UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Bruno Cavaler Ghisi

ARQUITETURA MODULAR PARA MIDDLEWARE DE TELEVISÃO DIGITAL

Dissertação submetido ao Programa de Pós Graduação da Universidade Federal de Santa Catarina para a obtenção do Grau de mestre em Ciência da Computação
Orientador: Prof. Dr. Frank Siqueira

Florianópolis

2011

Catalogação na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

G426a Ghisi, Bruno Cavaler
Arquitetura modular para middleware de televisão digital [dissertação] / Bruno Cavaler Ghisi ; orientador, Frank Augusto Siqueira. - Florianópolis, SC, 2011.
111 p.: il., grafs., tabs.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Ciência da computação. 2. Televisão digital. 3. Sistemas . embutidos de computador. 4. Java (Linguagem de programação de computador). I. Siqueira, Frank. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 681

Bruno Cavaler Ghisi

ARQUITETURA MODULAR PARA MIDDLEWARE DE TELEVISÃO DIGITAL

Esta Dissertação foi julgada adequada para obtenção do Título de "mestre", e aprovada em sua forma final pelo Programa de Pós Graduação em Ciência da Computação

Florianópolis, 25 de Agosto de 2011.

_____
Prof. Mario Antonio Ribeiro Dantas, Dr.
Coordenador do Curso

Banca Examinadora:

_____
Prof., Dr. Frank Siqueira,
Orientador
Universidade Federal de Santa Catarina

_____
Prof., Dr. Carlos Barros Montez,
Universidade Federal de Santa Catarina

_____
Prof., Dr. Eros Comunello,
Universidade Federal de Santa Catarina

_____
Prof., Dr. Magnos Martinello,
Universidade Federal do Espírito Santo

À minha família que é a inspiração maior.

AGRADECIMENTOS

Aos meus pais, Cícero e Eliane, pelo amor incondicional e estímulo durante estes anos. À Júlia, por todo apoio que uma grande irmã pode oferecer. À minha noiva, Amanda, pelo seu amor, incentivo e por ter criado o ambiente perfeito para que pudesse escrever esta dissertação.

Ao meu orientador, Professor Dr. Frank Siqueira, pelos ensinamentos e contribuições, que foram de extrema importância para este trabalho. Também por acreditar em mim, ser amigo e flexível para que pudesse explorar assuntos do meu interesse maior.

A Fundação CERTI e ao Centro de Convergência Digital por terem me fornecido um ambiente propício, com infraestrutura e suporte técnico (grandes amigos), incentivando a realização deste através de suas políticas.

Aos membros da banca examinadora por terem aceitado o convite e pelas suas contribuições para a conclusão deste trabalho.

Aos professores e funcionários do PPGCC por todos ensinamentos e profissionalismo durante este período em que fiz parte do programa.

Aos grandes amigos que encontrei durante o mestrado, Rafael Besen e Guilherme Schoepping, por todas as conversas que tornaram esta caminhada mais fácil.

A todos familiares e amigos que colaboraram para a finalização deste trabalho, meus sinceros agradecimentos!

Your time is limited, so don't waste it living someone else's life.

(Steve Jobs, 2005)

RESUMO

As especificações abertas de *middleware* de televisão digital não definem características internas de sua arquitetura. Como resultado, as implementações existentes não são tão flexíveis, extensíveis e adaptáveis como deveriam. Este trabalho propõe uma arquitetura modular para ambientes procedurais em *middlewares* de televisão digital, que permite a fácil extensão e atualização, fornecendo uma maneira simples de adicionar novos recursos e utilizar serviços internos à camada de *middleware*. Esta arquitetura é baseada no framework OSGi e seus componentes de software, chamados *bundles*. Uma categorização destes *bundles* é proposta, baseada em diferentes cenários de uso, de acordo com a indústria e as necessidades de inovação do mercado. Protótipos da arquitetura e *bundles* foram implementados e testes realizados para mostrar a viabilidade desta abordagem.

Palavras-chave: Televisão digital. Middleware. OSGi.

ABSTRACT

The currently open specifications for Digital Television (DTV) middleware do not define the internal characteristics of their architecture. As a result, the existing middleware implementations are not as flexible, extensible and adaptable as they should be due to the adoption of a closely-coupled architecture. This paper proposes a modular architecture for execution enviromnent on DTV middleware, which allows easy extension and updating, providing a simple way to add new features to the middleware. This architecture is based on the OSGi framework and its software components, called bundles. A categorization of these bundles was proposed, based on different usage scenarios that were identified, according to industry time-to-market and innovation needs. Prototypes of the architecture and of bundles have been implemented, and tests performed with these prototypes show the feasibility of this approach.

Keywords: Digital television. Middleware. OSGi.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AMS - Application Management System
API - Application Programming Interface
ATSC - Advanced Common Application Platform
ARIB - Association of Radio Industries and Businesses
AWT - Abstract Window Toolkit
BML - Broadcast Markup Language
CA - Certificate Authority
CDC - Connected Device Configuration
CLDC - Connected Limited Device Configuration
CSS - Cascading Style Sheets
DASE - DTV Application Software Enviromnent
DVB - Digital Video Broadcasting
EPG - Electronic Programming Guide
FP - Foundation Profile
HTML - HyperText Markup Language
ITU - International Telecommunication Union
ISDB - Integrated Services Digital Broadcasting
JSR - Java Specification Request
JAR - Java Archive
JVM - Java Virtual Machine
MHP - Multimedia Home Platform
NCL - Nested Context Language,
OSGi - Open Services Gateway Initiative
PBP - Personal Basis Profile
RFID - Radio Frequency Identification
SBTVD - Sistema Brasileiro de Televisão Digital Terrestre
XML - Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Em 2006, através de um ato federal (BRASIL, 2006), foi definida a criação do Sistema Brasileiro de Televisão Digital Terrestre (SBTVD). Com isto também criou-se o Fórum SBTVD, que é um grupo composto por representantes do setor de radiodifusão, do setor industrial, da comunidade científica e tecnológica. O Fórum SBTVD tem o objetivo de assessorar questões envolvendo definições acerca da televisão digital no país. O Brasil iniciou a definição do padrão de televisão digital tardiamente, em comparação com alguns outros países da Europa, por exemplo, onde já não é mais utilizado o sinal analógico para transmissão (*switch off*). O *switch off* no Brasil está planejado para 2016.

O Fórum SBTVD especificou um ambiente de *middleware* próprio para a interatividade chamado Ginga. O Ginga é a camada responsável que permite a execução de aplicações interativas dentro do padrão brasileiro de televisão digital.

Atualmente não existe uma implementação de referência completa deste *middleware* e um número limitado de implementações comerciais, que estão restritas à indústria. Além disto, as normas envolvendo o Ginga, ABNT NBR 15606, não definem uma forma como o mesmo deve ser implementado, limitando-se a descrever apenas o seu funcionamento e API (*Application Programming Interface*). Outras especificações abertas de *middlewares* de televisão digital, como MHP, DASE e ARIB, também não detalham questões arquiteturais e de implementação.

Este *middleware* permite que aplicações interativas sejam executadas no televisor ou *set-top box* de maneira padronizada e independente de fabricante. Fazem parte destes, por exemplo, aplicativos para integração com serviços *Web*, informações adicionais sobre o programa transmitido no vídeo, jogos, aplicações para governo (T-Gov), bancárias (T-Banking), comércio eletrônico (T-Commerce), e até aplicações que permitem integração com outros dispositivos externos.

Desta maneira, é importante o estudo e desenvolvimento de *middlewares* que possuam uma arquitetura que favoreça a modularização. Isto para que a indústria possa criar soluções que permitam a rápida adaptação ao mercado, criação de novas famílias de dispositivos e reuso de componentes. Além disto, também é importante

a adoção de tecnologias que promovam inovação, muitas vezes além de um padrão aberto estabelecido, para que seja possível disponibilizar serviços adicionais aos aplicativos e também para permitir o desenvolvimento de novos modelos de negócios.

1.1 OBJETIVOS

A seguir são apresentados o objetivo geral e os objetivos específicos que serviram de orientação ao desenvolvimento deste dissertação.

1.1.1 Objetivo Geral

Este trabalho tem como objetivo principal propor um modelo de arquitetura de *middleware* para sistemas de televisão digital.

1.1.2 Objetivos Específicos

Dentre os objetivos específicos, encontram-se:

1. A arquitetura deve ser compatível com os principais *middlewares* abertos de *televisão* digital;
2. A arquitetura deve favorecer a modularização. Desta forma, a organização da implementação com intuito de aumentar o reuso, aprimorar o gerenciamento e manutenção do código;
3. A arquitetura deve ser extensível possibilitando a adição de novas funcionalidades;
4. Implementar um protótipo da arquitetura proposta;
5. Realizar testes em uma ambiente real.

1.2 JUSTIFICATIVA

A motivação para esta dissertação surgiu do fato de existirem poucos trabalhos envolvendo questões sobre implementações de *middlewares* de televisão digital, sendo que a maior parte destes limita-se a estudos de grupos de pesquisa dentro da indústria.

Além disto, para que a interatividade no país avance significativamente e possa se tornar de fato popular, é necessário, dentre outros investimentos, que a indústria ganhe competitividade e disponibilize receptores interativos (com *middleware)* a um menor custo. Também é necessário o desenvolvimento de aplicativos que agreguem valor aos usuários (telespectadores) e promovam assim a interatividade. Desta maneira, a segunda motivação deste seria a vontade de realizar estudos para ajudar o progresso da interatividade no Brasil.

1.3 METODOLOGIA

A metodologia para a realização deste trabalho compreende o levantamento e estudo de especificações abertas para *middlewares* de televisão digital no âmbito mundial. Diante disto serão analisados pontos comuns destas especificações e os trabalhos desenvolvidos na literatura.

Após este levantamento será possível propor um modelo de arquitetura para a implementação de *middlewares* de televisão digital, bem como a extensão de funcionalidades para aplicações interativas.

Com a definição deste modelo de arquitetura, um protótipo dela será implementado e integrado a um *middleware* de televisão digital para posterior validação através de testes.

Diante da análise destes testes, serão propostas recomendações de uso e pontos de melhoria.

1.4 ORGANIZAÇÃO DESTA DISSERTAÇÃO

O capítulo 2 descreve sobre *middleware* de televisão digital e exemplifica alguns padrões estudados.

No capítulo 3 são apresentadas as principais características do *framework OSGi*, utilizado como base da arquitetura.

No capítulo 4 são descritos os trabalhos relacionados envolvendo *middlewares* de televisão digital e OSGi.

No capítulo 5 encontra-se a proposta de arquitetura desta dissertação.

O capítulo 6 apresenta questões específicas da implementação e testes realizados.

As conclusões resultantes da realização deste trabalho e as suas possibilidades de aprimoramento na forma de trabalhos futuros são apresentadas no capítulo 7.

2 MIDDLEWARE DE TELEVISÃO DIGITAL

Este capítulo objetiva apresentar a fundamentação teórica acerca de *middleware* de televisão digital.

2.1 ARQUITETURA DE MIDDLEWARE DE TELEVISÃO DIGITAL

Em sistemas distribuídos, *middlewares* permitem uniformidade na presença de diferentes *hardwares* e sistemas operacionais através da sua camada de *software* (TANENBAUM, 2002).

Um *middleware* de televisão digital define uma arquitetura em *software* que permite que aplicações funcionem de maneira independente do *hardware*. Esta arquitetura favorece a portabilidade, pois permite que aplicações sejam desenvolvidas utilizando a API (*Application Programming Interface*) pública disponibilizada e sejam transportadas para qualquer receptor digital (*set-top box*) que suporte o *middleware* adotado (MONTEZ, 2005).

Segundo a norma ITU J.200, uma arquitetura de alto nível de APIs e *middleware* de televisão digital deve possuir uma Máquina de Execução (*Execution Engine*) e/ou uma Máquina de Apresentação (*Presentation Engine*). Além destes dois componentes, pode-se ainda destacar a Ponte (*Bridge Elements*), responsável por integrar aplicações que usam ambas máquinas, e os Recursos Comuns, responsável por prover acesso comum, para ambas as máquinas, à recursos que estão integrados com a plataforma. A Figura 1 ilustra tais componentes descritos. No nível mais baixo está o *hardware* e o sistema operacional (*Operating System*), comumente uma versão de Linux embarcado. Logo acima, na primeira área cinza, encontra-se a área que foi chamada de Recursos Comuns. Acima dos Recursos Comuns estão as máquinas citadas, juntamente com a ponte e o gerenciador de aplicações (*App Lifecycle Monitor*) que controla a execução de aplicações gerenciando o que deve ser exibido. Na lateral direta está uma coluna chamada de *software* nativo (*Native Software*) que pode ser entendido como um *software* disponibilizado pelo fabricante, que não necessariamente foi implementado através das APIs do *middleware*, como o guia de programação, por exemplo. Novamente acima das caixas cinzas encontram-se as aplicações implementadas com a API do *middleware* (*Applications*) e por fim, acima, encontra-se a interação com o usuário

(*User Interaction*) que seria uma forma de entrada de dados, comumente através de um controle remoto, por exemplo.

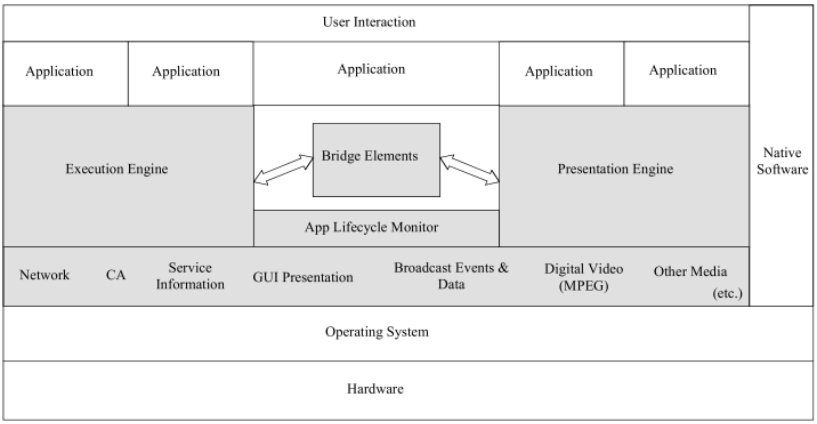

Figura 1 – Arquitetura de um *middleware* de televisão digital
Fonte: ITU J.200.

2.2 MÁQUINA DE EXECUÇÃO

A Máquina de Execução, ou Máquina Procedural, caracteriza-se por oferecer um ambiente que permite a execução de linguagens imperativas. Segundo Kowalsky (1988), um algoritmo pode ser descrito pela equação A = L + C, onde L seria a lógica e C seria o controle. Desta maneira, uma linguagem imperativa define tanto a lógica como o controle de um algoritmo. No controle é descrito cada passo, através de mecanismos de iterações e controles condicionais, por exemplo.

Em *middlewares* de televisão digital é comum o uso da plataforma Java como Máquina de Execução. Sua maior variação está relacionada às diferentes APIs providas por cada padrão. Entretanto todas estas APIs tem como base a especificação Java TV API (JAVA COMMUNITY PROCESS, 2006), que define um *framework* comum para o uso de Java em sistemas de televisão. Nestes ambientes que utilizam Java, existe a necessidade da integração de uma máquina virtual Java (JVM) para execução das aplicações.

2.2.1 Java TV

O Java TV fornece o *framework* base do ambiente que contém a Máquina de Execução. Através dele é disponibilizado uma API para recuperar tabelas do radiodifusor, acessar dados do EPG (*Electronic Programming Guide*) e executar mídia.

A máquina virtual destes ambientes de televisão digital se caracteriza por ser mais limitada que uma disponível na edição Java SE (*Standard Edition*). Desta maneira, ela é considerada uma máquina virtual do tipo Java ME (*Micro Edition*). Dentro desta edição, existem dois tipos de máquina virtuais: CLDC (*Connected Limited Device Configuration*), comuns em dispositivos com recursos computacionais mais limitados como celulares, e CDC (*Connected Device Configuration*), utilizada por ambientes como televisão e *set-top boxes*. A máquina virtual CDC (JAVA COMMUNITY PROCESS, 2005a) é compatível com a sintaxe da linguagem Java 1.4. As aplicação que rodam em máquina virtuais do tipo CDC são chamadas de *Xlet*.

No topo da máquina virtual CDC estão inseridos os perfis. Os perfis são responsáveis por adicionar bibliotecas a configuração básica da JVM, por exemplo, como o AWT (*Abstract Window Toolkit*), que é um *toolkit* gráfico, pacotes de rede, entrada e saída (IO), entre outros. Os perfis utilizados por *middlewares* TVD são: FP (*Foundation Profile*) e PBP (*Personal Basis Profile*). O FP (JAVA COMMUNITY PROCESS, 2005b) é o perfil que serve de base para qualquer outro e adiciona pacotes básicos da plataforma Java, como *java.lang*, *java.util*, *java.io*, *java.net*, entre outros. O PBP (JAVA COMMUNITY PROCESS, 2005c) está um nível acima do FP e fornece um ambiente de desenvolvimento de aplicações visuais com a biblioteca AWT (excluindo componentes pesados). Esta arquitetura de *middleware* TVD está descrita na Figura 2.

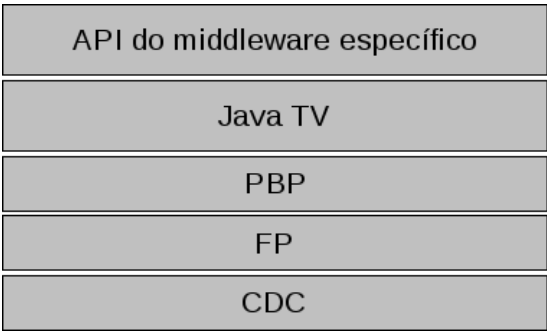

Figura 2 – Arquitetura da Máquina de Execução

2.2.2 Ciclo de Vida de uma Aplicação Xlet

Os *Xlets* são aplicações Java que estendem uma superclasse responsável por definir ações para seus diferentes estados durante seu ciclo de vida. Desta maneira, para criar um *Xlet*, um programador precisa implementar uma lógica para cada um destes estados. Estes estados são controlados pelo AMS (*Application Management System*) ou gerenciador de aplicação dentro de uma JVM. Desta maneira, a aplicação *Xlet* é executada dentro de um ambiente controlado chamado *sandbox*.

Os estados que um *Xlet* transita são (Figura 3):

1. Carregado (*loaded*): Este estado representa o objeto de aplicação sendo instanciado. Uma aplicação entra este estado apenas uma vez quando é carregada e ele representa o estado inicial.
2. Pausado (*paused*): Após a aplicação ter sido criada, o AMS invoca o método de inicialização (*initXlet*). Ela voltará ao estado de pausada caso esteja em execução e precisa ser interrompida sem ser destruída.
3. Iniciada (*started*): A aplicação é então inicializada após o AMS invocar o método de inicialização (*startXlet)*. Uma aplicação pode ter este método chamado mais de uma vez, caso tenha sido pausada e inicializada novamente.
4. Destruído (*destroyed*): Uma aplicação pode passar para o estado destruído (método *destroyXlet*) em qualquer dos estados diferentes do carregado. Ela pode requisitar sua

destruição para liberação de recursos ou então esta destruição ser oriunda de uma exceção lançada.

