UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

GUILHERME PETRY BREIER

OSGi-FEMTOJAVA: Plataforma Reconfigurável para Gerenciamento de Serviços Segundo o Padrão OSGi

Porto Alegre
2009

GUILHERME PETRY BREIER

OSGi-FEMTOJAVA: Plataforma Reconfigurável para Gerenciamento de Serviços Segundo o Padrão OSGi

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, da Universidade Federal do Rio Grande do Sul, como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.

Área de concentração: Controle e Automação

ORIENTADOR: Prof. Dr. Carlos Eduardo Pereira

Porto Alegre

2009

GUILHERME PETRY BREIER

OSGi-FEMTOJAVA: Plataforma Reconfigurável para Gerenciamento de Serviços Segundo o Padrão OSGi

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____

Prof. Dr. Carlos Eduardo Pereira, UFRGS

Doutor pela Universidade de Stuttgart – Stuttgart, Alemanha

Banca Examinadora:

Prof. Dr. João César Netto, UFRGS

Doutor pela Universite Catholique de Louvain – Lovaina, Bélgica

Prof. Dr.Walter Fetter Lages, UFRGS

Doutor pelo Instituto Tecnológico de Aeronáutica – São José dos Campos, Brasil

Prof. Dr. Marcelo Götz, UFRGS

Doutor pela Universität Paderborn – Paderborn, Alemanha

Coordenador do PPGEE: _____

Prof. Dr. Arturo Suman Bretas

Porto Alegre, abril de 2009.

AGRADECIMENTOS

Gostaria de agradecer primeiramente ao Professor Dr. Carlos Eduardo Pereira, pela orientação deste trabalho.

Agradeço também ao Programa de Pós-Graduação em Engenharia Elétrica – PPGEE pela oportunidade de aprender junto a excelentes professores.

Aos amigos do "Projeto 25 e seus eternos agregados" por serem pessoas especiais e com as quais sempre poderei contar em todos os momentos.

Aos amigos Diego Santini, Diogo Pereira, Felipe Martins, Jorge Alvez, Rodrigo Allgayer e especialmente ao amigo Marcos Dalte.

Aos amigos que sempre incentivaram o crescimento pessoal e científico, como os colegas Carlos Fernando Jung, e Professor Harald Alberto Bauer.

Aos Colegas da Escola Técnica Estadual Monteiro Lobato, por toda a dedicação e apoio em todos os momentos dessa trajetória.

A todos os amigos e amigas que fizeram parte dessa conquista.

Aos meus pais João Carlos e Helena e meus irmãos Luciano e Ana,

O meu mais sincero MUITO OBRIGADO!

RESUMO

Este trabalho apresenta o desenvolvimento de um sistema reconfigurável, baseado em OSGi (Open Services Gateway Initiative) para gateway residenciais utilizando-se da tecnologia FemtoJava, com o objetivo de automatizar a descoberta de serviços. Como sistemas de automação residencial/predial normalmente incluem produtos de diferentes fabricantes e possuem protocolos de comunicação diferentes, é necessário o uso de um agente que gerencie essa comunicação. Para que isso ocorra com sucesso, a OSGi Aliança oferece um conjunto de especificações numa plataforma aberta, em Java, para a entrega de vários tipos de serviços aos usuários finais. Utilizando as características do framework OSGi focados em ambientes inteligentes, é proposto um novo sistema de controle através desta arquitetura orientada a serviços para gateway residencial. O sistema foi implementado em uma plataforma reconfigurável da família Xilinx com o auxílio da ferramenta de síntese ISE. O trabalho também apresenta a validação da arquitetura proposta em um estudo de caso utilizando-se de dois módulos para interagir com o meio simulando um sistema de controle de temperatura.

Palavras-Chave: Gateway Residencial, OSGi framework, Ambientes Inteligentes.

ABSTRACT

This work presents the development of a reconfigurable system based on OSGi (Open Services Gateway Initiative) home gateway using the FemtoJava technology, aiming to automate the discovery of services. Automation home systems usually include products from different manufacturers and have different communication protocols, it is necessary to use an agent to manage this communication. For this to occur successfully, OSGi Alliance offers a range of specifications in an open platform, Java, for the delivery of various types of services to end users. Using OSGi framework features, intelligent environments focused on a new control system using this architecture targeted to services for residential gateway. The system was implemented on a reconfigurable platform family with the help of Xilinx ISE synthesis tool. The work also presents the validation of the architecture proposed in a case study using the two modules to interact with the environment simulating a temperature control system.

Keywords: Home gateways, OSGi framework, ambient intelligent

LISTA DE ILUSTRAÇÕES

Figura 1: Estrutura de um gateway de Serviço (GONG, 2000).....15
Figura 2: Relacionamento dos serviços com o resto da rede (MARPLES, 2001)...19
Figura 3: Relação dos componentes no OSGi (MARPLES, 2001).....22
Figura 4: Registro de serviços (MARPLES, 2001).....24
Figura 5: Comparativo entre arquiteturas de hardware (GÖTZ, 2007).....26
Figura 6: Reconfiguração de projeto em arquiteturas reconfiguráveis (GARCIA; et al., 2006)...27
Figura 7: Arquitetura interna de uma FPGA (MARTINS; et AL., 2003).....28
Figura 8: Fluxo de projeto do SASHIMI (WEHRMEISTER; PEREIRA; BECKER, 2006).....31
Figura 9: Arquitetura do software para o gateway de serviços (GONG, 2000).....32
Figura 10: Estrutura interna de um bundle (GONG, 2000).....34
Figura 11: Diagrama de estado de transição do bundle (OSGi, 2008).....35
Figura 12: Visualização da estrutura interna do bundle (OSGi, 2008).....37
Figura 13: Anatomia do bundle (OSGi, 2008).....37
Figura 14: Instalação de um bundle (OSGi, 2008).....38
Figura 15: Inicializando o bundle (OSGi, 2008).....39
Figura 16: Importando pacotes e obtendo serviços (OSGi, 2008).....40
Figura 17: Mecanismos de interfaceamento (OSGi, 2008).....40
Figura 18: Bundle HTTP exporta o pacote `http.service` (OSGi, 2008).....43
Figura 19: Importando o pacote (OSGi, 2008).....43
Figura 20: Arquitetura de referência genérica (HELAL, 2005).....47
Figura 21: Smart plugs (HELAL, 2005).....49
Figura 22: Registrando uma composição de serviços OSGi (REDONDO, 2007)..50
Figura 23: Instanciando um serviço OSGi composto (REDONDO, 2007).....51
Figura 24: Arquiteturas e integração com XbundLET (CABRER, 2006).....52
Figura 25: (a) MHP com OSGi (b) OSGi com MHP (CABRER, 2006).....53
Figura 26 Arquitetura de sistema para computação pervasiva (XU; XIN; LU, 2007).....56
Figura 27: Componentes implementados na presente proposta.....58
Figura 28: Arquitetura de hardware do FemtoOSGi.....64
Figura 29: Organização da memória de dados do FemtoOSGi.....65
Figura 30: Serial Server Bundle.....66
Figura 31: Interface do serviço.....66

Figura 32: Método Ativador. 68
Figura 33: *Listener* de Eventos. 68
Figura 34: Driver Locator. 69
Figura 35: A implementação do driver service. 70
Figura 36: Modem service. 72
Figura 37: Ativador do Bundle. 73
Figura 38: Blocos do protótipo. 74
Figura 39: Protótipo usado. 75
Figura 40: Estudo de caso para validação. 75
Figura 41: Manifesto do Sensor Bundle. 76
Figura 42: Bundle Ativador do Sensor. 76
Figura 43: Thread do sensor. 77
Figura 44: Manifesto do Atuador. 77
Figura 45: Bundle Ativador do Atuador. 78
Figura 46: Thread do Atuador. 78
Figura 47: Arquitetura FemtoOSGi. 80

LISTA DE TABELAS

Tabela 1: Resultado da Implementação usando FemtoJava 79
Tabela 2: Femtojava 80
Tabela 3: FEMTONODE 81

LISTA DE ABREVIATURAS

CPU Central Processing Unit

DSSS Direct Sequence Spread Spectrum

DVS Dynamic Voltage Scaling

FCC Federal Communications Commission

FPGA Field Programmable Gate Array

IEEE Institute of Electrical and Electronics Engineers

ISM Industrial, Scientific, Medical

ISO International Organization for Standardization

MANET Mobile Ad-Hoc Networks

OSI Open Systems Interconnection

OSGi Open Service Gateway initiative

PC Personal Computer

PDA Personal Digital Assistant

PIN Personal Identification Number

PPM Pulse Position Modulation

RF Rádio-Freqüência

RFU Reconfigurable Function Unit

SCO Synchronous Connection-Oriented

TDD Time-Division Duplex

TDMA Time Division Multiple Access

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

SUMÁRIO

1 INTRODUÇÃO 11
1.1 OBJETIVO 12
1.2 MOTIVAÇÃO 13
1.3 ORGANIZAÇÃO DO TRABALHO 14

2 CONCEITOS BÁSICOS 15
2.1 GATEWAYS DE SERVIÇO 15
2.1.1 O Ambiente de Aplicação 16
2.1.2 Java Embedded Server 16
2.2 ARQUITETURA ORIENTADA A SERVIÇOS 17
2.2.1 OSGi - Open Services Gateway Initiative 18
2.2.1.1 Operação 21
2.2.1.2 Especificações 24
2.3 SISTEMAS RECONFIGURÁVEIS 25
2.3.1 Tipos de sistemas 25
2.3.2 Reconfigurabilidade 27
2.3.3 Linguagem de programação através de descrição de hardware 28
2.3.3.1 VHDL... 29
2.3.4 Microcontroladores dedicados em Hardware Reconfigurável 29
2.3.4.1 SASHIMI/FemtoJava 30

