso dos sistemas de componentes, as interações entre componentes podem ser redirecionadas para outro componente, podem ser armazenadas para uma futura execução ou podem ocorrer normalmente caso não forem afetadas pela atualização. É preciso considerar se o estado do componente de software em memória é importante ou pode ser ignorado. No caso de ser importante é preciso que na aplicação da atualização o novo estado reflita as informações do estado antigo. Um outro ponto a considerar é que cada componente é desenhado para ter uma determinada interface e um certo comportamento, se as atualizações podem alterar essa interface ou esse comportamento então é preciso garantir que as interações do sistema continuarão funcionando após a atualização.

Apesar de muitas técnicas de atualização dinâmicas serem discutidas na literatura, a maioria utiliza aplicações genéricas para suas avaliações e as vezes com atualizações feitas especificamente para cobrir os pontos implementados pelos mecanismos. Ou seja, poucas dessas técnicas apresentam uma abordagem de atualização dinâmica em cima de um histórico de necessidades de atualizações de um sistema real em produção.

1.1 Objetivo e Abordagem

O principal objetivo deste trabalho é estudar os principais casos de atualizações que ocorrem em um sistema de uso extenso, o Openbus, identificar as interfaces necessárias para realização dessas atualizações de maneira dinâmica e analisar o custo/benefício destas interfaces. O OpenBus[4] é um barramento de aplicação que está em operação na Petrobras há 5 anos e é responsável pela comunicação de diversas aplicações de aquisição, processamento e interpretação de dados geofísicos e geológicos. Hoje o OpenBus é responsável por integrar 126 sistemas de uma das cinco maiores empresas de energia do mundo e quando precisa sofrer uma atualização, esta atualização é aplicada de modo estático. Durante a janela de atualização o OpenBus é desligado e os 126 sistemas ficam sem integração.

O OpenBus possui um histórico de atualizações de mais de 5 anos documentado no controlador de versão e esse histórico foi utilizado para identificar os principais tipos de modificações que ocorreram nesse sistema. Neste trabalho tratamos uma atualização como um conjunto de modificações efetuadas em um sistema.

Depois da identificação dos principais tipos de modificações que ocorreram no OpenBus, foi implementado um mecanismo que permite a atualização dinâmica de componentes. O foco do mecanismo de atualização dinâmica implementado é a atualização de componentes específicos e não trata o estado global do sistema de componentes de software, só o componente que está sendo atualizado em um devido momento.

Neste trabalho consideramos a atualização dinâmica como a alteração da implementação de componentes, dado que o suporte a reconfiguração dinâmica (conexões e reconexões em tempo de execução) já é fornecido pelo sistema de componentes utilizado pelo OpenBus.

O mecanismo foi utilizado para aplicar atualizações dinamicamente no OpenBus. Foi feito uma simulação utilizando as transições de versões que apresentam os casos mais interessantes para o estudo. Também foram feitos testes para medir a sobrecarga de processamento que esse mecanismo impõe aos componentes do sistema.

Para mensurar o potencial benefício do uso do mecanismo utilizamos o modelo de avaliação quantitativo proposto em [5], que leva em consideração a sobrecarga de processamento do mecanismo, os benefícios das atualizações e o tempo que o serviço fica indisponível no caso da atualização estática.

Também analisamos como o mecanismo de atualização dinâmica proposto neste trabalho ataca desafios comumente considerados na literatura, tais como: estado quiescente do componente, execução de chamadas externas durante atualização, ganchos para recuperação de estado e alteração da implementação. Também fizemos uma avaliação comparativa com os trabalhos relacionados e o mecanismo proposto, resumindo os pontos fortes dos trabalhos e suas possíveis aplicabilidades em cenários como os encontrados no OpenBus.

1.2 Estrutura do Documento

Este documento está organizado da seguinte maneira: o capítulo 2 apresenta os conceitos básicos inerentes a sistemas de componentes de software e sua atualização dinâmica. Detalhando a arquitetura do sistema de componentes SCS e os primeiros esforços de atualização dinâmica implementados para ele; o capítulo 3 descreve o mecanismo de atualização dinâmica implementado

para permitir a aplicação das atualizações propostas; o capítulo 4 se aprofunda nos detalhes do estudo do OpenBus, os principais tipos de atualizações e as versões utilizadas para a atualização; o capítulo 5 apresenta os experimentos feitos com o mecanismo para demonstrar sua expressividade, medir a sobrecarga de processamento e utiliza um modelo de avaliação quantitativo[5] para validar o uso da atualização dinâmica no sistema estudado; o capítulo 6 apresenta os trabalhos relacionados de atualização dinâmica e faz uma avaliação comparativa do mecanismo implementado com as abordagens apresentadas nos trabalhos relacionados, utilizando o arcabouço de avaliação gerado a partir do OpenBus; por fim, o capítulo 7 apresenta o resumo da avaliação, as considerações finais, consolida as contribuições e sugere alguns trabalhos futuros que darão continuidade ao estudo.

2
Atualização Dinâmica no Sistema de Componentes SCS

Esse capítulo apresenta os conceitos básicos inerentes a sistemas de componentes de software e sua atualização dinâmica. Primeiro nós apresentamos os principais pontos em comum dos sistemas de componentes de software, depois apresentamos as particularidades do arcabouço Sistema de Componentes de Software, o SCS[6] utilizado pelo OpenBus. Logo em seguida nós demonstramos a forte conexão dos sistemas de componentes de software e atualização dinâmica, além de introduzir os conceitos relativos a atualização dinâmica e os primeiros esforços de atualização dinâmica no arcabouço SCS.

O desenvolvimento baseado em sistemas de componentes consiste em compor sistemas a partir de unidades de software prontas e reutilizáveis. Um sistema é desenvolvido para ser um conjunto de partes menores ao invés de ser uma entidade monolítica. Essa abordagem diminui o custo de produção ao reutilizar componentes já existentes ao invés de criá-los de novo. Para essa metodologia funcionar é crucial que o mecanismo de composição seja simples e os componentes sejam de fácil reuso.

A simplicidade de composição do componente está fortemente relacionada ao modelo de componente utilizado. Um conceito importante que os modelos de componentes utilizam é a separação entre a definição do componente e sua implementação. Componentes são manipulados como caixas-pretas, ou seja, são manipulados com base exclusivamente na sua definição. A definição de um componente especifica um conjunto de conectores através dos quais é possível acessar os serviços do componente e fornecer os recursos esperados pelo componente, definidos como suas dependências. A construção de um sistema baseado em componentes é feita estabelecendo conexões entre componentes através da ligação de seus conectores, de forma que as dependências de um componente sejam supridas pelos serviços oferecidos por outro.

Considerando a necessidade de fácil reuso, o tamanho do componente, medido pela quantidade de recursos implementados, tem uma importância primordial. Em relação às funcionalidades de um componente, para que o componente possa ser amplamente reutilizável, essas funcionalidades devem ser logicamente relacionadas de forma a compor um conjunto de funcionali-

dades coeso. Já em relação às suas dependências, quanto menos dependências os componentes apresentarem mais robustos e pesados eles serão, gerando redundâncias de implementação e diminuindo a possibilidade de reutilização de outros componentes. Contudo, quanto mais funcionalidades forem delegadas a outros componentes, maiores serão suas dependências, dificultando sua utilização. Portanto, apesar de não ter uma medida certa e clara, é necessário definir adequadamente as funcionalidades de cada componente, definindo assim toda a arquitetura do sistema. Caso essa arquitetura deva ser modificada posteriormente, pode ser necessário fazer a alteração de grande parte do sistema.

Cada modelo de componentes de software define os tipos de conectores que os componentes podem fornecer. Os modelos[7][8][9][10][11] são semelhantes em suas abstrações, onde possuem dois tipos de interfaces: os serviços oferecidos e as dependências necessárias. Os modelos de componentes tipicamente disponibilizam mecanismos de manipulação, conexão e introspecção, por onde é possível acessar e conectar os componentes, além de obter descrições, em tempo de execução, das facetas disponibilizadas e das conexões realizadas.

2.1 SCS

O SCS[11] é um sistema de componentes inspirado no COM[9] e no CCM[8] e foi idealizado visando flexibilidade, simplicidade e facilidade de uso. Ele possui dois tipos de portas de serviços para os componentes: Facetas e Receptáculos. As facetas são portas de provisão de serviços. Receptáculos são portas através das quais um componente requisita um serviço que ele utiliza. Tanto as facetas quanto os receptáculos possuem uma determinada interface definida em IDL(*Interface Definition Language*). Os receptáculos podem ser de dois tipos: simples ou múltiplos. Os receptáculos simples só permitem a conexão de uma faceta, já os receptáculos múltiplos permitem a conexão de uma ou mais facetas. Um componente SCS pode ter múltiplas facetas e múltiplos receptáculos, porém o modelo pré-define as três facetas seguintes (Figura 2.1):

IComponent : Essa é a faceta que representa o "componente"e possui operações para ativação e desativação do mesmo. Essa faceta também contém operações para requisições de facetas. Esta faceta é obrigatória e todos os componentes possuem uma implementação para ela.

IReceptacles : Essa faceta contém operações para listar todas as conexões dos receptáculos e operações para conectar e desconectar facetas aos

receptáculos. Esta faceta não é obrigatória (por exemplo: no caso do componente não possuir dependências externas).

IMetaInterface : Essa faceta possui operações básicas para introspeção de facetas e receptáculos do componente. Esta faceta não é obrigatória.

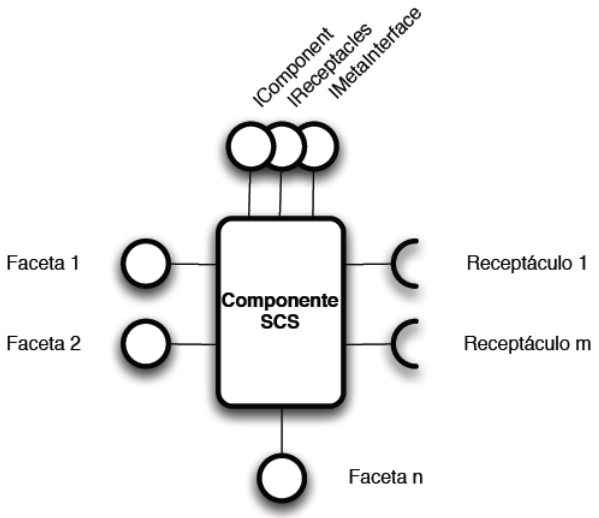

Figura 2.1: Representação de um componente SCS.

O SCS é baseado em CORBA[8] e possui implementações em C/C++, Lua e Java. As facetas tem suas interfaces definidas em IDL CORBA e são objetos remotos. Apesar do modelo SCS disponibilizar implementações base para as facetas IComponent, IReceptacles e IMetaInterface, elas são extensíveis e podem ser reimplementadas de acordo com a necessidade do componente. Essas facetas representam o comportamento básico de um componente.

A definição de um componente consiste no nome do componente e sua versão, seguido da lista de facetas e receptáculos. Cada faceta possui um nome, uma interface e uma implementação. Por sua vez cada receptáculo possui uma interface, um nome e uma cardinalidade (simples ou múltiplo).

2.1.1 Exemplo de uso SCS

A Figura 2.2 mostra a arquitetura de uma aplicação Lua que exemplifica o uso do modelo de componentes SCS. A ideia é simular comandos recebidos no controle remoto e repassados para uma televisão. Nessa aplicação existem dois componentes: TV e RemoteController. O Componente TV possui, além das três facetas básicas, uma faceta chamada Control do tipo IControl. O RemoteController possui um receptáculo, IControlRec, que aceita conexões de facetas do tipo IControl. Ele também possui, além das três facetas básicas,

uma faceta chamada RemoteInput do tipo IRemoteInput. Os dois componentes representam um controle remoto e uma televisão. O Controle remoto precisa de uma televisão para controlar, essa televisão por sua vez possui diversos controles pré-estabelecidos.

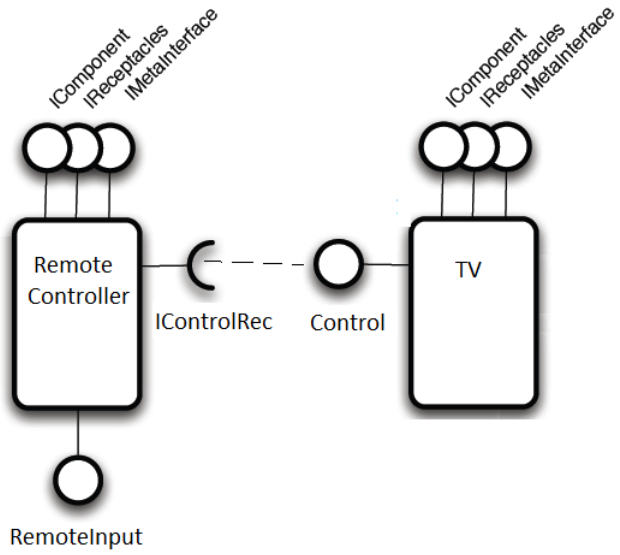

Figura 2.2: Exemplo com 1 conexão.

Abaixo seguem as interfaces em IDL das duas facetas: Control e RemoteInput.

```
1 interface IControl{
2   void volumeUp();
3   void volumeDown();
4   void changeChannelUp();
5   void changeChannelDown();
6   void changeChannel(in long channel);
7   void power();
8 }
```

Código 2.1: IDL da faceta Control.

```
1 interface IRemoteInput{
2   void buttonPress(in string button);
3 }
```

Código 2.2: IDL da faceta RemoteInput.

Depois de demonstradas as interfaces, criamos o componente TV:

```
1 ...
2 --create TV component
3 local TVcomponentId = { name = "TV", major_version = 1, minor_version = 0, patch_version = 0,
      platform_spec = "" }
4 local TVinstance = ComponentContext(orb, TVcomponentId)
5 --add the Control facet passing Name, Interface identifier and Implementation
6 TVinstance:addFacet("Control", "IDL:IControl:1.0", Control())
7
```

```
8  ...
```

Código 2.3: Criação do componente TV

Assim que o componente TV está pronto, criamos o componente RemoteController e o conectamos ao componente TV

```
1  ...
2  --create RemoteController component
3  local RemoteComponentId = { name = "RemoteController", major_version = 1, minor_version = 0,
       patch_version = 0, platform_spec = "" }
4  local RemoteInstance = ComponentContext(orb, RemoteComponentId)
5  --add the RemoteInput facet passing Name, Interface identifier and Implementation
6  RemoteInstance:addFacet("RemoteInput", "IDL:IRemoteInput:1.0", RemoteInput())
7  --add the IControlRec receptacle passing Name, Interface identifier and a boolean that represents if it
       accepts multiple connections
8  RemoteInstance:addReceptacle("IControlRec", "IDL:IControl:1.0", false)
9
10 ...
11 --create a proxy for the TV component using the reference
12 local TVIComponent = orb:newproxy(ior, "synchronous", "IDL:scs/core/IComponent:1.0")
13 --get the reference for the RemoteController component's IReceptacle facet
14 local RemoteIReceptacles = RemoteInstance.IComponent:getFacetByName("IReceptacles")
15 RemoteIReceptacles = orb:narrow(RemoteIReceptacles)
16 --get the reference for the TV component's Control facet
17 local TVControl = TVIComponent:getFacetByName("Control")
18 TVControl = orb:narrow(TVControl)
19 --connect TV's Control facet on RemoteController's Receptacle
20 RemoteIReceptacles:connect("IControlRec", TVControl)
21
22 ...
```

Código 2.4: Criação do componente RemoteController e conexão com o componente TV

E por último, utilizamos a faceta RemoteInput, do componente RemoteController, para enviar alguns comandos para o componente TV:

```
1  ...
2  --get IComponent reference
3  local remoteIComponent = orb:newproxy(ior, "synchronous", "IDL:scs/core/IComponent:1.0")
4
5  --get facet reference
6  local remote = remoteIComponent:getFacetByName("RemoteInput")
7  remote = orb:narrow(remote)
8
9  --use
10 remote:buttonPress("+")
11
12 ...
```

Código 2.5: Utilização da faceta do componente RemoteController

2.2 Atualização Dinâmica de Componentes

Sistemas de componentes de software são desenvolvidos para serem um conjunto de partes menores, mais especificamente, uma composição de componentes pré-existentes e reutilizáveis conectados entre si com pontos de entrada e saída bem definidos. Por serem compostos de partes acopláveis e desacopláveis, serem unidades de composição, que podem ser substituídas, removidas e adicionadas dependendo da funcionalidade desejada, os sistemas de componentes de software possuem uma arquitetura ideal para a atualização dinâmica[12]. Em sistemas baseados em componentes, essa forma de atualização pode ser feita através da substituição de componentes apenas reconectando-os de forma a alterar suas interações ou alterando a implementação de componentes em execução.

Como tipicamente cada componente encapsula a implementação de uma determinada parte do sistema, a substituição de um componente específico permite alterar o comportamento desta parte do sistema. A substituição de componentes é o mecanismo de atualização mais fundamental, pois efetivamente é capaz de mudar o comportamento do sistema, através da alteração na forma como o sistema é implementado. O componente substituído pode corrigir uma versão anterior, assim como apresentar uma implementação mais eficiente. Entretanto, a substituição de componentes ainda é dependente da arquitetura do sistema. É necessário que o sistema seja projetado adequadamente, ou seja, é necessário que as funcionalidades sejam adequadamente separadas em componentes distintos e que as interdependências sejam bem definidas de forma que essa estrutura não se altere com o tempo. A substituição de um componente por outro, que forneça serviços adicionais ou semanticamente diferentes, ou inclusive apresente dependências divergentes do componente anterior, resulta em alterações em outras partes do sistema, através de outras substituições ou inclusão de novos componentes.

O framework SCS, utilizado neste trabalho, já apresenta suporte a desconexões e conexões que permitem a atualização através da substituição de componentes. Neste trabalho apresentamos um estudo sobre atualizações dinâmicas de componentes, mais especificamente sobre atualizações dinâmicas que alteram a implementação de um determinado componente. O foco deste estudo é a atualização individual de um componente e como ele se comporta durante sua atualização, sendo assim, supomos que o estado global do sistema continuará válido após a atualização deste componente.

Apesar da arquitetura baseada em componentes ser adequada para atualização dinâmica, ainda existem diversos desafios [1] que muitas vezes não

são abordados pelos modelos de componentes de software.

2.2.1 Desafios

Taylor et al. [1] explicita os desafios mais comuns em atualização dinâmica:

1 - Continuidade do serviço : Resolver qual será o comportamento enquanto o componente está sendo atualizado. Enquanto a atualização está ocorrendo, o componente será substituído temporariamente por um outro? Ele irá continuar operando mas com um nível reduzido? O serviço pode interromper sua atividade? Erros podem ser tolerados enquanto determinado serviço está sendo atualizado? Para manter o componente funcionando durante a atualização existem diversas estratégias que podem ser adotadas dependendo do contexto. Um componente auxiliar pode ser colocado no lugar do componente que está sendo atualizado. Uma outra técnica seria somente interromper a funcionalidade que está sendo afetada pela atualização, porém as funcionalidade que não estão sendo atualizadas e dependem de uma outra que está sendo atualizada também precisam ser interrompidas senão podem gerar inconsistências.

2 - Melhor momento de atualização : Identificar as oportunidades que o componente tem para ser alterado. Kramer et at.[2] sugere a busca por um estado quiescente, um estado em que o componente de software está consistente e se encontra congelado e sem estímulos externos. Esse estado de repouso muitas vezes não é facilmente alcançável e outras técnicas são necessárias para forçar a atualização. Neamtiu et al. [13] estabelece marcações que o desenvolvedor do componente pode para definir no código fonte. Essas marcações representam pontos seguros (safe-points), onde o sistema indica que pode executar a atualização. Sempre que o sistema passa por um ponto seguro e existe uma atualização para ser executada, ele verifica a existência da atualização e atualiza. Escolher o melhor momento para atualizar, assim como decidir como será a continuidade do serviço, pode depender muito do contexto do componente. Essa decisão pode tanto ser tomada pelo componente, como o exemplo de pontos seguros, ou pelo sistema como um todo. A atualização de um componente, em alguns casos, pode ser autorizada somente se todas suas chamadas foram respondidas e ele não está interagindo com nenhum outro componente.