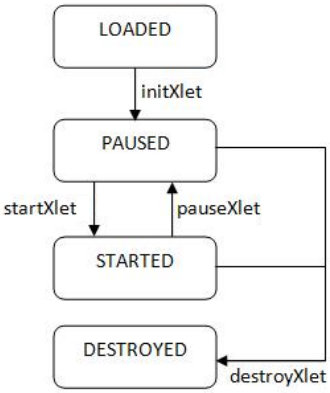

Figura 3 – Ciclo de vida de uma aplicação *Xlet*

2.3 MÁQUINA DE APRESENTAÇÃO

A Máquina de Apresentação, também chamada de máquina Declarativa, caracteriza-se por oferecer um ambiente que permite a execução de uma linguagem declarativa. O paradigma de programação declarativo descreve o que (*what*) deve ser computado mas não necessariamente como (*how*) deve ser computado (LLOYD, 1994). Desta maneira, uma linguagem declarativa, na equação de Kowalsky, descreveria a lógica, mas não necessariamente o controle, que seria responsabilidade do ambiente de execução.

2.4 PONTE

A Ponte oferece um mecanismo de mapeamento bidirecional que permite que aplicações de ambas máquinas interajam. A forma como essas aplicações devem interagir, seja via a invocação de uma função de outro ambiente ou manipulação de um arquivo declarativo, é dependente do padrão de *middleware* definido.

2.5 RECURSOS COMUNS

A camada de Recursos Comuns oferece a infra-estrutura básica que permite a integração entre o baixo nível e as máquinas. Esta camada integra-se com o sistema operacional e suas aplicações nativas, o que faz com que ela possua uma forte dependência da plataforma em questão.

Esta camada provê, para as máquinas, serviços como decodificação do vídeo, decodificação de imagens, tabelas de informações da emissora, infra-estrutura de certificados, rede, entre outros.

2.6 PADRÕES ABERTOS DE MIDDLEWARE DE TELEVISÃO DIGITAL

Padrões abertos são utilizados, pois garantem que sistemas compatíveis possam funcionar juntos, não importando qual o fabricante do equipamento (MORRIS, 2005), de modo que isto favorece a interoperabilidade. Em um cenário real, ainda assim podem existir problemas de interoperabilidade, visto que especificações são passíveis de diferentes interpretações e implementações.

Existem três principais padrões abertos de televisão digital no mundo: 1) DVB (*Digital Video Broadcasting*); 2) ATSC (*Advanced Common Application Platform*); 3) ISDB (*Integrated Services Digital Broadcasting*), representando tanto a versão japonesa como a derivação brasileira chamada ISDB-Tb. A adoção deles no mundo pode ser encontrada na Figura 4. O DVB encontra-se principalmente na Europa e o ATSC principalmente na América do Norte. O ISDB ganhou mais popularidade com adoção no Brasil e em outros países da América do Sul.

Estes padrões possuem variações e definem *middlewares* com características específicas. Os *middlewares* de cada um destes principais padrões serão descritos nos itens seguintes. O *middleware* do padrão ATSC chama-se DASE, do DVB chama-se MHP, do ISDB-Tb chama-se Ginga e do ISDB é muitas vezes referenciado pelo grupo que o criou, o ARIB. Neste trabalho utilizaremos a nomenclatura ARIB para citá-lo.

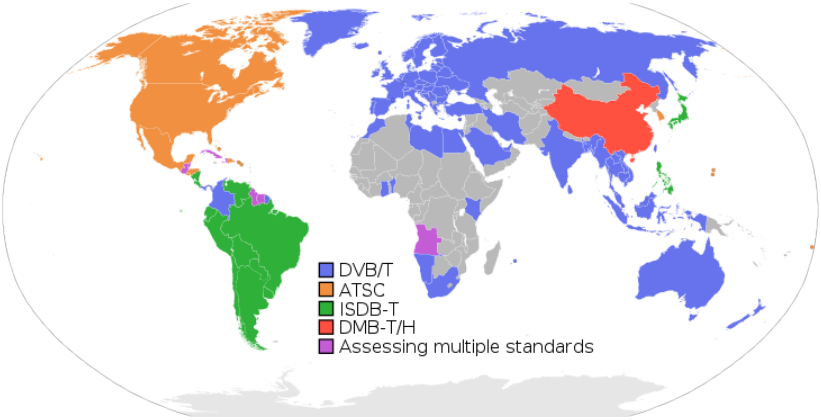

Figura 4 – Padrões de televisão digital e sua adoção no mundo
Fonte: WIKIPEDIA (2011).

2.6.1 MHP

O MHP (*Multimedia Home Platform*) (EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, 2003) é o *middleware* definido no padrão europeu DVB (DVB, 2004). Ele começou a ser definido em 1997 e divide-se em DVB-HTML, que é representado pela Máquina de Apresentação, e DVB-J, representado pela Máquina de Execução.

O DVB-HTML permite o desenvolvimento de aplicações através da linguagem declarativa HTML (*HyperText Markup Language*). Neste caso, o que a Máquina de Apresentação faz é se comportar de maneira similar a um navegador Web (*browser*), renderizando visualmente as *tags*. Por outro lado, DVB-J permite o desenvolvimento de aplicações através da linguagem Java.

A arquitetura alto nível do MHP é mostrada na Figura 5. O MHP está separado do Sistema Operacional que roda em um determinado *Hardware*. O MHP, na sua primeira camada, contém uma Máquina Virtual Java (JVM) para a execução de aplicações DVB-J e um Gerenciador de Aplicações para o controle do ciclo de vida das aplicações. O *User Agent* é o ambiente que renderiza as aplicações DVB-HTML e o *Plug-In* "A" seria a representação de algo que renderiza um conteúdo específico, ou seja, uma Aplicação do tipo "A", através da API de *Plug-In*.

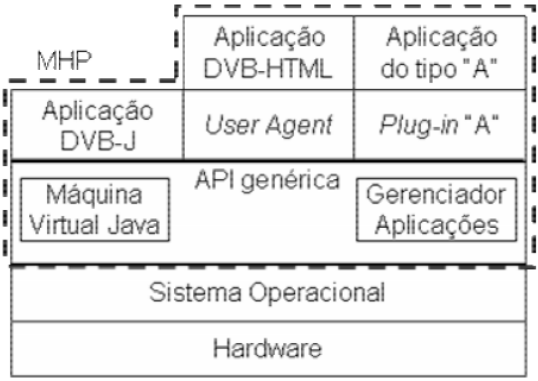

Figura 5 – Arquitetura alto nível do MHP
Fonte: MORENO (2006).

2.6.2 DASE

O padrão de *middleware* DASE (*DTV Application Software Enviromnent*) foi desenvolvido nos Estados Unidos pelo grupo ATSC.

O DASE define uma Máquina Declarativa (ADVANCED TELEVISION SYSTEMS COMMITTEE, 2003b) que possui um interpretador HTML, CSS (*Cascading Style Sheets*) e ECMAScript e uma Máquina de Execução (ADVANCED TELEVISION SYSTEMS COMMITTEE, 2003c) compatível com Java TV, entretanto com pacotes adicionais *org.atsc*. A Figura 6 apresenta sua arquitetura alto nível, onde a camada denominada "Decodificadores de Conteúdo Usuais" é equivalente à camada de Recursos Comuns descrita anteriormente, e as máquinas estão representadas pelo "Ambiente de Aplicação Declarativa" e "Ambiente de Aplicação Procedural".

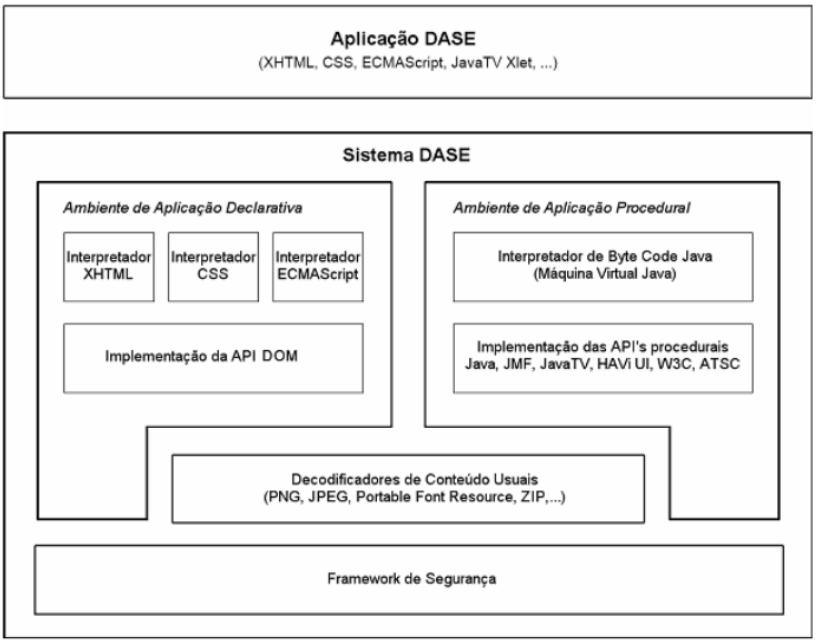

Figura 6 – Arquitetura alto nível do DASE
Fonte: MORENO (2006).

2.6.3 ARIB

O ARIB é o middleware do sistema japonês ISDB. No ambiente Declarativo é utilizada um linguagem chamada BML (*Broadcast Markup Language*) (ASSOCIATION OF RADIO INDUSTRIES AND BUSINESSES, 2002). Esta linguagem define um padrão de formato XML (*Extensible Markup Language*) próprio.

O ambiente Procedural chama-se ARIB-J (ASSOCIATION OF RADIO INDUSTRIES AND BUSINESSES, 2004). O ARIB-J fornece o desenvolvimento através da linguagem Java. A Figura 7 exibe a arquitetura alto nível do ARIB, onde é possível identificar um *Hardware*, o Sistema Operacional e a máquina virtual Java com suas bibliotecas. Nela também são representados o ambiente para execução do BML (*user agent*) e o ambiente dos *plugins*. As aplicações de cada tipo (Conteúdo BML e Conteúdo Java) estão representadas na parte superior da figura.

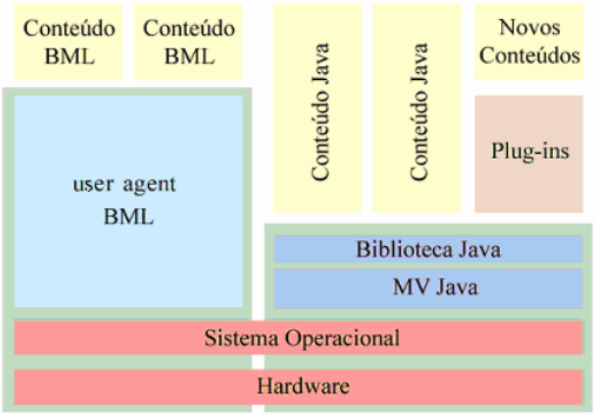

Figura 7 – Arquitetura alto nível do ARIB
Fonte: MORENO (2006).

2.6.4 Ginga

O *middleware* do padrão de televisão digital brasileiro, ISDB-Tb, é intitulado Ginga (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2010a). Ele foi especificado pelo SBTVD e divide-se no subsistema Ginga-NCL (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2011), que representa o ambiente declarativo, e Ginga-J (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2010c), que representa o ambiente imperativo.

O ambiente declarativo é baseado em uma linguagem chamada NCL (*Nested Context Language*), que também utiliza um formato específico do padrão XML. Além disto, este ambiente também faz uso da linguagem Lua (LERUSALIMSCHY, 2006) que adiciona características do paradigma de programação imperativa. O Ginga-J possui uma infra-estrutura de ambiente Java e adiciona uma API própria chamada Java DTV (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2010d). A camada de Recursos Comuns é chamada de Ginga-CC (*Ginga Common Core*).

A arquitetura do *middleware* Ginga é ilustrada na Figura 8. O *Hardware* e o Sistema Operacional encontram-se na camada de mais baixo nível. Logo acima está o Ginga-CC, que possui a infra-estrutura para prover serviços de rede, certificado, EPG, toolkit gráfico, entre outros. Acima do Ginga-CC estão a máquina de execução e a máquina

de apresentação, que são ligadas por uma ponte. O monitor de ciclo de vida da aplicação é responsável pelo gerenciamento das aplicações. Acima desta camada, estão as aplicações e a interação do usuário. Podem ainda existir softwares nativos, como EPG, aplicações nativas, entre outros.

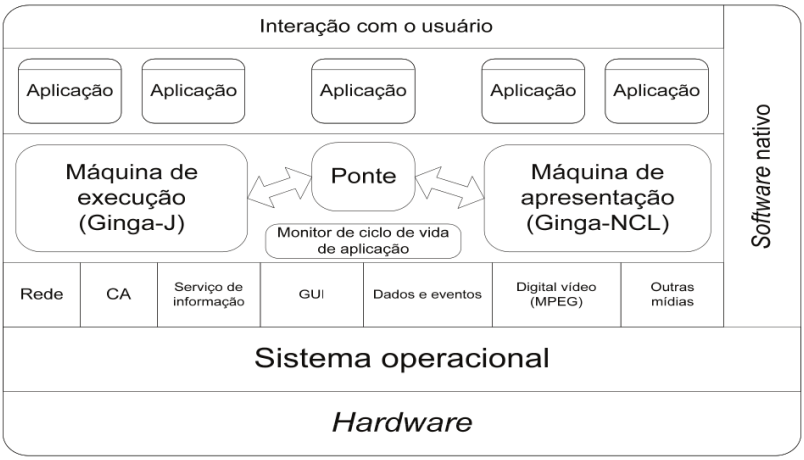

Figura 8 – Arquitetura alto nível do Ginga
Fonte: ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2010a.

A norma Ginga-NCL (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2010a) cita ainda a estrutura chamada *players*, onde seria possível adicionar exibidores de maneira desacoplada, como um *plug-in* através de um adaptador, conforme mostrado na Figura 9. Desta maneira é possível que implementações proprietárias de exibidores sejam integradas através de um adaptador conhecido pela máquina de apresentação Ginga NCL. Entretanto isto é apenas uma sugestão de arquitetura, visto que a especificação não define como essa estrutura deve ser implementada e nem padroniza uma API.

Figura 9 – *Players* de mono-mídia do Ginga
Fonte: ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2011.

3 FRAMEWORK OSGi

Este capítulo apresenta a fundamentação teórica acerca do padrão OSGi.

3.1 INTRODUÇÃO

A OSGi Alliance foi fundada em 1999. Ela tem a missão de criar especificações abertas para prover serviços para redes locais a dispositivos (OSGI ALLIANCE, 2007), trazendo a próxima geração de serviços de Internet para casas, carros, telefones celulares, *desktops*, e outros ambientes.

A tecnologia OSGi (*Open Services Gateway Initiative*) é um sistema que provê módulos dinâmicos para a plataforma Java. Enquanto a plataforma Java fornece a portabilidade para ser executado em diversas plataformas através de uma máquina virtual, a tecnologia OSGi provê as primitivas padronizadas que permitem que aplicativos possam ser construídos a partir de componentes pequenos e reutilizáveis. Estes componentes, por sua vez, podem ser compostos em novos aplicativos. (OSGI ALLIANCE, 2010).

Uma das motivações da criação do padrão OSGI foi o fato da plataforma Java possuir algumas deficiências com relação ao versionamento e dependência de bibliotecas. Devido a esta deficiência, muitos projetos desenvolveram formas proprietárias e não compatíveis de modularização, o que se torna algo ruim para a integração de sistemas de maneira ampla.

3.2 FRAMEWORK

O OSGI define um *Framework* que permite a instalação, desinstalação e atualização de *bundles*. Este *Framework* é responsável por manter a consistência dos serviços ao gerenciar a relação de dependência entre os *bundles* instalados. Um *bundle* é um pacote com conjunto de classes Java compiladas e empacotadas em um arquivo compactado JAR (*Java Archive*). Um arquivo de *bundle* JAR contém descritores que metadados que são utilizados pelo *Framework*.

Existe uma série de projetos que utilizam implementações de um *framework OSGi* em seu núcleo, tais como a IDE Eclipse (ECLIPSE

FOUNDATION, 2010a) e seus mecanismos de *plugins* e o servidor de aplicações GlassFish (JAVANET, 2010).

O framework OSGi é dividido nas camadas abaixo (Figura 10) que serão detalhadas nos próximos itens:

1. Camada de Segurança (*Security Layer*);
2. Camada de Módulos (*Module Layer*);
3. Camada de Ciclo de Vida (*Life Cycle Layer*);
4. Camada de Serviço (*Service Layer*).

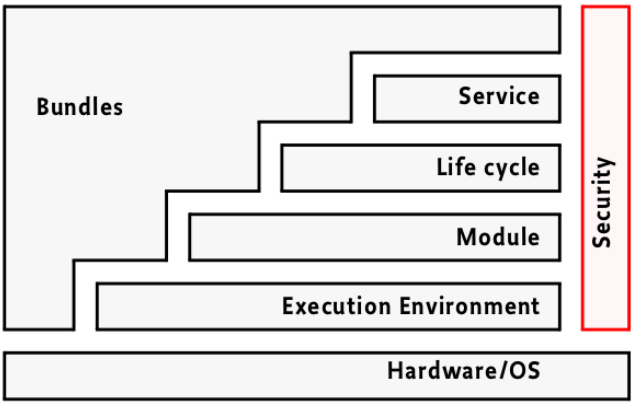

Figura 10 – Estrutura de camadas do *Franework* OSGi
Fonte: OSGI ALLIANCE (2007).

3.2.1 Camada de Segurança

A Camada de Segurança é baseada na segurança definida no Java 2, mas acrescenta uma série de restrições e preenche algumas das lacunas que existiam.

A Camada de Segurança não disponibiliza uma API para controle da aplicação, sendo que esta gerência de segurança é realizada pela Camada de Ciclo de Vida. Ela não é uma camada obrigatória para o funcionamento do *Framework*.

3.2.2 Camada de Módulos

A Camada de Módulos define um modelo de modularização em Java através de *bundles*. Esta camada possui regras para compartilhar pacotes entre os *bundles* ou para esconder pacotes de determinados *bundles*.

A Camada de Módulos pode ser utilizado sem a Camada de Ciclo de Vida e Camada de Serviço. A camada de Ciclo de Vida fornece uma API para gerenciar os pacotes na Camada de Módulo, enquanto que a Camada de Serviço fornece um modelo de comunicação para os *bundles*.

3.2.2.1 Carregamento de Classe

Muitos *bundles* podem ser gerenciados em uma única JVM. Dentro desta JVM, *bundles* podem esconder pacotes e classes de outros *bundles*, bem como compartilhar pacotes com outros *bundles*.

O principal mecanismo para ocultar e compartilhar pacotes é através do carregador de classes (*class loader). C*ada *bundle* tem um carregador de classe único. O carregador de classes forma uma rede de carregamento de classes com outros *bundles*, como mostrado na Figura 11 (OSGI ALLIANCE, 2007). Um *bundle class loader possui* um *parent/system class loader* para buscar o conjunto básico de classes.

Um *class space* engloba todas as classes que são alcançáveis por um carregador de classes. O carregador de classes pode carregar as classes de diversos locais, listados abaixo:

1. *Boot class path*: Contém os pacotes da linguagem Java e a implementação;
2. *Framework class path*: O *Framework* geralmente tem um carregador de classes com as classes de sua implementação separadas das classes de serviço;
3. *Bundles space*: Consiste do JAR associado ao *bundle* e qualquer JAR adicional que está relacionado ao *bundle*.