3 ARQUITETURA DO OSGI 32
3.1 ARQUITETURA 32
3.2 SERVIÇO 32
3.3 BUNDLE 34
3.3.1 Conteúdo de um Bundle 34
3.3.2 Bundle como módulo funcional 35
3.3.2.1 Ciclo de vida de um bundle 35
3.3.2.2 Bundle Class Loading 35
3.3.3 As ligações com o framework 36
3.3.3.1 O Manifesto 36
3.3.3.2 O Bundle ativador 37
3.3.4 Instalando um Bundle 38
3.3.5 Inicializando o Bundle 38
3.3.6 Importando pacotes e obtendo serviços 39
3.3.7 Atualizando um bundle 40
3.3.8 Finalizando a execução e removendo bundles 41
3.4 O FRAMEWORK 41
3.5 COOPERAÇÃO ENTRE BUNDLES E SERVIÇOS 42
3.5.1 Exportando e importando pacotes 42
3.5.2 Registrando e Obtendo Serviços 44
3.5.3 Pacotes versus Dependências de Serviços 45

4 ANÁLISE DO ESTADO DA ARTE 46
4.1 GATOR TECH 46
4.2 OSGI/BPEL 49

4.3 INTEGRAÇÃO TV DIGITAL E SISTEMA DE AUTOMAÇÃO RESIDENCIAL VIA XBUNDLE ... 51
4.4 DISCUSSÃO ... 54

5 PROPOSTA DO OSGI-FEMTOJAVA ... 55
5.1 COMPONENTES DE UM GATEWAY DE SUPORTE À COMPUTAÇÃO PERVASIVA ... 55
5.2 OSGI-FEMTOJAVA ... 58
5.2.1 Análise do *bytecodes* do OSGi no SASHIMI ... 58
5.2.2 Relevantes APIs do org.osgi.framework ... 59
5.2.2.1 `BundleContext` Interface ... 60
5.2.2.2 `ServiceReference` Interface ... 62
5.2.2.3 `ServiceRegistration` Interface ... 62
5.2.3 Adaptação no FemtoJava ... 63
5.3 IMPLEMENTAÇÃO ... 65
5.3.1 O Protótipo ... 73
5.4 ESTUDO DE CASO ... 75

6 CONCLUSÃO ... 82

REFERÊNCIAS ... 85

ANEXO A – PRINCIPAIS APIS OSGI ... 89

1 INTRODUÇÃO

A computação pervasiva tem sido tema de diversos trabalhos nos últimos anos (XU; XIN; LU, 2007). Essa emergente área de pesquisa propõe uma visão de futuro onde serviços computacionais são oferecidos para os usuários através de inúmeros dispositivos espalhados pelo ambiente. Os serviços são disponibilizados, tanto através da infra-estrutura existente dos computadores ligados fisicamente à rede quanto através de dispositivos móveis. Esse espalhamento da computação deve acontecer de maneira natural e imperceptível ao usuário. Dados pessoais, programas e arquivos de dados poderão ser acessados de qualquer lugar em qualquer momento. O poder de processamento será um recurso do ambiente, acessado quando necessário, da mesma forma que é hoje a eletricidade. O usuário não precisará ter ciência de qual máquina realiza o processamento necessário às suas aplicações, contanto que o resultado esperado seja obtido. Acredita-se que essa realidade será atingida através da aliança entre áreas de pesquisa como a computação móvel e a computação ciente do contexto. A computação ciente do contexto busca enriquecer a comunicação entre os seres humanos e os dispositivos computacionais, tornando sua atuação mais eficaz. As aplicações cientes do contexto conseguem perceber as modificações que ocorrem no ambiente e adaptar seu comportamento ao novo estado. Esse processo pode ser dividido em três etapas: monitoramento, reconhecimento de contexto e adaptação. Na etapa de monitoramento são coletadas, através de sensores, informações sobre o ambiente. Essas informações, entretanto, são, geralmente, de baixo nível de abstração e, portanto, dificilmente usadas diretamente por aplicações. A etapa de reconhecimento de contexto relaciona os dados obtidos do ambiente e transforma-os para que possam ser úteis às aplicações no processo de escolha do comportamento mais adequado a cada circunstância, habilitando a etapa de adaptação para efetivar a transformação do comportamento da aplicação de acordo com a nova situação do ambiente.

As arquiteturas orientadas a serviços (SOA - *Service-Oriented Architectures*) propõem-se a solucionar e preencher essa lacuna através de um novo paradigma de desenvolvimento de sistemas em computação pervasiva, mais completo, e propiciando interoperabilidade entre diversas tecnologias.

Neste contexto surge como proposta para a automação de serviços em aplicações de automação predial/residencial o OSGi (OSGI, 2008), baseado em Java, como uma proposta

interessante para o desenvolvimento. O OSGi oferece serviços para descoberta de dispositivos e serviços em execução com uma ampla capacidade de gerenciamento dos mesmos. Entretanto, o uso completo de toda a plataforma OSGi exige um alto custo computacional para o desenvolvimento de aplicações.

Por outro lado, sistemas computacionais usando arquiteturas reconfiguráveis proporcionam uma melhor otimização em função da aplicação, reduzindo o custo computacional e melhorando o desempenho do sistema (GÖTZ, 2008). A flexibilidade dessas arquiteturas facilita a otimização da lógica sintetizada, permitindo a síntese de microcontroladores dedicados e otimizados para uma determinada aplicação. Assim, apenas os recursos necessários para a aplicação são sintetizados, resultando em um processador menor, ocupando menos área na FPGA, com melhor desempenho e com menor potência dissipada. A metodologia do projeto SASHIMI (ITO; CARRO; JACOBI, 2001) difere da prática convencional de desenvolvimento de sistemas embarcados. A abordagem utilizada pelo SASHIMI permite a utilização da linguagem de alto nível JAVA não somente para programação, mas também para a especificação completa do sistema embarcado. Esta ferramenta permite fazer a síntese de uma máquina virtual Java (JVM) na forma de um processador que executa *bytecodes* Java nativamente chamado FemtoJava (ITO; CARRO; JACOBI, 2001). Através do FemtoJava podem ser executadas aplicações para sistemas embarcados usando como base a linguagem de programação Java. Com isso, pretende-se desenvolver uma aplicação usando componentes OSGi em uma plataforma reconfigurável.

1.1 OBJETIVO

Este trabalho tem como objetivo desenvolver um sistema embarcado, compatível com o padrão OSGi para a descoberta de dispositivos e serviços, implementado numa plataforma de hardware customizável e otimizando apenas os recursos necessários. Este gateway utiliza uma plataforma de hardware reconfigurável, onde é sintetizado um microcontrolador *softcore* FemtoJava (ITO; CARRO; JACOBI, 2001), cujo código é adaptado às necessidades da aplicação, sendo que somente o hardware necessário é sintetizado, reduzindo, assim, a área de hardware necessária.

A descrição da aplicação, para posterior síntese do microcontrolador em uma arquitetura reconfigurável, é realizada utilizando uma linguagem de programação orientada a

objeto, no caso Java. Para esta descrição, é utilizado um ambiente de desenvolvimento com uma ferramenta de síntese que gera os arquivos de descrição de hardware a partir da descrição da aplicação e do microcontrolador em Java, denominado SASHIMI (ITO; CARRO; JACOBI, 2001), o qual será detalhado posteriormente.

Deste modo, pode-se entender a contribuição deste trabalho por dois pontos de vista:

1) É adicionado ao RT-FemtoJava a capacidade de descoberta de serviços e carga dinâmica de aplicações, conceitos suportados por OSGi.
2) É gerado um sistema embarcado que implementa funcionalidades de OSGi, mas que tem seu hardware otimizado em função da aplicação proposta.

Basicamente, o que se busca é aproveitar a capacidade de configuração do Femtojava no sentido de gerar uma arquitetura customizada para o OSGi otimizada para o gerenciamento de serviços.

1.2 MOTIVAÇÃO

O principal objetivo do serviço do framework OSGi é a utilização da independência de plataforma da linguagem de programação Java (TM) e capacidade de carga dinâmica de códigos para fazer o desenvolvimento e a implantação de aplicativos dinâmicos para equipamentos com baixa capacidade de memória com mais facilidade.

Esta meta é atingida de duas maneiras. Em primeiro, o framework proporciona um modelo de programação consistente durante a aplicação em implementação, ele suporta o desenvolvimento e a utilização de serviços por uma dissociação da especificação do serviço (a interface) de sua implementação. Assim, desenvolvedores podem fornecer múltiplas implementações para a mesma interface de serviço.

Este suporte é fundamental, porque o framework foi projetado para ser executado em uma variedade de dispositivos e as diferentes características de hardware dos dispositivos podem afetar muitos aspectos do serviço de execução, mas ainda assim a interface de serviço estável garante a estabilidade global do software em execução no dispositivo.

Em segundo lugar, ele fornece um gerenciamento de ciclo de vida, funcionalidade que permite que desenvolvedores de aplicação particionem aplicações em pequenos componentes auto-instaláveis. Estes componentes são chamados bundles (OSGi, 2008). Os bundles podem

ser baixados em demanda e removidos quando já não são mais necessários. Quando um bundle está instalado e ativado no framework, ele pode registrar qualquer tipo de serviços que podem ser utilizados por outros bundles.