3 - Restauração ou transferência de estado : Restaurar e manter o estado do componente ou descartá-lo. O estado de um componente, na sua forma mais básica, são os valores que as variáveis do componente possuem em um certo instante de execução. Se o componente é atualizado, todo o estado do componente precisa ser mantido? Os primeiros cálculos após a atualização do componente precisam ser feitos utilizando os valores que ele tinha antes de atualizar? Nesse caso o estado do componente precisa ser mantido. Em outro exemplo, muitos dos filtros do Unix não precisam manter o seu estado já que a resposta de sua execução é só em função da entrada.

4 - Alteração de interface ou comportamento : Quebrar a compatibilidade ou não permitir a alteração da interface. Alterações de interface são inevitáveis quando há alteração de funcionalidade. Alterar a interface de um componente, no entanto, requer alterações em todos os componentes que utilizam o componente alterado. Muitos usuários podem não estar interessados na funcionalidade extra incluída na nova interface. Algumas técnicas para mitigar esse comportamento foram criadas por essa motivação. Uma técnica popular é a criação de adaptadores, esses componentes de fachada recebem a requisição com a interface antiga, fazem os ajustes necessários e repassam a chamada para a versão mais novas da interface. A inclusão de um novo componente na cadeia de chamada pode trazer danos ao desempenho. No caso de atualizar a interface uma segunda vez, ou terceira, sempre serão adicionados novos componentes de fachada? Essa e outras técnicas podem mitigar o problema mas com certeza não resolvem por completo.

Além dos desafios apresentados acima, o desenvolvimento de técnicas de atualização dinâmica é norteado por quatro objetivos principais: flexibilidade, robustez, facilidade de uso e baixa sobrecarga[3].

1 - Flexibilidade : é o critério mais importante na área de atualização dinâmica[14]: quanto menos flexível o sistema, mais provável que algum tipo de atualização não seja possível. Por outro lado, uma flexibilidade muito grande significa menos robustez em termos de implementação, mais complexidade na hora da atualização e provavelmente menos segurança. Para um sistema de atualização dinâmica ser flexível é necessário que ele suporte atualizações arbitrárias.

2 - Robustez : Sem nenhuma garantia se a atualização irá quebrar ou não o sistema em execução a atualização acaba não tendo muita utilidade,

mesmo que tenha muita flexibilidade. Os sistemas que necessitam de atualizações dinâmicas são sistemas que não podem parar para atualizar convencionalmente. Logo é necessário que a atualização não corrompa o sistema em execução. A robustez é dividida em cinco partes: segurança (safety), completude (completeness), pontualidade(well-timedness), simplicidade (simplicity) e desistência (rollback-enabled). A segurança é garantia de que o sistema não vai parar por alguma inconsistência. A completude é quando o programador garante que a atualização trata todas as mudanças que ela vai ocasionar. Pontualidade é a garantia de que a atualização não vai executar indefinidamente, que ela tem um ponto de parada. A simplicidade é o grau de complexidade que a técnica ou mecanismo causa na atualização, se a atualização for de simples implementação provavelmente não será tão flexível. A desistência existe para prevenir erros antes que eles ocorram, ela é utilizada para reverter um dano que uma atualização errada pode causar.

3 - Facilidade de uso : Poucos sistemas focam em usabilidade ao invés de flexibilidade e robustez. Como resultado muitos sistemas não separam as implementações relacionadas a atualização dinâmica e ao sistema em si. Isso causa um desenvolvimento menos modular e de manutenção mais difícil. Em geral as atualizações precisam ficar separadas da implementação do sistema ou as atualizações precisam ser geradas automaticamente a partir da diferença da nova com a velha implementação.

4 - Baixa sobrecarga : Um sistema é eficiente quando não possui sobrecarga durante a execução ou possui uma sobrecarga desprezível. Alguns sistemas só apresentam sobrecarga durante a atualização, para prover mecanismos de atualização mais flexíveis.

Para maximizar os benefícios da atualização dinâmica é preciso balancear a flexibilidade, robustez, facilidade de uso e a baixa sobrecarga. Encontrar o equilíbrio perfeito entre esses requisitos é uma tarefa árdua, se não for impossível.

Diversos trabalhos têm sido desenvolvidos no sentido de definir mecanismos e abstrações que ofereçam um melhor suporte a atualizações dinâmicas de aplicações, aliando técnicas adequadas de componentização, que possibilitam gerenciar a complexidade das alterações, a mecanismos de atualização dinâmica que permitem efetuar essas alterações sem interromper a execução dos serviços.

Um dos primeiros esforços nessa direção foi o desenvolvimento do sistema LuaOrb[15]. LuaOrb é uma infraestrutura de desenvolvimento de software

cujo núcleo é composto por *bindings* entre a linguagem interpretada Lua[16] e diferentes sistemas como CORBA[8], COM, Java e .NET. Este projeto permite realizar em tempo de execução tarefas como: conexão, adaptação, implementação e verificação de novos tipos de componentes. Baseado no LuaOrb, alguns mecanismos de mais alto nível para dar suporte a atualização dinâmica foram investigados, tais como: LuaDSI, um sistema que permite a atualização dinâmica de servidores CORBA[17]; *smart proxies*[18], um mecanismo que permite a reconfiguração do sistema de acordo com alterações identificadas por monitoramento. A seguir apresentamos mais alguns trabalhos nessa direção.

2.2.2 Atualização Dinâmica com o LOAF

O LuaOrb Adaptation Framework (LOAF)[19], junto com a implementação em Lua de parte do Modelo de Componente CORBA[8] (CCM - Corba Component Model) denominado LuaCCM, é um dos trabalhos desenvolvidos para dar suporte a atualização dinâmica de aplicações. Através do LOAF é possível alterar definições e implementações de componentes de software e reconfigurar e criar dinamicamente novas interações entre eles.

O LOAF[19] utiliza duas abstrações para representar as adaptações feitas em componentes de software: papel e protocolo. O papel representa uma atualização de um componente específico, já o protocolo é utilizado para aplicar novos papéis a um ou mais componentes do sistema. Alterar o papel de um componente pode ser criar novas portas para o componente ou alterar implementações de portas existentes. Utilizando a interface de adaptação do LuaCCM é possível aplicar papéis em componentes e estabelecer novas conexões entre eles através de scripts Lua, estes são capazes de representar naturalmente a abstração de protocolos.

Apesar da flexibilidade do LOAF, um dos assuntos que ele não abordou foi o tratamento de chamadas ao componente enquanto ele está sendo atualizado. Não há atomicidade nas alterações feitas por um papel, ou seja, entre a adição de duas portas num componente é possível que este receba alguma requisição. Isso pode gerar problemas, pois se os contextos de duas portas são dependentes, uma das portas novas pode receber uma requisição antes que a outra seja adiciona ou atualizada no componente.

Para avaliar o LOAF, alguns experimentos em diferentes domínios de aplicação foram desenvolvidos, como sistemas de gerenciamento de redes, CAD colaborativo, visualização distribuída, computação ubíqua e computação em grade. Mas vale ressaltar que uma questão importante relacionada ao LOAF é

que o modelo de componentes adotado, o CCM, em busca de ser um modelo de componentes satisfatoriamente completo acaba sendo um modelo bastante complexo e o LOAF, por sua vez, acaba herdando esta complexidade.

2.2.3 Atualização Dinâmica no SCS

Baseado no LOAF, Eduardo Portilho[20] sugeriu e implementou a primeira interface de atualização dinâmica, IAdaptable, para o sistema de componentes SCS[6] (Software Component System) em Java. Assim como o LOAF, as atualizações implementadas permitem efetuar alterações no conjunto de portas dos componentes e em sua implementação. Como o modelo de componentes SCS já permite a atualização de um sistema através de reconfiguração, o foco desse trabalho foi a alteração da implementação das facetas dos componentes. Segundo o autor, como foi desenvolvido em Java, é possível alterar a implementação de três maneiras: envio de byte-code Java, envio de código Java e envio de código Lua. Na primeira opção o desenvolvedor fornece um arquivo JAR (Java Archive) contendo as classes Java compiladas que serão usadas para substituir as implementações das facetas. Na segunda opção o desenvolvedor fornece os códigos Java que serão compilados e posteriormente serão utilizados para as novas implementações das facetas. Na terceira opção é enviado código Lua para a implementação do objeto Java e utilizando o *binding* LuaJava[21] a implementação em Java da faceta é substituída pela implementação em Lua. Apesar de permitir a atualização da implementação das facetas, não é possível alterar a interface de uma faceta. As funcionalidades presentes, além da atualização da implementação de facetas, incluem: remoção de facetas e inclusão novas facetas com novas interfaces.

O objetivo do Eduardo Portilho era comparar sua solução com o LOAF e analisar a flexibilidade dos mecanismos propostos considerando as linguagens de programação adotadas. Ele também analisou o desempenho da solução, incluindo na comparação aplicações desenvolvidas sem os mecanismos de adaptação, para mensurar a sobrecarga imposta por eles. Apesar de utilizar um modelo de componentes mais conciso do que o CCM, essa implementação em JAVA de atualização dinâmica para o SCS sofre com a rigidez do mecanismo de verificação de tipos da linguagem, dificultando bastante o processo de adaptação e prejudicando o uso da solução.

2.2.4 Abordagem

Neste trabalho, com o mecanismo de atualização dinâmica para o modelo de componentes SCS, abordamos soluções para os seguintes desafios apresentados nesta seção: Continuidade do serviço; Melhor momento de atualização; e Restauração ou transferência de estado. A alteração de interface, apesar de não ser possível alterar diretamente a interface de um componente, pode ser alcançada utilizando adaptadores. Nos objetivos do mecanismo apresentado incluem-se a flexibilidade da atualização a nível de faceta e a baixa sobregarga. No escopo deste trabalho nós não disponibilizamos nenhuma ferramenta para facilitar a geração automática de atualização e assumimos que a robustez da atualização foi testada préviamente pelo desenvolvedor.

3
Mecanismo de Atualição Dinâmica para o SCS

Esse capítulo apresenta o mecanismo de atualização dinâmica implementado para permitir a aplicação das atualizações propostas. Primeiro nós apresentamos a API do mecanismo e em seguida exemplificamos seu uso com a atualização de um componente. O mecanismo é composto de facetas que foram introduzidas no SCS com funcionalidades que permitem o controle do ciclo de vida e a atualização de componentes em tempo de execução.

O mecanismo foi escrito em Lua mas a concepção de suas interfaces foi feita de modo a possibilitar a implementação delas em outras linguagens. As principais funcionalidades oferecidas são as seguintes:

1) Incluir novas facetas no componente.

2) Remover facetas do componente.

3) Incluir novos receptáculos no componente.

4) Remover receptáculos do componente.

5) Alterar implementações das facetas do componente.

6) Controlar as requisições feitas ao componente enquanto a atualização está ocorrendo.

3.1
A Interface de Atualização Dinâmica Revisada

Para iniciar as experimentações de alterações dos componentes, introduzimos uma nova faceta, IBackdoor. Esta faceta, com uma implementação muito simples, permitiu o envio de código para ser executado no componente, facilitando a inspeção e possibilitando a alteração do componente em tempo de execução. Ao identificar as possibilidades de alterações, foram criadas funções mais específicas e esse código foi então migrado para uma faceta com uma interface mais especializada para atualização dinâmica.

```
1  interface IBackdoor{
2      any Backdoor(in Code patch);
3  }
```

Código 3.1: IDL da faceta inicial de alterações

```
local oo = require "loop.base"
local oil = require "oil"

--implementacao da faceta IBackdoor
local Backdoor = oo.class{}

function Backdoor:__init()
  return oo.rawnew(self, {})
end
--string Backdoor (in Code source);
function Backdoor:Backdoor(str)
  local f,e = loadstring(str)
  if not f then
    return tostring(e)
  else
    setfenv(f, setmetatable({ self = self.context}, { __index = _G }))
    local _,ret = pcall(f)
    return ret
  end
end

return Backdoor
```

Código 3.2: Implementação da faceta inicial de alterações

O segundo passo foi criar um controle de ciclo de vida do componente, podendo controlar o comportamento do componente no momento da atualização. Para isso foi criada uma nova faceta, ILyfeCycle, com a seguinte IDL:

```
module lifecycle{

    enum State {
        RESUMED,
        HALTED,
        SUSPENDED
    };

    exception CannotChangeState {string msg;}

    interface ILifeCycle{
        State getState();

        boolean changeState(in State state) raises (CannotChangeState);
    }
}
```

Código 3.3: IDL da faceta de ciclo de vida.

Esta nova faceta visa prover um controle para as requisições feitas no momento da atualização. Ela possui duas funções: uma para resgatar o estado em que se encontra o componente e outra para alterar o estado atual do componente. Os estados definidos por esta interface têm os seguintes comportamentos:

RESUMED: O componente está rodando normalmente e todas as requisições são processadas e respondidas.

SUSPENDED: O componente está suspenso e todas as requisições feitas a outras facetas são enfileiradas e serão executadas assim que o componente voltar para o estado de RESUMED.

HALTED: O componente está parado e todas as requisições feitas a outras facetas são descartadas.

Esse controle de ciclo de vida do componente permite que a atualização do componente seja atômica e evita que requisições sejam processadas no momento da atualização. A implementação dessa faceta utiliza o recurso de interceptação de chamadas às facetas dos componentes. Para cada chamada interceptada, o sistema verifica se o componente está no estado RESUMED e, caso esteja, a chamada é feita normalmente. Caso ocorra quando o componente se encontre no estado HALTED, essa chamada é descartada. No caso de uma chamada acontecer no momento em que o componente se encontra no estado SUSPENDED, a chamada é enfileirada e executada assim que o componente for para o estado RESUMED.

Após a experimentação preliminar com a faceta IBackdoor, as funcionalidades foram agrupadas na faceta IDynamicUpdatable com a seguinte IDL:

```
1  struct ComponentId {
2          string name; /* O nome identificador do componente. */
3          octet major_version; /* O numero principal da versao. */
4          octet minor_version; /* O numero secundario da versao. */
5          octet patch_version; /* O numero de revisao da versao. */
6          string platform_spec; /* A especificacao da plataforma necessaria para o funcionamento do
               componente. */
7        };
8
9        typedef sequence<octet> Code;
10
11       struct NewFacetDescription {
12         string name; /* O nome identificador da faceta*/
13         string interface_name; /* O nome identificador da interace da faceta */
14         Code facet_implementation; /* A implementacao da faceta */
15         Code facet_idl; /* A idl da faceta*/
16       };
17
18       struct FacetUpdateDescription{
19         NewFacetDescription description; /* Descricao e implementacao da nova faceta */
20         Code patchUpCode;/* Codigo de aplicacao da atualizacao */
21         Code patchDownCode;/* Codigo de rollback da atualizacao */
22         string key;/*Uma string para ser registrada como sendo a chave do objeto no ORB */
23       };
24
25       exception RawState { string msg; };
26       exception CannotFinishIfNotStarted { string msg; };
27
```

```
28  interface IDynamicUpdatable{
29    string GetUpdateState();
30    boolean ChangeUpdateState (in string state);
31    string StartUpdate () raises (CannotChangeState);
32    void FinishUpdate () raises (CannotFinishIfNotStarted);
33
34    string InsertFacet (in string updateKey, in FacetUpdateDescription facet);
35    string InsertFacetAsync (in FacetUpdateDescription facet);
36
37    FacetUpdateDescription RetrieveFacet (in string updateKey,in string facetName) raises(RawState);
38
39    string UpdateFacet (in string updateKey,in FacetUpdateDescription facet);
40    string UpdateFacetAsync (in FacetUpdateDescription facet);
41
42    string DeleteFacet (in string updateKey,in string facetName);
43    string DeleteFacetAsync (in string facetName);
44
45    string InsertReceptacle (in string updateKey, in string name, in string interface_name);
46    string InsertReceptacleAsync (in string name, in string interface_name);
47
48    string DeleteReceptacle (in string updateKey, in string name);
49    string DeleteReceptacleAsync (in string name);
50
51    string UpdateComponent(in string updateKey,in ComponentId newId,
52      in FacetUpdateDescriptions facets);
53    string UpdateComponentAsync(in ComponentId newId,
54      in FacetUpdateDescriptions facets);
55
56    string GetAsyncRet(in string key);
57
58    boolean RollbackFacet(in string updateKey, in string facetName);
59  };
```

Código 3.4: IDL da faceta de Atualização Dinâmica.

Essa faceta provê métodos para a atualização dinâmica de um componente, possibilitando a atualização de uma única faceta ou do componente inteiro. Ela possibilita também a remoção e adição de novas facetas. Como algumas atualizações podem demorar para serem efetuadas, foram criadas funções tanto para atualização síncrona, quanto para atualização assíncrona. A semântica das funções e estruturas definidas pela interface são as seguintes:

NewFacetDescription: Essa estrutura representa a implementação da faceta e contém: o nome da faceta (string name), o identificador da interface da faceta (string interface_name), o código da implementação da faceta (Code facet_implementation) e código da idl da faceta (Code facet_idl).

FacetUpdateDescription: Essa estrutura descreve uma atualização da faceta e contém: a implementação da faceta (NewFacetDescription description), o código que será executado no momento da atualização (Code patchUpCode), o código que será executado se a função RollbackFacet

for chamada (Code patchDownCode) e a chave do objeto corba (string key).

GetUpdateState: Essa função retorna o estado em que o componente irá entrar, caso uma atualização ocorra com ele (HALTED ou SUSPENDED).

ChangeUpdateState: Essa função permite trocar o estado em que o componente permanece no momento da atualização (HALTED ou SUSPENDED).

StartUpdate: Essa função permite iniciar as atualizações, ela tenta alterar o estado do componente para o estado de atualização e retorna uma chave que vai ser utilizada para acessar as funcionalidades de alteração do componente.

FinishUpdate: Essa função possibilita a finalização da atualização do componente, retornando ele para o estado RESUMED.

InsertFacet: Para essa função é fornecido a descrição da nova faceta. Primeiro é verificado se uma outra faceta com o mesmo nome já existe, depois é verificado se a implementação é compatível com a interface fornecida. Após as devidas verificações, a nova faceta é ativada. Depois é executado o código de atualização que pode executar qualquer atividade relevante para a faceta.

RetrieveFacet: Para essa função é fornecido o nome da faceta. Primeiro é verificado se a faceta com aquele nome existe, depois verifica se ela já foi atualizada alguma vez, se foi retorna o objeto de descrição da atualização da faceta. No caso da faceta não possuir atualizações uma exceção é lançada.

UpdateFacet: Para essa função é fornecido a descrição da atualização da faceta. Primeiro é utilizado o nome para recuperar a faceta do componente, depois é verificado se a nova implementação bate com a interface fornecida. Após as devidas verificações, a antiga implementação da faceta é desativada e a nova implementação é ativada. Em seguida é executado o código de atualização que, com acesso tanto a implementação nova quanto a implementação antiga, pode recuperar qualquer informação relevante do estado da faceta.

DeleteFacet: Para essa função é fornecido o nome da faceta. Primeiro é verificado se a faceta com o nome existe. Se a faceta existe, ela é desativada.

InsertReceptacle: Para essa função é fornecido o nome do novo receptáculo e sua interface. Primeiro é verificado se um outro receptáculo com o mesmo nome já existe e após essa verificação, o novo receptáculo é ativado.

DeleteReceptacle: Para essa função é fornecido o nome do receptáculo. Primeiro é verificado se o receptáculo com o nome existe. Se o receptáculo existe, ele é desativado.

UpdateComponent: Para essa função é fornecido: um novo identificador para o componente (contendo o número da nova versão) e as descrições das facetas a serem atualizadas. Para cada descrição de faceta é feito uma atualização na faceta conforme a função UpdateFacet.

Async: Todas as funções de atualização possuem uma variante assíncrona, ela permite invocar a atualização e a atualização será executada quando for possível em um momento oportuno para o componente.

GetAsyncRet: Essa função recebe uma chave previamente gerada pela chamada assíncrona e retorna o resultado caso ela já tenha ocorrido.

RollbackFacet: Essa função recebe o nome da faceta. Primeiro é verificado se uma faceta com o nome recebido realmente existe, depois é verificado se a faceta já foi atualizada alguma vez. Caso alguma atualização já tenha ocorrido na faceta, a faceta atual é desativada e a implementação antiga então é ativada. Em seguida é executado o código de *rollback* da atualização, que possui acesso tanto a implementação atual, quanto a implementação anterior da faceta, podendo recuperar qualquer informação relevante para a faceta. Por fim o componente é retirado do estado de atualização.