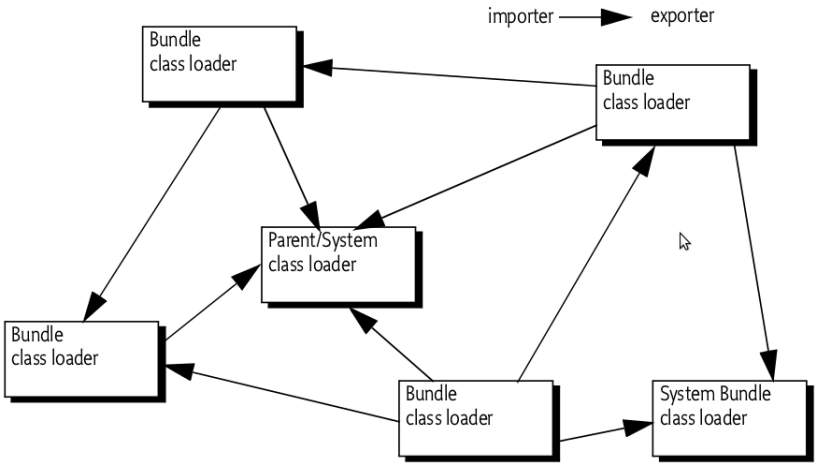

Figura 11 – Estrutura do carregador de classes do *Framework* OSGi
Fonte: OSGI ALLIANCE (2007).

3.2.3 Camada de Ciclo Vida

A Camada de Ciclo de Vida fornece uma API do ciclo de vida para os *bundles*. Ela define como os *bundles* são inicializados e parados, como são instalados, atualizados e desinstalados. Além disto, ela também fornece uma API de eventos para permitir que, por exemplo, um *bundle* gerenciador controle as operações da plataforma de serviço. O Ciclo de Vida necessita da Camada de Módulos, que provê os *bundles*, mas a Camada de Segurança é opcional.

Um *bundle* pode estar em um dos seis estados: instalado (*installed*), desinstalado (*uninstaled*), resolvido (*resolved*), inicializado (*starting*), parado (*stopping*) e ativo (*active*). Inicialmente um *bundle* é instalado e ele é armazenado na área de persistência do *Framework*, permanecendo lá até ser desinstalado. Se ocorrer alguma problema durante sua instalação, o *Framework* volta ao estado original, caracterizando uma operação atômica. Quando o *Framework* resolver com sucesso as dependências do *bundle*, ele entra no estado resolvido. Após isto, um *bundle* pode ser inicializado pelo *Framework*, se configurado para ser automático, ou então se explicitamente inicializado via código. Após o *bundle* ser inicializado, ele passa para o estado de ativo, que pode ser feito de duas maneiras: carregamento tardio (*lazy*

loading), quando uma de suas classes for carregada, ou explicitamente. Um *bundle* pode ainda ser parado, levando-o para este estado. Este processo está ilustrado na Figura 12.

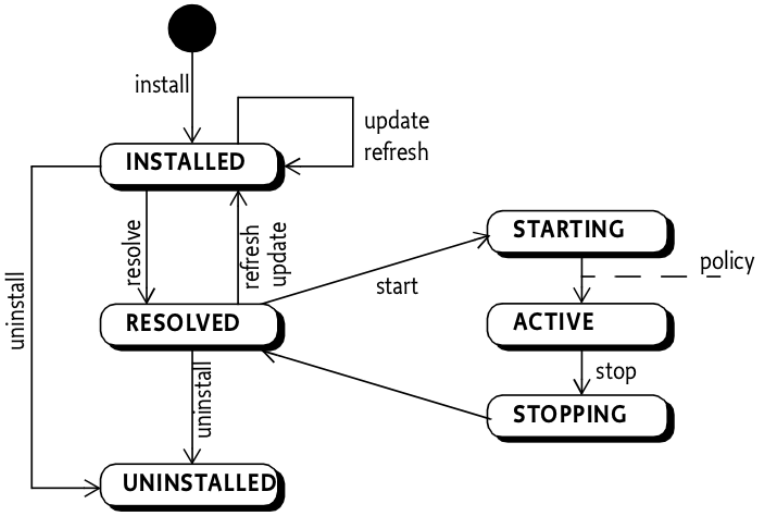

Figura 12 – Ciclo de vida de um *bundle*
Fonte: OSGI ALLIANCE (2007).

3.2.3.1 Contexto de um Bundle

A relação de um *bundle* dentro do *framework OSGi* é realizada através de seu contexto, mais especificamente da interface *BundleContext*. Ela define métodos para permitir a instalação de novos *bundles* no *Framework*, listar os *bundles* instalados, obter os serviços registrados e registrar serviços. Ela é importante para permitir integrações com o *framework OS*Gi dentro de um padrão definido.

3.2.4 Camada de Serviço

A Camada de Serviço provê um modelo conciso e coerente de programação, simplificando a implantação de *bundles* através do desacoplamento da especificação do serviço (interface) e de suas implementações (realização da interface). Este modelo permite aos

desenvolvedores vincular pacotes de serviços apenas com suas especificações de interface. A seleção de uma implementação específica pode ser feito em tempo de execução.

Bundles podem registrar novos serviços, receber notificações sobre o estado dos serviços, ou procurar serviços existentes. Este aspecto do *Framework* faz com que um *bundle* instalado se torne extensível após a implantação, ou seja, novos pacotes podem ser instalados para adicionar funcionalidades, e bundles existentes podem ser modificados e atualizados sem a necessidade de que o sistema seja reiniciado.

3.3 IMPLEMENTAÇÕES

As principais implementações do OSGi Release 4 (OSGI ALLIANCE, 2007) são Knoplerfish (MAKEWAVE, 2010), Apache Felix (APACHE SOFTWARE FOUNDATION, 2010), Equinox (ECLIPSE FOUNDATION, 2010b) e Oscar (O2W CONSORTIUM, 2010).

O Concierge (INSTITUTE FOR PERVASIVE COMPUTING, 2010) (RELLERMEYER, 2007) possui uma versão estável compatível com o Release 3, entretanto também possui uma versão em desenvolvimento que é compatível com o Release 4. O JEFFREE (OBJECTWEB CONSORTIUM, 2010) é compatível com a especificação Release 2.

Ainda existem outras iniciativas criadas para ambientes embarcados e Java ME (CLDC/MIDP), entretanto não estão de acordo com esta última especificação, como Jadabs-OSGi (SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH, 2010), compatível com o Release 1.1, e mBedded Server CLDC Edition (PROSYST SOFTWARE GMBH, 2010), compatível com o Release 3.

4 TRABALHOS RELACIONADOS

Este capítulo objetiva apresentar os trabalhos relacionados. A maioria dos trabalhos divide-se em propostas para arquiteturas de *middlewares* de televisão digital e também extensões destes para adição de serviços de maneira não padronizada.

4.1 MODELO DE COMPONENTES PARA MIDDLEWARE TVD

A seção contempla trabalhos que envolvem propostas de arquiteturas e estruturas para *middlewares* de televisão digital e outras plataformas relacionadas.

4.1.1 Philips OpenTV

Philips OpenTV é uma arquitetura para famílias de produtos de *set-top box*. A principal vantagem dessa arquitetura é fornecer uma estrutura que permita uma fácil adequação a diferentes clientes e mercados (LANGE, 2002). Os motivos pelos quais existem inúmeras diferenças de produtos são: custo de *hardware*, diferentes *chipsets* de diferentes fabricantes, entre outros.

As aplicações nativas do OpenTV são compiladas para um formato de *bytecode* independente de máquina chamado O-code, que posteriormente são interpretadas pelo interpretador OpenTV. A Figura 14 apresenta a arquitetura do OpenTV, onde existe um *hardware* que se comunica com uma camada de abstração do sistema operacional (*HW Abstraction Layer + OS Abstraction Layer*) para facilitar a portabilidade de plataforma. Acima está o ambiente de execução chamado OpenTV *Runtime q*ue se gerencia o interpretador de *bytecode* (OpenTV *interpreter*) onde são as aplicações (EPG, Menu, *Standby* e *Zapper*) são executadas. O interpretador pode ainda se comunicar diretamente com a camada de abstração do *hardware* através de *Native Tasks.*

Uma desvantagem desta arquitetura é que ela é totalmente dependente de um padrão proprietário e não existe uma camada de *middleware*.

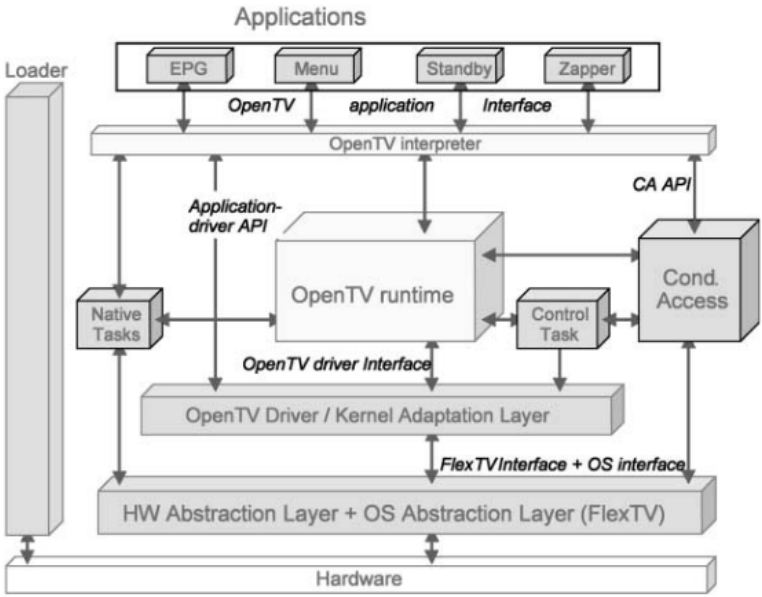

Figura 13 – Arquitetura OpenTV
Fonte: LANGE (2002).

4.1.2 FLEXCM

O trabalho (MIRANDA, 2008) surgiu para suportar o projeto e implementação de um *middleware* de televisão digital adaptativo, definindo um modelo de componentes chamado Flexcm e um ambiente de execução para estes componentes. A motivação surgiu dado que os modelos estudados previamente (COM, Koala, OpenCOM e EJB) não suportavam alguns dos requisitos como atualização de componentes, independência de linguagem, suporte para execução em sistemas embarcados e adaptação de software.

Neste ambiente, os componentes devem declarar explicitamente suas interfaces necessárias e devem conhecer apenas as interfaces, não as implementações. Por sua vez, o ambiente de execução que é encarregado de decidir qual das implementações destas interfaces será utilizada. Além disto, o modelo prevê formas para construir a representação das conexões atuais e reconfigurar as conexões em tempo de execução.

O trabalho prático foi realizado utilizando-se uma proposta de implementação de *middleware* Ginga chamada FlexTV (LEITE, 2005), desenvolvida para o Fórum SBTVD, que hoje encontra-se desatualizada devido a mudanças na especificação do Ginga-J. Parte desta implementação foi readequada para suportar o modelo de componentes Flexcm. A Figura 15 apresenta os principais componentes de alto nível que constituem o FlexTV. A camada verde chamada de gerenciamento do *middleware* seria equivalente ao Recursos Comuns descrito previamente. Dentro dela, existe uma série de componentes que foram empacotados, como os elementos gráficos, gerenciador de eventos de usuários, controlador de apresentação de mídias, de processamento de mídias, processador de fluxo de dados, demultiplexador (*demux*), sintonizador (*tuner*), servidor de informações sobre os fluxos, comunicação entre aplicações, canal de interação, gerenciador de perfis, persistência e acesso condicional.

Figura 14 – Arquitetura alto-nível proposta para o FlexTV
Fonte: MIRANDA (2007).

O estudo de caso desta proposta tem o foco na modularização da camada de Recursos Comuns e seu modelo traz algumas das vantagens do próprio *framework OSGi*, como desacoplamento da implementação e atualização em tempo de execução. Além disto, o modelo não é dependente de uma JVM.

4.2 MIDDLEWARE COM SUPORTE A OSGI

A maioria dos trabalhos disponíveis que integram OSGi em *middlewares* de televisão digital foi criado para fins de automação residencial. As implementações OSCAR e Knoplerfish são as mais

utilizadas, entretanto existem algumas diferenças nas estratégias de integração.

4.2.1 Estratégia de Integração

Existem algumas estratégias para integrar um *framework OSGi* em um *middleware* de televisão digital. Lin (LIN, 2009) define três classificações em seu trabalho, na qual foi utilizado a especificação de *middleware* MHP:

- MbO (MHP baseado em OSGi): OSGI é utilizado como base e as máquina de execução do *middleware* é empacotada como um *bundle*;
- ObM (OSGi baseado MHP): MHP é utilizado como base e o *framework OSGi* é empacotado como um Xlet;
- MnO (MHP vizinho ao OSGi): quando o MHP e OSGi não estão definidos como MbO ou ObM. MnO podem ser classificados em três subtipos:
    - MnO1j1p: MHP e OSGi executam em uma JVM em uma única plataforma;
    - MnO2j1p: MHP e OSGi executam em duas JVMs em uma única plataforma;
    - MnO2j2p: MHP e OSGi executam em duas JVMs em duas plataformas conectadas por uma rede doméstica.

A questão do MHP será abstraída no contexto deste trabalho para classificar os seguintes trabalhos de forma que o MHP poderia ser qualquer *middleware* de televisão digital.

4.2.2 Terminais Portáteis

Gama (GAMA, 2007) propõe uma arquitetura de televisão digital móvel baseada em OSGi e na especificação *Mobile Broadcast Service* API para terminais portáteis (JAVA COMMUNITY PROCESS, 2008).

A proposta tem como alvo JVMs mais limitadas, do tipo CLDC (*Connected Limited Device Configuration*). São citados alguns *bundles* que poderiam ser implementados, por exemplo, PVR (*Personal Video Recorder*) e EPG (*Electronic Program Guide*), bem como a possibilidade de executar a aplicação de controle como um *bundle*.

Entretanto, não foram descritos detalhes de integração e também não houve implementação real, logo nenhum teste foi realizado.

4.2.3 Automação Residencial

Os próximos trabalhos tratam da extensão de *middleware* de televisão digital utilizando OSGi para fins de automação residencial. A tecnologia base utilizada é a mesma proposta no presente, entretanto a finalidade não. Desta maneira, a maior parte destes trabalhos não realizam nenhuma classificação quanto ao uso de *bundles*, testes com implementação do *framework OSGi* e integrações com os diversos *middlewares* de televisão digital.

Tkachenko (TKACHENKO, 2006) propõem um ambiente chamado *DTV Home Networking Framework (DTV-HNF)* que abstrai os dispositivos que estão conectados e utiliza uma infraestrutura com OSGi, através de OSCAR e Knoplerfish, e UPnP (*Universal Plug and Play*). A integração proposta fazia uso de um Xlet que encaminhava as requisições para um *bundle* de controle de luz instalado na plataforma. Este *bundle* se comunicava via *socket* com uma outra aplicação que simulava a luz. Esta proposta segue o modelo MnO2j2p. A arquitetura citada está descrita na Figura 15.

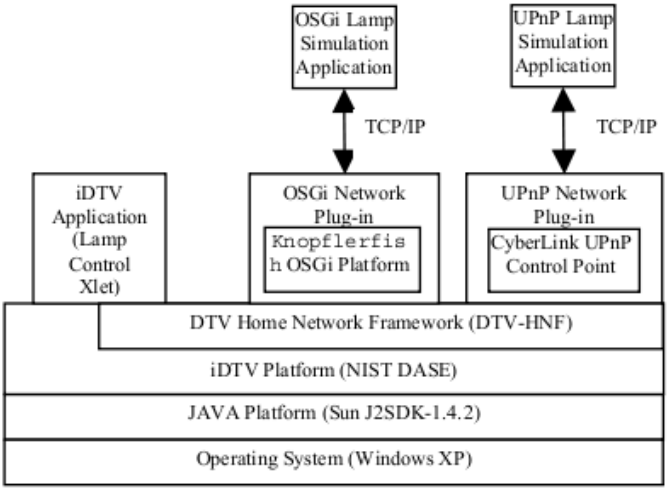

Figura 15 – Proposta de arquitetura TKACHENKO
Fonte: TKACHENKO (2006).

O testes foram realizados apenas em um PC utilizando o *middleware* DASE. A JVM utilizada era Java SE (Sun J2SDK-1.4.2) e o sistema operacional Windows XP.

Redondo (REDONDO, 2007) propõe a integração de um *framework OSGi*, no contexto do trabalho foi utilizado o OSCAR, no *middleware* MHP. Um conceito chamado *XbundLET*, que seria um componente híbrido para OSGi e MHP, foi desenvolvido para uso em ambas as plataformas, conforme ilustra a Figura 16. O *XbundLET* implementa ambas as interfaces de *Xlet* e *Bundle* e desta maneira integra seus ciclos de vida. O trabalho descreve principalmente a comunicação do MHP para OSGi e segue o modelo MnO1j1p.

Os testes foram realizados no ambiente PC e o gerenciador de aplicações era responsável por inicializar o ambiente de integração.

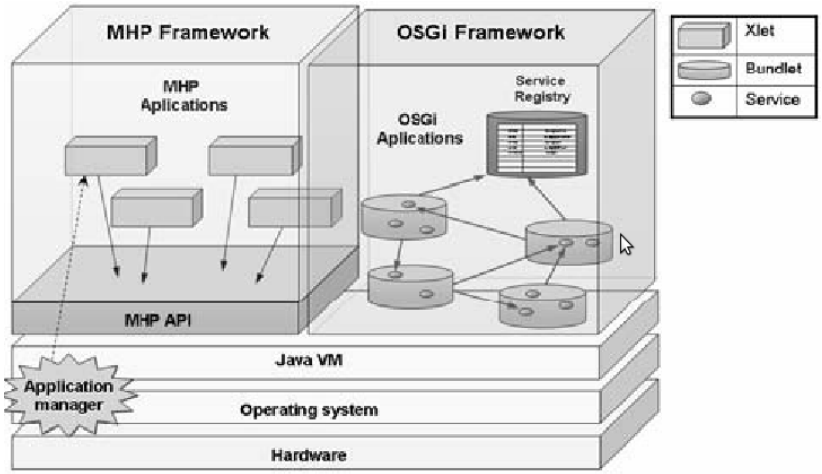

Figura 16 – Proposta de arquitetura Xbundlet
Fonte: REDONDO (2007).

Yang (YANG, 2007) propõe uma arquitetura em MHP utilizando a implementação Knoplerfish. Esta arquitetura permitia a comunicação bidirecional entre o Xlet e o *bundle*. A estratégia de integração adotada foi o uso de um *bundle* e um Xlet que exportavam seus contextos, através do *XCExporterXlet* e *BCExporterBundle*, respectivamente. A integração entre os ambientes era feita através do

BundleContextBridge e *XletContextBridge* (Figura 17). Esta proposta segue o modelo de integração MnO1j1p.

Os testes foram realizados no PC com um *bundle* que permitia abrir uma determinada porta. A JVM utilizada era Java SE (Sun 1.4.2).

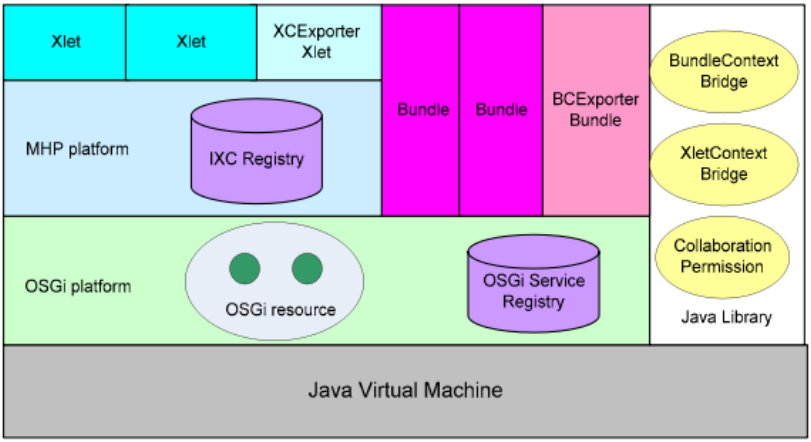

Figura 17 – Proposta de arquitetura YANG
Fonte: YANG (2007).