Este aspecto dinâmico torna o software extensível no equipamento após a implantação: novos bundles podem ser instalados para adicionar funcionalidades ou bundles existentes podem ser atualizados para corrigir falhas sem a necessidade de reiniciar todo o sistema.

Além de todas essas características, a plataforma OSGi proporciona mecanismos para a descoberta de dispositivos e serviços no ambiente. Essa característica permite que se descubra os equipamentos disponíveis no ambiente em que está inserido o gateway e então permite que os serviços sejam executados através desses hardwares.

1.3 ORGANIZAÇÃO DO TRABALHO

Essa dissertação está estruturada da seguinte maneira: no capítulo 2 são apresentados os conceitos das tecnologias abordadas neste trabalho. O capítulo 3 apresenta a arquitetura que compõe o framework OSGi. No capítulo 4, é realizado um estudo da arte de gateways utilizando a tecnologia OSGi. No capítulo 5, é apresentada a arquitetura de hardware proposta por este trabalho, o gateway OSGi em FemtoJava desenvolvido, abordando também a validação do mesmo. Por fim, no capítulo 6, é apresentada a conclusão deste trabalho apontando possíveis direções para trabalhos futuros.

2 CONCEITOS BÁSICOS

Este capítulo descreve conceitos cuja compreensão é fundamental para o entendimento do que aqui se propõe. Inicialmente é descrito o conceito de gateway de serviços, núcleo funcional dos chamados Ambientes Inteligentes (AmI) (VALTCHEV; FRANKOV, 2002). A proposta OSGi, utilizada neste trabalho é então descrita. No final do capítulo, o conceito de sistemas reconfiguráveis é apresentado.

2.1 GATEWAYS DE SERVIÇO

Provedores de serviços tradicionais, como os de telefone e TV a cabo, têm a sua própria rede direcionada para residências. No entanto, na visionária rede residencial, esta configuração irá em breve tornar-se complexa para gerenciar um portfólio diversificado de serviços, tais como segurança, vigilância da saúde, telefonia e meios de comunicação de áudio/vídeo, cada um possivelmente utilizando diferentes tecnologias da comunicação. Também irá perder oportunidades interessantes para a integração. Em razão disso, utiliza-se um dispositivo centralizado com interface externa conectada a internet, e também as redes internas conectadas aos dispositivos e aparelhos (Figura 1). Este dispositivo é chamado de gateway (GONG, 2000).

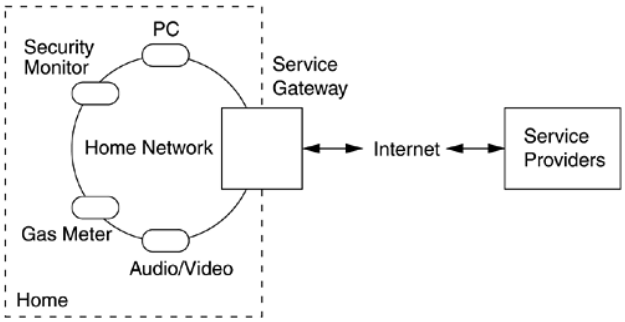

Figura 1: Estrutura de um gateway de Serviço (GONG, 2000).

Uma típica plataforma de gateway de serviço consiste em:

- Um processador;
- Memória;

- Dispositivo de armazenamento de dados (disco ou flash RAM);
- Rede TCP/IP;
- Uma porta de comunicação;
- Um sistema operacional ou um sistema operacional em tempo real.

2.1.1 O Ambiente de Aplicação

Tanto a instalação como a configuração do dispositivo deve ser simples e automática, enquanto a complexidade de aparelhos e serviços deve ser gerida remotamente pelos provedores de serviços. Muitas vezes, um serviço como a gravação de um programa de televisão depende da presença de outro serviço, como um guia eletrônico de programação. Assim, deve ser possível para o gateway residencial detectar a necessidade de um serviço, descobrir a sua presença ou ausência, e garantir que ele esteja disponível. O Java Embedded Server da Sun Microsystems (SUN, 2008) está direcionado para o mercado residencial de gateway e é concebido para responder a estes desafios. Ao adicionar o Java Embedded Server a um produto de hardware, fabricantes de dispositivos podem facilmente transformar qualquer dispositivo terminal de banda larga, como uma DSL/modem ou *set-top box*, em um gateway residencial.

2.1.2 Java Embedded Server

Desde a sua criação, a linguagem de programação Java e plataforma Java têm sido amplamente aceitas como a forma de programação para a Internet. Dado que a linguagem de programação Java é uma plataforma neutra, desenvolvedores podem escrever e testar aplicações em um ambiente desktop, como o Linux ou o Windows e, em seguida, implementá-los no dispositivo alvo. Uma aplicação escrita para um dispositivo pode ser implantada diretamente em outro dispositivo, desde que o outro dispositivo possua uma MVJ. Ela encurta ciclos de desenvolvimento e estimula o código de qualidade.

Depois que percebeu o grande potencial de aplicações em dispositivos inteligentes conectados, e os pontos fortes da plataforma Java para enfrentar esta classe de software, a SUN Microsystems desenvolveu e lançou a primeira versão do *Java Embedded Server* em outubro de 1998. Sua arquitetura representa uma interface de programação unificada que permite que os serviços trabalhem em conjunto. Um serviço como um componente, pode ser programado para executar qualquer protocolo ou executar qualquer função de forma isolada ou cooperativa. Mais especificamente, o *Java Embedded Server* aborda as seguintes questões que são importantes para prover serviços para o gateway residencial:

• Entrega do serviço *Just-in-time*: O framework permite que um serviço possa ser baixado através da rede quando o serviço é necessário. O serviço pode ser utilizado uma vez e então descartado ou ele pode ser mantido persistentemente sobre o gateway residencial para um período mais longo.

• Serviço de atualizações e versões: O framework pode ser utilizado para verificar rapidamente a versão de um serviço que está sendo executado em um gateway e para atualizar dinamicamente este serviço a partir de uma localização remota. Isto é muito útil para desenvolvedores de software para dispositivos. Estes dispositivos têm sido tradicionalmente carregados com uma aplicação em ambiente estático. Um bug crítico na aplicação pode ser caro para consertar. Utilizando o *Java Embedded Server*, isto já não é uma restrição. Uma nova versão do software pode ser carregada pelo framework através da rede.

• Serviço de descoberta e dependência: O framework prevê um mecanismo de descoberta de serviço com o qual um componente descarregado pode consultar um serviço de registro no framework e obter e utilizar um serviço já existente. O framework também resolve relações de dependência quando um serviço depende do outro para funcionar.

2.2 ARQUITETURA ORIENTADA A SERVIÇOS

A arquitetura orientada a serviços (SOA – *Service-Oriented Architecture*) é uma arquitetura de sistema que integra diferentes sistemas distribuídos em cima de uma rede com um procedimento padronizado. Cada sistema exporta suas próprias características para a rede como uma unidade de serviço. A lógica interna e implementação do serviço são auto-contidas e encapsuladas no sistema. O sistema expõe somente interfaces de serviços em forma de

métodos exportados. A arquitetura transforma aplicações em componentes de software reusáveis, gerando interoperabilidade entre diversas tecnologias. SOA é definido como um paradigma para organizar e utilizar capacidades distribuídas que podem ser controladas por diferentes domínios (NAKAMURA; et al., 2004).

Para o sucesso das aplicações em SOA, é essencial que haja a capacidade de identificar os serviços e suas características. O relacionamento entre os componentes que estão inseridos em uma aplicação precisa estar descrito em um contrato, que pode ser uma interface Java ou um arquivo XML (*eXtensible Markup Language*). Dessa forma, cria-se uma rotina que facilita e explicita a dependência, a manutenção e a adaptação.

A flexibilidade permite aos componentes estarem ligados, de uma forma que possibilite a combinação semântica, a utilização de bibliotecas comuns e a divisão até de estados de execução. A principal vantagem de SOA é a flexibilidade e adaptação. Várias tecnologias são desenvolvidas incorporando os conceitos de SOA, como *web services*, OSGi, entre outras (GONG, 2000).

2.2.1 OSGi - Open Services Gateway Initiative

O OSGi é uma arquitetura orientada a serviços para gateways residenciais, e no framework é proposto um componente de execução (bundle) que exporta os serviços oferecidos e pode, também, obter outros serviços por meio do OSGi *Service Registry* (OSGI, 2008). Um bundle nada mais é do que um arquivo Java (JAR) que representa os componentes mínimos que podem ser instalados, desinstalados e atualizados.

A camada de serviços sobre a qual o serviço executa o OSGi é específica. As especificações de entrega são abertas, em uma arquitetura comum que permite a implantação e gestão de serviços de forma coordenada. Esta arquitetura comum pode então mapear componentes fisicamente e logicamente. Um típico ambiente no qual o OSGi é destinado a ser implantado é mostrado na Figura 2.

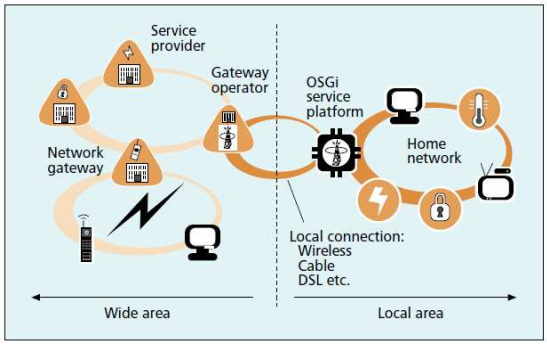

Figura 2: Relacionamento dos serviços com o resto da rede (MARPLES, 2001).