O Componente, agora, possui seis facetas básicas para controle como demonstrado na Figura 3.1:

3.2 Exemplo de uso do mecanismo

Para exemplificar o uso mais comum da atualização de uma faceta utilizando o mecanismo criado, utilizei a aplicação SCS descrita na subseção 2.1.1. Como mostrado na figura 3.4, o componente agora também possui as três facetas ILyfeCycle, IBackdoor e IDynamicUpdatable. Primeiro criamos a nova implementação "Controlv2.lua"da faceta Control, depois utilizamos a faceta IDynamicUpdatable para aplicar a atualização.

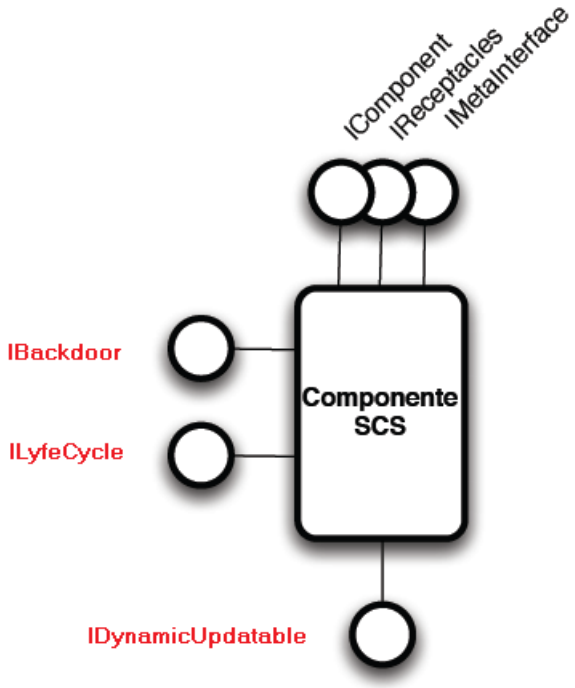

Figura 3.1: Nova estrutura básica do componente.

O código 3.5 apresentado abaixo altera a implementação da faceta Control do componente TV. Após obter acesso ao componente TV, ele recupera a referência para a faceta IDynamicUpdatable e logo após utiliza o método UpdateFacet para atualizar a faceta Control com a nova implementação "Controlv2.lua". Durante a atualização, o componente, por padrão, enfileira todas as chamadas que ele recebe para executar posteriormente. Após a atualização não é necessário refazer a conexão e o componente processa todas as chamadas enfileiradas. A figura 3.3 apresenta o diagrama de sequência das chamadas utilizadas na atualização da faceta.

```
1 ...
2     --update the facet Control with the new implementation "Controlv2.lua"
3     local ret = IUpdateFacet:UpdateFacet(updateKey,
4           {description={name="Control",
5             interface_name="IDL:IControl:1.0",
6             facet_idl=oil.readfrom("idl/IControl.idl"),
7             facet_implementation=oil.readfrom("Controlv2.lua")},
8           patchUpCode="_newFacet.channel_=__oldFacet.channel;_newFacet.volume_=__oldFacet.volume"
                ,patchDownCode="_newFacet.channel_=__oldFacet.channel;_newFacet.volume_=__oldFacet
                .volume",key="Control"})
9       ...
```

Código 3.5: Atualização da faceta Control do componente TV

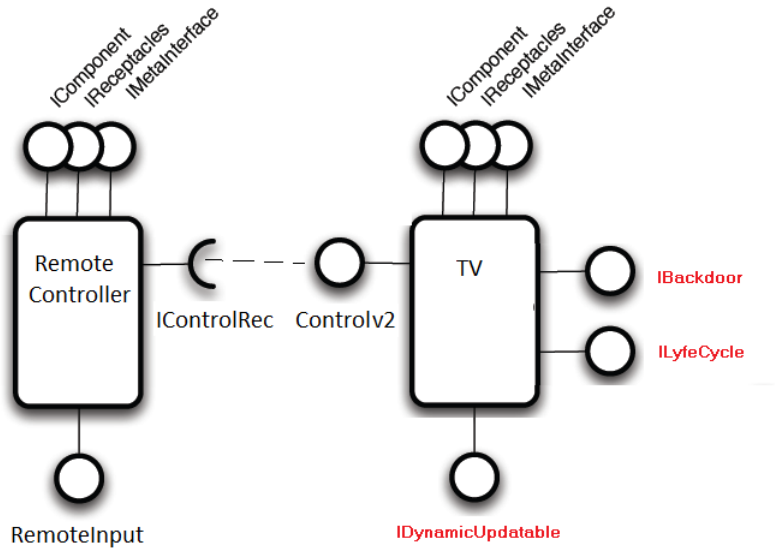

Figura 3.2: Atualização utilizando a IDynamicUpdatable.

3.3 Análise do Mecanismo Implementado

O mecanismo proposto é apresentado como um conjunto de interfaces que dão suporte a atualização dinâmica de componentes com uma implementação base para referência. A principal estratégia é possibilitar que o desenvolvedor da aplicação possa reimplementar estas interfaces e decidir qual a política de atualização desejada. Dessa maneira são disponibilizados ganchos para que os desafios apresentados na seção 2.2 sejam superados.

A continuidade do serviço pode ser definida na implementação da faceta ILyfeCycle sendo que a implementação padrão é enfileirar as chamadas para serem executadas após a atualização.

A escolha do melhor momento para atualizar é feita também na implementação da faceta ILyfeCyle e o comportamento padrão é verificar se existem requisições em andamento, se não existirem, o componente inicia a atualização.

A restauração ou transferência de estado pode ser feita utilizando um gancho no momento da atualização. Este gancho permite a execução de códigos implementados pelo desenvolvedor da atualização que tem acesso a faceta antiga e a faceta nova, podendo assim, recuperar as informações necessárias.

O mecanismo desenvolvido não permite a alteração da interface de métodos. A fim de evitar a quebra da comunicação com códigos que utilizem aquela faceta, uma decisão de projeto foi impossibilitar a alteração da interface da faceta. No caso de ser necessário alterar a interface de uma faceta, também será necessário alterar os clientes que utilizam aquela faceta. Como os clientes deverão ser alterados, eles podem ser alterados para utilizar uma nova faceta

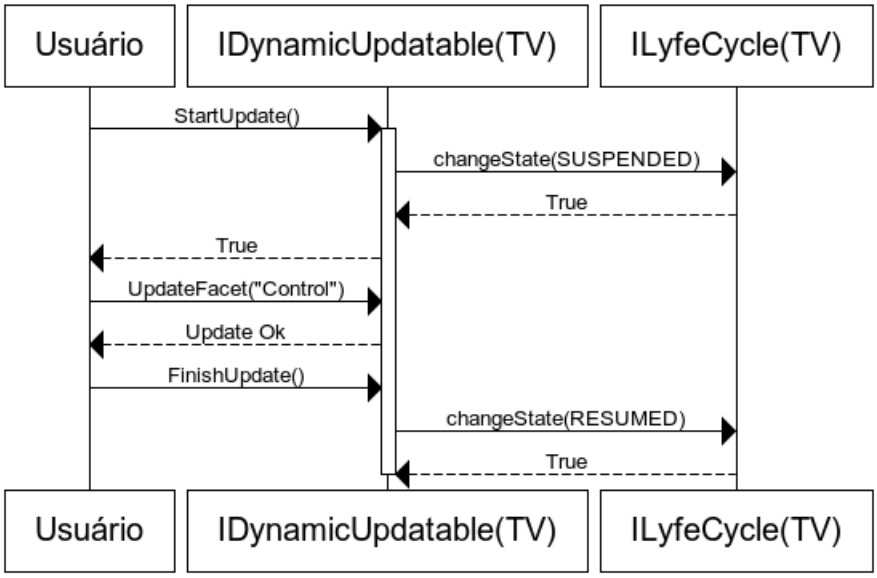

Figura 3.3: Diagrama de sequência das interações.

com uma nova interface, ao invés de alterar a interface da faceta já existente.

O mecanismo permite a atualização a nível de faceta. Possibilitando inclusive a atualização da faceta IDynamicUpdatable, que contém a funcionalidade de atualização dinâmica.

Os testes para medir a sobrecarga foram apresentados na seção 5.2 e apesar de nós não disponibilizamos nenhuma ferramenta para facilitar a geração automática de atualizações ou até mesmo aplicar atualizações automaticamente, no futuro poderíamos construir uma outra camada com estas funcionalidades e que utiliza o mecanismo apresentado como base.

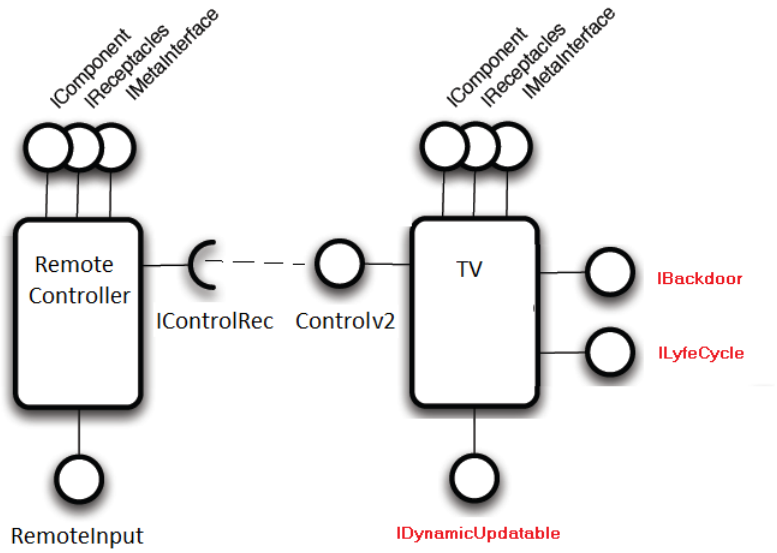

Figura 3.4: Atualização utilizando a IDynamicUpdatable.

4
O Caso de Estudo OpenBus

Nesse capítulo estudamos o OpenBus[4] e suas atualizações. Primeiramente nós apresentamos a arquitetura do OpenBus e a diferença entre as versões 1.4, 1.5 e 2.0. Logo após nós apresentamos também as atualizações que o OpenBus sofreu ao longo do tempo, assim como os detalhes dessas atualizações. Por último nós apresentamos uma análise dessas atualizações em que nós identificamos os principais tipos de modificações que ocorrem em um sistema de componentes de software em produção.

4.1
OpenBus

O OpenBus é um sistema desenvolvido pelo Instituto Tecgraf de Desenvolvimento de Software Técnico-Científico da PUC-Rio (Tecgraf/PUC-Rio) para possibilitar a integração de aplicações através de uma arquitetura orientada a serviços. O OpenBus representa um barramento de serviços ao qual as aplicações podem se conectar e consumir ou oferecer seus serviços para outras aplicações. Ele também provê suporte para registro e descoberta de serviços. Como o barramento é o meio no qual toda interação e comunicação entre as aplicações é feita, o OpenBus realiza controle de acesso, que basicamente consiste na autenticação de todo acesso a sistemas através do barramento, permitindo assim identificar de forma segura a origem de toda comunicação feita entre as aplicações.

Por convenção, o sistema OpenBus deve sempre manter compatibilidade, a nível de API, com a versão imediatamente anterior à corrente.

O esquema de versionamento do OpenBus utiliza um formato X.Y.Z, onde os campos X e Y representam a versão da interface dos componentes do sistema e o campo Z é incrementado quando ocorre alguma modificação que não altera a interface dos componentes. Como podemos ver pela figura 4.1 existem três versões principais do OpenBus: 1.4, 1.5 e 2.0 e as atualizações que continham alteração de interface foram a primeira em 03/2010 e a quinta (última) em 10/2011.

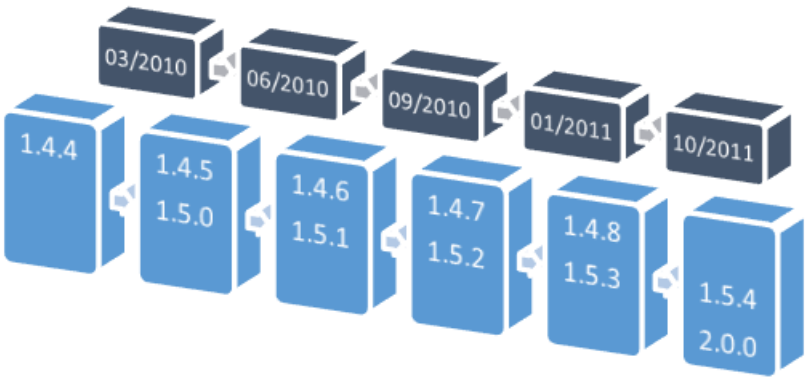

Figura 4.1: Evolução do OpenBus

4.1.1 Arquitetura das versões 1.4 e 1.5

Como apresentado na figura 4.2, o OpenBus nas versões 1.4 e 1.5 oferece três serviços básicos: Access Control Service, Register Service e Session Service.

Serviço de Acesso (ou Access Control Service) :

É o ponto de entrada no barramento. Sua referência é conhecida por todos. Para que uma aplicação possa entrar no barramento para consumir ou prover serviços, é necessário que ela se autentique utilizando uma credencial de acesso. A autenticação pode ser realizada através de um par usuário/senha ou através de um certificado digital. Após a autenticação, uma credencial é emitida para a aplicação para que a mesma possa acessar os serviços disponibilizados pelo barramento. Essa credencial é composta por uma identificador único e por um nome de uma entidade à qual pertence a credencial. Essa credencial tem um ciclo de vida e pode ser renovada para que não expire.

Serviço de Registro (ou Register Service) :

É o repositório responsável por controlar as ofertas de serviços disponíveis no barramento. Uma aplicação que queira oferecer um serviço deve explicitamente registrar sua oferta no serviço de registro. Aplicações que desejam utilizar um serviço podem obter a localização e as propriedades de provedores desse serviço através de consultas ao serviço de registro.

Serviço de Sessão (ou Session Service) :

É o serviço que permite criar sessões de comunicação entre apliações. Oferece um mecanismo simplificado de troca de mensagens entre as aplicações que compartilham uma mesma sessão.

As versões 1.4 e 1.5 também oferecem uma interface de serviço de acesso a dados. Essa interface foi criada como modelo para uma aplicação que disponibiliza serviços de acesso a dados hierárquicos. A aplicação que implementar essa interface pode ser disponibilizada no barramento como um serviço extra para a utilização de outros apliações, como é o caso da aplicação App1 na figura 4.2 que representa o Serviço de Dados (ou Database Service).

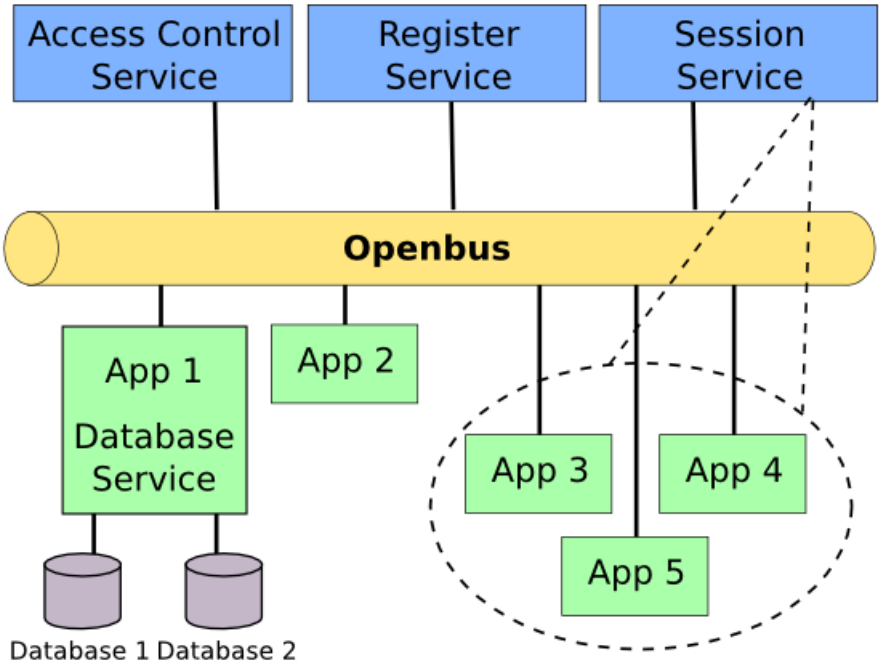

Figura 4.2: Representação do OpenBus 1.4 e 1.5.

4.1.2 Arquitetura da versão 2.0

Na versão 2.0 a arquitetura foi modificada. O Serviço de Acesso e o Serviço de Registro foram unificados, como serviços núcleo, em um único componente que representa o barramento. O Serviço de Sessão, que agora se chama Serviço de Colaboração, foi removido dos serviços básicos e passou a ser um serviço extra que pode ter sua oferta registrada no barramento como qualquer outro.

Nessa versão o barramento passou a persistir todo o seu estado no diretório de dados. Dessa forma, sempre que ele é iniciado e já existe uma base de dados populada, esses dados serão recarregados e farão parte do estado inicial do barramento.

Apesar de manter compatibilidade com a versão 1.5, essa nova versão introduz muitas melhorias no quesito de segurança, porém essas melhorias não serão usufruídas por clientes que acessam o barramento com versões antigas das bibliotecas de acesso. O barramento permite que aplicações que usam a biblioteca de acesso da versão 1.5 se comuniquem com clientes que utilizam a biblioteca de acesso 2.0, e vice-e-versa, porém, por motivos de compatibilidade, essas comunicações se realizam com o mesmo nível de segurança que existia na versão 1.5.

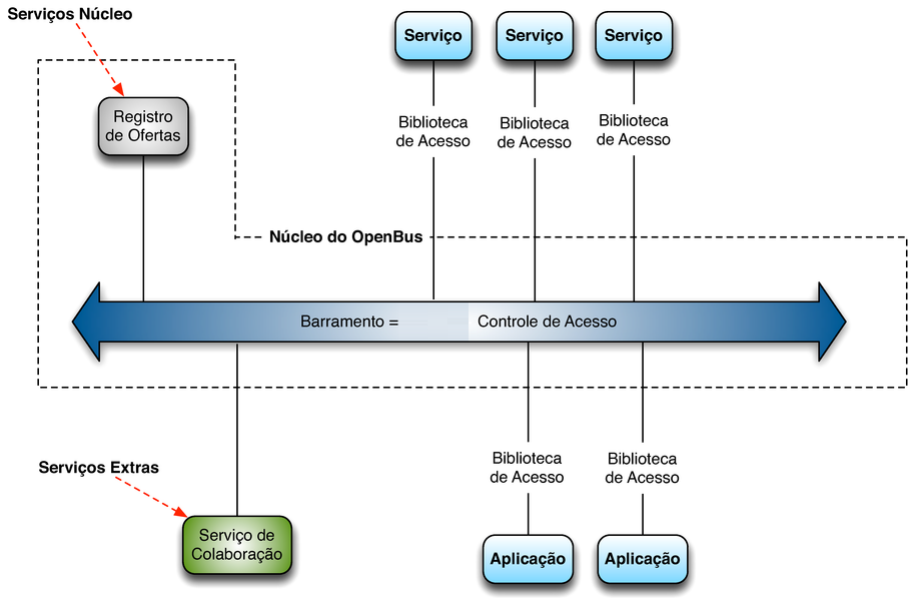

Figura 4.3: Representação do OpenBus 2.0.

4.1.3 Atualizações da versão 1.4 até a versão 2.0

A seguir detalhamos a arquitetura das diferentes versões do OpenBus no escopo de componentes, facetas e conexões. A figura 4.4 ilustra a arquitetura do sistema na versão 1.4, com os componentes: Access Control Service (ACS), Register Service (RS) e o Session Service(SS). Esses componentes, juntos, representam o barramento do OpenBus. A comunicação entre esses componentes é feita através das conexões de seus receptáculos e facetas. Os componentes SS e RS possuem um receptáculo que se conectam a faceta IComponent do ACS e o ACS possui um receptáculo q se conecta a faceta IComponent do RS.

Serviço de Acesso (ou ACS) :

Com as facetas ILeaseProvider (ou ILP) e IAccessControlService(ou IACS) oferece o serviço de autenticação e credenciamento dos componentes no barramento. Possui um receptáculo RegistryServiceReceptacle(ou RSR) que expressa sua dependência externa ao Serviço de Registro.