Lin (LIN, 2008) separa o AMS do MHP, empacotando-o como um *bundle* (Figura 18) chamado *MHPAPPmanager*. Assim, o AMS irá automaticamente detectar e coletar todos os serviços disponibilizados, sejam MHP ou OSGi, através do *broker*. O *broker* permitia a comunicação bidirecional entre os ambientes. Esta proposta segue o modelo de implementação MbO.

Os testes foram realizados em um PC com dois cenários: 1) controle de acesso físico que continha um *bundle* para uma *webcam* que mostrava quem estava do outro lado de uma porta e um *bundle* que controlava a porta e; 2) sistema de recomendação baseado no perfil do usuário que continha um *bundle* com a lógica de recomendação e um *bundle* que gerenciava o perfil do usuário através de RFID (*Radio Frequency Identification*). A JVM utilizada era Java SE (Sun 1.4.1.17) e o *framework OSGi* utilizado foi o OSCAR.

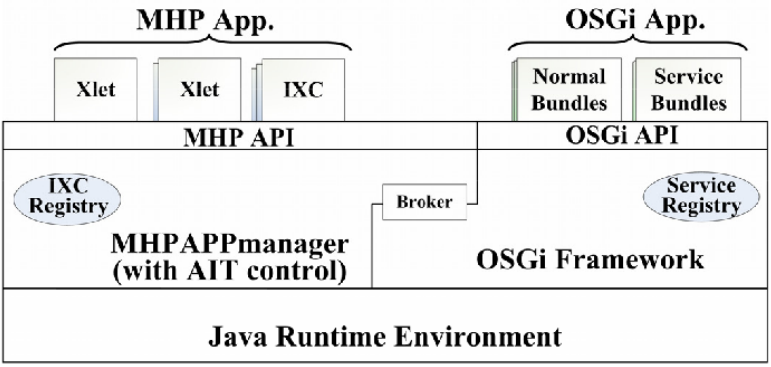

Figura 18 – Proposta de empacotamento do AMS como *bundle*
Fonte: LIN (2008).

Lucena (LUCENA, 2009) trabalhou a integração do Ginga e OSGI através do Knoplerfish. A estratégia de integração segue alguns conceitos apresentados no trabalho de Lin (LIN, 2008) através dos exportadores de contexto, OSGi e *Xlet*, e registro em ambos ambientes, entretanto sem empacotar o AMS como *bundle*. Foi também proposto um modelo de integração com o ambiente declarativo (Ginga-NCL) utilizando a Ponte, que não aconteceu em outros trabalhos. Esta proposta segue o modelo MnO1j1p.

Os testes foram realizados em PC com aplicações *Xlets* que integravam-se com *bundles* e 1) coletavam a temperatura do ambiente; 2) controlavam as lâmpadas; 3) controlavam um robô de Lego via Bluetooth e; 4) tiravam fotos através de uma *webcam*. Os testes realizados na parte declarativa incluiam a impressão de um documento e a integração com uma *webcam* que tirava fotos. A JVM utilizada era uma Java SE (Sun 1.4.2.17 e Sun 1.6.0.4).

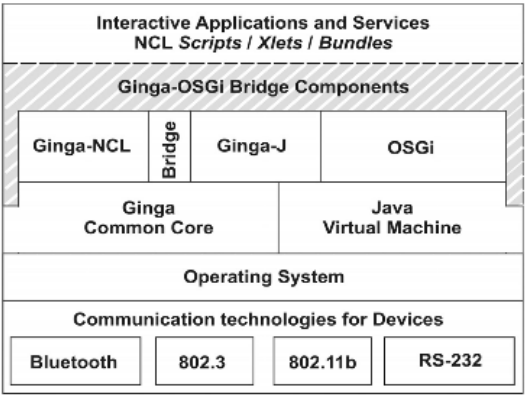

Figura 19 – Proposta de arquitetura LUCENA
Fonte: LUCENA (2009).

A Tabela 1 apresenta o resumo comparativo, dos trabalhos relacionados no âmbito de automação residencial, de acordo com o modelo de integração, *middleware e* framework OSGi utilizados, e também o ambiente de testes.

Os trabalhos relacionados realizaram testes apenas em ambientes de PC com JVMs não padrões de ambientes embarcados. Além disto, não ficam claros os motivos da escolha dos *frameworks* OSGi utilizados (OSCAR e Knoplerfish). Por fim, a maior parte dos trabalhos utilizou o modelo de integração MnO1j1p e o *middleware* MHP.

Tabela 1- Resumo comparativo dos trabalhos relacionados de automação residencial

Trabalho	Modelo	Middleware	OSGi	Testes
Tkachenko	MnO2j2p	DASE	OSCAR Knoplerfish	PC
Redondo	MnO1j1p	MHP	OSCAR	PC
Yang	MnO1j1p	MHP	Knoplerfish	PC
Lin	MbO	MHP	OSCAR	PC
Lucena	MnO1j1p	Ginga	Knoplerfish	PC

5 PROPOSTA DA ARQUITETURA MODULAR

Este capítulo tem como objetivo apresentar a proposta da arquitetura desenvolvida, descrevendo suas principais vantagens e desvantagens.

5.1 CAMADA DE SERVIÇOS

Este trabalho propõe a integração de uma Camada de Serviços em *middlewares* de televisão digital com objetivo de utilizar uma arquitetura modular interna à implementação do *middleware.* Esta arquitetura apóia-se no conceito de camadas que encapsulam funcionalidades permitindo a modularização.

Esta camada seria implementada através de OSGi. Como citado no Capítulo 2, as principais especificações de *middleware* de televisão digital (DASE, MHP, Ginga e ARIB) possuem uma máquina de execução baseado em Java TV e por consequência necessitam de uma JVM. A máquina de apresentação não é dependente da JVM, mas suas APIs poderiam ser implementadas utilizando-a. Isto, por exemplo, devido ao fato que o Ginga não obriga a máquina de execução em ambientes móveis, por exemplo celular, apesar de a mesma estar disponível em muitos modelos. Desta maneira, não seria necessário utilizar bibliotecas que não fossem padrão. Além disto, o uso de OSGi interno ao *middleware* poderia trazer benefícios citados no Capítulo 3, mas não apenas servindo como uma extensão, como exemplo dos casos em automação residencial (Capítulo 4).

Esta arquitetura poderia ser utilizada para implementar parte de uma especificação de *middleware* por meio de serviços específicos, criação de aplicativos que poderiam se comportar como *widgets*[1] e disponibilização de extensões proprietárias que poderiam ser utilizados em aplicações terceiras. A camada de Recursos Comuns inicializa o *middleware*, que por sua vez inicializa a camada de Serviços. A camada de Serviços é controlada por uma camada de gerenciamento responsável por buscar os *bundles* de serviços e inicializá-los para posterior uso. Um *bundle* de serviço contém uma interface bem definida e a lógica necessária para prover determinado serviço.

[1]Aplicações que obedecem a um propósito específico. Tornaram-se populares em ambientes de desktop, exibindo aplicações específicas para tempo, calendário, dicionário, entre outros.

A principal desvantagem do uso desta arquitetura é a necessidade da presença de uma JVM e o aumento de recursos necessários (*footprint*), já que é necessária uma camada adicional para gerenciamento e uso. Com relação a armazenamento, o *framework OSGi* pode ser implementado através de um arquivo JAR de 300KB (OSGI ALLIANCE, 2010), o que não seria um tamanho tão significativo em comparação com outros ambientes, dado as vantagens oferecidas.

Por ser dependente a uma JVM, ela possui uma limitação na modularização de ambientes que necessitem da JVM, como por exemplo, a máquina de execução ou a máquina de apresentação, caso fosse implementada utilizando-a. Ela não permite a modularização das camadas de mais baixo nível como a de Recursos Comuns, apesar de que seria possível invocar seus serviços através de uma ponte em casos específicos.

Os principais benefícios obtidos com a adoção desta arquitetura modular no cenário de televisão digital são: 1) novas extensões podem ser facilmente criadas, satisfazendo as necessidades de mercado; 2) produtos podem ser personalizados através da composição de *bundles*, 3) implementações de *middleware* podem ser gerenciadas para correção de *bugs* ou atualização de funcionalidade e; 4) um *software* de terceiro pode ser eficientemente integrado com o *middleware* através de APIs previamente definidas.

O uso desta arquitetura traz a vantagem principal de permitir a modularização e extensabilidade, mas em contrapartida necessita de mais recursos computacionais.

O *framework OSGi* seria parte da implementação do *middleware*, representando uma camada de serviços, de acordo com a imagem base do padrão ITU (Figura 20). Esta camada de serviços seria responsável pelo gerenciamento dos *bundles e* poderia servir ao *middleware* e aplicações. A Máquina de Execução e a Máquina de Apresentação poderiam ser implementadas através da camada de Serviços, o que poderia trazer variações da Figura 20.

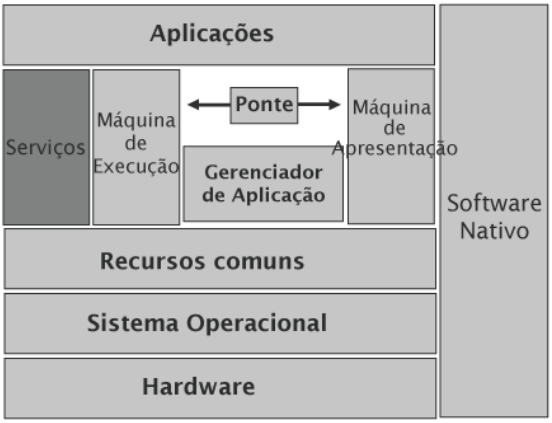

Figura 20 – Camada de Serviços integrado a um arquitetura de *middleware* de televisão digital

5.2 ESTRATÉGIA DE INTEGRAÇÃO

Conforme a seção 4.2.1, a estratégia de integração adotada segue o modelo MnO1j1p (LIN, 2009), executando em uma única JVM dentro de uma única plataforma. Ela não segue o modelo ObM, pois o *framework* OSGi não está empacotado como um *Xlet* e também não segue o modelo MbO, pois as APIs máquina de execução também não são empacotadas como um *bundle*. Além disto, não era desejado seguir o modelo MnO2j1p, pois em ambientes reais dificilmente duas JVM estariam disponíveis, já que seriam necessários muitos recursos computacionais. Por fim, também não segue o modelo MnO2j2p, pois está apenas em uma plataforma, no caso específico, o receptor ou televisão digital.

5.3 CAMADA DE GERENCIAMENTO

A gestão de instalação, desinstalação e atualização de cada tipo de *bundle* deve ser realizada internamente pelo *middleware*, bem como a sua inicialização. Desta forma, é necessário uma camada de gerenciamento dentro deste ambiente de Serviços, já que ela também deve realizar o controle de quais tipos de *bundles* estão visíveis em cada contexto, sejam serviços de *middleware*, extensão proprietária ou

aplicações *widget*. Ele também fornece mecanismos para listar os serviços disponíveis e invocar seus métodos.

5.4 CATEGORIZAÇÃO

Três tipos de pacotes (*bundles* OSGi) foram definidos nesta Camada de Serviços:

1. Serviço de *middleware*;
2. Extensão proprietária;
3. Aplicação *widget*.

As extensões proprietárias e serviços de *middleware* podem definir uma interface de serviço de modo a separar a implementação do serviço real a partir da especificação. Esta interface não está presente em aplicações *widget*, que são *bundles* que não fornecem serviços externos, entretanto devem implementar uma outra interface padrão para que possam ser executadas pelo AMS.

Devido à possibilidade de desacoplar a interface da implementação do serviço, é possível que extensões proprietárias possam ser compiladas utilizando uma determinada interface padrão e a plataforma pode fornecer uma implementação distinta. Por exemplo, no caso de um serviço de recomendação, pode existir uma interface padrão para recomendação de um programa, entretanto poderiam haver diferentes implementações que fazem uso de diferentes serviços *Web* e algoritmos. Os serviços de *middlewares* também podem ser beneficiados com isto. Este desacoplamento permite o uso de implementações de terceiros e atualizações de implementações pelo fabricante.

A Figura 21 apresenta a camada de serviços proposta, onde na base está um *framework* OSGi, que possui uma série de *bundles*, categorizados pelos tipos citados, e controlados por uma camada de gerenciamento.

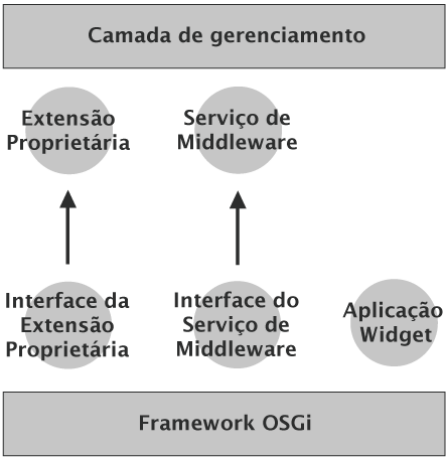

Figura 21 – Categorização dos *bundles*

5.4.1 Serviço de Middleware

Serviços de *middleware* são *bundles* que oferecem APIs comuns para a implementação do *middleware*, como uma API para implementação de um serviço de preferências de usuário, o *parsing* de um protocolo de determinada especificação, funções para acessar um fluxo de dados de transporte ou EPG, por exemplo.

Esta implementação de *bundle* pode também integrar-se com a camada de Recursos Comuns (baixo nível), através de JNI (*Java Native Interface*), que é um mecanismo de *binding* Java para C/C++. Desta maneira, a camada de Recursos Comuns poderia prover dados disponíveis no baixo nível e utilizar serviços de *bundles* também, por exemplo, para disponibilizar à outras camadas. Este *bundle* também pode ser implementado apenas em Java. Quando houver ao menos uma integração JNI, este *bundle* será classificado como nativo, enquanto que no segundo, como regular, conforme a Tabela 2.

Tabela 2- Classificação dos bundles serviços de *middleware* quanto a integração

Tipo	**Integração**
Nativo	Baixo nível - JNI (C/C++)
Regular	Java

Bundles deste tipo são visíveis apenas pela camada interna do middleware e seu conjunto de bibliotecas, de modo que não podem ser utilizados por aplicativos externos (Xlets). Também é possível definir interfaces para estes serviços, já que diferentes implementações poderiam ser utilizadas, dependendo das características da plataforma ou necessidades para a contratação de empresas terceiras.

5.4.2 Extensão Proprietária

Extensões proprietárias são *bundles* que oferecem APIs não comuns, adicionando novos serviços e capacidades as aplicações (*Xlets*). Eles trazem extensibilidade à arquitetura, mas não utilizam um padrão determinado por especificações de middleware de televisão digital, logo não seriam portáveis em dispositivos que não o implementam. Por exemplo, um *bundle* deste tipo poderia fornecer um de interpretador para uma linguagem de programação, um algoritmo específico para um sistema de recomendação ou uma forma de integração a um serviço de rede social.

Estes *bundles* são visíveis através de aplicações, de forma que um desenvolvedor pode utilizar em tempo de execução. Elas não seguem uma API de *middleware* específica, mas são alternativas que permitem à indústria gerar inovação, sem nenhuma dependência, através da adição de novos *bundles* independentes. Assim como os serviços de *middleware*, as interfaces podem ser definidas de maneira separada da implementação. Estes *bundles* também podem ser classificadas como regular e nativo, de acordo com a integração, Java ou JNI citado no item anterior.

As implementações em torno de automação residencial descritas nos trabalhos relacionados também poderiam ser classificadas como *bundles* do tipo extensão proprietária. Desta forma, seria possível desenvolver uma aplicação *Xlet* que faz uso de determinado serviço para gerenciar um ambiente, por exemplo.

5.4.3 Aplicação Widget

Aplicações *widget* são *bundles* que funcionam como aplicativos instalados na plataforma. Este tipo de *bundle* pode ser classificado em dois tipos: nativo e regular. A diferença entre eles é que *bundles* do tipo serviços de *middleware* são visíveis para o nativo, enquanto que um regular só enxerga *bundles* do tipo extensões proprietárias, conforme descrito na Tabela 3. Desta maneira não faria sentido uma aplicação de terceiros possuir algum tipo de acoplamento com a implementação interna do *middleware*, via algum *bundle* serviço de *middleware*, já que a mesma deve permanecer desconhecida por questões de segurança e separação de camadas.

Tabela 3- Visibilidade dos bundles serviços de middleware e extenseões proprietárias para aplicação widget

Aplicação widget	Serviços de middleware	Extensões proprietárias
Nativa	Visível	Visível
Regular	Não visível	Visível

Uma aplicação EPG, por exemplo, pode ser desenvolvida utilizando o conceito de aplicação *widget*, para exibição da programação, fazendo uso de um serviço de *middleware* nativo, com integração a camada baixo nível para acesso as tabelas do *broadcast*, sendo classificada como uma aplicação *widget* nativa. Além disto, a indústria pode criar lojas de aplicativos e disponibilizar *widget* para a sua própria plataforma utilizando seus próprios padrões de extensões proprietárias, tais como aplicações bancárias (*T-banking*), comércio eletrônico (*T-Commerce*), saúde (*T-Health*), aplicações integradas a redes sociais, jogos, entre outros. Elas poderiam vir instaladas por padrão ou ser instaladas via algum mecanismo proprietário, seja pela *Internet* ou por um adaptador local.

6 IMPLEMENTAÇÃO DA ARQUITETURA

Este capítulo descreve a estratégia de integração da arquitetura, detalhes da implementação da arquitetura, escolha do *framework* OSGi, *middleware* e os testes realizados. Alguns exemplos dos tipos de *bundles* descritos na Seção 5 foram também implementados a fim de validar o conceito proposto.

6.1 DEFINIÇÃO DO MIDDLEWARE

O *middleware* escolhido para a realização dos testes foi o Ginga devido ao cenário nacional de implantação da tecnologia no país.

Como não existe uma implementação completa do Ginga e pelo fato do autor trabalhar diretamente com uma implementação proprietária da Fundação CERTI (CERTI, 2011), foi escolhido por utilizá-la para os testes. Esta implementação ainda não está completamente compatível com toda a especificação, entretanto está bem avançada e é capaz de executar aplicações em ambas as máquinas, cobrindo boa parte da API do Ginga com exceção da Ponte (interpretador NCL, módulo *persistent* do Lua, módulo *settings* do Lua, módulo *event* do Lua, módulo *canvas* do Lua e Java DTV).

6.2 DEFINIÇÃO DO FRAMEWORK OSGi

Com objetivo de definir um *framework OSGi* que se adequasse ao cenário restrito de um *middleware* de televisão digital, foram inicialmente levantado os principais produtos disponíveis. Um critério para esta seleção era a necessidade de possuírem licenças software livre, onde o código-fonte estivesse disponível. Com isto chegou-se à seguinte lista:

1. OSCAR 1.0.5: Licença BSD, mantido pelo *OW2 Consortium*.
2. Knoplerfish 3.1.0: Licença BSD, mantido pela *MakeWave*.
3. Felix 3.2.2: Licença Apache v2.0, mantido pela *The Apache Software Foundation*.
4. Equinox 3.6.2: Licença *Eclipse Public*, Mantido pela *The Eclipse Foundation*.

5. Concierge 1.0.0: Licença BSD, mantido pelo *Institute for Pervasive Computing, ETH Zurich*.
6. Concierge (*branch*) 4.2. É uma versão mais atual do Concierge que ainda não foi lançada e está em desenvolvimento.