Neste diagrama pode-se observar que a plataforma de serviço funciona como o portal de redes externas e sua funcionalidade é controlada pelo gateway. Esta plataforma de serviços é o principal componente da arquitetura OSGi. Ela funciona como ambiente de execução dos serviços. Normalmente, a plataforma de serviços seria capaz de suportar uma variedade de serviços de voz, dados, internet e serviços de comunicação multimídia.

A plataforma de serviços fornece um ambiente de execução que os provedores de serviços podem usar para prestar serviços aos clientes em seus próprios ambientes utilizando dispositivos em sua rede local. Isso permite a prestação de serviços muito além do que é possível com um navegador web.

As metas para as especificações OSGi foram originalmente *set-top boxes* digitais e analógicos, gateways residenciais e modems. Como a norma está em desenvolvimento, tem-se encontrado aplicabilidade em eletrônicos, computadores, carros e outras áreas onde os benefícios de um ambiente operacional uniforme, abstração de hardware, e serviços de gerenciamento de ciclo de vida são apreciados. As especificações OSGi permitem uma nova categoria de inteligentes dispositivos devido ao seu suporte, flexibilidade, gestão e implantação de serviços.

O mercado para a implementação remota e gerenciamento de serviços só pode crescer quando um padrão está disponível, aceito e promovido por um grande grupo de empresas representativas no mundo atual.

Isto poderia ser criado através de uma norma, explicitamente ou pelo surgimento de uma norma que emergisse desde o início de implantação no mercado.

As companhias, membros fundadores da OSGi, reconheceram os perigos desta segunda opção e, em preferência a este processo *ad-hoc*, uniram suas forças para criar as especificações necessárias para permitir que o mercado usasse um mesmo padrão. Um conjunto de princípios foi estabelecido para orientar o desenvolvimento dessas especificações tais como:

Plataforma Independente: O ambiente de software OSGi pode ser implementado em várias plataformas, com capacidades muito diferentes.

Aplicação Independente: OSGi prevê uma plataforma "horizontal" que é aplicável em qualquer ambiente computacional.

Suporte a Serviços Múltiplos: Ambientes OSGi são capazes de acolher múltiplas aplicações de diferentes fornecedores de serviços em uma única plataforma de serviços.

Suporte para Colaboração de Serviços: O ambiente OSGi permite que serviços implantados forneçam funcionalidade para outros serviços. As aplicações podem descobrir dinamicamente estes serviços e adaptar seu comportamento para a configuração do ambiente e dos outros serviços que estão presentes.

Segurança: Um ambiente OSGi pode concomitantemente suportar muitos serviços de diferentes fornecedores de serviços. A segurança entre estes serviços é de importância primordial.

Simplicidade: O ambiente OSGi oferece um serviço de ambiente em que a complexidade da gestão do serviço de ambiente pode ser colocada nas mãos de profissionais sob a forma de um operador de gateway. Isto não impede, todavia, os indivíduos de realizar a sua própria configuração, conforme o caso.

Expandindo a Figura 2, a visão do mundo OSGi compreende as seguintes entidades:

Plataforma de Serviços *(Service Plataform):* A plataforma de serviços é um ambiente de execução para implementar serviços remotamente de aplicações que são normalmente administrados por um aplicativo de gerenciamento. O aplicativo de gerenciamento é específico para um operador de rede ou gateway.

Provedor de Serviços *(Service Provider):* Os provedores de serviços entregam aplicações para o operador do gateway, que combinam uns com os outros e verifica a veracidade da agregação antes da entrega, ao cliente, da plataforma do serviço.

Operador *(Gateway Operador):* Responsável pelo correto funcionamento do gateway, o operador oferece um ambiente seguro para os prestadores de serviços para executar e, portanto, garante a integridade do meio ambiente. O operador é responsável não só por instalar e remover aplicações, garantir recursos suficientes, como também pelas permissões que estão à sua disposição. O operador também monitora as portas de entrada para detectar falhas e ataques de segurança.

Acesso à Internet *(Internet Access):* Um gateway poderá fornecer acesso à internet como parte de suas ofertas. No entanto, também é possível que o gateway operador pode especializar-se exclusivamente na plataforma de serviços de gestão e só acessar o serviço plataformas pré-existentes durante uma conexão com a internet. Neste caso, qualquer fornecedor independente de serviços de internet, poderia então fornecer a conectividade.

Rede local e dispositivos *(Local Network and Devices):* as redes locais e dispositivos conectados à plataforma de serviços podem ser diretamente ligados, como através de uma porta USB, ou indiretamente, através de redes locais. Esta ligação entre o mundo externo da Internet ou de outros de rede local torna a plataforma de serviços mais do que apenas um computador, uma ponte entre os domínios.

A relação entre cada um destes itens é apenas vagamente definido pelo OSGi para permitir uma variedade de modelos de negócios de serviços a ser prestados.

2.2.1.1 Operação

O ambiente OSGi consiste de um framework que fornece o núcleo de especificação da plataforma de serviços OSGi. Ele fornece, para fins gerais, um modo seguro, gerenciado pelo framework Java que suporta a implantação do serviço de aplicações extensíveis e que permitem fazer downloads. O framework roda em cima de uma Java Runtime Environment, como mostrado na Figura 3. Bundles podem acessar recursos no framework, na VM, e no sistema operacional, conforme necessário. Mecanismos são fornecidos para assegurar que os bundles utilizem o código adequado quando acessam capacidades nativas. Um bundle pode incluir variantes do código nativo para lhe permitir ser usado em diferentes plataformas nativas.

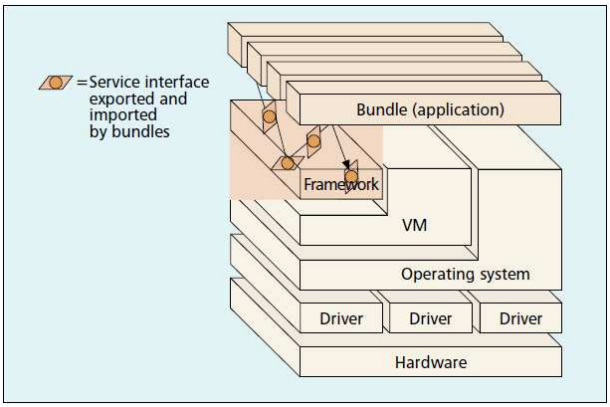

Figura 3: Relação dos componentes no OSGi (MARPLES, 2001).

Dispositivos OSGi compatíveis podem baixar e instalar bundles e removê-los quando eles já não são necessários. Instalado, o bundle pode registrar um número de serviços que podem ser compartilhados com outros bundles sob rigoroso controle do framework. Um bundle também pode exportar ou importar pacotes Java sob esse mesmo controle restrito.

O framework gerencia a instalação e atualização dos bundles em um modo dinâmico e escalonado, e gerencia as dependências entre bundles e serviços, proporcionando ao bundle desenvolver com recursos necessários e tirando vantagem da independência e capacidade de carga dinâmica de código na plataforma Java, a fim de facilmente desenvolver, implantar em grande escala, os serviços para os dispositivos com baixa capacidade de memória.

O framework fornece uma descrição concisa e um modelo consistente de programação para os desenvolvedores de bundles, simplificando o desenvolvimento e implantação de serviços, dissociando o serviço da especificação de sua implementação. Este modelo permite, aos desenvolvedores, vincular ao bundle de serviços exclusivamente de sua interface de especificação. A seleção de uma aplicação específica, otimizado para uma determinada necessidade ou de um determinado fornecedor, pode ser, assim, executada.

Um modelo de programação consistente ajuda desenvolvedores de bundles a lidar com questões de escalabilidade críticas, porque o framework se destina a ser executado em uma variedade de dispositivos com diferentes características de hardware, o que pode afetar muitos aspectos de um serviço de implementação. Interfaces consistentes garantem que os componentes do software podem ser misturados e combinados, e ainda resultar em sistemas estáveis.

O framework permite bundles disponíveis para selecionar uma aplicação em tempo de execução através do framework de registro serviço. Bundles registram novos serviços, recebem notificações sobre o estado dos serviços, ou procuram os serviços existentes para se adaptar às atuais capacidades do dispositivo. Este aspecto do framework instalado num bundle torna extensível após implantação: novos bundles podem ser instalados para adicionar recursos, ou bundles já existentes podem ser modificados e atualizados sem exigir que o sistema seja reiniciado.

No ambiente OSGi, bundles encapsulam aplicações. Um bundle inclui classes Java e outros recursos que em conjunto podem fornecer funções para os usuários finais e, opcionalmente, fornecer elementos para outros bundles. Estas capacidades denominan-se serviços exportados. Um bundle é implementado como um Java Archive file (JAR), contendo os recursos para implementar nenhum ou muitos serviços. Estes recursos podem ser da classe de arquivos da linguagem de programação Java, bem como outros dados, como arquivos HTML, arquivos de ajuda, ícones, e assim por diante. Contém um arquivo manifesto descrevendo o conteúdo do JAR e fornecendo informações sobre o bundle. Este arquivo utiliza cabeçalhos para especificar parâmetros para que o framework funcione corretamente na instalação e ativação de um bundle.

Dependências de estado sobre outros recursos tais como pacotes Java, devem estar disponíveis para o bundle antes de ser publicado. O framework deve resolver estes pacotes antes de se iniciar um bundle.

Designa uma classe especial no bundle para agir como bundle ativador. O framework deve instanciar essa classe e invocar métodos para iniciar e parar o bundle respectivamente. A implementação do bundle e do bundle ativador permite que o bundle inicialize.

Pode conter opcionalmente documentação dentro do JAR. Isto pode, por exemplo, ser usado para armazenar o código fonte de um pacote. Os sistemas de gestão podem remover esta informação para poupar espaço de armazenamento.