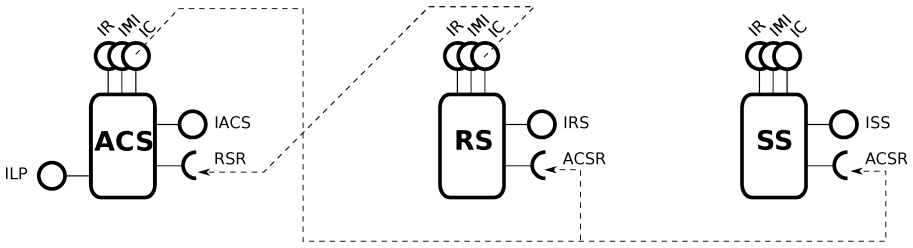

Figura 4.4: Arquitetura do OpenBus 1.4.

Serviço de Registro (ou RS) :

Com sua faceta IRegistryService (ou IRS) oferece o serviço de cadastro e oferta de componentes no barramento. Possui um receptáculo AccessControlServiceReceptacle(ou ACSR) que expressa sua dependência externa ao Serviço de Acesso.

Serviço de Sessão (ou SS) :

Com sua faceta ISessionService oferece o serviço de compartilhamento de comunicação entre componentes. Possui um receptáculo AccessControlServiceReceptacle (ou ACSR) que expressa sua dependência externa ao Serviço de Acesso

A figura 4.5 ilustra a versão 1.5. As facetas sombreadas representam aquelas que foram mantidas para compatibilidade com a versão 1.4.

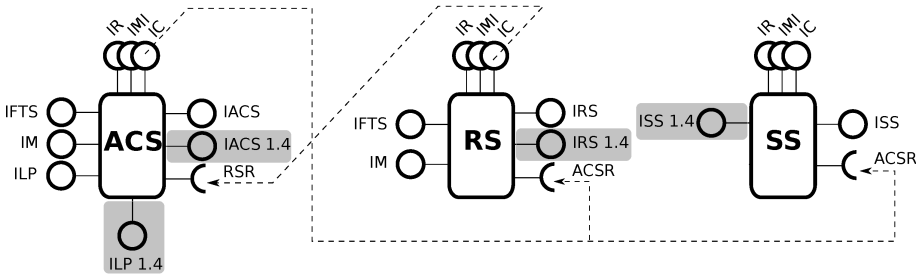

Figura 4.5: Arquitetura do OpenBus 1.5.

Serviço de Acesso (ou ACS) :

Foram adicionadas as facetas IFaultToleranceSupport (ou IFTL) e IManagement (ou IM) que introduzem serviços de tolerância a falha e gerenciamento de credenciais de acesso. As facetas ILP e IACS foram atualizadas e foi mantida a compatibilidade com a versão 1.4.

Serviço de Registro (ou RS) :

Foram adicionadas as facetas IFaultToleranceSupport (ou IFTL) e IManagement (ou IM) que introduzem serviços de tolerância a falha e geren-

ciamento de ofertas de componentes. A faceta IRS foi atualizada e foi mantida a compatibilidade com a versão 1.4.

Serviço de Sessão (ou SS) :

A faceta ISS foi atualizada e foi mantida a compatibilidade com a versão 1.4.

Na versão 2.0, apresentada na figura 4.6, foi criado um novo componente OpenBus com novas facetas. O ACS e o RS da versão 1.5, que aparecem no sombreado da imagem, foram mantidos com as suas interfaces, porém só funcionam como uma fachada para a nova versão. O Serviço de Sessão foi removido do conjunto básico de serviços e passou a ser um serviço extra ofertado no barramento.

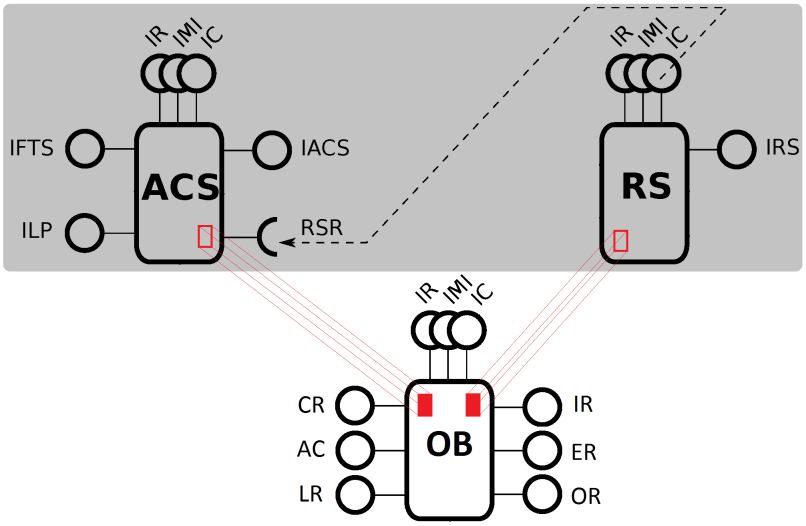

Figura 4.6: Arquitetura do OpenBus 2.0.

Serviço de Acesso (ou ACS) :

Foram removidas a faceta IM e as facetas IACS e ILP que davam suporte à versão 1.4. Foram atualizadas as facetas IFTS, ILP e IACS para manter a compatibilidade com a versão 1.5.

Serviço de Registro (ou RS) :

Foram removidas as faceta IM, IFTS e a faceta IRS que dava suporte à versão 1.4. Foi atualizada a faceta IRS para manter a compatibilidade com a versão 1.5.

OpenBus (ou OB) :

As facetas CertificateRegistry, AccessControl, LoginRegistry, InterfaceRegistry e EntityRegistry oferecem os serviços de controle de acesso,

autenticação e credenciamento no barramento. A faceta OfferRegistry oferece o mesmo serviço, anteriormente oferecido pelo RS, de cadastro e busca de ofertas de componentes no barramento.

A seguir vamos analisar o histórico de atualizações com mais detalhes. A primeira versão que possui notas de lançamento (*release notes*) do OpenBus é a versão 1.4.4, a última versão antes da atualização para a versão 1.5. Analisando as notas de lançamento das versões do OpenBus, a partir da versão 1.4.4, foi possível encontrar cinco atualizações do OpenBus. A primeira atualização foi da versão 1.4.4 para a versão 1.5.0. Nessa atualização, os componentes da 1.4.4 foram atualizados também para 1.4.5 e a compabitilidade com a versão 1.4 foi mantida. Os componentes foram atualizados para levar em consideração a coexistência das duas versões diferentes. Em todas as atualizações subsequentes os componentes da versão anterior são atualizados junto com a nova versão para manter a compatibilidade.

Atualizações:

1) 1.4.4 para 1.5.0 (1.4.5)

2) 1.5.0 para 1.5.1 (1.4.6)

3) 1.5.1 para 1.5.2 (1.4.7)

4) 1.5.2 para 1.5.3 (1.4.8)

5) 1.5.3 para 2.0.0 (1.5.4)

Nessas atualizações foi possível classificar oito tipos diferentes de modificações:

A) Bugfix:

Esse tipo de modificação é necessária quando o sistema está apresentando algum problema ou comportamento inesperado. Exemplos: correção de erros que possam causar inconsistências ou terminar a execução do programa, tratamentos de exceções que não eram feitos, alteração de comportamentos inesperados. Exemplo nas notas de lançamento: [OPENBUS-352] - Serviço de Sessão caía por falta de recursos: “Too many open files raised”

B) Melhorias, otimizações e alteração de comportamento:

Esse tipo de modificação representa alterações na implementação ou configuração do componente mas que não incluam novas funcionalidades.

Exemplos: otimização de código, remoção de código não utilizado, melhoras na compilação e configuração do sistema. Exemplo nas notas de lançamento: [OPENBUS-537] - Registrar nos logs o nome das operações interceptadas.

C) Novas funcionalidades previstas:

Esse tipo de modificação inclui um novo comportamento para o componente, comportamento esse que pode ser utilizado por uma das interfaces já definidas pelo componente. Exemplos: novas políticas de execução, novas opções de configuração, qualquer adição de nova funcionalidade que não altere a interface do componente. Exemplo nas notas de lançamento: [OPENBUS-383] - Adicionar o nome da interface como critério para busca.

D) Novas funcionalidades não previstas:

Esse tipo de modificação inclui um novo comportamento para o componente, porém esse novo comportamento não está mapeado nas interfaces já definidas pelo componente, exigindo alterações nestas interfaces para poder ser utilizado. Exemplos: novos métodos, qualquer adição de nova funcionalidade que precisa de mudanças na interface do componente, seja na faceta ou nos receptáculos. Exemplo nas notas de lançamento: [OPENBUS-2068] - Renomear a operação 'getServices' do 'OfferRegistry'.

E) Estrutural/Organizacional:

Esse tipo de modificação não contempla alterações na implementação do sistema e sim alterações de documentação, comentários, localização de recursos. Exemplos: documentação, testes, alteração da localização dos arquivos nos diretórios, alteração de instalador. Exemplo nas notas de lançamento: [OPENBUS-2209] - Correções e melhorias nos manuais (de introdução e de instalação).

F) Alteração/Atualização de dependência externa]:

Esse tipo de modificação representa uma alteração em dependências externas do sistema. Exemplos: trocar a versão de uma dependência externa, passar a usar ou deixar de usar uma dependência externa. Exemplo nas notas de lançamento: [OPENBUS-214] - Atualizar a biblioteca luasocket para a versão 2.0.2.

G) Alteração do Framework:

Esse tipo de modificação representa uma alteração no framework de componentes de software. Exemplos: alteração do modelo de componentes ou da sua implementação. Exemplo nas notas de lançamento: [OPENBUS-1373] - Atualizar OpenBus para utilizar o SCS 1.3.0.

H) Alteração do Ambiente de Execução:

Esse tipo de modificação representa uma alteração no ambiente de execução. Exemplos: atualização do sistema operacional, atualização da máquina virtual ou ambiente de execução do sistema. Exemplo nas notas de lançamento: [OPENBUS-1313] - Modificar a LuaVM para suportar inteiros de 64bits.

Após classificar os tipos de modificações, pudemos separar e quantificar a ocorrência desses tipos em cada uma das atualizações analisadas. A figura 4.7 apresenta o resultado dessa análise. Nela podemos ver que os principais tipos de modificações que aconteceram no OpenBus foram dos tipos E (Estrutural/Organizacional) e A (Bugfix). Podemos notar também que ao ocorrer modificações do tipo D (Novas funcionalidades não previstas) os campos X e Y da versão X.Y.Z são incrementados de acordo com a regra seguida pelo OpenBus.

Atualização	1.4.x	1.5.x	2.0.x	A	B	C	D	E	F	G	H
0	1.4.4			0	0	0	0	0	0	0	0
1	1.4.5	1.5.0		59	32	18	4	68	5	2	0
2	1.4.6	1.5.1		29	3	0	0	13	0	1	0
3	1.4.7	1.5.2		9	2	0	0	6	0	1	0
4	1.4.8	1.5.3		29	15	6	0	53	0	1	1
5		1.5.4	2.0.0	28	15	7	11	29	0	0	0

Figura 4.7: Detalhes das versões.

5
Aplicação e Avaliação do Mecanismo de Atualização Dinâmica

Nesse capítulo nós apresentamos os detalhes dos experimentos realizados com o mecanismo descrito no capítulo 3. Avaliamos primeiro a expressividade das interfaces oferecidas pelo mecanismo aplicando as atualizações do OpenBus. Avaliamos depois o desempenho da implementação com testes de sobrecarga e o uso do modelo de avaliação quantitativo[5] e validamos o uso da atualização dinâmica no sistema estudado.

5.1
Aplicações das atualizações no OpenBus

Escolhemos as atualizações 1 e 5 da lista da seção 4.1.3 (da versão 1.4.4 para a versão 1.5.0 e da versão 1.5.3 para a versão 2.0.0) do OpenBus para a simulação. Essas atualizações foram escolhidas por apresentarem não só alterações de facetas como também adições e remoções de facetas. Dentre as atualizações ocorridas no OpenBus, estas foram as únicas que apresentaram modificações do tipo D, que influenciam a interface do componente.

Na aplicação destas atualizações nós utilizamos o mecanismo de atualização dinâmica, que inclui as facetas IBackdoor e IDynamicUpdatable, detalhado no capítulo 3. Antes de executar o experimento alteraramos a implementação do OpenBus tanto na versão 1.4 quanto na versão 1.5 para incluir a oferta da faceta IDynamicUpdatable de cada componente básico do OpenBus no barramento.

Outra medida que teve que ser tomada foi a cópia das novas dependências para o servidor onde o componente está rodando, antes da aplicação da atualização. Isso foi necessário por não haver um empacotador que possibilitasse o envio de todas as informações necessárias para a atualização do componente.

Atualização da versão 1.4 para 1.5

Analisando novamente a arquitetura da versão 1.4 (figura 5.1) e da versão 1.5 (figura 5.2) podemos identificar claramente as alterações estruturais que ocorrem na atualização. No componente ACS são incluídas 4 novas facetas e

são atualizadas duas facetas para manter a compatibilidade entre as versões. No componente RS são incluídas três facetas e uma é atualizada para manter a compatibilidade. No componente SS só uma faceta é incluída e uma faceta é atualizada para manter a compatibilidade.

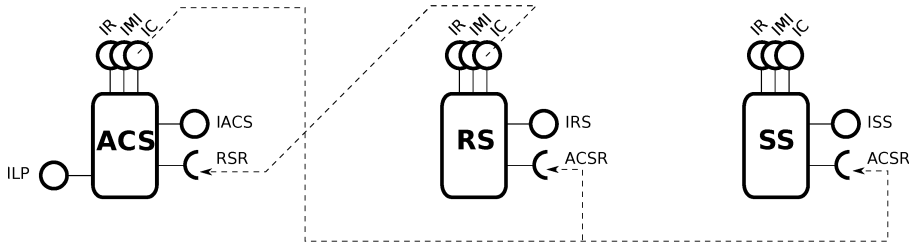

Figura 5.1: Arquitetura do OpenBus 1.4.

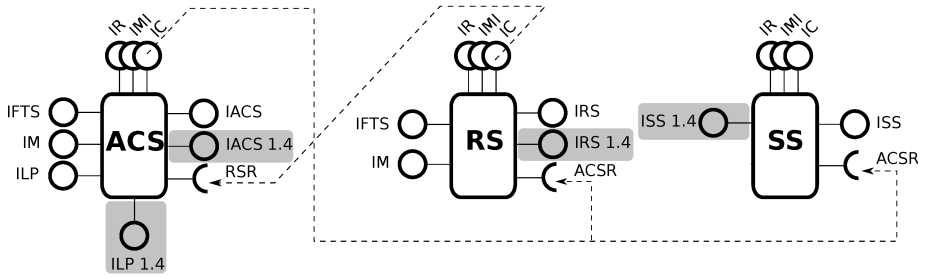

Figura 5.2: Arquitetura do OpenBus 1.5.

Abaixo seguem os exemplos de atualização de cada componente da versão 1.4 para a versão 1.5. Para cada componente foram analisadas todas as facetas, suas novas implementações e se possuíam alguma informação que deveria para ser persistida mesmo após a atualização.

Atualização do Registry Service (ou RS) : Neste componente primeiro foi adicionada a faceta IRS da versão 1.5 e copiada todas as ofertas já registradas na versão 1.4. Depois a faceta IRS da versão 1.4 foi atualizada para ser somente uma fachada. Depois foram adicionadas as facetas IM e IFTS. Por último foi atualizado o IReceptacles com uma nova implementação.

No código 5.1 abaixo apresenta o início da atualização do componente RS. Na terceira linha executamos o método InsertFacet() da faceta IDynamicUpdatable do componente RS. Nesta execução utilizamos como parâmetro as informações da nova faceta IRegistryService_v1.5 que está sendo criada, sua implementação e o código que recupera as referências das ofertas de serviços armazenadas no estado da faceta da versão 1.4. Logo após, na linha 12, executamos o método UpdateFacet() para realizar a atualização da antiga faceta IRegistryService.

```
1  ...
2    --Insert RS1.5 copy offers from 1.4
3    local ret =updateRS:InsertFacet(updateKey,{
4              description={name="IRegistryService_v1.5",
5                interface_name=props.REGISTRY_SERVICE_INTERFACE.value,
6                facet_idl=oil.readfrom("idl/1.5/registry_service.idl"),
7                facet_implementation=oil.readfrom("RS15Facet.lua")},
8              patchUpCode="_newFacet.offersByIdentifier_=_self.context.IRegistryService.offersByIdentifier;
                    _newFacet.offersByCredential_=_self.context.IRegistryService.offersByCredential",
9              patchDownCode="",
10             key=props.REGISTRY_SERVICE_KEY.value})
11   ...
12   --Update RS1.4
13    ret = updateRS:UpdateFacet(updateKey,
14       {description={name="IRegistryService",
15        interface_name=props.REGISTRY_SERVICE_INTERFACE_V1_04.value,
16        facet_idl=oil.readfrom("idl/1.4/registry_service.idl"),
17        facet_implementation=oil.readfrom("RS14Facet.lua")},
18       patchUpCode="",
19       patchDownCode="",
20       key=props.REGISTRY_SERVICE_KEY_V1_04.value})
21   ...
```

Código 5.1: Atualização da faceta IRS do componente Registry Service ou RS da versão 1.4 para 1.5

No código 5.2 abaixo apresenta a inserção das facetas IM e IFTS no componente RS. Executamos o método InsertFacet() da faceta IDynamicUpdatable do componente RS. Nesta execução utilizamos como parâmetro as informações das novas facetas IManagement_v1.5 e IFaultTolerantService_v1.5 que estão sendo criadas e suas respectivas implementações RSManagementFacet.lua e RSFaultToleranceFacet.lua.

```
1  ...
2    --Insert IM
3    ret=updateRS:InsertFacet(updateKey,{
4              description={name="IManagement_v1.5",
5                interface_name=props.MANAGEMENT_RS_INTERFACE.value,
6                facet_idl=oil.readfrom("idl/1.5/registry_service.idl"),
7                facet_implementation=oil.readfrom("RSManagementFacet.lua")},
8              patchUpCode="",
9              patchDownCode="",
10             key=props.MANAGEMENT_RS_KEY.value})
11   ...
12   --Insert IFTS
13   ret=updateRS:InsertFacet(updateKey,{
14             description={name="IFaultTolerantService_v1.5",
15               interface_name=props.FAULT_TOLERANT_SERVICE_INTERFACE.value,
16               facet_idl=oil.readfrom("idl/1.5/fault_tolerance.idl"),
17               facet_implementation=oil.readfrom("RSFaultToleranceFacet.lua")},
18             patchUpCode="",
19             patchDownCode="",
20             key=props.FAULT_TOLERANT_RS_KEY.value})
21   ...
```

Código 5.2: Atualização das facetas IM e IFTS do componente RS

No código 5.3 abaixo apresenta a atualização da faceta de controle de receptáculos do componente RS. Executamos o método UpdateFacet() da faceta IDynamicUpdatable do componente RS. Nesta execução utilizamos como parâmetro as informações da nova implementação do IReceptacle que está sendo atualizado, a nova implementação e o código que recupera o estado da antiga implementação da faceta IReceptacles.

```
1  ...
2    --Update IReceptacle
3    ret = updateRS:UpdateFacet(updateKey,
4        {description={name="IReceptacles",
5         interface_name=IDL:"scs/core/IReceptacles:1.0",
6         facet_idl=oil.readfrom(os.getenv("IDL_PATH") .. "/scs.idl"),
7         facet_implementation=oil.readfrom("RSIRECFacet.lua")},
8        patchUpCode="_newFacet._nextConnId = oldFacet._nextConnId, newFacet._maxConnections =
             _oldFacet._maxConnections, _newFacet._numConnections = oldFacet._numConnections,
             _newFacet._receptsByConId = _oldFacet._receptsByConId",
9        patchDownCode="",
10       key=props.RSIRECEPTACLES_KEY.value})
11   ...
```