Para análise comparativa desta lista, foram utilizados os seguintes quesitos:

1. Compatibilidade com CDC/FP/PBP: é necessário que seja possível executar o *framework OSGi* em uma JVM com a configuração CDC e perfis FP e PBP. Para validar isto, verificou-se a documentação do projeto;
2. Tamanho: o tamanho não deve ter um impacto significativo no tamanho total do *middleware*. Para validar isto, foi contabilizado o tamanho total do(s) arquivo(s) JAR da implementação do *framework*;
3. Versão de OSGi: deve ser levado em conta qual a versão da especificação do OSGi é suportada. Para validar isto, verificou-se a documentação do projeto;
4. Projeto Ativo: o projeto deve estar ativo para facilitar o suporte e ter acesso às melhorias implementadas. Para validar isto, foi verificada a data de lançamento da última versão e a atividade nas listas de emails do projeto;
5. Facilidade de integração: deve ser fácil integrá-lo. Para validar isto, foi criado uma aplicação com suporte a instalação, desinstalação e inicialização de *bundles;*
6. Tempo de inicialização: o tempo de inicialização deve ser baixo. Para validar isto, foi criado uma aplicação para invocar o *framework OSGi* e medir o tempo anterior e posterior à chamada de inicialização básica;
7. Alocação de memória: deve ser baixa. Para validar isto, foi criado uma aplicação para invocar o *framework OSGi* e medir a alocação de memória, através da coleta anterior à chamada de inicialização e posterior.

De acordo com estes testes realizados (Anexo A), foi decidido pelo uso do Concierge (*branch*), pois o mesmo apresentou um bom desempenho na questão de inicialização, tamanho e alocação de memória. A escolha de utilizar o *branch* foi devido ao fato do mesmo estar ativo e em desenvolvimento para ser compatível com a versão OSGi R4, possuindo assim uma fácil integração através da API

Framework Launcher. O uso do R4 permitirá que outros *frameworks* possam ser utilizados com poucas mudanças de código.

6.3 ESTRATÉGIA DE INTEGRAÇÃO

A estratégia de integração segue o modelo MnO1j1p (LIN, 2009), executando em uma única JVM dentro de uma única plataforma. Para abstrair as questões da integração do *framework OSGi* e gerenciamento dos *bundles*, foi criada uma classe denominada *OSGiConfigurator* (Figura 22). O *OSGiConfigurator* é responsável por inicializar o *framework* OSGi e instalar os *bundles* do tipo Serviço de *Middleware*, Extensão Proprietária e Aplicação *Widget (*regular e nativa*)*. Para isto, ele possui uma instância agregada do *BundleLocator*, que possui métodos para retornar o caminho dos *bundles* de acordo com o seu tipo, e também do *OSGiFramework* para inicializar o contêiner, instalar os *bundles* e recuperar um determinado serviço. A Figura 22 apresenta este diagrama de classes.

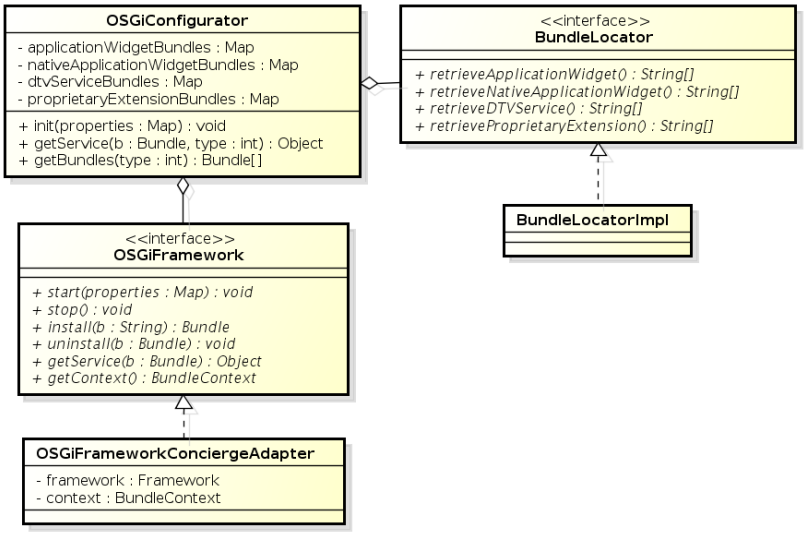

Figura 22 – Diagrama de classes da estratégia de integração

É possível desacoplar a implementação de *framework* OSGi através da realização da interface *OSGiFramework*. Está sendo utilizado um *adapter*[2] para integrar uma implementação de *framework*, no caso, o Concierge (*branch*), através do *OSGiFrameworkConciergeAdapter*. O *OSGiFrameworkConciergeAdapter* possui uma referência para a implementação padrão do Concierge. O *BundleLocatorImpl* realiza a interface *BundleLocator* e implementa seus métodos, sendo capaz de buscar numa estrutura de sistemas de arquivos os determinados caminhos para os *bundles*, que poderiam estar estruturados de maneira bem simples, por exemplo, em três pastas distintas, uma para cada tipo.

O *OSGiConfigurator* é uma referência *singleton*[3] no sistema, que pode ser inicializado nos Recursos Comuns ou no *system classloader* da JVM. Este local de inicialização é representado no diagrama de sequência da Figura 23 pelo *MW Actor*. Durante a sua inicialização, o *OSGiConfigurator* recebe um mapa de parâmetros que são encaminhados para a instância do *OSGiFramework*, responsável por criar o *Framework* e inicializá-lo através de alguma implementação interna de *factory*[4] (mensagem 1.1). O *OSGiConfigurator* então invoca uma chamada para a instância de *BundleLocator* retornar dos bundles do tipo aplicação *widget* regular (mensagem 1.2) e itera sobre esta lista de caminhos para registrar os *bundles* através de uma chamada *install* para o *OSGiFramework* (mensagem 1.3). Uma referência deste *bundle* é mantida internamente para controle de acesso (por exemplo, uma aplicação *widget* regular não pode visualizar *bundles* do tipo serviços de *middleware)*. Após isto, o procedimento de busca de caminho e instalação do *bundle* é repetido para os tipos de *bundle* aplicação *widget* nativa (mensagem 1.4 e 1.5), serviço de *middleware* (mensagem 1.6 e 1.7) e extensão proprietária (1.8 e 1.9).

[2]Padrão de projeto que permite que um objeto utilize serviços de outros objetos com interfaces diferentes por meio de uma interface única.
[3]Padrão de projeto que permite que apenas uma objeto de uma classe seja instanciado.
[4]Padrão de projeto que define uma interface para criação de objetos, entretanto permite que as subclasses especifiquem a forma disto.

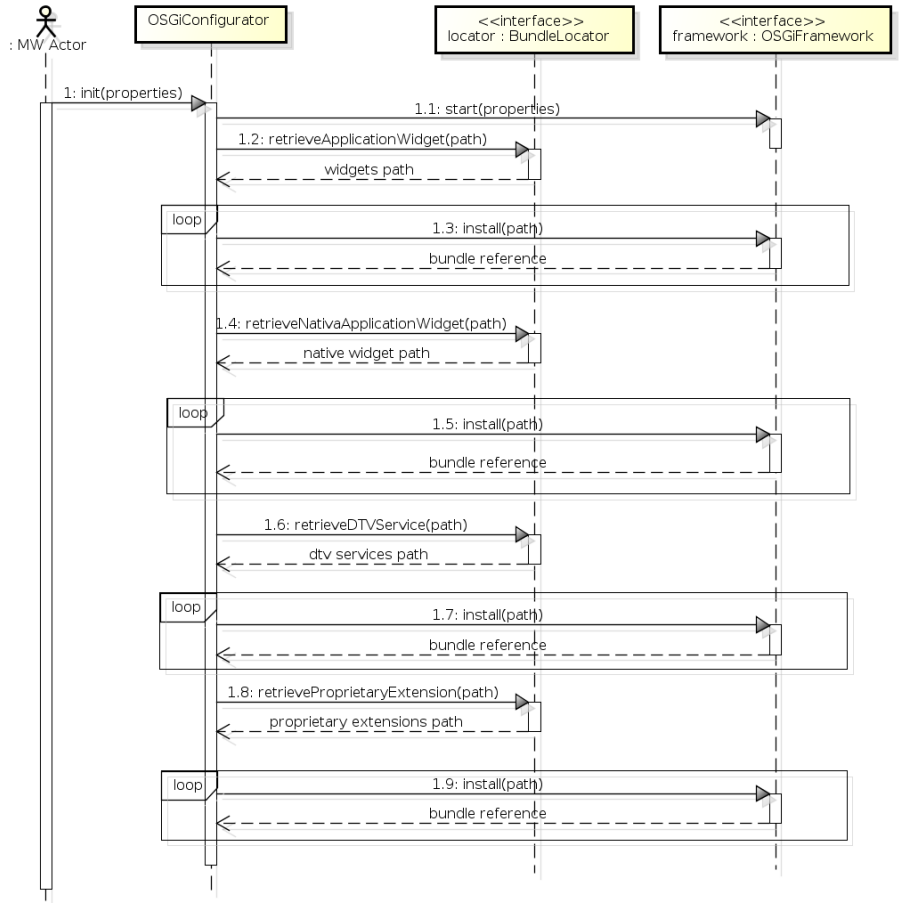

Figura 23 – Diagrama de sequência da inicialização da inicialização do OSGiConfigurator

Para que um *Xlet* possa verificar os serviços de extensões proprietárias disponíveis, foi necessário efetuar modificações no AMS (gerenciador de aplicações da máquina virtual, descrito na seção 2.2.2). Um *Xlet* em seu primeiro estado, recebe uma referência para o seu contexto através de um objeto do tipo *XletContext*. O diagrama de classes da Figura 24 apresenta as mudanças necessárias para realizar esta integração. A classe *XletBundleContext* estende a antiga implementação do *XletContext* e adiciona dois métodos: 1) para

recuperar um determinado serviço e 2) para retornar uma lista dos serviços disponíveis em formato *String*. O AMS instancia um *XletBundleContext* para cada *Xlet* e encaminha uma referência de uma *facade*[5], que externaliza os métodos disponíveis do *OSGiConfigurator* para o *XletBundleContext*. Se o *Xlet* for uma aplicação *widget* nativa, é utilizado uma *facade* sem restrições quanto a filtragem de *bundles*, chamada de *OSGiConfiguratorXletFacade*. Caso não seja uma aplicação *widget* nativa, é utilizado o *OSGiConfiguratorLimitedXletFacade*. Ambas as classes fazem chamada para o *OSGiConfigurator* especificando o tipo de *bundle* quando invocado o método *getService*. O *OSGiConfigurator*, por sua vez, faz determinada restrição lógica de acordo com a lista de serviços que possui para cada tipo de *bundle*.

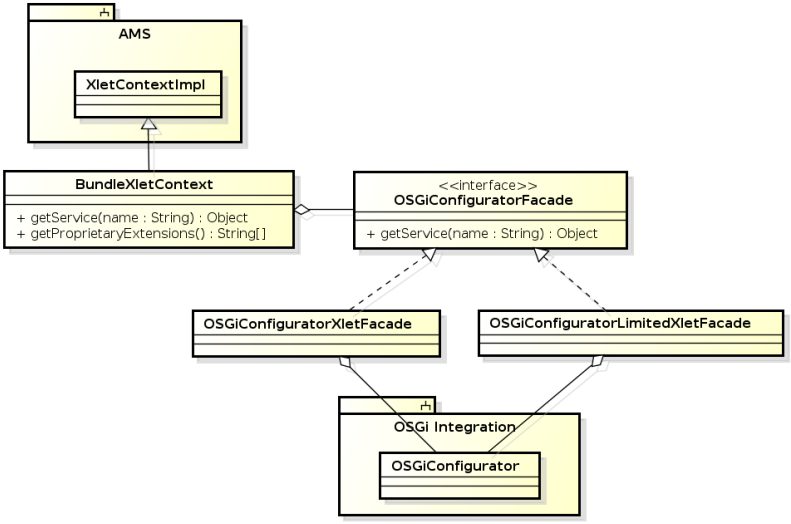

Figura 24 – Diagrama de classes da integração com AMS

Uma *bundle* do tipo serviço de *middleware*, utilizado internamente na implementação, está disponível para acesso diretamente via a referência do *OSGiConfigurator*. Os *bundles* do tipo aplicação *widget* precisam de uma modificação no AMS para que ele possa inicializar estas aplicações. O serviço de uma aplicação w*idget*

[5]Padrão de projeto que disponibiliza uma interface simplificada para uma as funcionalidades de uma API.

implementa uma interface chamada *BundleXlet* que possui métodos que representam o ciclo de vida de um *Xlet*, conforme descrito na Figura 25. Desta maneira, o AMS recupera o serviço desta aplicação e invoca os determinados métodos de acordo com a transição de estado. Esta interface estende a interface padrão do *Xlet* para manter também a compatibilidade de execução no AMS padrão.

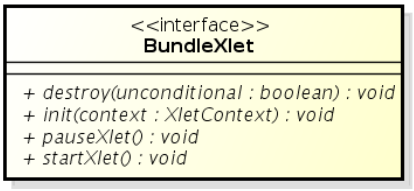

Figura 25 – Interface *BundleXlet*

6.4 PROTÓTIPOS DE BUNDLES

Esta seção descreve três tipo de *bundles* que foram desenvolvidos para validação como exemplo.

6.4.1 Serviço de Middleware (criação de imagem)

O *bundle* do tipo serviço de *middleware* para criação de imagens é parte de uma API específica chamada Java DTV (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2010d), que está disponível no Ginga-J. O Java DTV define uma classe chamada *Image*, localizado no pacote *com.sun.dtv.lwuit*, que é responsável pela instanciação de imagens que podem ser utilizadas pelo LWUIT (*Lightweight User Interface Toolkit*). O LWUIT é o *toolkit* alto nível de GUI disponível na máquina de execução. Esta classe define métodos para criação de imagens através de um caminho, parâmetros como largura e altura, entre outros. A classe *Image* não precisa de nenhum tipo de integração nativa diretamente, e se for implementada utilizando o pacote *java.awt* pode ter o código-fonte escrito puramente em Java.

Um *bundle* foi desenvolvido com um serviço que oferece a criação de imagens. O serviço foi dividido em três arquivos Java (trechos relevantes da implementação encontram-se no Anexo B):

• *DTVImageService*: a interface que declara os métodos para criação de imagens;

• *DTVImageServiceAWTImpl*: implementa a interface *DTVImageService* e contém o algoritmo utilizando o pacote *java.awt*. Poderia ser criada uma outra implementação deste serviço, por exemplo, utilizando um *toolkit* gráfico nativo diretamente sem AWT;

• *DTVImageServiceActivator*: implementa a interface *org.osgi.framework.BundleActivator* e contém os métodos para registrar e cancelar registro do serviço através do *org.osgi.framework. BundleContext*;

6.4.2 Extensão Proprietária (integração com a rede social Twitter)

O microblog Twitter (TWITTER, 2011) permite que mensagens sejam associadas a um tema, definido por uma *hashtag* (isto é, um sinal "#" seguido por uma palavra-chave) no corpo da mensagem. O Twitter fornece *Web Services* para acessar as últimas mensagens (*tweets*), que contêm uma *hashtag* específica.

Um *bundle* extensão proprietária foi criado com uma interface para recuperar *tweets* contendo uma *hashtag* específica. A interface define um método de pesquisa que recebe uma S*tring* que representa a consulta e retorna uma lista das *strings* de mensagens.

A biblioteca Twitter4J (YANAMOTO, 2011) foi utilizada para integrar à API do Twitter. A biblioteca Twitter4J possui o código-fonte implementando utilizando a sintaxe da linguagem Java 1.5, mas possui compilação (*target*) para Java 1.4, requerida pela plataforma de desenvolvimento utilizada. Alguns outros ajustes foram feitos para torná-la compatível com a plataforma, como a remoção de chamadas para o método *split* em objetos *String*, uma vez que este método não está disponível na pilha Java ME CDC.

A Figura 26 mostra essa dissociação entre a interface e a implementação real. O serviço foi dividido em três arquivos Java:

• *TwitterService*: a interface que declara o método que recebe uma String de consulta e retorna uma lista de mensagens em String.

•*TwitterServiceImpl*: implementa a interface e delega para a implementação da biblioteca Twitter4J.

•*TwitterServiceActivatorImpl*: implementa a interface *org.osgi.framework.BundleActivator* e contém os métodos para registrar

e cancelar registro do serviço através do *org.osgi.framework.BundleContext*.

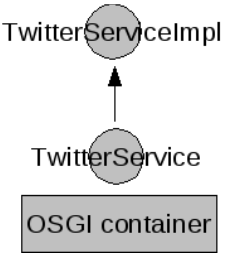

Figura 26 – *Bundle* extensão proprietária (Twitter)

Também foi necessário definir a visibilidade da biblioteca Twitter4J e seus pacotes importados no arquivo de manifesto que descreve o *bundle para que o* .

6.4.3 Aplicação Widget (cliente do Twitter)

Foi desenvolvida uma aplicação *Xlet* para testar o uso da extensão proprietária Twitter. Este aplicativo recupera os últimos *tweets* de acordo com uma *hashtag* específica.

Por exemplo, um programa de esportes poderia definir uma *hashtag* #java e usuários poderia enviar mensagens com esta *hashtag*. Desta maneira os telespectadores poderiam acompanhar a discussão filtrada pela *hastag* #java. Esta *hashtag* é compilada na aplicação *Xlet* e não é possível alterá-la dinamicamente devido a limitações desta implementação neste código de teste (Código 4 do Anexo B).

Durante o método de inicialização do *Xlet*, é realizado um *cast* do *XletContext* para o *BundleXletContext*. Isso é necessário para obter uma referência para este contexto que possui acesso aos serviços dos *bundles* registrados. Neste ponto, o aplicativo precisa saber que serviços adicionais estão disponíveis na plataforma e podem ser utilizados. Depois que o aplicativo tem uma referência para o contexto do *bundle*, ele pode recuperar o serviço do Twitter. Então, a API Java *Reflection* foi usada para chamar os métodos fornecidos pelo serviço Twitter utilizando seu objeto de referência.

Assim, quando o *Xlet* é iniciado, ele chama um método que recupera os últimos *tweets* contendo uma *hashtag* específica e mostra na tela, como ilustrado pela Figura 27.

Figura 27 – Aplicação *widget* mostrando os últimos *tweets*

6.5 MODELO DE TESTES

Os testes da implementação desta arquitetura foram realizados em uma implementação do *middleware* Ginga. Esta implementação possui uma versão para a arquitetura x86 e MIPS.

O objetivo dos testes é determinar o *overhead*, neste caso o aumento de processamento e memória, para esta solução integrada. Não é objetivo do teste proposto levantar possíveis problemas na implementação, por isto as medidas coletadas serão relativas à adição da arquitetura. Desta maneira foram consideradas as variáveis:

A. Memória. O consumo de memória em *bytes* do teste executado.
B. Tempo: A diferença em milissegundos do início e fim do teste executado.

Estas variáveis foram combinadas com os sete cenários de execução que tinham objetivo de mensurar o *overhead* na inicialização (item 1, 2 e 3) e em chamada (item 4, 5, 6 e7):

1. Inicialização do *middleware* sem a integração OSGi: O *middleware* inicializado sem considerar a integração da arquitetura proposta. Este teste serve de base para calcular o *overhead* adicional nos outros cenários descritos.