Quando um bundle é iniciado, a sua funcionalidade é fornecida e os serviços estão expostos a outros bundles instalados no ambiente OSGi. Eles podem então usar o framework para acessar a esta funcionalidade, como mostrado na Figura 4.

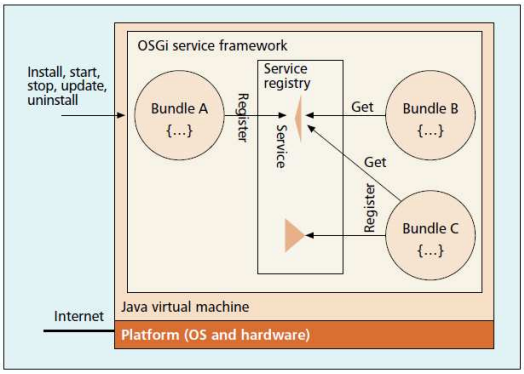

Figura 4: Registro de serviços (MARPLES, 2001).

2.2.1.2 Especificações

Serviços OSGi são especificados por uma interface Java, juntamente com um documento detalhando a semântica por trás daquela interface. Estes serviços são amplamente descritos na OSGi Service Platform Release, e incluem:

Log Service: Fornece os meios para registrar informações de bundles para o sistema e também fornece acesso a esta informação.

HTTP Server: um poderoso motor de *servlet*, utilizável por bundles. O serviço HTTP é um dos mais simples, seguro e confiável para implantar *servlets*.

Device Access: Um modelo para o gerenciamento de dispositivos e *drivers* de dispositivo no mundo externo.

Configuration Management: Um completo modelo para gestão de bundles remotos e locais. Bundles estão com informações de configuração sempre atualizadas e se for necessário que sejam feitas alterações de configurações, elas possam serem feitas imediatamente sem exigir reinicialização.

User Management: Este serviço gerencia a autenticação de usuários e atividades sobre a plataforma de serviços. É um modelo muito genérico, permitindo diversas subjacentes sistemas de autenticação.

2.3 SISTEMAS RECONFIGURÁVEIS

Os sistemas de computação reconfigurável são plataformas onde a arquitetura pode ser modificada ao longo do tempo para melhor se adequar à aplicação que será executada. Deste modo, o processador de um sistema reconfigurável passa a trabalhar com uma arquitetura desenvolvida exclusivamente para aquele determinado tipo de aplicação, permitindo uma eficiência muito maior do que a normalmente encontrada em processadores de uso geral. Isto ocorre porque o hardware é otimizado para executar os algoritmos necessários para aquela aplicação específica; ou seja, para se obter um melhor desempenho de um algoritmo, este deve ser executado num hardware específico para ele. Sistemas reconfiguráveis representam uma nova idéia na computação, na qual algum agente de hardware de propósito geral é configurado para realizar uma tarefa específica, mas pode ser reconfigurado sob demanda para realizar outras tarefas específicas.

2.3.1 Tipos de sistemas

Um sistema computacional pode ser implementado utilizando três diferentes tipos de arquiteturas de hardware: microprocessador de propósito geral (GPP), circuitos integrados de aplicação específica (ASICs) ou dispositivos programáveis em hardware (FPGAs) (COMPTON; HAUCK, 2002). Os sistemas implementados utilizando microprocessador de propósito geral apresentam um menor custo e facilidade de implementação. Porém, por não serem especificamente projetados para a tarefa na qual estão sendo utilizados, muitas vezes perdem em desempenho, ou seja, a capacidade de processamento mais rápido e na precisão da tarefa à qual o dispositivo se destina. Com a intenção de buscar um maior desempenho, utilizam-se circuitos integrados de aplicação específica (ASIC). Os ASICs, por serem otimizados para desenvolver tarefas específicas, apresentam um desempenho superior aos GPPs. Entretanto, apesar disso possuem um alto custo de produção, sendo seu uso justificado a partir da produção de um grande volume de unidades, de forma a baratear as unidades individuais deste componente.

Assim, pode-se colocar os microprocessadores de propósitos gerais e os ASICs em extremos opostos, considerando flexibilidade e desempenho, conforme apresentado na

Figura 5. Os GPPs apresentam grande flexibilidade de aplicações e baixo custo, mas perdem em eficiência. Os ASICs apresentam alto nível de eficiência, porém, alto custo de implementação. Além disso restringem-se a uma aplicação específica.

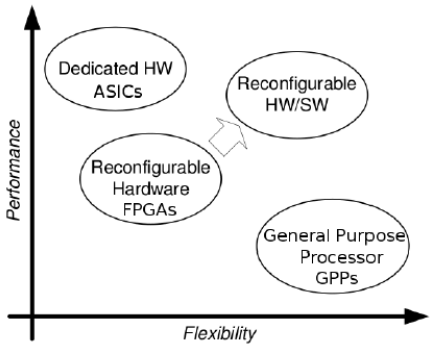

Figura 5: Comparativo entre arquiteturas de hardware (GÖTZ, 2007).

Entre a flexibilidade da arquitetura dos GPPs e o desempenho dos ASICs, pode-se destacar a presença dos dispositivos de lógica programáveis FPGAs (Field- Programmable Gate Array) e CPLDs (Complex Programmable Logic Device) em uma posição intermediária a estas características. A flexibilidade funcional é comumente obtida através de atualizações de software, mas desta forma a mudança é limitada somente à parte programável dos sistemas. A Figura 6 apresenta a reconfiguração de um projeto utilizando hardware reconfigurável (GARCIA; et al., 2006). Em dispositivos reconfiguráveis, como FPGAs, têm-se quatro tipos de reconfiguração:

• Reconfiguração total: É a forma de configuração onde o dispositivo reconfigurável é inteiramente alterado.

• Reconfiguração parcial: É a forma de configuração que permite que somente uma porção do sistema digital seja reconfigurada. Uma reconfiguração parcial pode ser não-disruptiva ou disruptiva. A reconfiguração parcial é não-disruptiva quando as porções do sistema que não estão sendo reconfiguradas permanecem completamente funcionais durante o ciclo de reconfiguração e a reconfiguração parcial é disruptiva quando outras partes do sistema são afetadas, tipicamente necessitando de uma parada no sistema inteiro.

• Reconfiguração dinâmica: A reconfiguração dinâmica é a forma de reconfigurar o dispositivo em tempo de execução, sendo denominada como run-time reconfiguration (RTR), on-the-fly reconfiguration ou in-circuit reconfiguration. Reconfiguração dinâmica é outra

forma de expressar a reconfiguração parcial não-disruptiva. O termo implica não haver necessidade de reiniciar o circuito ou remover elementos durante a reconfiguração.

• Chaveamento de contexto: É a capacidade de um dispositivo ou sistema de ser configurado em tempo de execução, durante a operação do sistema, em função de um conjunto de arquivos de configuração pré-carregados em uma memória de controle interna ao dispositivo. Pode estar associado a procedimentos de reconfiguração parcial ou total.

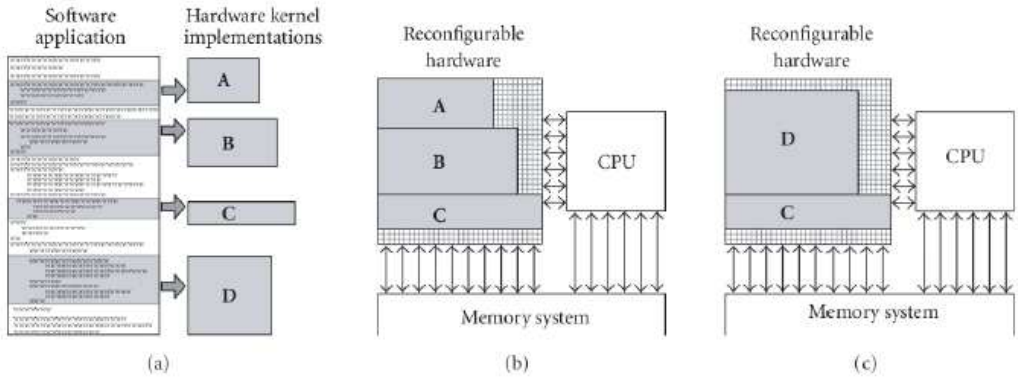

Figura 6: Reconfiguração de projeto em arquiteturas reconfiguráveis (GARCIA; et al., 2006).

2.3.2 Reconfigurabilidade

As arquiteturas reconfiguráveis ou arquiteturas de sistemas computacionais reconfiguráveis são aquelas onde os blocos (módulos) lógicos construtivos básicos podem ser reconfigurados, na sua lógica ou funcionalidade interna, e os blocos de interconexão responsáveis pela interligação desses blocos lógicos construtivos e pela definição da estrutura da arquitetura também podem ser reconfigurados. Esses blocos lógicos construtivos normalmente implementam ou são as unidades funcionais de processamento, armazenamento, comunicação ou entrada e saída de dados (TODMAN; et al., 2005).

Os FPGAs (Field Programable Gate Arrays) representam uma classe de dispositivos de lógica programável com a capacidade de reconfiguração e de poder ser configurado diversas vezes após a sua fabricação. Diferentemente dos dispositivos programáveis de menor

densidade tais como os CPLDs, os FPGAs apresentam uma grande densidade interna de blocos lógicos configuráveis (CLBs) que podem ser interconectados através de uma rede interna de roteamento de sinais, como representado na Figura 7.