Código 5.3: Atualização da faceta IReceptacle do componente RS

Atualização do Access Control Service (ou ACS) : Neste componente primeiro foi adicionada a faceta IACS da versão 1.5 e copiadas todas as credenciais já registradas e autorizadas na versão 1.4. Depois a faceta IACS da versão 1.4 foi atualizada para ser somente uma fachada. Depois foi adicionada a faceta ILP da versão 1.5 sem copiar nenhuma informação da versão 1.4. Em seguida foi atualizada a versão 1.4 da faceta ILP para servir como fachada para a nova versão. Logo após foram adicionadas as facetas IM e IFTS. Por último foi atualizado o IReceptacles com uma nova implementação.

```
1  ...
2    --Insert ACS1.5 -- copy credentials entries from 1.4
3    local ret =updateACS:InsertFacet(updateKey,{
4             description={name="IAccessControlService_v1.5",
5               interface_name=props.ACCESS_CONTROL_SERVICE_INTERFACE.value,
6               facet_idl=oil.readfrom("idl/1.5/access_control_service.idl"),
7               facet_implementation=oil.readfrom("ACS15Facet.lua")},
8             patchUpCode="_newFacet.entries = self.context.IAccessControlService.entries",
9             patchDownCode="",
10            key=props.ACCESS_CONTROL_SERVICE_KEY.value})
11   ...
12   --Update ACS1.4
13   ret = updateACS:UpdateFacet(updateKey,
14       {description={name="IAccessControlService",
15        interface_name=props.ACCESS_CONTROL_SERVICE_INTERFACE_V1_04.value,
16        facet_idl=oil.readfrom("idl/1.4/access_control_service.idl"),
17        facet_implementation=oil.readfrom("ACS14Facet.lua")},
18       patchUpCode="",
```

```
19        patchDownCode="",
20        key=props.ACCESS_CONTROL_SERVICE_KEY_V1_04.value})
21    ...
22    --Insert ILP1.5 no state
23    local ret =updateACS:InsertFacet(updateKey,{
24              description={name="ILeaseProvider_v1.5",
25                interface_name=props.LEASE_PROVIDER_INTERFACE.value,
26                facet_idl=oil.readfrom("idl/1.5/access_control_service.idl"),
27                facet_implementation=oil.readfrom("ILP15Facet.lua")},
28              patchUpCode="",
29              patchDownCode="",
30              key=props.LEASE_PROVIDER_KEY.value})
31    ...
32    --Update ILP1.4 no state
33     ret = updateACS:UpdateFacet(updateKey,
34        {description={name="ILeaseProvider",
35         interface_name=props.LEASE_PROVIDER_INTERFACE_V1_04.value,
36         facet_idl=oil.readfrom("idl/1.4/access_control_service.idl"),
37         facet_implementation=oil.readfrom("ILP14Facet.lua")},
38        patchUpCode="",
39        patchDownCode="",
40        key=props.LEASE_PROVIDER_KEY_V1_04.value})
41    ...
42    --Insert IM
43    ret=updateACS:InsertFacet(updateKey,{
44              description={name="IManagement_v1.5",
45                interface_name=props.MANAGEMENT_ACS_INTERFACE.value,
46                facet_idl=oil.readfrom("idl/1.5/access_control_service.idl"),
47                facet_implementation=oil.readfrom("ACSManagementFacet.lua")},
48              patchUpCode="",
49              patchDownCode="",
50              key=props.MANAGEMENT_ACS_KEY.value})
51    ...
52    --Insert IFTS
53    ret=updateACS:InsertFacet(updateKey,{
54              description={name="IFaultTolerantService_v1.5",
55                interface_name=props.FAULT_TOLERANT_SERVICE_INTERFACE.value,
56                facet_idl=oil.readfrom("idl/1.5/fault_tolerance.idl"),
57                facet_implementation=oil.readfrom("ACSFaultToleranceFacet.lua")},
58              patchUpCode="",
59              patchDownCode="",
60              key=props.FAULT_TOLERANT_ACS_KEY.value})
61    ...
62    --Update IReceptacle
63     ret = updateACS:UpdateFacet(updateKey,
64        {description={name="IReceptacles",
65         interface_name=IDL:"scs/core/IReceptacles:1.0",
66         facet_idl=oil.readfrom(os.getenv("IDL_PATH") .. "/scs.idl"),
67         facet_implementation=oil.readfrom("ACSIRECFacet.lua")},
68        patchUpCode="_newFacet._nextConnId_=_oldFacet._nextConnId,_newFacet._maxConnections_=_
              _oldFacet._maxConnections,_newFacet._numConnections_=_oldFacet._numConnections,_
              _newFacet._receptsByConId_=_oldFacet._receptsByConId",
69        patchDownCode="",
70        key=props.ACSIRECEPTACLES_KEY.value})
71    ...
```

Código 5.4: Atualização do componente Access Control Service ou ACS da versão 1.4 para 1.5

Atualização do Session Service (ou SS) : Neste componente primeiro foi adicionada a faceta ISS da versão 1.5 e copiadas todas as sessões em andamento na versão 1.4. Depois a faceta ISS da versão 1.4 foi atualizada para ser somente uma fachada.

```
1  ...
2    --Insert SS1.5 copy sessions from SS1.4
3    local ret =updateSS:InsertFacet(updateKey,{
4              description={name="ISessionService_v1.5",
5                interface_name=props.SESSION_SERVICE_INTERFACE.value,
6                facet_idl=oil.readfrom("idl/1.5/session_service.idl"),
7                facet_implementation=oil.readfrom("SS15Facet.lua")},
8              patchUpCode="_newFacet.sessions_=_self.context.ISessionService.sessions;_newFacet.observed_=
                   _self.context.ISessionService.observed;_newFacet.invalidMemberIdentifier_=_self.context.
                   ISessionService.invalidMemberIdentifier",
9              patchDownCode="",
10             key=props.SESSION_SERVICE_KEY.value})
11   ...
12   --Update SS1.4
13    ret = updateSS:UpdateFacet(updateKey,
14        {description={name="ISessionService",
15         interface_name=props.SESSION_SERVICE_INTERFACE_V1_04.value,
16         facet_idl=oil.readfrom("idl/1.4/session_service.idl"),
17         facet_implementation=oil.readfrom("SS14Facet.lua")},
18        patchUpCode="",
19        patchDownCode="",
20        key=props.SESSION_SERVICE_KEY_V1_04.value})
21   ...
```

Código 5.5: Atualização do componente Session Service ou SS da versão 1.4 para 1.5

Atualização da versão 1.5 para 2.0

Na atualização do OpenBus da versão 1.5 (figura 5.3) para a versão 2.0 (figura 5.4), os desenvolvedores tomaram a decisão de juntar os dois componentes, ACS e RS em um só componente, o OB. Essa decisão resultou numa restrição muito forte: o componente OB que representa a versão 2.0, precisa rodar no mesmo processo que os componentes ACS e RS da versão 1.5. Na implementação atual do OpenBus 2.0, os três componentes (ACS, RS e OB) compartilham o estado do sistema através da referência de dois objetos. Ou seja, os três componentes precisam estar necessariamente no mesmo processo e ter acesso a esses dois objetos compartilhados. Essa decisão de implementação tomada pela equipe do OpenBus acarreta em um problema. Hoje o mecanismo de autenticação do OpenBus é feito através de um SDK distribuído pela equipe do OpenBus. Esse SDK é uma biblioteca que possui a referência para os serviços básicos do OpenBus. Essas referências ficam implícitas na biblioteca e são feitas por meio de arquivo de configuração,

logo elas não podem ser trocadas em tempo de execução. Isso impede a utilização imediata do mecanismo desenvolvido neste trabalho e qualquer outro mecanismo de atualização não possuiria as ferramentas necessárias para juntar os dois componentes ACS e RS no mesmo processo ou alterar as referências dos clientes em tempo de execução. Para contornar esse problema propomos duas abordagens: 1) é sugerido uma nova arquitetura (figura 5.5) que explicita a dependência que os componentes ACS e RS, da versão 1.5, possuem do componente OB, da versão 2.0; 2) é sugerido o mapeamento das referências dos serviços básicos do OpenBus como componentes, assim poderemos reconfigurar os receptáculos e alterar as referências a esses serviços caso seja necessário. Essas alterações demandaram o desenvolvimento da funcionalidade de adição e remoção de receptáculos no mecanismo de atualização dinâmica.

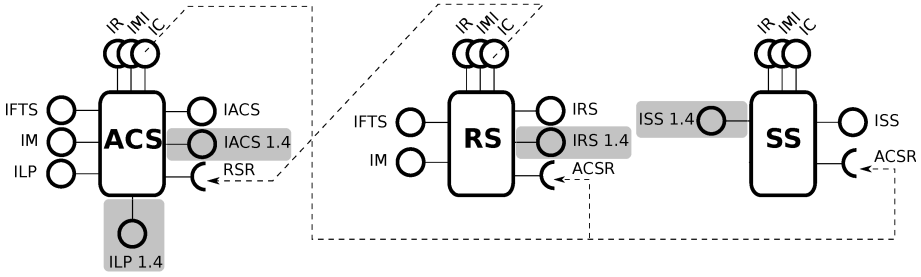

Figura 5.3: Arquitetura do OpenBus 1.5.

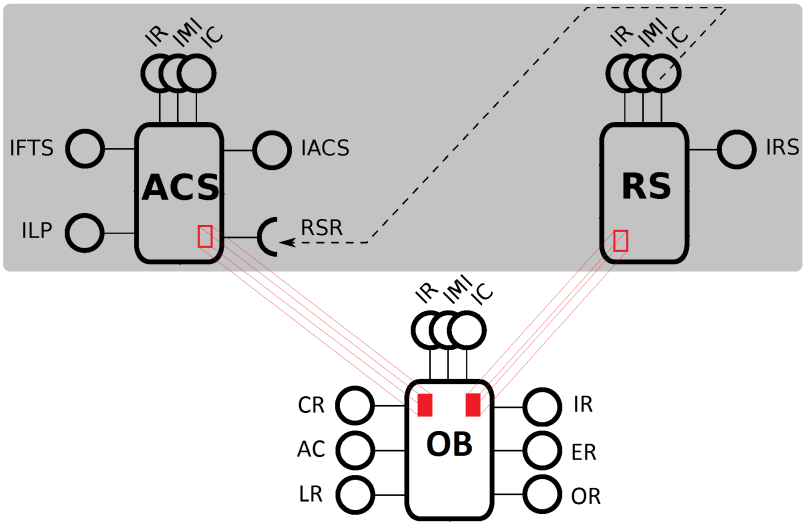

Figura 5.4: Arquitetura do OpenBus 2.0.

1) Nova arquitetura

Com essa nova arquitetura da versão 2.0 (figura 5.5), podemos fazer o comparativo com a versão 1.5 (figura 5.3) e identificar as alterações ocorridas. No

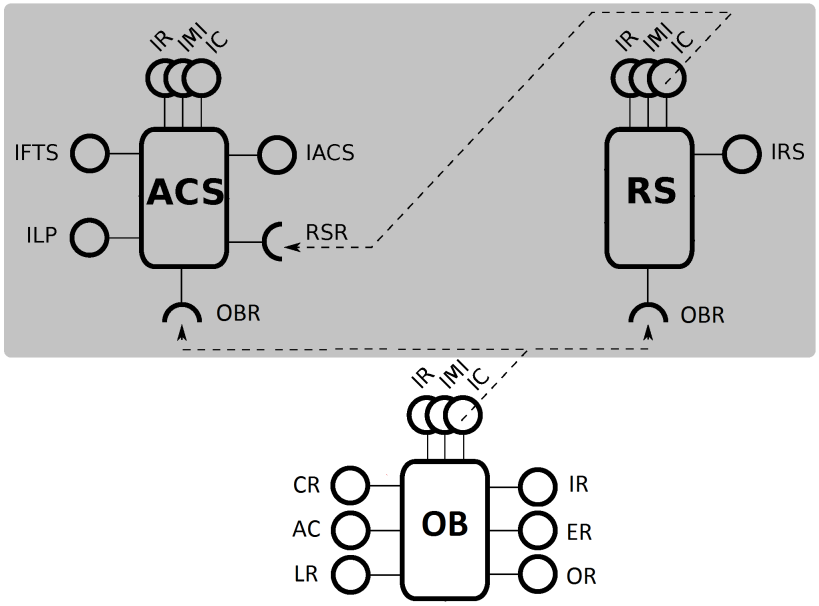

Figura 5.5: Nova Arquitetura proposta para o OpenBus 2.0.

componente ACS são excluídas três facetas e adicionado um novo receptáculo. No componente RS também são excluídas três facetas, adicionado um novo receptáculo e além dessas alterações é excluído o receptáculo ACSR. Ainda, na arquitetura da versão 2.0, o componente SS não existe mais como um serviço básico do OpenBus, e foi criado um novo componente único, o OB, que representa todos os serviços básicos. Abaixo seguem as atualizações dos componentes ACS e RS.

Atualização do Registry Service (ou RS) : Neste componente primeiro foi adicionado o receptáculo OBR para o IComponent do componente OB da versão 2.0, depois conectamos o componente OB no receptáculo. Depois a faceta IRS foi atualizada para funcionar como fachada do componente OB. Depois foram removidas as facetas IRS da versão 1.4, IFTS e IM. Por fim foi removido o receptáculo ACSR.

```
1  ...
2    --Insert Receptacle for OB IComponent
3    local ret =updateRS:InsertReceptacle(updateKey,"OBR","IDL:scs/core/IComponent:1.0")
4    ...
5     --Update IRS1.5 and Connect OB into Receptacle
6     ret = updateRS:UpdateFacet(updateKey,
7         {description={name="IRegistryService_v1.5",
8                 interface_name=props.REGISTRY_SERVICE_INTERFACE.value,
9                 facet_idl=oil.readfrom("idl/1.5/registry_service.idl"),
10                facet_implementation=oil.readfrom("RS15Facet.lua")},
11            patchUpCode="local oil = require'oil';local ior = oil.readfrom('obcomponent.ior');local
                  OBComponent =self.context.orb:newproxy(ior);self.context.IReceptacles:connect('OBR',
                  OBComponent)",
```

```
12              patchDownCode="",
13              key=props.REGISTRY_SERVICE_KEY.value})
14    ...
15     --Remove IRS1.4
16     ret =updateRS:DeleteFacet(updateKey,"IRegistryService")
17    ...
18     --Remove IFTS
19     ret =updateRS:DeleteFacet(updateKey,"IFaultTolerantService_v1.5")
20    ...
21     --Remove IM
22     ret =updateRS:DeleteFacet(updateKey,"IManagement_v1.5")
23    ...
24     --Remove receptacle ACSR
25     ret =updateRS:DeleteReceptacle(updateKey,"ACSR")
26    ...
```

Código 5.6: Atualização do componente Registry Service ou RS da versão 1.5 para 2.0

Atualização do Access Control Service (ou ACS) : Neste componente primeiro foi adicionado o receptáculo OBR para o IComponent do componente OB da versão 2.0, depois conectamos o componente OB no receptáculo. Depois a faceta IACS, ILP e a IFTS foram atualizadas para funcionarem como fachada do componente OB. Depois foram removidas as facetas de IACS e ILP da versão 1.4 e a faceta IM.

```
1   ...
2      --Insert Receptacle for OB IComponent
3     local ret =updateACS:InsertReceptacle(updateKey,"OBR","IDL:scs/core/IComponent:1.0")
4     ...
5      --Update IACS1.5 and Connect OB into Receptacle
6      ret = updateACS:UpdateFacet(updateKey,
7          {description={name="IAccessControlService_v1.5",
8                  interface_name=props.ACCESS_CONTROL_SERVICE_INTERFACE.value,
9                  facet_idl=oil.readfrom("idl/1.5/access_control_service.idl"),
10                 facet_implementation=oil.readfrom("ACS15Facet.lua")},
11             patchUpCode="local oil = require'oil';local ior = oil.readfrom('obcomponent.ior');local
                   OBComponent =self.context.orb:newproxy(ior);self.context.IReceptacles:connect('OBR',
                   OBComponent)",
12             patchDownCode="",
13             key=props.REGISTRY_SERVICE_KEY.value})
14    ...
15     --Update ILP1.5
16     ret = updateACS:UpdateFacet(updateKey,
17         {description={name="ILeaseProvider_v1.5",
18                 interface_name=props.LEASE_PROVIDER_INTERFACE.value,
19                 facet_idl=oil.readfrom("idl/1.5/access_control_service.idl"),
20                 facet_implementation=oil.readfrom("ILP15Facet.lua")},
21             patchUpCode="",
22             patchDownCode="",
23             key=props.LEASE_PROVIDER_KEY.value})
24    ...
25     --Update IFTS1.5
26     ret = updateACS:UpdateFacet(updateKey,
```

```
27          {description={name="IFaultTolerantService_v1.5",
28                  interface_name=props.FAULT_TOLERANT_SERVICE_INTERFACE.value,
29                  facet_idl=oil.readfrom("idl/1.5/fault_tolerance.idl"),
30                  facet_implementation=oil.readfrom("ACSFaultToleranceFacet.lua")},
31              patchUpCode="",
32              patchDownCode="",
33              key=props.FAULT_TOLERANT_ACS_KEY.value})
34      ...
35       --Remove IACS1.4
36       ret =updateACS:DeleteFacet(updateKey,"IAccessControlService")
37      ...
38        --Remove ILP1.4
39       ret =updateACS:DeleteFacet(updateKey,"ILeaseProvider")
40      ...
41        --Remove IM
42       ret =updateACS:DeleteFacet(updateKey,"IManagement_v1.5")
43      ...
```

Código 5.7: Atualização do componente Access Control Service ou ACS da versão 1.5 para 2.0

2) Novo mapeamento de referências

Na versão 1.5 do OpenBus, os componentes ACS e RS eram, obrigatoriamente, disparados em processos separados. Os clientes do OpenBus utilizam um SDK distribuído pela equipe do Openbus para desenvolver as aplicações que utilizam o barramento para troca de informações. Esse SDK mantém, internamente, as referências para os componentes ACS e RS. Ou seja, todos os clientes que utilizam o SDK distribuído, não possuem as dependências do ACS e do RS mapeadas como receptáculos. Enquanto o cliente executar, não existe a possibilidade de atualizar as referências que ele possui para acessar o OpenBus.

Para atualizar o OpenBus da versão 1.5 para a versão 2.0, teríamos que criar um novo componente OB da versão 2.0 e no mesmo processo criar um componente ACS e um componente RS para prover compatibilidade para a versão 1.5. Depois que isto estivesse concluído, transformaríamos os componentes ACS e RS originais da versão 1.5 em *proxies* para os novos componentes ACS e RS como aparece na figura fig:proxy.

Para podermos fazer a reconfiguração e acabar com o componente intermediário, seria necessário mapear as referências utilizadas no SDK como receptáculos, de acordo com a figura 5.7. Com essas referências mapeadas seria possível alterar as conexões dos componentes do SDK e passar a utilizar as novas referências dos serviços básicos.

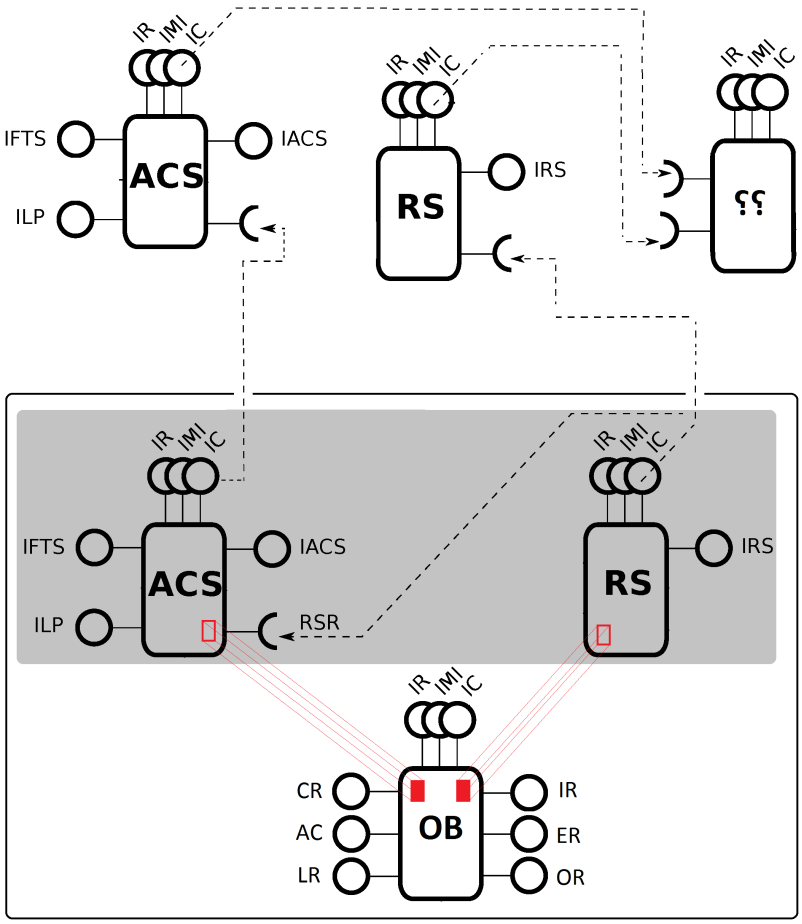

Figura 5.6: Exemplo utilizando os antigos componentes como proxies.