2. Inicialização do *middleware* com a integração OSGi; O *middleware* inicializado com a integração da arquitetura proposta, entretanto sem nenhum *bundle* para ser instalado.
3. Inicialização do *middleware* com a integração OSGi e o *bundle* "Olá Mundo" instalado: O *bundle* "Olá Mundo" disponibiliza um serviço que imprime tal texto no console. Este serviço representa a um *bundle* muito simples, que será utilizado como referência para valores de uma baixa alocação de memória e processamento. Ele foi instalado como extensão proprietária.
4. *Xlet* que imprime "Olá Mundo": *Xlet* que apenas imprime o texto "Olá Mundo" no console.
5. *Xlet* que invoca um serviço "Olá Mundo": *Xlet* que faz uma chamada para o serviço do *bundle* "Olá Mundo", responsável por imprimir o texto "Olá Mundo" no console. É análogo ao cenário 4, entretanto faz uso de um *bundle* extensão proprietária.
6. *Xlet* que cria uma imagem via API do *middleware*: Xlet que faz uma chamada para a classe *Image* do Java DTV, responsável por criar uma imagem, cuja altura e largura são especificadas como parâmetros da chamada.
7. *Xlet* que cria uma imagem via API do *middleware* (implementação com *bundle*): Xlet que faz uma chamada para a classe *Image* do Java DTV, responsável por criar uma imagem na tela. A implementação de *Image* faz uso do *bundle* Imagens (seção 6.4.1), que disponibiliza um serviço que retorna uma imagem com determinada altura e largura. Este serviço representa um *bundle* real para ser utilizado por *middlewares*. É análogo ao cenário 6, entretanto a implementação faz uso de um *bundle* do tipo serviço de *middleware*.

Desta maneira, as duas variáveis foram combinadas com os cenários de testes, que foram realizados nos dois ambientes descritos (x86 e MIPS). Com objetivo de aumentar a confiabilidade dos resultados, cada item de teste foi realizado 100 vezes através de um *script* automatizado, reiniciando a JVM em cada teste. Os dados foram coletados e análises estatísticas realizadas obtendo medidas tais como mediana, desvio padrão, média e os quartis. O código foi instrumentado usando chamadas para capturar o estágio inicial do tempo e memória livre, bem como o estágio final. Para capturar o tempo corrente, foi utilizado a chamada Java *System.currentTimeMillis()* e para a memória

disponível foi utilizado a chamada *Runtime.freeMemory()*, ambos disponíveis no pacote padrão da linguagem (*java.lang)*.

É importante destacar que a JVM faz uso de mecanismos de *garbage collector* (GC) para seu gerenciamento de memória automático e ele não é controlado pelo programador. A especificação da JVM não define uma forma como o GC deve funcionar e isto é de responsabilidade da implementação (LONDHOLM, 1999). Ela apenas define que uma área de memória *heap* é criada quando a JVM é inicializada. Desta forma, para evitar possíveis coletas no *heap* pelo *GC* nos testes envolvendo a variável tempo, foram feitos alguns ajustes finos (*fine tuning)*, como o aumento da memória *heap* padrão para 15MB, que seria suficiente para a execução dos testes e alocação das classes de teste com folga. O tamanho da memória *heap* é dependente da plataforma alvo e do projeto do produto. Desta maneira, este aumento da memória *heap* para os testes poderia mascarar algum problema na implementação da arquitetura, entretanto foi realizado para evitar qualquer variação devido aos ambientes distintos, em especial a JVM.

No teste de inicialização, a lógica inicial de captura foi inserida no carregamento da classe (bloco estático) do AMS , gerenciador de aplicação, e a lógica final foi inserida na invocação do método de inicialização (*startXlet*) de um Xlet de teste que não possui uma implementação vazia. Neste cenário de teste, o *framework* OSGi foi inicializado pelo próprio AMS de forma a medir o tempo relativo para inicializar e instalar os *bundles*.

Nos testes de chamada, a lógica inicial de captura foi inserida dentro da inicialização do Xlet (*startXlet*) e a final ao fim deste método. Desta maneira foi mensurada apenas esta diferença.

6.5.1 Execução dos Testes

Os testes realizados na arquitetura x86 foram executados em um PC com o sistema operacional Ubuntu 9.10, processador Intel Core 2 Duo 2.4 GHz, 3GB de RAM. Para a escolha da JVM neste ambiente, buscou-se uma compatível com a configuração CDC e os perfis FP e PBP, possuindo também um *target* de compilação para x86. Além disto, era desejado o uso de uma JVM madura e de preferência que possuísse alguma licença de software livre. Desta maneira, decidiu-se pelo uso do *phoneMe Advanced* (JAVANET, 2011), versão MR 2, que é um projeto disponibilizado com a licença GPL-2.0 e amplamente utilizado. O

phoneME Advanced foi ainda compilado para utilizar o *toolkit* gráfico *DirectFB* na versão 1.4.3 (DIRECTFB, 2011).

Os testes realizados na arquitetura MIPS foram realizados utilizando um ambiente próximo do real, com um sistema operacional Linux embarcado, MIPS 32 bits, família 24K, de 450 Mhz, e com 512MB de memória disponíveis na plataforma. Foi utilizada uma JVM comercial, diferente da *phoneME Advanced*, que estava integrada à plataforma, compatível com CDC/FP/PBP.

6.5.1.1 Inicialização do Middleware

Foram coletados os tempos de inicialização e alocação de memória do *middleware* sem a integração OSGi (1), com a integração OSGi (2) e com a integração OSGi e o *bundle* "Olá Mundo" instalado (3).

No ambiente x86, houve um aumento do tempo médio de inicialização com a adição de OSGi (Tabela 4 e Figura 28). Isto representa um aumento de 8,85% do cenário 1 o 2, sendo que este aumento é maior para o cenário 3, em torno de 11,5% (Tabela 5). Entretanto, a diferença total em milisegundos não é tão significativa considerando o valor médio de 41,88. No ambiente MIPS, houve uma variação proporcionalmente maior de 2 para o 3, totalizando um valor absoluto de 1007,11 milisegundos. Considerando que o *framework* OSGi é inicializado uma vez na plataforma, isto não seria um problema recorrente, logo o tempo decorrido poderia ser aceitável em determinado conjunto de requisitos. Este incremento, no contexto do MIPS, pode ser explicado pelo tempo de resposta para acesso ao sistema de arquivos para a localização do arquivo do *bundle* e instalação. Isto necessita ser investigado mais detalhadamente na plataforma para identificar formas de reduzir o tempo de carregamento do *bundle*.

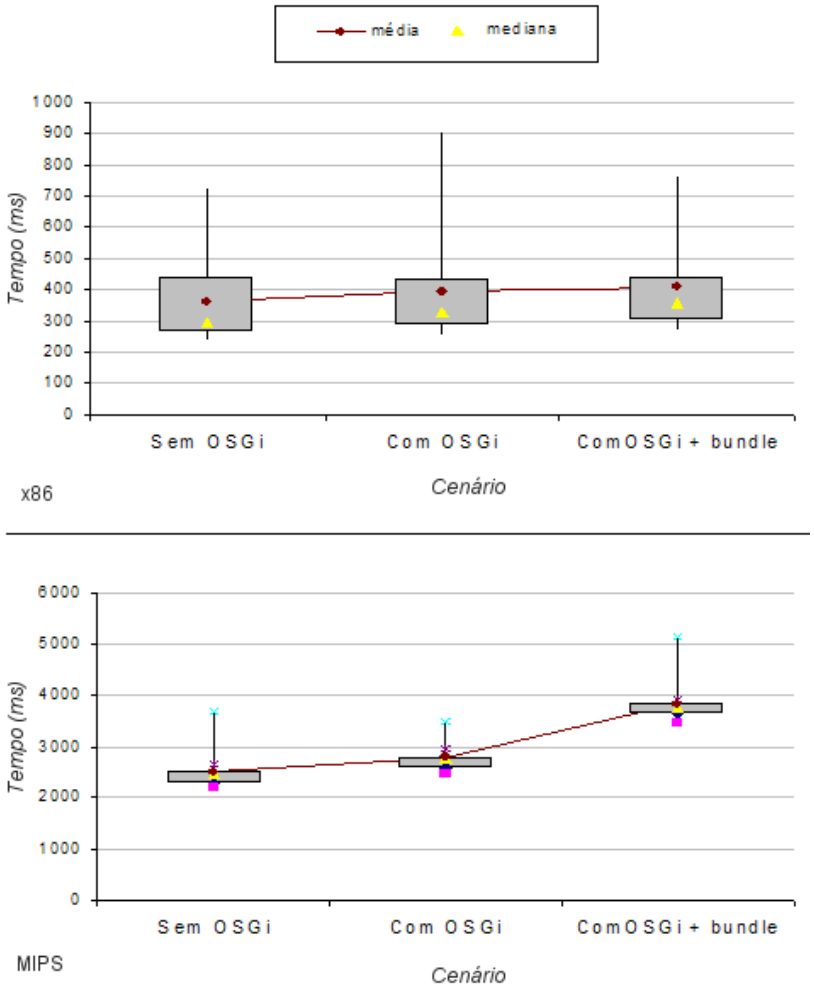

Figura 28 – Tempos médios de inicialização

Tabela 4- Tempos de inicialização

Cenário	Menor	Média	Maior	Quartil Inferior	Quartil Superior	Mediana	Desvio Padrão
Sem OSGi (x86)	237	364,38	719	267,75	439,75	293	138,79

Com OSGi (x86)	255	396,63	901	287,75	438	329	150,93
Com OSGi + bundle (x86)	272	406,26	756	301,25	441,5	358	145,21
Sem OSGi (MIPS)	2191	2518,15	3691	2301,5	2665,25	2442,5	282,09
Com OSGi (MIPS)	2442	2804,09	3500	2596	2963,5	2773,5	260,82
Com OSGi + bundle (MIPS)	3452	3811,2	5121	3615,25	3905,25	3770,5	289,17

Tabela 5- Diferença dos tempos médios de inicialização

Cenário	**Milisegundos (ms)**	**Percentual (%)**
Com OSGi - Sem OSGi (x86)	32,25	8,7
Com OSGi e bundle – Com OSGi (x86)	9,63	2,42
Com OSGi e bundle – Sem OSGi (x86)	41,88	11,49
Com OSGi - Sem OSGi (MIPS)	285,94	11,35
Com OSGi e bundle – Com OSGi (MIPS)	1007,11	35,91
Com OSGi e bundle – Sem OSGi (MIPS)	1293,05	51,34

Existe um aumento na alocação de memória *heap* com a integração do OSGi (Figura 29 e Tabela 6). Evidentemente isto é devido ao carregamento de classes internas da implementação do *framework* e estrutura de dados para o gerenciamento. No ambiente x86, houve um aumento médio de 5,9% do cenário 1 para o 2. Este aumento é muito maior com a adição do *bundle*, em torno de 156,6%, conforme mostrado na Tabela 7. Neste caso há ainda um carregamento tardio de outras classes na memória *heap* para o gerenciamento dos *bundles*, entretanto é possível que se possa otimizar investigando a JVM. Este aumento representa uma diferença total de 1013KB do cenário *C* para o A, que relativamente, dependendo dos requisitos e da memória disponível, também poderia ser aceitável. No ambiente MIPS, existe uma variação aproximada de 601KB do *middleware* sem a integração OSGi para o com a integração OSGi e *bundle*, que também poderia ser aceitável considerando um área total de *heap* de 15MB. Neste cenário

representaria em torno de 36,9% de aumento na alocação de memória. Este valor não aumentaria proporcionalmente com a adição de *bundles*, já que as classes base estariam carregadas na área de *heap* permanente.

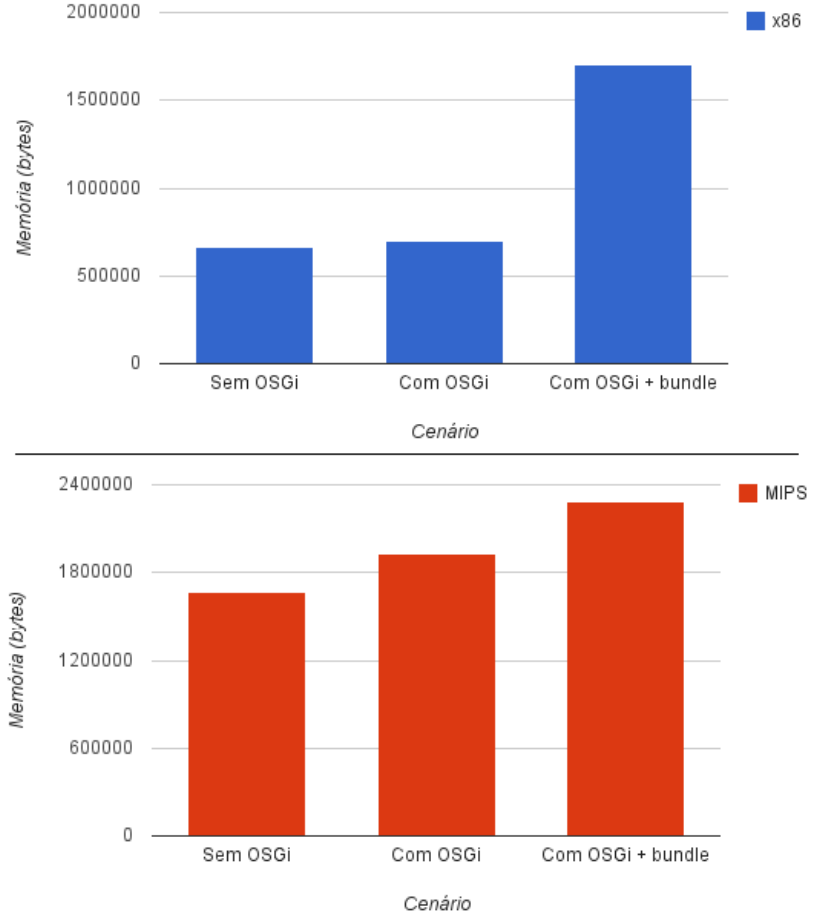

Figura 29 – Valor de alocação de memória na inicialização

Tabela 6- Valor de alocação de memória na inicialização

Cenário	Sem OSGi	Com OSGi	Com OSGi + bundle
x86	662077	701334	1699002
MIPS	1667122	1931889	2283396

Tabela 7- Diferença do valor de alocação de memória na inicialização

Cenário	Com OSGi- Sem OSGi	Com OSGi e bundle - Com OSGi	Com OSGi e bundle - Sem OSGi
x86	39257 (5,9%)	997668 (142,2%)	1036925 (156,6%)
MIPS	264767 (15,8%)	351507 (18,1%)	6162274 (36,9%)

6.5.1.2 Chamada em Xlet

Foi desenvolvido um *Xlet* que faz uma chamada ao método *createImage* da API Java DTV. Este método retorna um objeto do tipo *com.sun.dtv.lwuit.Image*, passando como parâmetro um valor inteiro de altura e outro de largura. Foram realizados testes encapsulando esta lógica em um *bundle* do tipo extensão de *middleware* regular (seção 6.4.1, referenciado como com OSGi) e também com a implementação padrão sem integração com a arquitetura (referenciada como sem OSGi).

Conforme a Figura 30 e Tabela 8, é possível verificar que o tempo médio de inicialização tende a ser próximo de zero na invocação do método, tanto no cenário x86, com valores médios de 2 milisegundos, como no cenário MIPS. Neste último a chamada leva mais tempo, devido principalmente à integração da JVM com o *toolkit* gráfico nativo, onde a diferença média apresentada foi de apenas 1,76 milisegundos, que representaria 3,43% (Tabela 9). Dependendo dos requisitos, isto também não seria impactante, bem como seria ainda possível verificar se o *overhead* é originado no baixo nível ou na JVM.

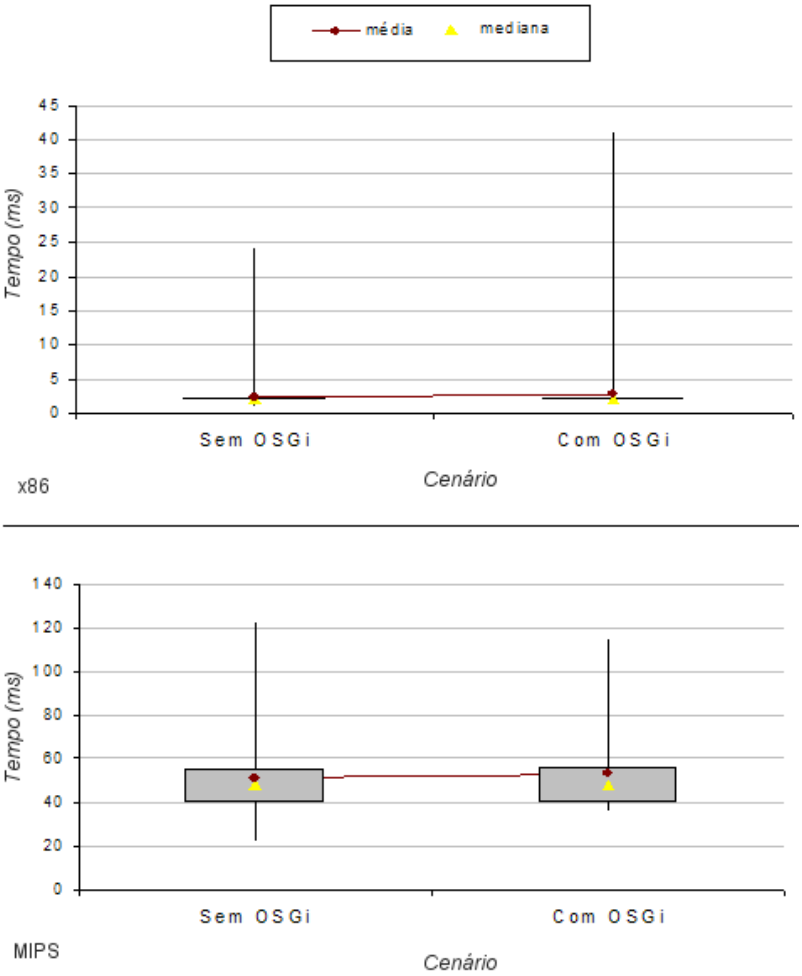

Figura 30 – Tempo médio de invocação do createImage

Tabela 8- Tempos de invocação do createImage

Cenário	Menor	Média	Maior	Quartil Inferior	Quartil Superior	Mediana	Desvio Padrão
Sem OSGi (x86)	1	2,34	24	2	2	2	2,22

Com OSGi (x86)	2	2,69	41	2	2	2	4,11
Sem OSGi (MIPS)	22	51,3	122	40,75	55	48	19,29
Com OSGi (MIPS)	36	53,06	115	40	56	48,5	18,57

Tabela 9- Diferença dos tempos médios de invocação do createImage

Cenário	**Milisegundos (ms)**	**Percentual (%)**
Com OSGi - Sem OSGi (x86)	0,35	14,95
Com OSGi - Sem OSGi (MIPS)	1,76	3,43

Houve um aumento na alocação de memória de 2,84KB para 6,18KB no x86, conforme exibido na Tabela 10. Este valor é um pouco mais que o dobro (Figura 31). No MIPS, houve uma variação de aproximadamente 5,26KB. De fato existe uma alocação maior na chamada de serviços de *bundles*, devido principalmente a toda a camada de gerenciamento do *framework* e da implementação da arquitetura. No entanto, este tipo de alocação de memória *heap* que é originada pela invocação dos *bundles* tende a ser desalocada pelo GC da JVM durante a execução, visto que a mesma não mantém nenhum tipo de referência fixa. Como foi dito, na especificação da JVM não existe obrigatoriedade quanto à forma de implementação do GC, entretanto em alguns testes práticos realizados, notou-se que o GC é invocado quando se necessita de espaço *heap*, logo ele segue uma curva de crescimento que é interrompida por quedas bruscas de liberação de memória.