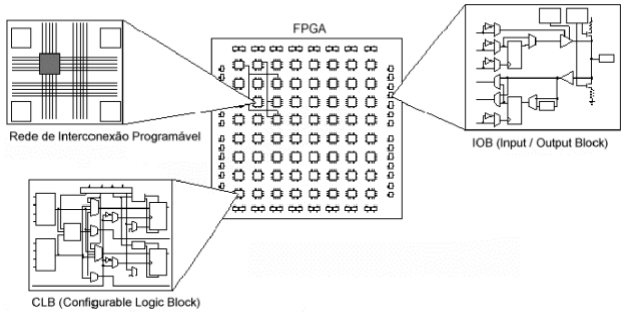

Figura 7: Arquitetura interna de uma FPGA (MARTINS; et AL., 2003).

O arquivo de configuração é obtido a partir da síntese de arquivos de descrição de hardware (HDL) ou de esquemáticos. Apesar de terem programação volátil, placas com FPGAs normalmente possuem algum tipo de memória não-volátil para que a programação seja feita de forma autônoma e rápida.

2.3.3 Linguagem de programação através de descrição de hardware

As linguagens de descrição de hardware, ou HDLs, foram desenvolvidas a partir da necessidade de criação de uma maneira simples e completa de se especificar um sistema digital. Seu principal mérito é permitir que todas as etapas de um projeto (especificação, simulação, descrição de implementação e síntese) sejam realizadas a partir de uma única notação.

Existem diversas linguagens desenvolvidas com capacidade para descrever a especificação e implementação de circuitos digitais, como por exemplo: VHDL (IEEE 1076), Verilog (IEEE 1364), ABEL (Advanced Boolean Equation Language), HardwareC, ISPS (Instruction Set Interchange Format), e AHDL (Altera Hardware Description Language). Dessas linguagens, duas tornaram-se padrão e são amplamente utilizadas atualmente: o

VHDL, e o Verilog, ambos são normas da IEEE. A popularização das HDLs deu-se principalmente pelo surgimento das ferramentas de síntese para dispositivos programáveis, que permitem a implementação de um hardware digital a partir de sua descrição VHDL ou Verilog.

2.3.3.1 VHDL

VHDL (Hardware Description Language) é uma linguagem de descrição de hardware voltada para a simulação de sistemas eletrônicos, sendo posteriormente utilizada para síntese de circuitos digitais em dispositivos programáveis, como por exemplo, FPGAs. Inicialmente, a linguagem VHDL foi utilizada pelo departamento de defesa dos Estados Unidos para descrição técnica e projeto de novos circuitos integrados, visando acelerar o projeto e auxiliar na manutenção dos equipamentos eletrônicos utilizados pelas forças armadas. A linguagem VHDL permite a descrição de hardware em três níveis de abstração: circuitos digitais (portas lógicas básicas), máquina de estados e programação estruturada.

2.3.4 Microcontroladores dedicados em Hardware Reconfigurável

Atualmente, plataformas híbridas, onde tem-se a combinação de uma FPGA com uma CPU (ASIC) denominadas de CSoC (*Configurable System on a Chip*), apresentam um ganho em flexibilidade, consumo de energia e capacidade de processamento. Pode-se ter essa implementação de duas maneiras, denominadas de *hardcore* e *softcore*. Um *softcore* é uma CPU sintetizada e mapeada aos recursos lógicos de uma FPGA. A grande vantagem de se poder usar um processador *softcore* está no fato de que seu código pode ser adaptado às necessidades da aplicação, sendo que somente o hardware necessário é sintetizado no FPGA, reduzindo assim a quantidade de lógica necessária e o consumo de energia.

O aumento da abstração na descrição da aplicação e, consequentemente, na síntese do microcontrolador, facilita a implementação do projeto e a reutilização de código. A utilização de linguagem *Assembly* e C, largamente utilizadas em sistemas embarcados, tornam o código complexo e difícil de ser reaproveitado. Assim, a linguagem Java tem sido muito utilizada. Projetistas de sistemas embarcados têm adotado linguagem Java em função da independência do hardware hospedeiro oferecida por essa tecnologia, conferindo alto grau de portabilidade às aplicações. Portanto, portabilidade e reuso de código, através da orientação a objetos da linguagem Java, podem proteger os investimentos prévios em desenvolvimento de software (TAN et al., 2006).

Foi nesta perspectiva que o ambiente SASHIMI foi desenvolvido para sintetizar um microcontrolador dedicado (*softcore*), a partir de uma descrição em linguagem de alto nível.

2.3.4.1 SASHIMI/FemtoJava

O SASHIMI (*Systems As Software and Hardware In MIcrocontrollers*) é uma ferramenta que se destina à síntese de sistemas microcontrolados descritos em linguagem Java [UFRGS(2006)]. O alvo dos sistemas sintetizados pelo SASHIMI é o microcontrolador FemtoJava (ITO; CARRO; JACOBI, 2001), gerado de forma particular para cada aplicação desenvolvida. A abordagem SASHIMI permite a utilização de uma linguagem de alto nível (Java) não somente para programação, mas para a especificação completa do sistema contendo o microcontrolador.

As ferramentas do ambiente SASHIMI suportam a extração automática do subconjunto de instruções Java, necessário para implementar o software da aplicação. Para cada sistema pode ser gerado um microcontrolador específico (com o conjunto de instruções adaptado) e automaticamente adaptar o seu software. Portanto, o resultado é o melhor aproveitamento dos recursos de hardware e a obtenção de um software otimizado para cada aplicação. A Figura 8 ilustra o fluxo de projeto definido para o SASHIMI, partindo do código fonte da aplicação, descrito em linguagem de programação Java, até a síntese do processador FemtoJava e software dedicado descrito em VHDL.

O microcontrolador FemtoJava é composto por uma unidade de processamento baseada em arquitetura de pilha, memórias RAM e ROM integradas, portas de entrada e saída

mapeadas em memória e um mecanismo de tratamento de interrupções com dois níveis de prioridade. Na memória ROM, o programa Java descrito é carregado. A arquitetura da unidade de processamento implementa um subconjunto de instruções da Máquina Virtual Java, e seu funcionamento é consistente com a especificação da Máquina Virtual Java. A utilização de registradores para armazenar elementos da pilha possibilita, ainda, ganho de desempenho, redução na área ocupada em FPGA e o processamento realizado simultaneamente aos acessos à memória.

Em (WEHRMEISTER; PEREIRA; BECKER, 2006) foi desenvolvida uma API para incorporar, ao ambiente SASHIMI, o padrão RTSJ (*Real Time Specification for Java*), dando suporte a tarefas concorrentes e permitindo a especificação de restrições temporais.

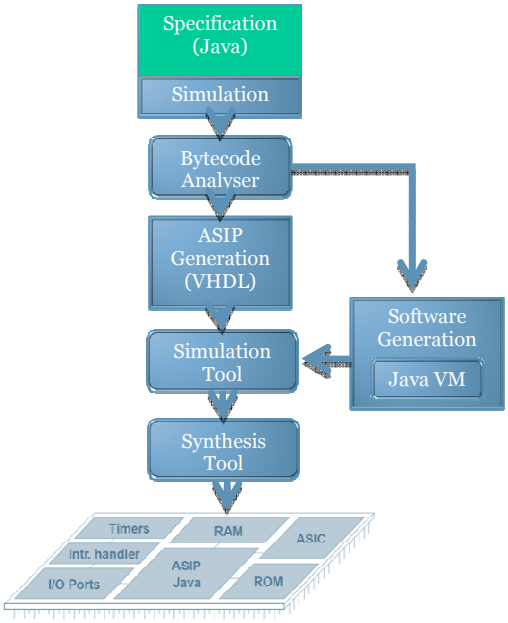

Figura 8: Fluxo de projeto do SASHIMI (WEHRMEISTER; PEREIRA; BECKER, 2006).

A estrutura de escalonamento desenvolvida consiste em uma tarefa adicional encarregada de alocar o processador para aquelas tarefas da aplicação que estão prontas para execução, com base na política de escalonamento implementada. Essa abordagem é a mesma utilizada em sistemas operacionais de tempo-real. O Femtojava suporta até oito tarefas estáticas (não há criação de tarefas em tempo de execução), periódicas e independentes. Assim, possibilita-se o desenvolvimento de sistemas de tempo-real embarcados otimizados.

3 ARQUITETURA DO OSGI

Neste capítulo é aprofundada a especificação do *OSGi Service Gateway* através da sua arquitetura, visto que o OSGi não se limita a indicar apenas um outro conjunto de APIs. Também se abordará, por meio de alguns exemplos, como são implementados os serviços através de bundles e como os mesmos se relacionam com o framework.

3.1 ARQUITETURA

A arquitetura do gateway de serviços usando o framework OSGi é representada na Figura 9. Ela geralmente consiste de várias camadas. O sistema operacional e a plataforma Java compõem a base da execução ambiente. O OSGi framework é executado sobre a máquina virtual Java. Sobre o framework são executados os bundles de serviços.

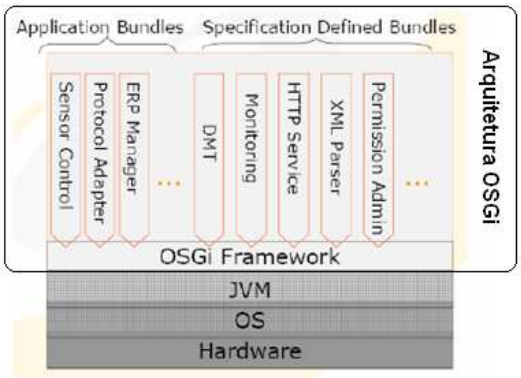

Figura 9: Arquitetura do software para o gateway de serviços (GONG, 2000).