5.2 Testes de sobrecarga

Para controlar as requisições e o estado do componente no momento da atualização, a faceta ILifeCycle utiliza um interceptador que inspeciona todas as chamadas feitas ao componente. Esse interceptador verifica o estado do componente: no caso de ser SUSPENDED, enfileira a chamada para ser executada posteriormente; no caso de ser HALTED, descarta a chamada; e no caso de ser RESUMED, executa a chamada. Para possibilitar a consistência da atualização dinâmica, esse interceptador precisa sempre verificar o estado do componente antes de passar a chamada adiante. Fizemos um teste de sobrecarga para verificar qual o impacto dessas interceptações no funcionamento do sistema de componentes. Para medir esse impacto utilizamos o componente TV 2.3, descrito na subseção 2.1.1, em dois cenários: com o mecanismo de atualização dinâmica (logo, com as interceptações) e sem o mecanismo de atualização dinâmica (sem as interceptações).

Em ambos os cenários registramos 10 (dez) vezes o tempo necessário para executar um loop de 10.000 (dez mil) chamadas ao método Control:power()

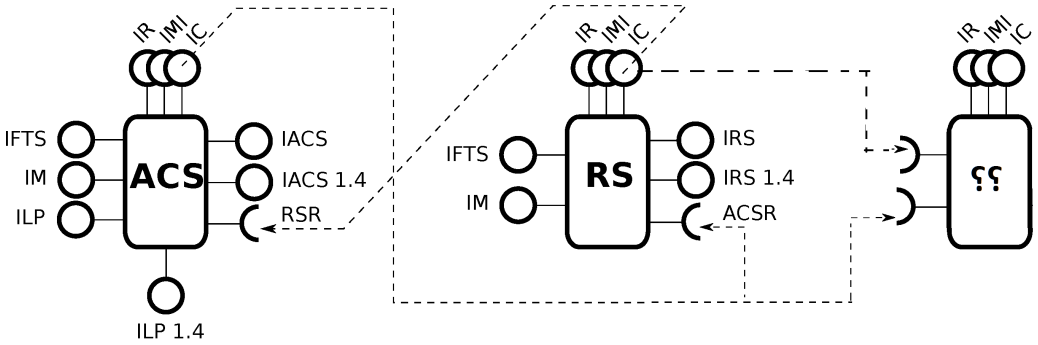

Figura 5.7: Exemplo de mapeamento das referências dos serviços básicos.

da faceta Control do componente TV. Depois fizemos uma média aritmética com os valores marcados em cada loop, removendo o maior e o menor valor. Para minimizar o efeito da rede nas chamadas remotas, executams na mesma máquina tanto o componente quanto a aplicação que efetuou as chamadas ao componente.

Marcação	Tempo	Média
1	4100,3	0,41003
2	4087,2	0,40872
3	4101,3	0,41013
4	4279,2	0,42792
5	4273,2	0,42732
6	4260,3	0,42603
7	4227,2	0,42272
8	4213,3	0,42133
9	4172,2	0,41722
10	4247,3	0,42473

Figura 5.8: Tempo em ms de 10.000 chamadas com o mecanismo.

No cenário com o mecanismo a média do tempo por chamada foi de 0,42 ms, com um desvio padrão de 0,006. No cenário sem o mecanismo a média do tempo por chamada foi de 0,40 ms, com um desvio padrão também de 0,005. O desvio padrão apresentado mostra que não há muita dispersão estatística nos valores medidos. Utilizando o tempo médio de chamada nos dois cenários, constatamos que o mecanismo causa uma sobrecarga de processamento de aproximadamente 5%.

5.3 Aplicação do modelo quantitativo

Avaliações em sistemas de atualização dinâmica normalmente são feitas com base na cobertura de tipos de atualização que o mecanismo se propõe a realizar e na sobrecarga de processamento que os mecanismos de atualização

Marcação	Tempo	Média
1	3935,2	0,39352
2	3906,2	0,39062
3	4012,3	0,40123
4	4143,2	0,41432
5	4007,3	0,40073
6	3975,2	0,39752
7	3919,3	0,39193
8	4088,2	0,40882
9	3925,2	0,39252
10	3932,3	0,39323

Figura 5.9: Tempo em ms de 10.000 chamadas sem o mecanismo.

dinâmica causam. Existem poucas avaliações que consideram os benefícios das atualizações dinâmicas. O modelo de avaliação quantitativo proposto por Gharaibeh et al. [5] foi criado para suprir uma demanda de avaliação complementar às avaliações tipicamente feitas em sistemas de atualização dinâmica. Esse modelo traz uma opção de avaliar se o benefício de atualizar dinamicamente justifica o custo da perda de desempenho que o mecanismo possa trazer. Em princípio, funcionar durante 24 horas nos 7 dias da semana é sempre uma vantagem, porém, isso pode não se aplicar a todos os sistemas. Essa avaliação permite ponderar sobre os benefícios de aceitar uma sobrecarga de processamento afim de evitar que o sistema fique desligado e inoperante.

Esse modelo, baseado em um modelo do mercado financeiro para avaliar preços de opções de investimento, apresenta uma maneira de calcular o ganho que a atualização dinâmica pode trazer para o sistema analisado. Ele utiliza como parâmetro o valor que as atualizações trazem para o sistema, o tempo demandado para aplicar a atualização, a frequência de atualizações e a sobrecarga que o mecanismo de atualização dinâmica causa para permitir a atualização dinâmica no sistema. O modelo atribui uma receita diária, fictícia ou não, para o sistema e observa como essa receita é afetada de acordo com os efeitos de cada atualização.

5.3.1 Modelo

Gharaibeh et al. consideram dois tipos principais de atualização:

Modelo 1: Atualização estática Nesse modelo o valor do sistema aumenta de acordo com as novas funcionalidades que são adicionadas e os erros que são corrigidos. O sistema deixa de funcionar enquanto está desligado para a aplicação de atualizações e isso gera perda de receita.

Modelo 2: Atualização dinâmica Nesse modelo o valor do sistema também aumenta de acordo com as novas funcionalidades que são adicionadas e os erros que são corrigidos. O sistema pode perder desempenho pela sobrecarga que o mecanismo de atualização dinâmica pode causar e isso pode diminuir a sua receita. Também, durante a atualização, o sistema opera parcialmente e isso também gera uma perda de receita.

Em ambos os modelos, o autor assume que a atualização foi previamente testada em um ambiente próprio e que ela não ocasionará erros ou inconsistências para os sistemas. Para ilustrar a variação da receita do sistema nos diferentes tipos de atualização podemos utilizar a figura 5.10. A figura mostra a variação da receita nos dois modelos, Modelo 1: Atualização estática (linha em negrito) e Modelo 2: Atualização dinâmica. Nos dois modelos, o sistema inicia com uma receita fixa, sendo a receita do segundo modelo inferior, pois a sobrecarga do mecanismo de atualização dinâmica pode ter um impacto (-Cvm) na receita. Em um dado momento (sigma), a atualização já está pronta para ser aplicada. No modelo dinâmico, a atualização pode então ser aplicada imediatamente, e enquanto isso ocorre, o sistema fica operando parcialmente (-Coa) e perde parte da sua receita. Depois de um tempo (Toa), a atualização foi aplicada no segundo modelo e ele volta a operar gerando uma receita maior devida às novas funcionalidades ou correções que foram aplicadas na atualização. No modelo estático é necessário esperar um tempo oportuno (Toff) para que o sistema possa ser desligado e depois um tempo para que a atualização seja aplicada. Depois de aplicada a atualização (Tfa), o sistema é religado e volta a operar também com uma receita maior devida às novas funcionalidades ou correções que foram aplicadas na atualização.

É importante notar que a sobrecarga de processamento que o mecanismo de atualização dinâmica causa no sistema é um fator importante nesse modelo. Essa sobrecarga pode causar uma perda de receita muito grande no longo prazo. Se a frequência de atualizações for pequena, o impacto que a sobrecarga de processamento causa no sistema vai ser muito significativo, dado que existe perda de receita diária por causa da sobrecarga. O primeiro parâmetro utilizado para o cálculo do modelo é o impacto da sobrecarga de processamento do mecanismo de atualização dinâmica.

Os diferentes tipos de modificações apresentados nas atualizações contribuem diferentemente para aumentar o valor do sistema. Enquanto novas funcionalidades e melhorias aumentam diretamente o seu valor, as correções de bugs evitam que o valor do sistema seja diminuído, aumentando o valor indiretamente. Atribuir valores exatos a cada modificação requer um estudo

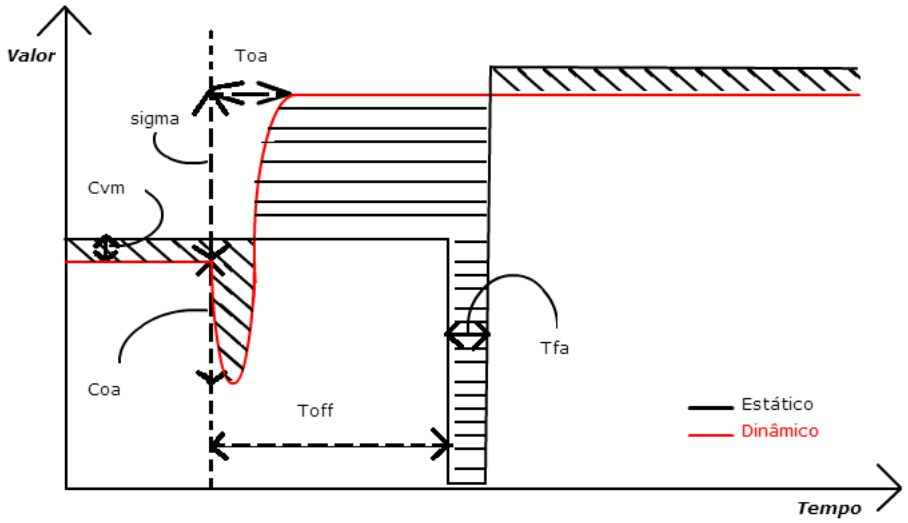

Figura 5.10: Exemplo do cálculo de receita dos modelos de atualização.

técnico e aprofundado de cada modificação e como ela afeta o valor do sistema como um todo. Associar esses valores exatos é possível, porém, caso não haja muitas informações detalhadas sobre o conteúdo de cada modificação, os autores do modelo sugerem utilizar heurísticas para aproximar os valores das modificações de acordo com o seu tipo. O segundo parâmetro utilizado no modelo é o valor que as atualizações agregam ao sistema. Ele é necessário para quantificar o aumento de valor após a atualização.

Junto com a sobrecarga de processamento e o valor das atualizações são utilizados a frequência com que o sistema é atualizado e qual a perda de desempenho durante a atualização dinâmica.

5.3.2 Parâmetros para análise do OpenBus

Para a análise do OpenBus utilizamos a atualização 4 (das versões 1.4.7 e 1.5.2 para as versões 1.4.8 e 1.5.3), por ser a atualização que mais demorou para ser lançada, mais precisamente 9 meses após o lançamento da atualização 3 (das versões 1.4.6 e 1.5.1 para as versões 1.4.7 e 1.5.2). As correções, novas funcionalidades e otimizações foram feitas ao longo de 9 meses e foram aplicadas todas em conjunto. Um dos objetivos na utilização dessa atualização é exemplificar que o método de atualização dinâmica pode trazer um ganho,

mesmo no longo prazo, se as atualizações forem sendo aplicadas a medida que forem desenvolvidas.

Na aplicação do modelo de avaliação só foram consideradas as modificações que aumentam o valor do sistema: Atualizações do tipo A (Bugfix), B (Melhorias, otimizações e alteração de comportamento) e C (Novas funcionalidades previstas na interface). De acordo com a tabela 4.7, foram ao todo 50 modificações (29 do tipo A, 15 do tipo B e 6 do tipo C) ao longo de 9 meses. Seguindo a heurística apresentada pelos autores, assumi que após a aplicação da atualização o sistema dobra seu valor, logo, em média, a aplicação de cada modificação aumenta o valor do sistema em 2% (100/50 = 2).

No cenário atual, para uma atualização estática do OpenBus é necessário um aviso prévio aos usuários de pelo menos 24 horas. As atualizações duram em torno de 10 minutos a 1 hora, dependendo das alterações da versão. Para a aplicação deste modelo utilizaremos um tempo médio de 35 minutos para a aplicação de uma atualização estática (Tfa) no OpenBus e utilizamos um tempo de espera para a aplicação da atualização estática (Toff) de 24 horas. Como tempo de aplicação de uma atualização dinâmica (Toa), utilizaremos um tempo médio (auferido através de testes) de 1 minuto. Considerei a sobrecarga de 5%, como medido na seção 5.2, devido ao impacto causado pelo mecanismo de atualização dinâmica e considerei que a aplicação, mesmo enquanto está sendo dinamicamente atualizada, opera com metade da sua capacidade (logo, metade do seu valor).

O Modelo foi aplicado em dois cenários diferentes:

Atualização integral :

Necesse cenário a atualização foi aplicada na íntegra no final de 9 meses.

Atualização gradual :

Necesse cenário a atualização foi aplicada gradualmente ao longo dos 9 meses. As 50 modificações foram divididas em 18 atualizações. As duas primeiras atualizações possuem 1 modificação cada e aumentam em 2% o valor do sistema. A escolha dessas duas atualizações mais simples no início é devido a informação obtida através da equipe do OpenBus que após a implantação é feito um planejamento das funcionalidades da próxima versão, logo não teriam muitas funcionalidades no primeiro mês após a última atualização. As outras 16 atualizações possuem 3 modificações cada e aumentam em 6% o valor do sistema.

5.3.3 Resultados

Após a definição dos parâmetros foi medido o valor do sistema aplicando o modelo. Na figura 5.11, podemos verificar a alteração do valor do sistema após uma única atualização. Na figura é apresentado os valores iniciando em 1 hora antes da aplicação da atualização dinâmica e até 1 hora depois da aplicação da atualização estática, tempo total de 26 horas e 35 minutos. Analisando esta imagem podemos perceber que nesse curto espaço de 26 horas, a atualização dinâmica se torna vantajosa. Porém, não podemos esquecer, que esta atualização demorou aproximadamente 6.696 horas (9 meses) para ser aplicada. Como podemos verificar na figura 5.12, a vantagem que a atualização dinâmica tem no curto prazo é diluída.

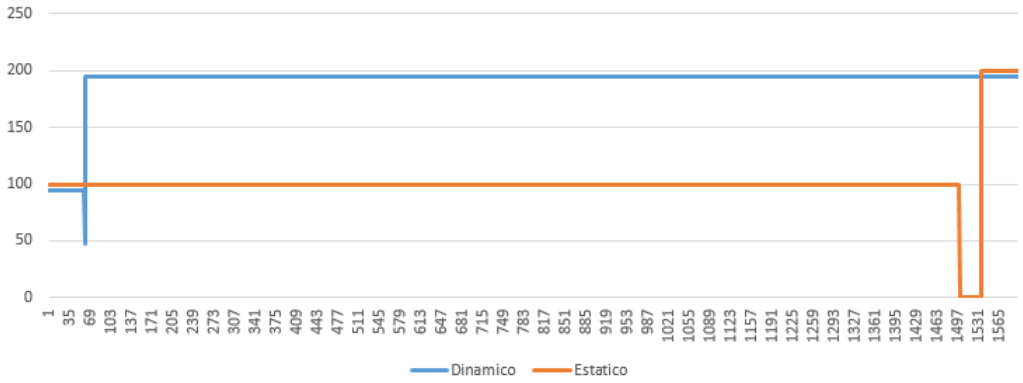

Figura 5.11: 26 horas. Aplicação do modelo para atualização 4 (1.5.2 para 1.5.3) utilizando Toa= 1 minuto, Tfa=35 minutos e Toff = 24 horas

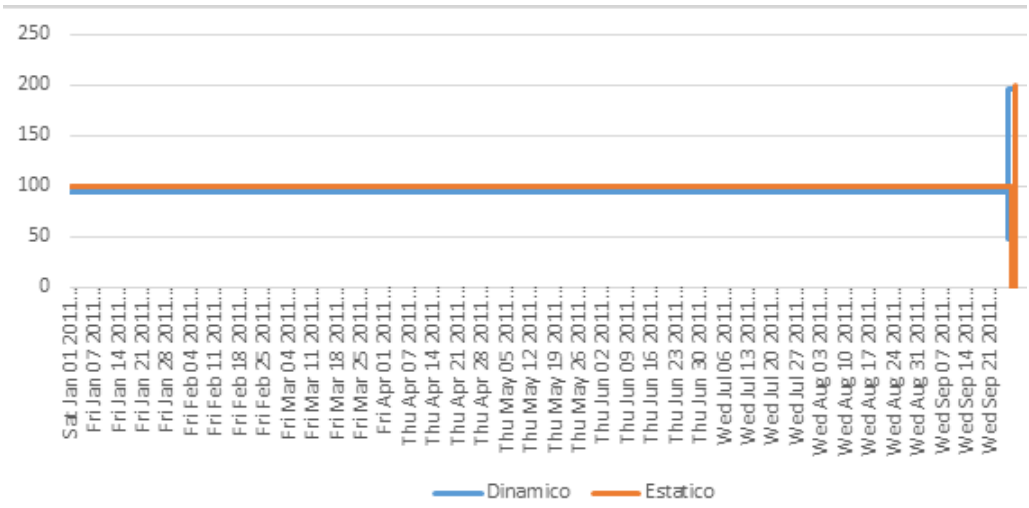

Figura 5.12: 9 Meses. Aplicação do modelo para atualização 4 (1.5.2 para 1.5.3) utilizando Toa= 1 minuto, Tfa=35 minutos e Toff = 24 horas

Na figura 5.13, podemos verificar o valor do sistema após sucessivas modificações.

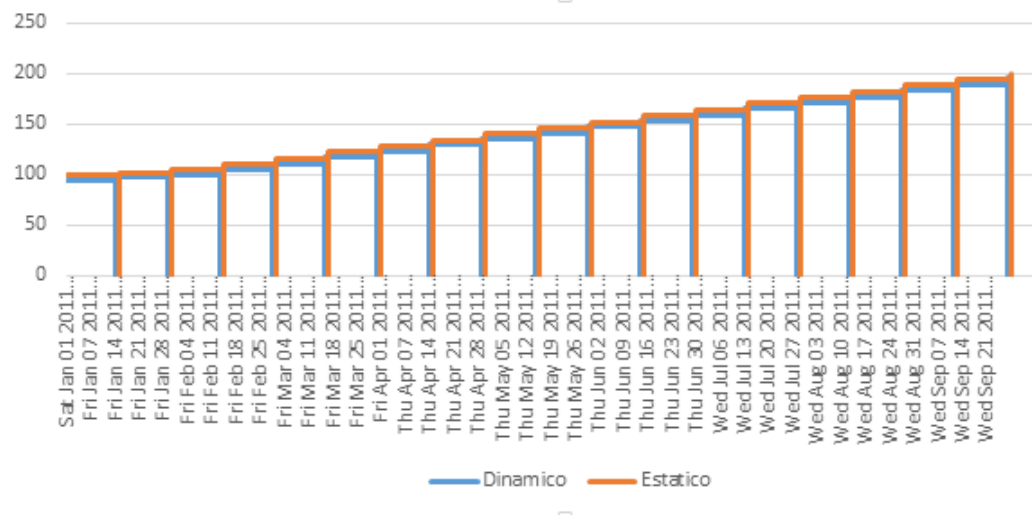

Figura 5.13: Aplicação do modelo para atualização 4 (1.5.2 para 1.5.3) Toa= 1 minuto, Tfa=35 minutos e Toff = 24 horas

Toa e Tfa	Dinâmico	Estático	$\vec{\Delta}$	%
Integral 26 horas	304.877	162.000	142.877	+88,20%
Integral 9 meses	37.224.248	38.736.100	1.511.852	-3,90%
Gradual	54.059.947	55.767.290	1.707.343	-3,06%

Tabela 5.1: Valores dos sistemas do inicio das atualizações até 1 mês depois de terminada a aplicação. 01/01/2011 até 01/11/2011

Observando a tabela 5.1, podemos verificar que as diferenças nos valores finais dos sistemas são expressivas. Na atualização integral, se analisarmos somente as 26 horas demonstradas na figura 5.11 vemos que a atualização dinâmica traz uma vantagem de 88% em relação a atualização estática, porém, a cada minuto que passa, devido a sobrecarga do mecanismo, o sistema deixa de ganhar 5 pontos de receita, levando aproximadamente 20 dias para zerar esta diferença. No final de 9 meses a diferença em percentual da atualização estática para a atualização dinâmica é de aproximadamente 3,9%. Na atualização gradual a diferença é um pouco menor, de 3,06%. A atualização estática é mais vantajosa em 1.707.343 pontos, isso significa que se a sobrecarga fosse de aproximadamente 2%, não haveria perda de receita. Logo podemos concluir que, apesar da sobrecarga do mecanismo ser baixa e aceitável, ela ainda tem um impacto muito grande no longo prazo, sugerindo uma nova investigação para tentar diminuir a sobrecarga do mecanismo.