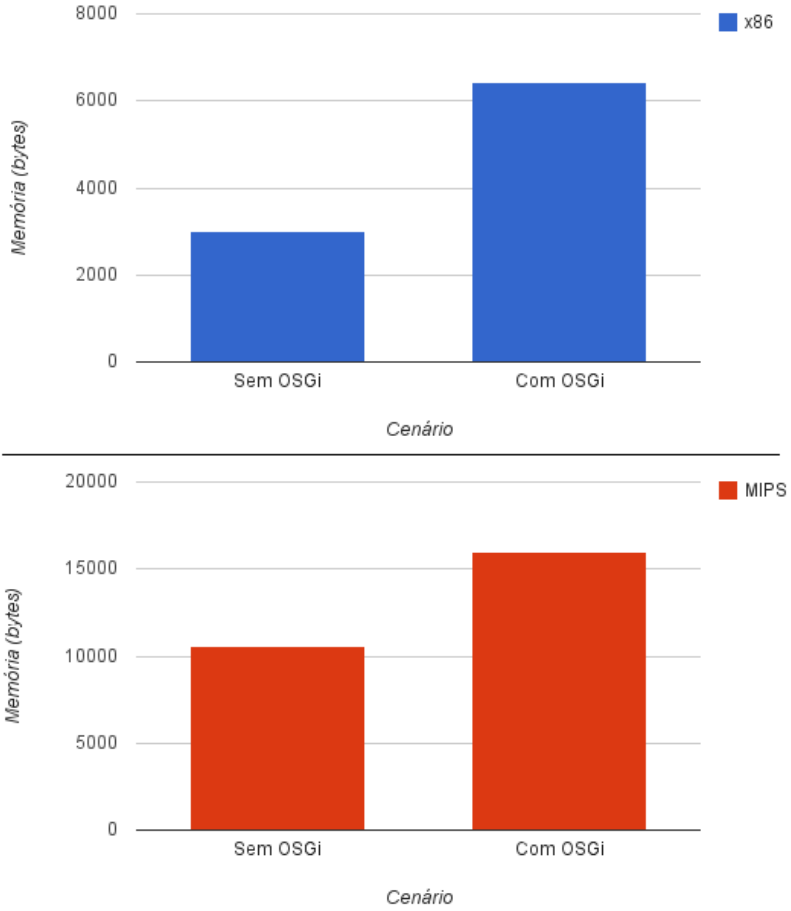

Figura 31 – Valor de alocação na invocação do createImage

Tabela 10- Valor de alocação de memória na invocação do createImage

Cenário	Sem OSGi (bytes)	Com OSGi (bytes)
x86	3016	6441
MIPS	10580	15968

Tabela 11- Diferença do valor de alocação de memória na invocação do createImage

Cenário	Alocação (bytes)	Percentual (%)
Com OSGi- Sem OSGi (x86)	3425	113,5
Com OSGi- Sem OSGi (MIPS)	5388	50,9

Foi também implementado um *Xlet* que imprime “Olá Mundo” no console. Os testes foram realizados encapsulando esta chamada dentro de um *bundle* extensão proprietária regular (descrito como com OSGi) e outro que imprimia diretamente o texto diretamente sem a chamada ao *bundle (*descrito como sem OSGi). Foram coletados valores muito baixos que traziam dificuldades de medir com precisão. No ambiente x86, o tempo médio das chamadas eram muito pequenos (menores que 1 milisegundo) em ambos os casos (Tabela 12). De qualquer maneira, foi possível verificar que o *overhead* de processamento neste cenário era muito baixo.

No MIPS, entretanto ocorreu um aumento em torno de 10, 72 milisegundos na chamada (Figura 32 e Tabela 13). Isto pode ser crítico em alguns cenários, por exemplo, como em uma função primitiva de pintura chamada recorrentemente. Desta maneira, é necessário investigar na plataforma qual o recurso que tende a possuir maior demanda, se na JVM ou sua integração nativa.

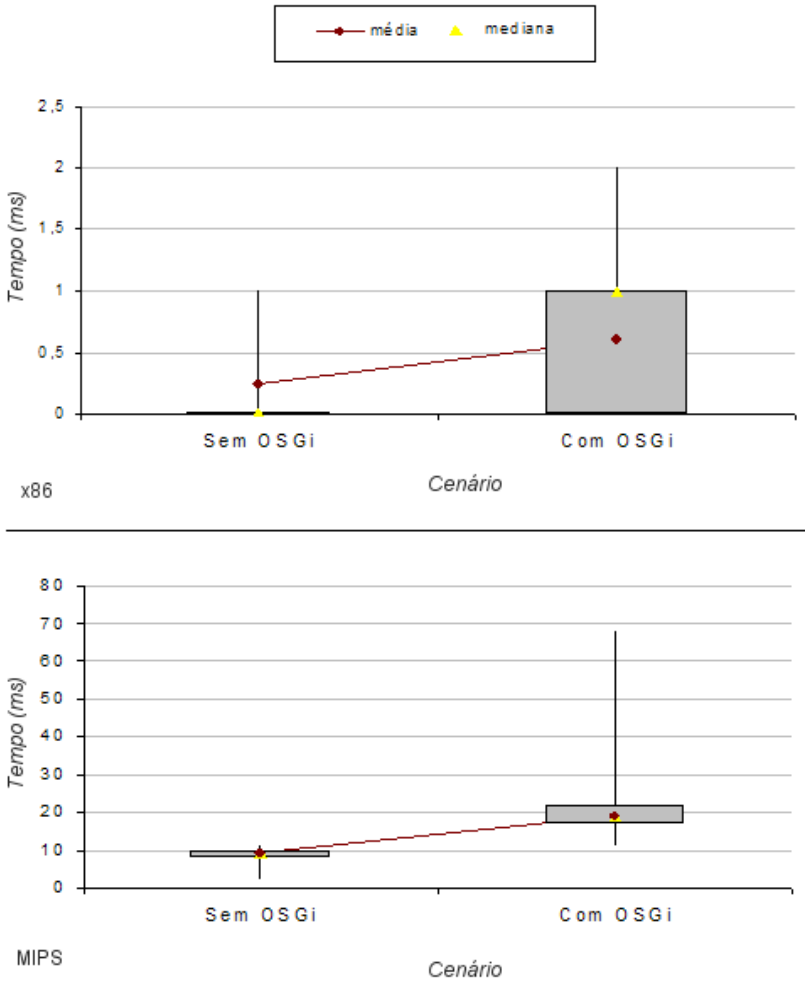

Figura 32 – Tempo médio de invocação do "Olá Mundo"

Tabela 12- Tempos de invocação do "Olá Mundo"

Cenário	Menor	Média	Maior	Quartil Inferior	Quartil Superior	Mediana	Desvio Padrão
Sem OSGi (x86)	0	0,24	1	0	0	0	0,42

Com OSGi (x86)	0	0,6	2	0	1	1	0,51
Sem OSGi (MIPS)	2	9,01	11	8	10,25	9,5	1,98
Com OSGi (MIPS)	11	19,73	68	17	22,25	19	6,62

Tabela 13- Diferença dos tempos médios de invocação do "Olá Mundo"

Cenário	**Milisegundos (ms)**	**Percentual (%)**
Com OSGi- Sem OSGi (x86)	0,36	150
Com OSGi- Sem OSGi (MIPS)	10,72	118,97

Houve um aumento na alocação de memória de aproximadamente 0,35KB para 4,05KB, no cenário x86, conforme descrito na Tabela 14. Este aumento proporcionalmente é significante, em torno de 1029%, mas como a chamada "Olá Mundo" possui uma lógica muito simples, seria esperado um aumento proporcional alto com a adição da camada de gerenciamento. Além disto, este aumento deixa evidente um *overhead* de alocação de memória da chamada OSGi (Figura 33), que neste caso é expresso pela média da diferença das chamadas no valor de 3,70KB (Tabela 15). No cenário MIPS, houve uma variação de aproximadamente 5,35KB. Conforme citado anteriormente, ambos tenderiam a ser desalocados pelo GC.

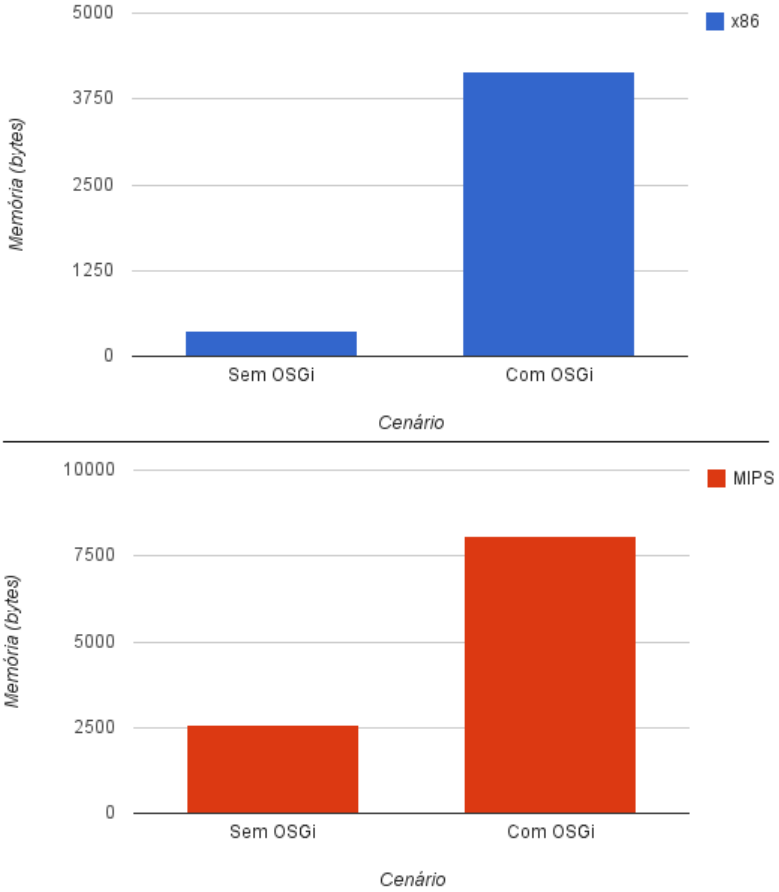

Figura 33 – Valor de alocação da invocação do ”Olá Mundo”

Tabela 14- Valor de alocação de memória na invocação do “Olá Mundo”

Cenário	Sem OSGi (bytes)	Com OSGi (bytes)
x86	368	4156
MIPS	2576	8062

Tabela 15- Diferença do valor de alocação de memória na invocação do "Olá Mundo"

Cenário	Alocação (bytes)	Percentual (%)
Com OSGi- Sem OSGi (x86)	3788	1029
Com OSGi- Sem OSGi (MIPS)	5486	212,9

Já era esperada uma grande variação dos resultados obtidos em ambos ambientes (x86 e MIPS). Isto porque o ambiente MIPS reflete uma plataforma real, que é caracterizada por ser mais restrita e com limitações de recurso. Além disto, existem soluções diferentes nas camadas abaixo do *middleware*, com implementações distintas, que podem possuir gargalos, como, por exemplo, a JVM. Desta maneira, fica evidente a necessidade de investigar cada plataforma e realizar ajustes e *tuning* para alcançar um resultado mais satisfatório.

Os testes no ambiente x86 mostraram resultados iniciais satisfatórios, que apesar um pouco distante do real, poderiam mostrar a viabilidade da solução. Os testes no ambiente MIPS mostram que é possível a integração da solução, entretanto é necessário efetuar otimizações na plataforma em questão para identificar e corrigir possíveis gargalos.

As chamadas a *bundles* ficaram muito próximas em ambos ambientes e cenários (com OSGi e sem OSGI), com exceção do "Olá Mundo" no MIPS. Em ambos ambientes houve um aumento na alocação de memória *heap* em cada chamada, que tende a ser desalocada pelo GC durante a execução. Entretanto em ambientes com pouca memória *heap* disponível, isto poderia ser um problema, dado que esta invocação do GC consome processamento e o usuário poderia ter a impressão de travamentos constantes. Além disto, também houve um aumento no tempo de inicialização, devido ao próprio *framework* OSGi. A inicialização é realizada apenas uma vez e o valor absoluto não era tão crítico, mas teria que avaliar melhor os requisitos para verificar se poderia ser um problema. Um ponto importante é o aumento na alocação de memória *heap* com a inicialização do *framework*, já que seria um espaço que teria que ser previsto previamente para o funcionamento.

7 CONCLUSÃO E TRABALHOS FUTUROS

Foi possível desenvolver uma proposta de arquitetura que fosse compatível com os principais *middlewares* abertos de televisão digital (objetivo específico 1). Através do uso do OSGi é possível modularizar a implementação do *middleware* (objetivo específico 2), bem como permitir novas extensões através dos *bundles* de extensão proprietária (objetivo específico 3). O protótipo implementado (objetivo específico 4) e os resultados coletados (objetivo específico 5) demonstram que pode ser viável o uso de um *framework* OSGi em *middlewares* de televisão digital.

O uso de um *framework* OSGi traz uma série de benefícios para manutenção e reuso da arquitetura interna ao *middleware*. A proposta de modularizar o *middleware* traz benefícios para rápida componentização de produtos e manutenção. É possível não apenas adicionar funcionalidades, mas também utilizá-lo para disponibilizar serviços às implementações e padronização para criação de aplicativos. Um fator negativo é que ele é dependente de uma JVM, logo não se aplicaria em cenários que não a disponibilizassem, mas que no entanto está disponível em *middlewares* de televisão digital, especificamente na máquina de execução, descrita nas especificações abertas citadas. A arquitetura proposta também pode trazer competitividade à indústria, de forma que um fabricante de *middleware* poderia desenvolver tipos de *bundles* comuns que poderiam ser utilizados em implementações de *middlewares* diferentes ou aplicações executadas em plataformas distintas.

Os resultados dos testes realizados demonstram que diferentes implementações de JVM irão ter desempenhos diferentes em plataformas distintas. Particularmente, há diferenças significativas entre um ambiente em PC e um ambiente embarcado e restrito, como uma televisão. Desta forma, é importante a realização de testes envolvendo o cenário específico em questão. A partir destes testes específicos é possível avaliar o que é passível de otimização e os impactos causados pelas alterações efetuadas. Com isto, é possível decidir pelo uso, total ou parcial da solução. O uso parcial poderia ser compreendido pela não implementação de todos tipos de *bundles*. Por exemplo, em um contexto onde existe um *overhead* muito alto de processamento, poderia fazer sentido apenas o uso de *bundles* do tipo aplicações *widgets*, já que um

bundle do tipo serviço de *middleware* que estivesse relacionado a renderização de componentes poderia trazer impactos visuais.

Os conceitos propostos poderiam ser utilizados também em outros ambientes que não em receptores convencionais de televisão digital, como por exemplo, no ambiente móvel (celular). A proposta (GAMA, 2007) segue esta idéia, entretanto seu detalhamento compreende apenas o nível conceitual e não são apresentados detalhes de implementação e viabilidade, que são descritos neste trabalho no contexto de televisão digital. O trabalho (MIRANDA, 2008) define um modelo de componentes que pode ser utilizado para a implementação da camada de Recursos Comuns, que estaria mais relacionado a implementação do *middleware*, entretanto não define uma forma que aplicações poderiam utilizar-se de extensões ou serem instaladas. Este modelo de componentes poderia ser utilizado em conjunto para o desenvolvimento desta camada. O trabalho (LANGE, 2002), por sua vez, define uma infraestrutura proprietária não dependente de JVM, mas que não pode ser aplicada facilmente em outros *middlewares* de televisão digital que possuem especificações abertas.

Nenhum dos trabalhos citados envolvendo OSGi (TKACHENKO, 2006) (REDONDO, 2007) (YANG, 2007) (LIN, 2008) (LUCENA, 2009) deixa claro a realização de testes para a escolha do *framework* OSGi. Nos testes realizados desta proposta, o Concierge apresentou melhor desempenho (de acordo com os critérios definidos), o que é diferente da escolha dos trabalhos citados. Estes trabalhos apenas realizaram testes em ambientes de PC e utilizam JVMs que não são consideradas como soluções finais, já que não seguem a configuração CDC, como a JVM Sun 1.4, com exceção do (TKACHENKO, 2006). Desta forma, não estava claro a viabilidade do uso de um *framework* OSGi nestes ambientes restritos como em *middlewares* de televisão digital. Além disto, nenhum destes trabalhos propõe uma categorização do uso de *bundles* - detalhados nesta proposta como serviços de *middlewares*, extensões proprietárias e aplicações *widgets* - e nem suportam a implementação de *middleware* utilizando estes *bundles*, limitando-se ao uso de estender o *middleware* para fins de automação residencial.

7.1 TRABALHOS FUTUROS

Trabalhos futuros poderiam compreender a implementação completa de um *middleware* utilizando a arquitetura proposta, de forma a extrair um conjunto de *bundles* do tipo serviços de *middleware* que poderiam ser utilizados em diferentes *middlewares* de televisão digital. Desta maneira, poderia ser criado um repositório que seria utilizado para o desenvolvimento das diferentes especificações.

Com relação ao desenvolvimento de *bundles*, seria interessante a criação de um conjunto de aplicações *widgets* e realização de estudo das novas camadas necessárias para a integração destas, por exemplo, com uma loja virtual, que poderia trazer novas necessidades envolvendo segurança e integração com pagamento. É também possível realizar o *porting* de *bundles* utilizados em diferentes plataformas para o tipo extensão proprietária regular, de maneira que seja possível disponibilizá-los em uma implementação de *middleware*, facilitando o reuso destes componentes.

É importante a realização de novos testes em ambientes distintos para que possam ser verificadas novas restrições e recomendações de uso. Estes ambientes distintos englobariam processadores, memória, JVMs e *chipsets*, por exemplo. Também seria interessante a realização de mais testes para verificar possíveis otimizações na implementação da camada de gerenciamento da arquitetura proposta.

Também é necessário analisar a possibilidade do desenvolvimento de mecanismos para que ambientes declarativos possam fazer uso de extensões proprietárias via Ponte ou serem empacotados como aplicações *widgets*. No que tange o domínio de automação residencial, seria importante o desenvolvimento de uma solução completa utilizando os *bundles* propostos integrado a um *middleware* com a implementação da arquitetura.

7.2 TRABALHOS PUBLICADOS

A pesquisa decorrente do presente resultou na publicação de um artigo (GHISI, 2011) sobre a arquitetura citada e a categorização dos tipos de *bundles*. Além disto, durante a realização deste trabalho,

também foram publicados mais dois artigos, indiretamente relacionados com o tema, sendo um sobre aplicações interativas com integração de redes sociais (GHISI, 2010a) e outro sobre modelos conceituais de T-Commerce no Brasil (GHISI, 2010b).

REFERÊNCIAS

APACHE SOFTWARE FOUNDATION. **Apache Felix**. Disponível em: <http://felix.apache.org/site/index.html>. Acesso em: 16 dez. 2010.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15606-1**. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital, Parte 1: Codificação de dados. Rio de Janeiro, 2010a.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15606-2**. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital, Parte 2: Ginga-NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações. Rio de Janeiro, 2011.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15606-3**. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital, Parte 3: Especificação de transmissão de dados. Rio de Janeiro, 2010b.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15606-4**. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital, Parte 4: Ginga-J - Ambiente para a execução de aplicações procedurais. Rio de Janeiro, 2010c.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 15606-6**. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital - Parte 6: Java DTV 1.3. Rio de Janeiro, 2010d.

ASSOCIATION OF RADIO INDUSTRIES AND BUSINESSES. **ARIB STD-B23**, Version 1.1-E1 , Application Execution Engine Platform for Digital Broadcasting. ARIB Standard, 2004.

ASSOCIATION OF RADIO INDUSTRIES AND BUSINESSES. **ARIB STD-B24**, Version 5.2-E1, Volume 2: Data Coding and Transmission Specification for Digital Broadcasting, ARIB Standard, 2002.