3.2 SERVIÇO

Um serviço faz algo útil e é implementado na linguagem de programação Java. Por exemplo, um servidor Web é um serviço, assim como a aplicação de um pedido de relatórios de consumo de energia, bem como o software de segurança que liga as luzes. A lista de possíveis serviços não tem limites.

Para desenvolver um serviço, inicialmente deve se definir uma interface que diz o que faz o serviço, juntamente com as correspondentes classes de execução que esclarecem a forma como o serviço será executado. Por exemplo, o seguinte é um serviço que reproduz música digital. Primeiro, é mostrada a sua interface de serviço:

```
package player.service;
// imports
public interface DigitalPlayer {
   /**
    * Plays music read from the stream.
    */
   public void play(InputStream musicStream);
}
```

A classe que implementa a interface pode parecer como:

```
package player.impl.mp3;
import player.service.DigitalPlayer;
// other imports
public class MP3Player implements DigitalPlayer {
   public void play(InputStream inStream) {
      // Read MP3 encoded bytes from the stream,
      // send to the appropriate codec
      // available in this service implementation.
   }
}
```

A separação entre a interface da implementação garante que a interface para o serviço de chamadas se mantenha estável, mesmo se a execução sofra alterações. No exemplo, o código usando a interface `DigitalPlayer` não será afetado se a classe `MP3Player` for substituída por uma classe `RealAudioPlayer`, que iria implementar o mesmo serviço, mas decifrar o fluxo em um determinado formato totalmente diferente.

3.3 BUNDLE

Um bundle é uma forma de empacotamento para os serviços, como já se comentou. Também é uma unidade funcional com um ciclo de vida de operações e de capacidade de carregar classe. Por último, contém bem definidos os métodos que lhe permitem ser ligado ao framework.

3.3.1 Conteúdo de um Bundle

Um bundle é um arquivo JAR que contém as classes e recursos tais como imagens, páginas HTML, e outros arquivos de dados. Bundles são normalmente veículos de entrega e implantação de serviços. Por exemplo, um pacote que contém o serviço de `music player` também pode incluir arquivos MP3 para tocar. Outro bundle que contém um serviço de controle de iluminação também pode vir com um conjunto de páginas HTML e imagens que servem como sua interface. O código de empacotamento, juntamente com os dados de implantação, torna-se muito conveniente. A Figura 10 apresenta uma estrutura interna de um pacote que contém o arquivo JAR para um serviço.

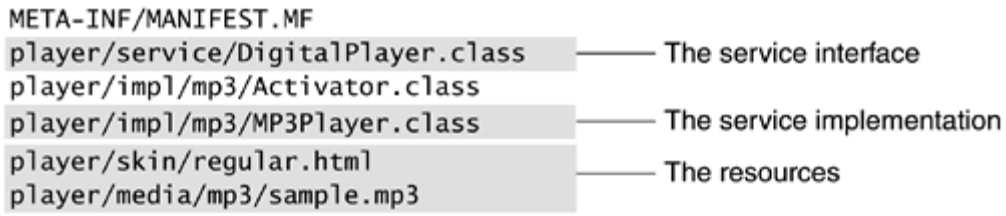

Figura 10: Estrutura interna de um bundle (GONG, 2000).

O conteúdo do bundle mostra que contém a classe de arquivos na interface e implementação para o serviço de música digital, assim como os recursos de uma amostra de música MP3 e uma interface de usuário em HTML para o serviço.

3.3.2 Bundle como módulo funcional

Um bundle não é apenas um arquivo estático. É associado a operações do ciclo de vida e é auto-contido com relação de class loading.

3.3.2.1 Ciclo de vida de um bundle

Um bundle pode assumir o estado de instalado, iniciado, atualizado, parado, e desinstalados pelo framework. Cada uma destas ações move o bundle para um novo estado. A Figura 11 mostra o conjunto de estados de transição.

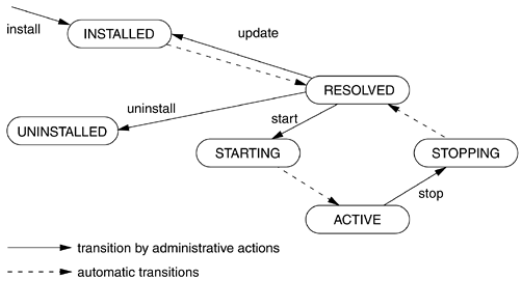

Figura 11: Diagrama de estado de transição do bundle (OSGi, 2008).

Quando um pacote é iniciado, ele se torna ativo, o que significa que ele funciona de acordo com a prescrição do desenho. Por exemplo, depois que um pacote HTTP é iniciado, ele está pronto para aceitar pedidos de Web browsers.

3.3.2.2 Bundle Class Loading

Cada bundle ativo tem a sua própria classe de carregamento que, por padrão, pode carregar somente classes dentro do pacote em si. O código dentro de um bundle não pode fazer referência às classes dentro de outro bundle. Esta é a forma como bundles alcançam um isolamento a partir de outro.

Esta funcionalidade protege as classes em funcionamento em um bundle, de conflitos com as de outra. Por exemplo, se um bundle define uma classe `exemplo.class` e um outro bundle define uma classe com o mesmo nome, mas com semântica inteiramente diferente, o pacote exemplo não será exportado e compartilhado, os dois bundles podem coexistir, e ambas as versões do `exemplo.class` podem ser carregados e executados sem problema algum dentro da máquina virtual Java.

3.3.3 As ligações com o framework

Em razão dos bundles serem instalados no framework, são definidos mecanismos de interface padrão dentro de cada bundle a partir do qual o framework aprende e aceita o bundle. Eles são o manifesto e o ativador do bundle.

3.3.3.1 O Manifesto

O manifesto é um arquivo padrão de entrada em um arquivo JAR. Inclui meta-informações sobre o arquivo em si, como o checksums e assinaturas de cada um dos arquivos JAR quando o arquivo é assinado digitalmente. Um conjunto adicional de cabeçalhos foram definidos na especificação OSGi Service Gateway de qual framework pode adquirir conhecimento sobre o bundle para hospedá-lo com sucesso. Este arquivo é mostrado como a entrada `META-INF/MANIFEST.MF` (Figura 12) no exemplo e pode-se declarar da seguinte maneira:

```
Bundle-Activator: player.impl.Activator
Export-Package: player.service
Import-Package: http.service
```

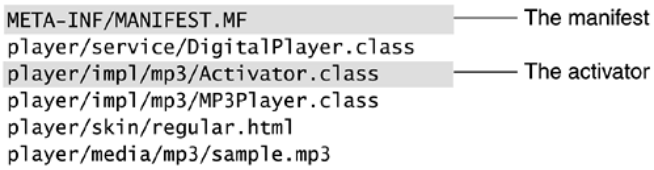

Figura 12: Visualização da estrutura interna do bundle (OSGi, 2008).

Esta declaração identifica ao framework que este ativador é a classe `player.impl.Activator`. O bundle exporta as classes no pacote `player.service` e exige o pacote `http.service` para ser exportado por outro bundle.

3.3.3.2 O Bundle ativador

O ativador é executado pelo bundle para executar operações no momento em que o bundle é iniciado e parado. Ele implementa os métodos `start` e `stop` na interface `org.osgi.framework.BundleActivator`, que é definida como segue:

```
public interface BundleActivator {
   public void start(BundleContext context);
   public void stop(BundleContext context);
}
```

Na Figura 13 é ilustrada toda a estrutura interna de um bundle.

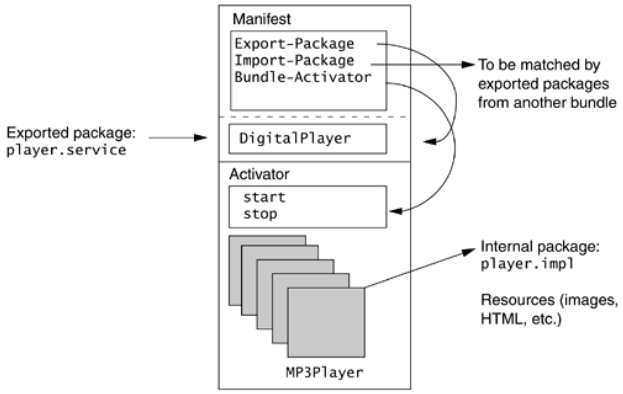

Figura 13: Anatomia do bundle (OSGi, 2008).

Devido à interação entre bundles e o framework ser padronizada, o bundle pode ser desenvolvido fora do seu ambiente de hospedagem.

3.3.4 Instalando um Bundle

Para instalar um bundle (Figura 14), simplesmente se fornece ao framework o arquivo JAR do bundle. O arquivo JAR do bundle pode residir em um servidor Web ou no sistema de arquivo local.

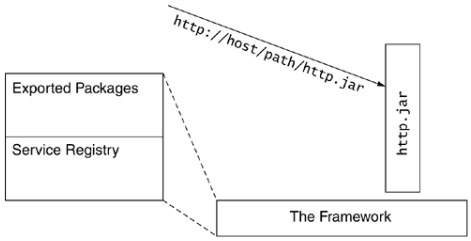

Figura 14: Instalação de um bundle (OSGi, 2008).

O framework recupera o conteúdo do bundle a partir de determinado local para instalá-lo no gateway. Uma vez instalado, o bundle é atribuído a um único integrante, e seu estado se torna instalado. Um evento é difundindo e notifica os ouvintes que o pacote foi instalado.