Esta aplicação do modelo não leva em consideração multas de SLA ou perda de confiabilidade por falta de serviço, o que traria uma vantagem para a atualização dinâmica. Então é válido lembrar que o OpenBus integra cerca de 126 sistemas e em alguns cenários uma perda de 4% é preferível a um impacto maior na operação do sistema ou a uma queda em sua confiabilidade.

Outro resultado que podemos verificar é que, apesar das diferenças da atualização dinâmica e estática de 3,06% e 3,9% serem próximas, os valores agregados dos sistemas por sua vez, são bem distantes. Isso mostra que a atualização gradual, seja ela estática ou dinâmica é bem mais vantajosa do que a atualização integral, pois ela antecipa os lucros resultante das atualizações. Uma dúvida natural que aparece sobre atualizações graduais é o risco de comprometer o funcionamento de sistemas com requisitos rígidos de disponibilidade. O trabalho de Dumitras et al.[22] apresenta um estudo que categoriza os riscos e problemas que ocorrem no processo de atualização dinâmica e também propõe e avalia um framework para diminuir estes riscos.

5.4 Considerações Finais

Neste capítulo nós demonstramos expressividade das interfaces do mecanismo de atualização dinâmica utilizando as atualizações da versão 1.4 para 1.5 e da 1.5 para 2.0 do OpenBus.

Depois mostramos os resultados dos testes de sobrecarga, resultando em uma sobrecarga extra de 5% de processamento devido a interceptação e identificação do estado do componente. Descrevemos também o modelo de avaliação quantitativo e aplicamos o modelo no mecanismo. Os resultados da aplicação do modelo de avaliação se resumem em três pontos: a disposição de aceitar a perda de receita para não ficar offlina; a necessidade de diminuir a sobrecarga do mecanismo; e a atualização gradual de sistemas é mais vantajosa independente de ser estática ou dinâmica.

6
Trabalhos Relacionados

O capítulo 6 apresenta os trabalhos relacionados de atualização dinâmica e faz uma avaliação comparativa do mecanismo implementado com as abordagens apresentadas nos trabalhos relacionados, utilizando o arcabouço de avaliação gerado a partir do OpenBus; Os trabalhos escolhidos apresentam mecanismos de atualização dinâmica e arquiteturas que possibilitem a atualização e manutenção de múltiplas versões. Apesar do mecanismo desenvolvido neste trabalho não propor uma solução para manutenção de múltiplas versões, ele provê as funcionalidades básicas para atualização dinâmica e permite que o desenvolvedor estenda-o para outros fins. Existe um trabalho, utilizando o SCS[23], que implementa um suporte para múltiplas versões que pode ser integrado ao mecanismo desenvolvido afim de ser aplicado ao OpenBus.

6.1
Ginseng

O Ginseng[13] é um mecanismo composto de 3 entidades: um compilador, um gerador de patches e um sistema de execução. Para começar, o programa tem uma versão inicial (v0.c), então o compilador gera um executável atualizável (v0) junto com alguns meta dados dessa versão inicial (d0). Primeiro, o programa na versão 0 começa a ser executado. Quando o programa muda para uma nova versão (v1.c), o programador fornece o código novo (v1.c), o código antigo (v0.c) e os meta dados da versão em execução (d0) para o gerador de patches, que gera o patch (p1.c). Este representa as diferenças entre as versões. O patch criado (p1.c) é enviado junto com os meta dados da versão em execução (d0) para o compilador que gera um patch dinâmico a fim de que o sistema de execução carregue o patch dinâmico no programa em execução completando a atualização dinâmica.

A escolha do momento certo em que o sistema de execução irá carregar o patch dinâmico é delegada aos programadores da aplicação. Esses momentos são definidos como pontos seguros (safe-points). O Mecanismo disponibiliza uma primitiva DSUUpdate() que representa uma tentativa de atualização caso exista um patch a ser aplicado no sistema. Os autores avisam que tratamento

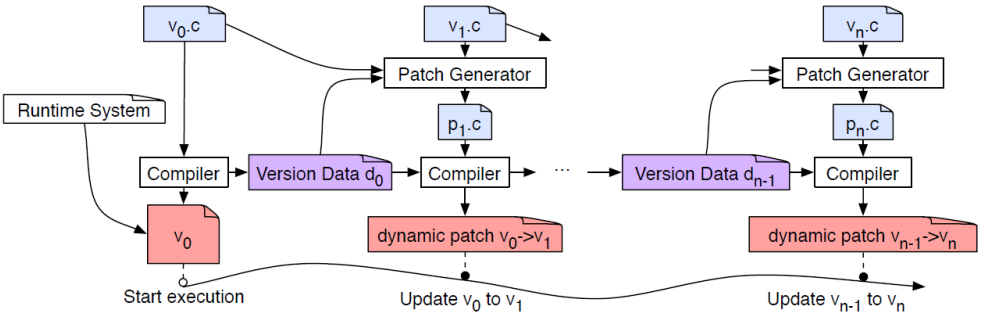

Figura 6.1: Processo de Atualização do Ginseng

de sinais pode ser utilizado para forçar uma atualização a qualquer momento, mas advertem que pode haver inconsistência caso a atualização altere alguma função que esteja sendo executada.

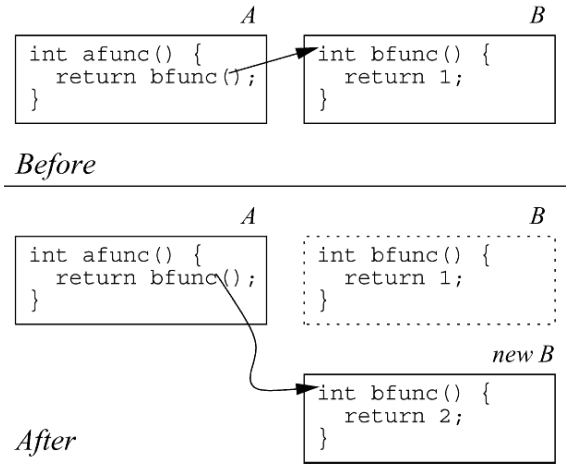

Figura 6.2: Atualização de uma função B no Ginseng.

A granularidade das atualizações no Ginseng é bem pequena visto que atualizações são aplicadas a módulos individualmente e podemos alterar funções e tipos estruturados. Com essa granularidade existe a possibilidade de alterarmos um método, incluindo um novo parâmetro. Caso aconteça isso, os códigos que utilizam esse método precisam também ser alterados para incluir o novo parâmetro, ou poderá haver alguma inconsistência no sistema.

Quanto a granularidade, o mecanismo que desenvolvemos se assemelha ao Ginseng, pois a faceta de um componente pode ser considerada um módulo individual de um programa. Porém o mecanismo desenvolvido não permite a alteração da interface de métodos. A fim de evitar a quebra da comunicação com códigos que utilizem aquela faceta, uma decisão de projeto foi impossibilitar a alteração da interface da faceta. No caso de ser necessário alterar a interface de uma faceta, também será necessário alterar os clientes que utilizam aquela faceta. Como os clientes deverão ser alterados, eles podem ser alterados

para utilizar uma nova faceta com uma nova interface, ao invés de alterar a interface da faceta já existente.

Diretrizes	Ginseng
Continuidade do serviço	A execução do sistema é interrompida enquanto ele está sendo atualizado.
Melhor momento de atualização	Definição de Safe-Point, a decisão da escolha do melhor momento é do desenvolvedor.
Restauração ou transferência de estado	O estado é transferido utilizando uma função de transformação do estado para a nova versão.
Alteração de interface ou comportamento	Permite a alteração de interface, porém é necessário atualizar também todos os pontos que serão afetados pela alteração.
Flexibilidade	Permite alteração a nível de funções (Linguagem C)
Robustez	Checagem estática de código pelo gerador de atualizações.
Facilidade de Uso	Gerador automático de atualizações utilizando código antigo e código novo
Baixa sobrecarga	A sobrecarga do Ginseng pode variar de 0 a 32% de acordo com o tipo de atualização

Tabela 6.1: Resumo do Ginseng

6.2 JVOLVE

Bem similar ao Ginseng (DSU), o JVOLVE[24] atualiza o sistema em execução através de patches gerados com diff entre o código da versão nova e o código da versão antiga. O Mecanismo utiliza a ferramenta UPT (Update Preparation Tool) para gerar os patches que serão aplicados no sistema. No JVOLVE os autores estenderam uma máquina virtual Java (Jikes RVM) programada em Java para permitir adicionar, editar e remover classes, métodos, campos e alterar definições de interfaces de classes e métodos. A única res-

trição de alteração que o JVOLVE tem é a permutação na hierarquia das classes. Assim como o Ginseng, o JVOLVE utiliza o modelo de pontos seguros (safe-point,), porém ele tenta inferir quais seriam os pontos seguros da aplicação.

O JVOLVE entende como ponto seguro o fato da máquina virtual não estar executando nenhum método restrito. E método restrito pode ser identificado pelos casos a seguir: métodos alterados na atualização e métodos especificados pelo usuário.

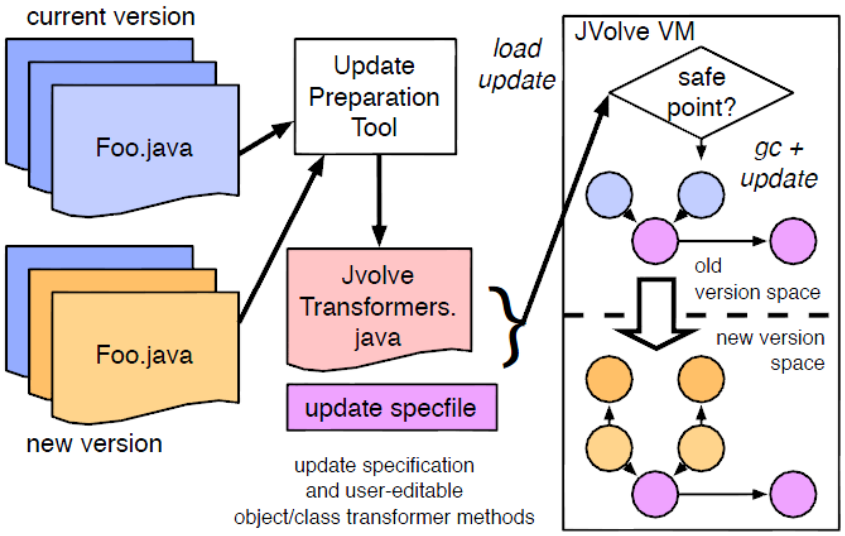

Figura 6.3: Processo de Atualização do JVOLVE.

Ao gerar os patches de atualização, a máquina virtual espera um ponto seguro(safe-point). Quando esse ponto seguro é encontrado, a máquina virtual dispara o coletor de lixo (*garbage collector*) e aplica todas as modificações necessárias para a atualização.

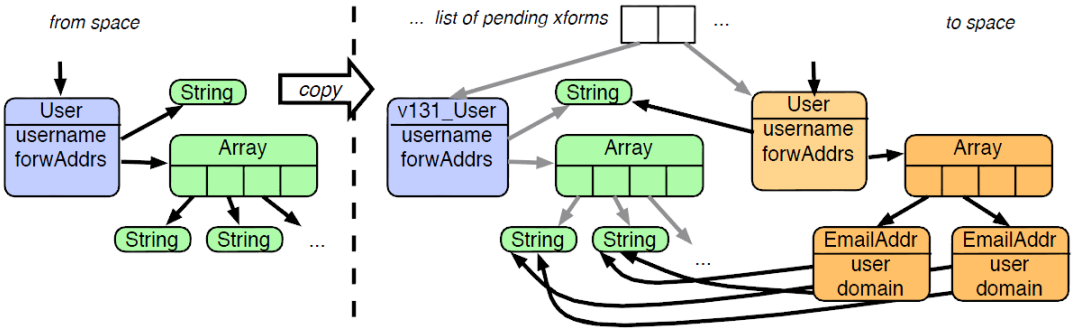

Figura 6.4: Atualização de uma Classe no JVOLVE.

O ponto seguro demonstrado tanto no JVOLVE quanto no Ginseng se apresenta como a opção mais coerente de iniciar a atualização. O desenvolvedor da aplicação é quem detém o maior conhecimento sobre a implementação do sistema e por isso a decisão de delegar a ele a marcação dos pontos

seguros para se aplicar a atualização é ideal. O mecanismo implementado se assemelha a essa decisão no momento que permite que o desenvolvedor estenda a faceta ILyfeCycle e determine se é possível aplicar a atualização. Na implementação da função changeState(), que representa a transição para o início da atualização, o desenvolvedor pode fazer a checagem do estado do componente e verificar se ele se encontra em um ponto seguro para atualizar.

6.3 Imago

Os autores do Imago[22] apresentam o mecanismo como uma saída para atualizar dinamicamente e evitar os erros de configuração e erros no procedimento de atualização. Para alcançar isso, o Imago consiste em instalar a nova versão em um universo paralelo (Unew, uma coleção de recursos diferente da utilizada pela versão nova, seja com hardware diferente ou através de virtualização) e transferindo os dados persistidos, oportunamente, para a nova versão(Unew). O mecanismo foi projetado para isolar o sistema em produção (Uold) da atualização (Unew). Quando a sincronização dos dados persistidos do universo novo (Unew) com o universo antigo (Uold) termina, o Imago faz a transição do universo antigo (Uold) para o universo novo (Unew).

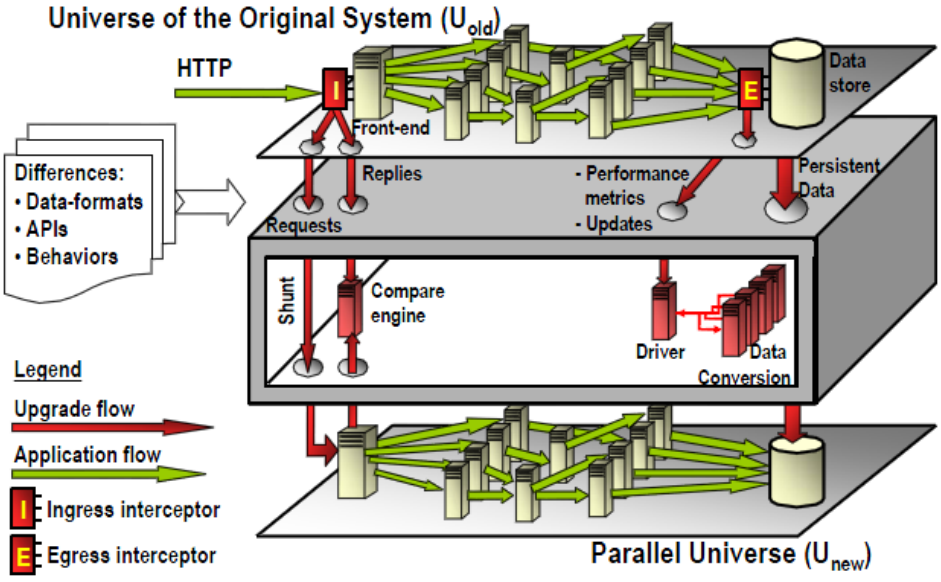

Figura 6.5: Processo de Atualização do Imago.

O Mecanismo assume alguns pontos bem fortes:

Pontos de Ingresso (I) e Egresso (E) fixos e conhecidos

A Carga de trabalho do sistema é baseada na sua maioria em requisições somente de leitura

O Sistema provê ganchos para descarregar atualizações em andamento e mecanismos para ler objetos de dados armazenados no universo antigo (Uold) sem atrapalhar o fluxo natural.

De acordo com os autores, os pontos fortes do Imago são: Isolamento; Atomicidade e Fidelidade. O isolamento é obtido através do universo paralelo (Unew) criado para a nova versão. Assim, a atualização não interfere na versão que está em execução (Uold). A atomicidade é obtida, pois como o sistema é baseado na sua maioria em requisições de leitura, o sistema consegue sincronizar todos os dados persistidos e fazer o chaveamento do universo antigo (Uold) para o universo novo (Unew) e não ter as duas versões rodando ao mesmo tempo. A Fidelidade é obtida com um estado de teste. Quando os dados são sincronizados, o sistema vai para o estado de teste, que ele pode utilizar tanto uma carga de teste preparada, quanto as chamadas reais do sistema para comparar a saída gerada pelo universo antigo (Uold) com o universo novo (Unew) e ver se a atualização é fiel ao sistema antigo.

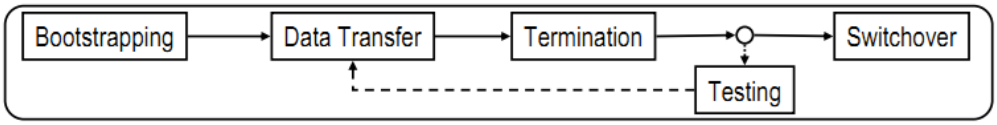

Figura 6.6: Processo de chaveamento do Imago

O Imago assume que os dados persistidos representam o estado do sistema, e que a nova versão não precisa de nenhuma outra informação do sistema em execução. A implementação do Imago consiste em 4 principais módulos: Update driver, responsável por transferir dados do universo antigo(Uold) para o universo novo(Unew); Compare Engine, que compara as saídas geradas na fase de testes; E interceptor, que monitora o log do banco de dados para a sincronização dos estados; e o I interceptor, que intercepta as chamadas para o sistema.

Esse mecanismo é um dos poucos que inclui a fase de testes em seu processo de atualização. Apesar de muito importante, a fase de testes é ocasionalmente deixada de lado pelos mecanismos de atualização dinâmica. Em sua maioria, os mecanismos de atualização dinâmica, assumem que as atualizações já foram testadas antes de serem aplicadas aos sistemas em funcionamento.

Os pontos de ingresso e egresso fixos e bem definidos são características compartilhadas pelos componentes de software. Como a atualização no Imago consiste em sincronizar o estado da aplicação atual com o estado da aplicação futura, testar a aplicação e depois substituir a aplicação antiga pela aplicação nova, para simular este processo precisaríamos desenvolver alguma faceta extra

do componente que permita a leitura dos dados armazenados por ele e também incluir algum mecanismo extra de teste de componentes.

6.4 Upstart

O Upstart[25] é uma técnica de atualização dinâmica, cujo principal objetivo é prover tolerância a falha para os objetos do sistema. Os autores caracterizam o sistema distribuído como uma coleção de objetos que se comunicam através de RPC (Remote Procedure Call). A atualização envolve seis componentes: oldClass, a versão antiga do objeto a ser atualizado; newClass, a versão nova do objeto; TF, uma função de transformação que gera o estado da versão nova a partir do estado da versão velha; SF, uma função de agendamento que identifica o momento certo de aplicar a atualização; pastSO, um objeto que dá suporte a uma versão antiga do objeto; e futureSO, um objeto que dá suporte a uma versão futura do objeto. Caso o estado do objeto não seja importante, a função de transformação pode ser omitida. Se o Sistema não permite suporte a versões passadas ou futuras, os objetos de simulação podem ser omitidos também.

Os objetos de simulação que dão suporte a versão passada e a versão futura são simplesmente objetos que adaptam a chamada dos métodos e delegam a execução para a versão certa no caso de haver uma atualização que altere a interface do objeto. O Sistema insere um proxy que recebe todas as chamadas para aquele objeto. Independente de versão, o objeto só possui 1 ponto de entrada. Todas as chamadas de objetos são rotuladas com a versão. Assim, o proxy consegue identificar se deve enviar para um objeto de simulação ou se deve enviar para o objeto real.

Um método de atualização ocorre da seguinte maneira: No início existe um objeto na versão 1. Ao iniciar o processo de atualização é instalado um objeto de simulação futuro para tratar chamadas para a versão 2. Após o término da atualização, é criado um objeto de simulação da versão passada para tratar chamadas da versão 1 e a versão do objeto é chaveada da 1 para a 2.