ADVANCED TELEVISION SYSTEMS COMMITTEE. **ATSC A-101**. DTV Application Software Environment Level 1 (DASE-1) PART 1: Introduction, Architecture, and Common Facilities, 2003a.

ADVANCED TELEVISION SYSTEMS COMMITTEE. **ATSC A-102**. DTV Application Software Environment Level 1 (DASE-1) PART 2: Declarative Applications and Environment, 2003b.

ADVANCED TELEVISION SYSTEMS COMMITTEE. **ATSC A-103**. DTV Application Software Environment Level 1 (DASE-1) PART 3: Procedural Applications and Enviromnent, 2003c.

BRASIL. **Decreto nº 5.820 -** 29 de junho de 2006. Dispõe sobre a implantação do SBTVD-T, estabelece diretrizes para a transição do sistema de transmissão analógica para o sistema de transmissão digital do serviço de radiodifusão de sons e imagens e do serviço de retransmissão de televisão, e dá outras providências. Diário Oficial da União, 30 jun. 2006, p51.

CERTI. **Fundação CERTI**. Disponível em: <http://certi.org.br>. Acesso em: 10 mar. 2011.

DIRECTFB. **DirectFB**. Disponível em <http://directfb.org>. Acesso em: 16 jan. 2011.

ECLIPSE FOUNDATION. **Eclipse IDE**. Disponível em: <http://www.eclipse.org>. Acesso em: 16 dez 2010a.

ECLIPSE FOUNDATION. **Equinox**. Disponível em: <http://www.eclipse.org/equinox>. Acesso em: 16 dez. 2010b.

EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. **ETSI TS 102 812**. Digital Video Broadcasting (DVB), Multimedia Home Platform (MHP) Specification 1.1.3. Junho de 2003.

GAMA, Kiev Santos. **An osgi middleware for mobile digital tv applications**. Mobility ’07: Proceedings of the 4th international conference on mobile technology, applications, and systems and the 1st

international symposium on Computer human interaction in mobile technology, pages 690–695, New York, NY, USA, 2007.

GHISI, Bruno Cavaler; SIQUEIRA, Frank. **An extensible architecture for DTV middleware**. EuroITV'11, Portugal, 2011.

GHISI, Bruno Cavaler; LOPES, Guilherme Figueiredo; SIQUEIRA, Frank. **Integração de Aplicações para TV Digital Interativa com Redes Sociais**. Workshop de TV Digital Interativa, Webmedia 2010, Brasil, 2010a.

GHISI, Bruno Cavaler; LOPES, Guilherme Figueiredo; SIQUEIRA, Frank. **Conceptual Models for T-Commerce in Brazil**. Workshop on Interactive Digital TV in Emergent Countries, EuroITV'10, Finland, 2010b.

INSTITUTE FOR PERVASIVE COMPUTING. **Concierge OSGI**. Disponível em: <http://concierge.sourceforge.net>. Acesso em: 16 dez. 2010.

JAVA COMMUNITY PROCESS. **JSR 218**: Connected Device Configuration 1.1, 2005a.

JAVA COMMUNITY PROCESS. **JSR 219**: Foundation Profile 1.1, 2005b.

JAVA COMMUNITY PROCESS. **JSR 927**: Java TV API 1.1. 2006.

JAVA COMMUNITY PROCESS. **JSR 217**: Personal Basis Profile 1.1.2. 2005c.

JAVA COMMUNITY PROCESS. **JSR 272**: Mobile Broadcast Service API for Handheld Terminals. 2008.

JAVANET. **GlassFish Open Source Application Server**. Disponível em <https://glassfish.dev.java.net>. Acesso em: 16 dez. 2010.

JAVANET. **PhoneME Advanced**. Disponível em: <http://phoneme.java.net>. Acesso em: 16 jan. 2011.

KOWALSKY, Robert. **Algorithm = Logic + Control**. Communications of the ACM 22, 424-436, Inglaterra, 1979.

LANGE, Fons de. **The Philips-Open TV Product Family Architecture for Interactive Set-Top Boxes**. Proceeding PFE '01 Revised Papers from the 4th International Workshop on Software Product-Family Engineering, Inglaterra, 2002.

LEITE, Luiz Eduardo; BATISTA, Carlos Eduardo C. F.; SOUZA, Guido Lemos de; KULESKA, Raoni, ALVES, L.G.P.; BRESAN, G.; RODRIGUES, Rogério Ferreira; SOARES, Luiz Fernando Gomes. **FlexTV- Uma Proposta de Arquitetura de Middleware para o**

Sistema Brasileiro de TV Digital. Revista de Engenharia de Computação e Sistemas Digitais, n. 2, p. 2949, nov. 2005.

LERUSALIMSCHY, Roberto. **Programming in Lua**, second edition. 2006.

LONDHOLM, Tim; YELLIN, Frank. **The Java Virtual Machine Specification**, Second Edition, Section 3.5.3. Addison-Wesley, 1999.

LIN, Cheng-Liang; WANG, Pang-Cheieh; HOU, Ting-Wei. **A Wrapper and Broker Model for Collaboration between a Set-top box and Home Service Gateway**. IEEE Transactions on Consumer Electronics, 54(3):1123--1129, August 2008.

LIN, Cheng-Liang; WANG, Pang-Chieh; HOU, Ting-Wei. **Classification and Evaluation of Middleware Collaboration Architectures for Converging MHP and OSGi in a Smart Home**. Journal of Information Science and Engineering 25, 1337-1356, 2009.

LLOYD, J. W. **Practical advantages of declarative programming**. Joint Conference on Declarative Programming, Inglaterra, 1994

LUCENA, Vicente Ferreira, CHAVES, João Edgar; VIANA, Nairon Saraiva; MAIA, Orlewilson Bentes. **A Home Automation Proposal Built on the Ginga Digital TV Middleware and the OSGi**

Framework. IEEE Transactions on Consumer Electronics, 53(3):1254-1262, August 2009.

MAKEWAVE. **Knoplerfish OSGi** - open Source Service OSGi Service Platform. Disponível em: <http://www.knopflerfish.org>. Acesso em: 16 dez 2010.

MONTEZ, Carlos; BECKER, Valdecir. **TV Digital Interativa: conceitos, desafios e perspectivas para o Brasil**. Florianópolis: Ed. da UFSC, 2005. 2a edição.

MIRANDA, Sindolfo Luiz de. **Flexcm: Um Modelo de Componentes para Sistemas Adaptativos**. Dissertação de Mestrado – programa de Pós Graduação em Informática, Universidade Federal da Paraíba. Paraíba, 2008.

MIRANDA, Sindolfo Luiz de; LEITE, Luiz Eduardo Cunha; LEMOS, Guido; MEIRA, Silvio. **FLEXCM – A Component Model for Adaptive Embedded Systems**. Computer Software and Applications Conference, 2007.

MORENO, Marcio F. **Um Middleware Declarativo para Sistemas de TV Digital Interativa**. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro. Rio de Janeiro, 2006.

MORRIS, Steven, SMITH-CHAIGNEAU, Anthony. **Interactive TV Standards** – A Guide to MHP, OCAP and JavaTV. ISBN-13 978-0-240-80666- 2. Elsevier, Focal Press, 2005.

O2W CONSORTIUM. **O2W Forge: Project Info - OSCAR - OSGi Framework**. Disponível em: <http://forge.ow2.org/projects/oscar>. Acesso em: 16 dez. 2010.

OBJECTWEB CONSORTIUM. **JEFFREE**: Java Embedded Framework FREE. Disponível em: <http://jeffree.ow2.org>. Acesso em: 16 dez. 2010.

OSGI ALLIANCE. **OSGi Service Platform Release 4**, Version 4.1. Abril 2007.

OSGI ALLIANCE. **OSGI Alliance**. Disponível em: <http://www.osgi.org>. Acesso em: 14 dez. 2010.

PROSYST SOFTWARE GMBH. **Mbedded Server CLDC Edition Overview**. Disponível em: <http://dz.prosyst.com/pdoc/mbs_cldc_2.0/um/overview.html>. Acesso em: 16 dez. 2010.

REDONDO, Rebeca P. Diaz; VILAS, Ana Feliz; CABRER, Manuel Ramos; ARIAS, José J. Pazos. **Exploiting OSGi capabilities from MHP applications**. Journal of Virtual Reality and Broadcasting, Volume 4 (16). Germany, 2007.

RELLERMEYER, Jan S.; ALONSO, Gustavo: **Concierge. A Service Platform for Resource-Constrained Devices**. Proceedings of the 2007 ACM EuroSys Conference. Portugal, 2007.

SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH. **Jadabs OSGi J2ME**. Disponível em <http://jadabs.berlios.de/jadabs-cldc/multiproject/osgi-j2me/index.html>. Acesso em: 16 dez. 2009.

TANENBAUM, Andrew S; STEEN, Maarten van. **Distributed Systems**: Principles and Paradigms. 2002.

TKACHENKO, Dmitry; KORNET, Nikolay; LAGUNOV, Alexander; KRAVTSOV, Denis; KURBANOV, Andrey; KHANDELWAL, Rashej; LI, Luyang. **A Possible Extension for iDTV Platform to Support Interactions with Home Appliances.** IEEE Consumer Communications and Networking Conference, 228—232, 2006.

TWITTER. **Twitter**.
Disponível em: <http://twitter.com>. Acesso em: 27 mar. 2011.

WIKIPEDIA. **DMB-T/H**. Disponível em: <http://en.wikipedia.org/wiki/DMB-T/H>. Acesso: 16 dez. 2011.

YANAMOTO, Yusuke. **Twitter4J: A Java library for the Twitter API**. Disponível em: <http://twitter4j.org>. Acesso em: 27 mar. 2011.

YANG, Ming-Chien; SHENG, Norman; HUANG, Brandon; TU Jethro. **Collaboration of Set-Top Box and Residential Gateway Platforms**. IEEE Transactions on Consumer Electronics, 53(3):905--910, August 2007.

ANEXO A – Testes para a escolha do Framework OSGi

De acordo com o critério 1 (compatibilidade com CDC/FP/PBP), todos as implementações são compatíveis com a configuração CDC, sendo que alguns são desenvolvidos com o conjunto restrito de bibliotecas oriundas do CDC e outros possuem uma compilação específica para gerar o binário.

Em relação ao critério 2 (tamanho), o Equinox apresenta o maior tamanho do arquivo de biblioteca (Figura 34), com 1148,3KB, enquanto que o Concierge apresentara o menor tamanho, de 87KB. O OSCAR e o Concierge (*branch*) também têm tamanho relativamente pequeno, com 202,6KB e 217,9KB, respectivamente. O Felix e o Knoplerfish ficaram próximos da média de 424KB, com 414,6KB e 471,9KB respectivamente.

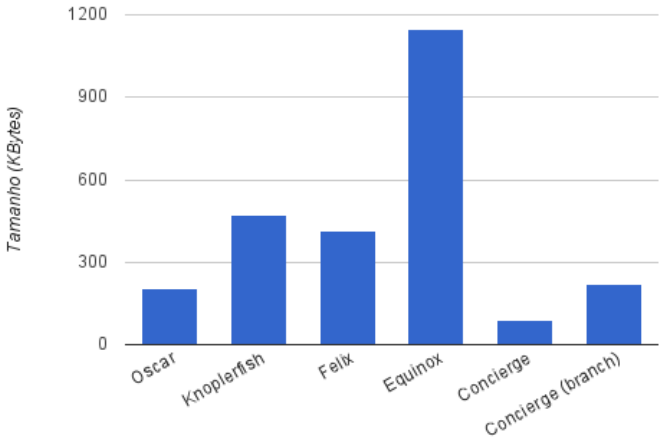

Figura 34 – Gráfico tamanho x Framework OSGi

De acordo com o critério 3 (versão do OSGi), o Concierge e o OSCAR são os únicos que não são compatíveis com a versão OSGi R4, sendo compatíveis com a versão OSGi R3, o que poderia explicar parte de seu menor tamanho, já que a especificação incorporou novas funcionalidades na versão R4. Com relação a inatividade (critério 4), eles também são os únicos que estão inativos, motivo pelo o qual não atualizaram a versão também. Além disto, ambos também não possuem uma fácil integração (critério 5). No caso do OSCAR, se faz necessário

estender classes do projeto para adequar código, já que ele não foi desenvolvido para tal finalidade. No caso do Concierge, ele inicializa diretamente um *shell* e não possui uma forma fácil de instalar *bundles* estendendo o código, o que torna ainda mais difícil esta integração. Além disto, na versão OSGi R4 foi adicionada a API *Framework Launcher* para facilitar a integração de código, motivo pelo qual as demais implementações apresentaram uma integração mais fácil. Esta API favorece também a portabilidade, já que seria mais fácil integrar uma implementação diferente com poucas mudanças (apenas na inicialização do *framework* e eventuais parâmetros proprietários). A Tabela 16 detalha os *frameworks* quanto à versão do OSGi, atividade e integração.

Tabela 16- Framework OSGi x Versão, Atividade e Integração

Framework	**Versão OSGi**	**Atividade**	**Integração**
OSCAR	R3	2005 (Inativo)	Média
Knoplerfish	R4	2011 (Ativo)	Fácil
Felix	R4	2011 (Ativo)	Fácil
Equinox	R4	2011 (Ativo)	Fácil
Concierge	R3	2009 (Inativo)	Difícil
Concierge (*branch*)	R4 (não compatível)	2011 (Ativo)	Fácil

Os testes de processamento e memória foram realizados em um PC com uma JVM Java SE 1.6 (sun-java6-jre), sistema operacional Ubuntu 9.10, Intel Core 2 Duo 2.4 GHz, 3GB de RAM. Apesar desta JVM não ser uma do tipo CDC, ela é suficiente como indicativo para ajudar na escolha identificando os gargalos de cada implementação de *framework* OSGi. Com objetivo de tentar isolar possíveis problemas, os testes foram executados 100 vezes de maneira automatizada através de um *script*, reiniciando a JVM em cada teste.

Foi inserida uma chamada *System.currentTimeMillis()* do pacote *java.lang* antes da aplicação de teste invocar o *Framework* (anterior) e uma chamada após (posterior). O tempo de inicialização foi calculado com a diferença deste tempo posterior em relação ao anterior. Para o cálculo do consumo de memória, foi inserida uma chamada *Runtime.freeMemory()* do pacote *java.lang* antes da aplicação de teste

invocar o *Framework* (anterior) e uma chamada após (posterior). Esta chamada retorna o valor total de memória disponível (em *bytes*) na JVM. O consumo de memória foi calculado com a diferença entre o consumo de memória posterior e o anterior.

De acordo com o tempo de inicialização (critério 6), o Equinox apresentou a maior média (Figura 35), ultrapassando 1 segundo, enquanto que o Concierge e Concierge (*branch*) apresentaram as duas menores. O OSCAR obteve o terceiro melhor tempo. O Concierge (*branch*) apresenta um tempo médio de inicialização 41,25% menor que o OSCAR e 69,48% menor que Knoplerfish, utilizados em alguns trabalhos descritos anteriormente. Mais detalhes sobre estes tempos podem ser encontrados na Tabela 17.

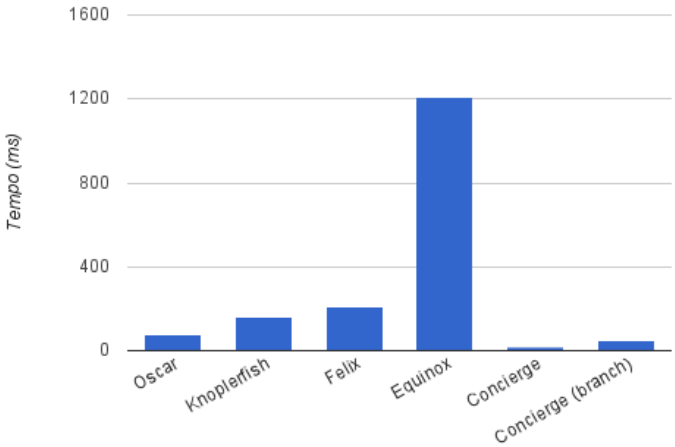

Figura 35 – Tempo médio de inicialização dos Frameworks OSGi

Tabela 17- Tempo médio de inicialização dos Frameworks OSGi

Tempo (ms)	OSCAR	Knoplerfish	Felix	Equinox	Concierge	Concierge (branch)
Maior	119	219	290	1546	18	94
Menor	62	124	159	1210	11	36
Média	80	154	212	895	14	47

O Equinox apresentou a maior alocação de memória (critério 7), em torno de 6214KB, enquanto que o Concierge e Concierge (*branch*) apresentaram os menores valores e bem abaixo da média, com 478KB e 759KB, respectivamente. O OSCAR também obteve o terceiro menor consumo, enquanto que o Feliz e Knoplerfish obtiveram uma média de 3500KB. A Figura 36 ilustra o gráfico comparativo. Os tempos coletados (em *bytes*) podem ser encontrados na Tabela 18.

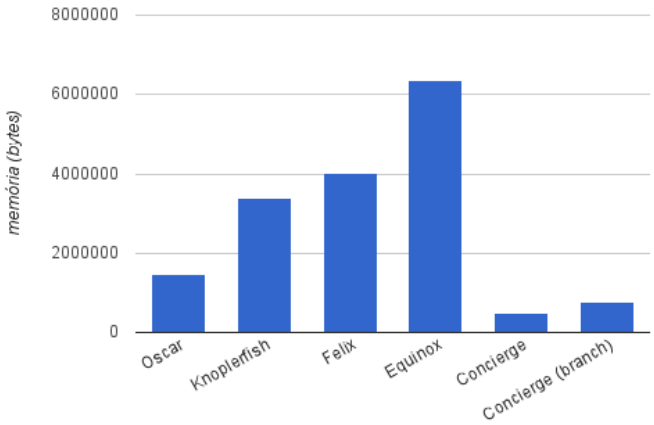

Figura 36 – Alocação de memória na inicialização dos Frameworks OSGi

Tabela 18- Alocação de memória na inicialização dos Frameworks OSGi

Memória (bytes)	OSCAR	Knoplerfish	Felix	Equinox	Concierge	Concierge (branch)
Média	1471808	3394336	4006704	6363417	490408	777344

ANEXO B – Exemplo de bundle

Código 1: DTVImageService.java.

```
public interface DTVImageService {

      public BufferedImage createImage(Integer
w, Integer h);
...
}
```

Código 2: DTVImageServiceAWTImpl.java.

```
public class DTVImageServiceAWTImpl implements
DTVImageService {

      public BufferedImage createImage(Integer
width, Integer height) {
            GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment
();
            GraphicsDevice gs =
ge.getDefaultScreenDevice();
            GraphicsConfiguration gc =
gs.getDefaultConfiguration();
            BufferedImage bufferImage =
gc.createCompatibleImage(width.intValue(),
height.intValue());
            return bufferImage;
      }
...
}
```

Código 3: DTVImageServiceActivator.java.

```
public class DTVImageActivator  implements
BundleActivator{

	private ServiceRegistration registration;

	public void start(BundleContext context)
throws Exception {
		registration =
context.registerService(DTVImageService.class.get
Name(), new DTVImageImpl(), null);
	}

	public void stop(BundleContext arg0) throws
Exception {
		registration.unregister();
	}
}
```

Código 4: TwitterXlet.java.

```
public class TwitterXlet implements BundleXlet{

public void initXlet(XletContext context)
throws XletStateChangeException {
      bundleXletContext = (BundleXletContext)
context;
}
public void startXlet() throws
XletStateChangeException {
      Object service =
bundleXletContext.getService("br.ufsc.ppgcc.bru
no.osgi.service.TwitterService");
      Method method =
service.getClass().getMethod("search", new
Class[] { String.class });
      List result = (List)
method.invoke(service, new Object[] {
"#java" });
...
}
...
}
```