O Modelo ainda permite que o suporte a versões antigas sejam desligados de acordo com a necessidade do administrador do sistema. Com os mecanismos de objetos futuros e passados, o sistema consegue dar suporte a diferentes versões do mesmo objeto ainda que a interface deste seja alterada sem precisar atualizar todos os clientes que o utilizam.

O Upstart é um modelo que, conceitualmente, se adequaria perfeitamente a necessidade do OpenBus. O OpenBus possui a necessidade de manter a

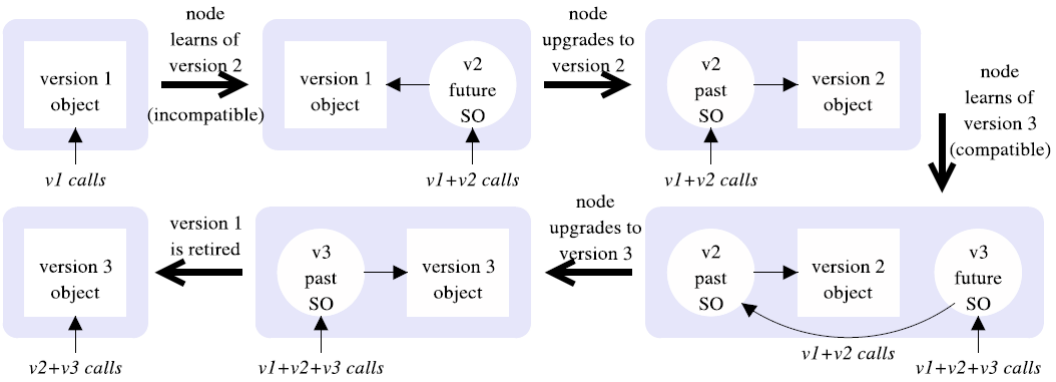

Figura 6.7: Processo de Atualização do Upstart

compatibilidade com a última versão. Essa necessidade poderia ser sanada utilizando os objetos de simulação para manter a compatibilidade com a versão anterior. A função de transformação utilizada pelo Upstart pode ser comparada ao código executado durante a atualização no mecanismo desenvolvido nesse trabalho. O propósito desse código é resgatar o estado corrente do componente e persistir as informações necessárias, exatamente o objetivo da função de transformação.

6.5 PKUAS

O PKUAS[26] se baseia em uma gerência de versão em tempo de execução. Para isso todas as chamadas possuem dois campos necessários: a versão do componente; e uma flag que indica se a versão é necessária. Assim como no Upstart, todo componente só possui um ponto de entrada, independente de versão, definido como Skeleton. Ao receber uma chamada, o Skeleton do componente encaminha essa chamada para a versão correspondente.

O PKUAS possui dois métodos de atualização possíveis:

Atualização forçada: Nessa política, todas as instâncias do componente serão atualizadas instantaneamente e no final todas as versões antigas serão destruídas. Para componentes em que o estado não importa, as chamadas em andamento continuam em execução, as novas chamadas são bloqueadas e quando as chamadas em andamento terminarem, todas as instâncias da versão antiga são destruídas e a versão nova inicia a execução. Para componentes em que o estado não pode ser ignorado, o mecanismo espera o estado quiescente, que pode ser definido pelo usuário no ciclo de vida do componente durante a implementação. Quando o estado é alcançado, utiliza uma função de transformação para recuperar o estado do componente na versão antiga e gera o estado da versão nova.

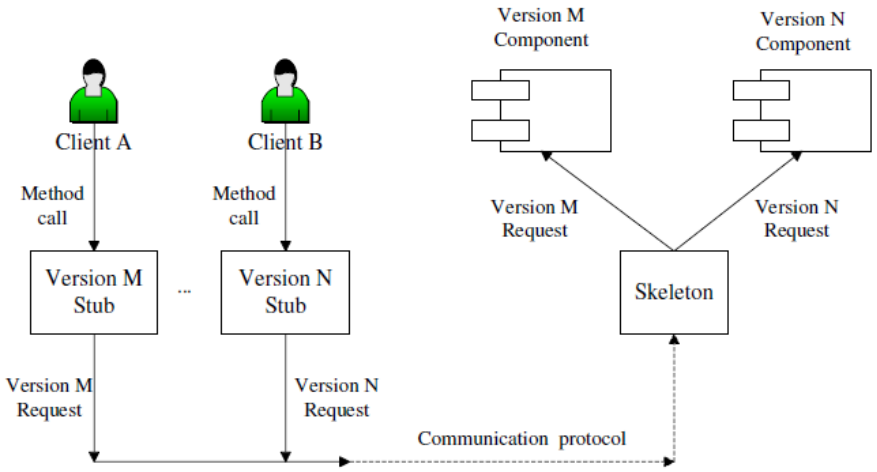

Figura 6.8: Gerência de versão em tempo de execução do PKUAS

Com isso, todas as instâncias das versões antigas são destruídas e a versão nova inicia a execução.

Atualização gradual: Nessa política, todos as instâncias do componente continuam rodando sem afetar os seus clientes. Chamadas de versões antigas são direcionadas para versões antigas. Quando uma chamada para uma versão nova ocorrer, a chamada é encaminhada para a versão nova. Administradores do sistema podem desligar versões antigas quando elas não estão executando chamadas. Um detalhe interessante no PKUAS é o flag enviado pelo cliente que identifica se a versão é necessária ou não. Caso a versão não seja necessária e não esteja em funcionamento, a versão mais nova que for compatível com aquela chamada responde e envia para o cliente a versão que respondeu aquela chamada. Assim, o cliente pode tomar a decisão de atualizar ou não o seu código.

Esse modelo se apresenta como a solução mais completa de atualização dinâmica. Ele se assemelha ao mecanismo desenvolvido em dois aspectos: possibilita o desenvolvedor definir qual é o melhor momento no ciclo de vida do componente para atualizar; e definir uma função de transformação para recuperar e persistir o estado do componente que está sendo atualizado. O PKUAS também leva em conta a convivência de múltiplas versões. Apesar do mecanismo desenvolvido nesse trabalho não possuir nenhuma gerência específica para múltiplas versões, ele possibilita a atualização do componente e deixa essa decisão de como gerenciar as múltiplas versões para o desenvolvedor.

6.6 Considerações Finais

Neste capítulo apresentamos alguns dos trabalhos relevantes de atualização dinâmica apresentados na literatura. Fizemos também uma comparação dos trabalhos apresentados com o trabalho desenvolvido e com o OpenBus, procurando pontos de semelhança e de diferenças. Podemos verificar que todos os trabalhos permitem alguma funcionalidade para a recuperação de estado dos sistemas durante a atualização, porém o Imago só permite a recuperação do estado que está persistido e ignora as informações em memória do sistema em execução. Vimos também que o GINSENG, o JVOLVE e o PKUAS deixam a decisão de escolher o melhor momento para atualizar nas mãos do desenvolvedor, seguindo o mesmo intuito deste trabalho. E por último podemos verificar que o Upstart e o PKUAS apresentam soluções de gerência de múltiplas versões que podem ser aplicadas no caso do OpenBus, que sempre precisa prover compatibilidade com uma versão anterior.

Diretrizes	JVOLVE
Continuidade do serviço	A execução do sistema é interrompida enquanto ele está sendo atualizado.
Melhor momento de atualização	Definição de Safe-Point, a decisão da escolha do melhor momento é do desenvolvedor.
Restauração ou transferência de estado	O estado é transferido utilizando uma função de transformação do estado para a nova versão.
Alteração de interface ou comportamento	Permite a alteração de interface, porém é necessário atualizar também todos os pontos que serão afetados pela alteração.
Flexibilidade	Permite alteração a nível de classe e métodos (Linguagem Java) e não permite alteração de hierarquia de classes.
Robustez	Checagem estática de código pelo gerador de atualizações.
Facilidade de Uso	Gerador de atualizações automático utilizando código antigo e código novo.
Baixa sobrecarga	A sobrecarga imposta pelo JVOLVE está associada ao garbage collection e ao carregamento de classes, já contidos na linguagem Java.

Tabela 6.2: Resumo do JVOLVE

Diretrizes	Imago
Continuidade do serviço	A Execução continua até que seja possível alterar as chamadas para o novo universo paralelo.
Melhor momento de atualização	Espera o estado quiescente para aplicar a atualização.
Restauração ou transferência de estado	Possui um ciclo de sincronização que recupera e restaura o estado do sistema.
Alteração de interface ou comportamento	Não permite a alteração da interface.
Flexibilidade	Substituição obrigatória de todo os sistema, criando um universo paralelo.
Robustez	Possui uma fase de testes que visa garantir a robustez da atualização.
Facilidade de Uso	Não possui nenhum facilitador para geração automática de atualização.
Baixa sobrecarga	Por criar um Universo paralelo o Imago não causa nenhuma sobrecarga, porém, necessita de uma infra-estrutura inteiramente nova pois não pode compartilhar nenhum recurso com o sistema que vai ser atualizado.

Tabela 6.3: Resumo do Imago

Diretrizes	Upstart
Continuidade do serviço	A execução do sistema é interrompida.
Melhor momento de atualização	Interrompe a execução e utiliza o último estado válido persistido.
Restauração ou transferência de estado	Possui um mecanismo de tolerância a falha que frequentemente persiste o estado do sistema.
Alteração de interface ou comportamento	Permite a alteração de interface e chamadas a interface antiga são direcionadas para suas devidas versões.
Flexibilidade	Permite atualizações a nível de objeto, a nova versão é constituída de um novo objeto que tomará o lugar do objeto da versão antiga.
Robustez	Não há nenhuma informação sobre robustez.
Facilidade de Uso	Possui um gerenciador de versões que permite a a convivência de múltiplas versões
Baixa sobrecarga	Não há informações sobre o impacto deste mecanismo ou se há sobrecarga de processamento.

Tabela 6.4: Resumo do Upstart

Diretrizes	PKUAS
Continuidade do serviço	A execução do sistema é mantida até que não seja necessária a versão antiga.
Melhor momento de atualização	Espera o estado quiescente ou interrompe a execução.
Restauração ou transferência de estado	Utiliza funções de transformação que podem recuperar o estado do componente antigo e aplicá-lo na nova versão.
Alteração de interface ou comportamento	Permite a alteração de interfaces e mantém a compatibilidade com múltiplas versões
Flexibilidade	As atualizações são feitas a nível de componente, sendo toda versão nova é uma nova instância de componente criada.
Robustez	Não há nenhuma informação sobre robustez.
Facilidade de Uso	Possui um gerenciador de versões que permite a a convivência de múltiplas versões.
Baixa sobrecarga	Não há informações sobre o impacto deste mecanismo ou se há sobrecarga de processamento.

Tabela 6.5: Resumo do PKUAS

7
Conclusão

Apresentamos neste trabalho um estudo sobre atualizações dinâmicas em sistemas de componentes de software. Nesse estudo analisamos os principais tipos de modificações que ocorrem nos sistemas de componentes em produção. São elas: A) Bugfix, B) Melhorias, otimizações e alterações de comportamento, C) Novas funcionalidades previstas na interface, D) Novas funcionalidades não previstas na interface, E) Atualizações estruturais ou operacionais, F) Alteração de dependência externa, G) Alteração em frameworks e H) Alteração do ambiente de execução. Para identificar esses tipos de modificações utilizamos o histórico das evoluções do sistema OpenBus ao longo de 5 anos de operação.

Após a classificação das modificações, nós desenvolvemos um mecanismo de atualização dinâmica que acomodasse as atualizações apresentadas no OpenBus. Para demonstrar a expressividade das interfaces do mecanismo, ele foi utilizado para simular e experimentar algumas das atualizações que o OpenBus sofreu durante esses anos. O mecanismo implementado foi submetido a um teste de sobrecarga de processamento para verificar o impacto que ele causa nas interações entre os componentes. Identificamos uma sobrecarga de processamento de 5% no tempo de execução.

Com o resultado do teste de sobrecarga utilizamos um modelo quantitativo para verificar o custo e benefício de utilizar a atualização dinâmica no OpenBus. Nos resultados da aplicação do modelo pudemos verificar que é viável a utilização do modelo de atualização dinâmica e que a sobrecarga imposta pelo mecanismo é bem pequena.

Por último fizemos uma avaliação comparativa com os trabalhos relacionados e o mecanismo proposto, resumindo os pontos fortes dos trabalhos e suas semelhanças com o mecanismo implementado.

Podemos citar como principal contribuição deste trabalho a definição e validação das interfaces definidas para atualização dinâmica dos componentes. Junto com esta contribuição segue a implementação em Lua das interfaces apresentadas. Outras contribuições contidas neste trabalho são: o levantamento dos tipos de modificações que um sistema de componente de software em

produção sofre; e a análise comparativa dos trabalhos relacionados com o mecanismo desenvolvido.

Apesar do mecanismo de atualização dinâmica desenvolvido neste trabalho ter sido suficiente para a simulação e aplicação das atualizações apresentadas no OpenBus, ainda existem diversas funcionalidades que precisam ser exploradas a fim de aprimorar este mecanismo. A seguir segue a lista desses aprimoramentos que serão incorporados no mecanismo no futuro:

Otimização do mecanismo :

Na aplicação do modelo quantitativo verificamos que se a sobrecarga do mecanismo fosse de 2%, ao invés de 5%, não haveria perda de receita. Existem alguns pontos que podem ser otimizados no interceptador para obtermos uma sobrecarga menor. Podemos também implementar uma política que só ativará o interceptador em momentos de atualização, retirando integralmente a sobrecarga imposta pelo mecanismo.

Empacotamenteo da atualização :

O mecanismo atual permite que seja enviado o novo código de implementação da faceta, porém, as dependências, precisam estar na máquina em que o componente está executando, no momento da atualização. É necessário a implementação de algum sistema de empacotamento da atualização para que seja possível descrever as dependências para que elas sejam resolvidas na hora da aplicação da atualização. Já existe uma infraestrutura de implantação[27] desenvolvida para o SCS que pode ser integrada ao mecanismo desenvolvido afim de resolver esta necessidade.

Alteração de interface compatível :

A implementação atual do mecanismo não permite a alteração da interface da faceta atualizada. Essa decisão foi tomada para evitar erros de atualização, porém dado a natureza das chamadas remotas feitas ao componente, poderíamos permitir a atualização da interface caso seja mantida a compatibilidade estrutural [15].

Histórico de atualização :

Apesar do mecanismo apresentado neste trabalho permitir a recuperação da implementação da faceta para verificar qual a implementação atual, ele não possui nenhuma facilidade para verificar o histórico de atualizações e qual a versão exata daquela faceta está rodando. É necessário desenvolver alguma integração com o controle de versão para permitir

uma clara visualização de qual versão está executando em um determinado momento.

Integração contínua :

Como apresentado nos resultados do modelo quantitativo 5.3.1, a aplicação das atualizações a medida que as novas funcionalidades estão sendo desenvolvidas é uma das maiores vantagens da atualização dinâmica. Porém, a aplicação das atualizações podem causar erros ou problemas de integração entre componentes. A Integração contínua consiste na prática de testes e implantações automáticas que permitam que as atualizações sejam aplicadas mais frequentemente e sem causar erro.

8
Referências Bibliográficas

[1] TAYLOR, R. N.; MEDVIDOVIC, N.; OREIZY, P. Architectural styles for runtime software adaptation. **2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture**, Ieee, p. 171–180, set. 2009. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5290803>. 1, 2.2, 2.2.1

[2] KRAMER, J.; MAGEE, J. The Evolving Philosophers Problem : Dynamic Change Management . n. November 1990, p. 1–33, 1991. 1, 2.2.1

[3] HICKS, M.; NETTLES, S. Dynamic software updating. **ACM Transactions on Programming Languages and Systems**, v. 27, n. 6, p. 1049–1096, nov. 2005. ISSN 01640925. Disponível em: <http://portal.acm.org/citation.cfm?doid=1108970.1108971>. 1, 2.2.1

[4] TECGRAF. **Openbus - enterprise integration application middleware.** 2006. Disponível em: <http://jira.tecgraf.puc-rio.br/confluence/display/OpenBus/Home>. 1.1, 4

[5] GHARAIBEH, B.; RAJAN, H.; CHANG, J. Analyzing software updates: should you build a dynamic updating infrastructure? **Fundamental Approaches to Software**, 2011. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-19811-3_26>. 1.1, 1.2, 5, 5.3

[6] MWLAB: MIDDLEWARE LABORATORY PUC-RIO. **SCS: Software Component System**. 2009. Disponível em: <http://www.tecgraf.puc-rio.br/~scorrea/scs.>. 2, 2.2.3

[7] BRUNETON, E. et al. The F RACTAL component model and its support in Java. p. 1257–1284, 2006. 2

[8] OMG: OBJECT MANAGEMENT GROUP. **CORBA component model**. 2006. Disponível em: <http://www.omg.org/technology/documents/formal/components.htm>. 2, 2.1, 2.1, 2.2.1, 2.2.2

[9] MICROSOFT. **COM - Componet Object Model Technology**. Disponível em: <http://www.microsoft.com/com>. 2, 2.1

[10] COULSON, G. et al. A component model for building systems software. In: **IASTED Conf. on Software Engineering and Applications**. [S.l.: s.n.], 2004. p. 684–689. 2

[11] AUGUSTO, C. E. L. et al. SCS - Sistema de Componentes de Software. Departamento de Informática - Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). 2007. Disponível em: <https://jira.tecgraf.puc-rio.br/confluence/download/attachments/18612229/scsOverview.pdf>. 2, 2.1

[12] OREIZY, P.; GORLICK, M. An architecture-based approach to self-adaptive software. **Systems and Their**, 1999. 2.2

[13] NEAMTIU, I. et al. Practical dynamic software updating for C. In: **Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation - PLDI '06**. New York, New York, USA: ACM Press, 2006. p. 72 – 83. ISBN 1595933204. Disponível em: <http://portal.acm.org/citation.cfm?doid=1133981.1133991>. 2.2.1, 6.1

[14] NETTLES, S. M.; TANNEN, V. DYNAMIC SOFTWARE UPDATING. 2001. 2.2.1

[15] CERQUEIRA, R. **Um Modelo de Composiçao Dinâmica entre Sistemas de Componentes de Sofware**. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, 2000. 2.2.1, 7

[16] IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; CELES, W. The evolution of Lua. **Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III**, ACM Press, New York, New York, USA, p. 2–1–2–26, 2007. Disponível em: <http://portal.acm.org/citation.cfm?doid=1238844.1238846>. 2.2.1

[17] CATUNDA, M.; RODRIGUEZ, N.; IERUSALIMSCHY, R. Dynamic extension of CORBA servers. **Euro-Par'99 Parallel Processing**, 1999. Disponível em: <http://medcontent.metapress.com/index/A65RM03P4874243N.pdf http://link.springer.com/chapter/10.1007/3-540-48311-X_192>. 2.2.1

[18] MOURA, A. de; URURAHY, C. Dynamic support for distributed auto-adaptive applications. **Distributed Computing Systems**

Workshops, 2002. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1030811>. 2.2.1

[19] FIGUEIRó, R. **Um Framework para Adaptaçao Dinâmica de Sistemas Baseados em Componentes Distribuídos**. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, 2004. 2.2.2

[20] PORTILHO, E. **Um Estudo de Técnicas para a Adaptaçao de Componentes de Software em Java**. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, 2008. 2.2.3

[21] LuaJava - A script tool for java. Disponível em: <http://www.keplerproject.org/luajava>. 2.2.3

[22] DUMITRAS, T.; NARASIMHAN, P. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system. **Proceedings of the 10th ACM/IFIP/USENIX**, 2009. Disponível em: <http://dl.acm.org/citation.cfm?id=1657005>. 5.3.3, 6.3

[23] ROENICK, H. Atualização Dinâmica de Componentes de Software. p. 1–12, 2009. 6

[24] SUBRAMANIAN, S.; HICKS, M.; MCKINLEY, K. S. Dynamic Software Updates : A VM-centric Approach. n. 1. 6.2

[25] AJMANI, S.; LISKOV, B.; SHRIRA, L. Modular Software Upgrades for Distributed Systems. 6.4

[26] XIAO, Z. et al. Towards Dynamic Component Updating: A Flexible and Lightweight Approach. **2009 33rd Annual IEEE International Computer Software and Applications Conference**, Ieee, p. 468–473, 2009. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5254224>. 6.5

[27] JUNIOR, A. A. B. **Implantação de Componentes de Software Distribuídos Multi-Linguagem e Multi-Plataforma**. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, Brasil, ago. 2009. 7