3.3.5 Inicializando o Bundle

O framework examina o cabeçalho do manifesto do bundle, e extrai as seguintes informações sobre o novo bundle:

• O que este bundle oferece para a exportação (`Export-Package` cabeçalho)

• Quais pacotes este pacote precisa de importação (`Import-Package` cabeçalho)

• Qual é a classe ativadora para este bundle (`Bundle-Activator` cabeçalho)

Se o bundle precisa importar pacotes, o framework verifica qualquer um dos bundles com o estado de resolvidos se estes têm os pacotes para exportar.

Se o bundle está resolvido ou não importa qualquer pacote, ele entra no estado RESOLVED. Em seguida, exporta os seus pacotes, como declarado no cabeçalho do manifesto `Export-Package`, para o framework.

O framework cria um objeto `BundleContext` para o bundle, solicita o método `START` do ativador do bundle, e passa em `BundleContext` como o argumento. O bundle deve registrar todos os serviços que presta no método `start` neste momento. Por um breve período em que o método `start` é executado, o bundle fica temporariamente no estado iniciado.

Em caso de sucesso na ativação, o bundle é alterado para o estado ativo, e um outro evento é difundido e notifica os ouvintes interessados que o bundle foi iniciado (Figura 15).

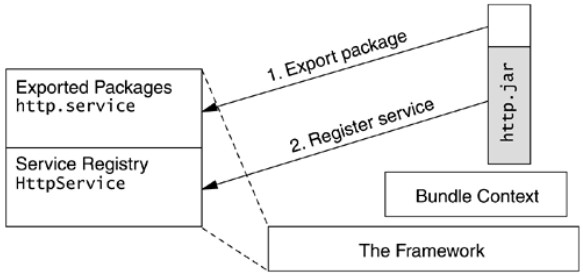

Figura 15: Inicializando o bundle (OSGi, 2008).

3.3.6 Importando pacotes e obtendo serviços

O framework retorna uma referência para o serviço solicitado, representado por um objeto `ServiceReference`, chamando para o bundle (`mp3.jar`, neste exemplo). O chamado então procura obter o serviço em si e invoca a API apropriada definida pelo serviço (Figura 16).

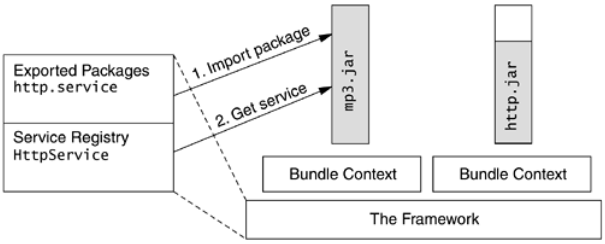

Figura 16: Importando pacotes e obtendo serviços (OSGi, 2008).

Um aspecto importante da interação é que a interface `HttpService` é o contato entre a chamada do bundle `mp3.jar` e o bundle prestador de serviços `http.jar`, e é a única interface do pacote `http.service` exportado por este último (Figura 17).

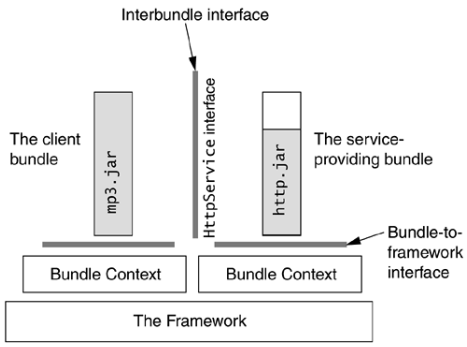

Figura 17: Mecanismos de interfaceamento (OSGi, 2008).

3.3.7 Atualizando um bundle

Uma das características mais importantes do framework é a capacidade de atualizar um bundle em tempo de execução, o que permite que uma nova versão de um bundle execute no gateway sem reiniciá-lo. Isto é essencial para a entrega de correções e aperfeiçoamentos à característica destacada em gateways no campo.

O novo bundle deve fazer mudanças apenas na execução, e não nos pacotes exportados. Esta prática minimiza interrupções em bundles que dependem do bundle que sofreu a atualização.

3.3.8 Finalizando a execução e removendo bundles

Ao finalizar a execução de um bundle, o framework chama o método `stop` do bundle ativador. Embora o método esteja sendo executado, o bundle rapidamente é levado ao estado de parado. O framework, em seguida, executa automaticamente as seguintes tarefas:

1. Cancela o registro do serviço prestado pelo pacote. Durante este processo, um evento é difundido a notificar todos ouvintes interessados que o serviço não está mais registrado;

2. Libera quaisquer serviços em uso pelo bundle;

3. Remove qualquer evento acrescentado pelo bundle;

4. Move o bundle de volta para o estado RESOLVIDO;

5. Transmite um outro evento para notificar ouvintes que o bundle foi interrompido.

Quando desinstalado, o pacote é retirado do gateway, um evento é comunicado aos ouvintes que o bundle foi desinstalado, e seu estado se tornou desinstalado.

Mesmo quando um bundle está parado ou desinstalado, os pacotes que exportou permanecem em vigor para garantir que outros bundles que dependem dos pacotes continuem a ter acesso a eles.

3.4 O FRAMEWORK

O framework proporciona um ambiente de hospedagem para os bundles. As suas responsabilidades são:

• Gerenciamento do ciclo de vida dos bundles;

• Resolver interdependências entre os bundles tornando classes e recursos disponíveis a partir de um bundle;

• Manter um registro dos serviços;

• Monitorar eventos e notificação ouvintes quando um bundle sofre alterações do estado, quando um serviço é registrado ou não, ou quando o framework encontra algum erro.

O framework é a base comum em que todos os serviços dos bundles instalados e registrados venham a desempenhar.

Como demonstrado anteriormente, o pacote do ativador inicia e pára os dois métodos, passando por um objeto `org.osgi.framework.BundleContext` com um único argumento. Para as classes dentro do bundle, esse objeto é a "janela" para o framework e representa o ambiente de execução no qual o bundle se encontra.

3.5 COOPERAÇÃO ENTRE BUNDLES E SERVIÇOS

É referido que bundles são isolados uns dos outros, porque eles têm a sua própria classe para carregar. No entanto, bundles e os seus serviços também têm de cooperar entre si para alcançar alguma funcionalidade global. Uma prática é construir uma aplicação utilizando bundles como blocos.

O compartilhamento é realizado pelos pacotes exportadores e importadores para os bundles e para os registros e para a obtenção de serviços.

3.5.1 Exportando e importando pacotes

Um bundle pode exportar nenhum, alguns ou todos os seus pacotes. Um bundle manifesta a sua intenção de exportar pacotes declarando os pacotes com o `Export-Package` no cabeçalho do seu manifesto. Múltiplos pacotes podem ser separados por vírgulas. Cada pacote pode ser opcionalmente seguido por uma vírgula e uma versão especificação. Por exemplo,

```
Export-Package: com.acme.foo; specification-version=2.0,
   com.acme.bar
```

declara que pretende exportar o pacote versão 2.0 do pacote `com.acme.foo` e versão 0,0 do pacote `com.acme.bar` (porque nenhuma versão é fornecida).

Um bundle pode também declarar pacotes a importar, o que indica que o pacote precisa usar as classes específicas, a fim de executar. O framework faz a ligação e garante que

o importador encontra o exportador antes que o primeiro possa ser iniciado. Esse processo é chamado de resolução, e um bundle é dito ser resolvido se os pacotes necessários para a importação já foram exportados por outros bundles presentes no framework.

Um bundle importar pacotes usando no cabeçalho `Import-Package` no seu manifesto. Por exemplo,

```
Import-Package: com.acme.foo; specification-version=1.0,
   com.acme.bar
```

indica que o pacote precisa importar pelo menos a versão 1.0 do pacote `com.acme.foo` e qualquer versão do pacote `com.acme.bar`.

Quando os pacotes são exportados por um bundle, eles são exportados para o framework. A Figura 18 e a Figura 19 ilustram o processo dos pacotes sendo exportados e importados dentro do framework, respectivamente.

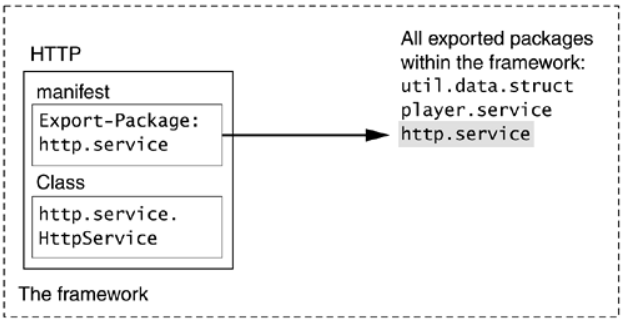

Figura 18: Bundle HTTP exporta o pacote `http.service` (OSGi, 2008).

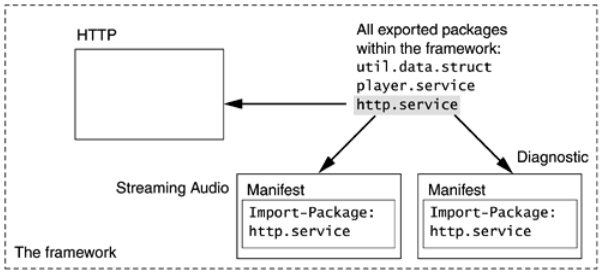

Figura 19: Importando o pacote (OSGi, 2008).

Internamente, o framework interpreta as diretivas dos pacotes importadores e exportadores dos manifestos nos bundles e linka adequadas estruturas de dados em sua class loaders.

Se um pacote já foi exportado, posteriores tentativas por outros bundles de exportar o mesmo pacote são ignorados pelo framework, mesmo se um bundle apresenta uma versão superior. Assim, se um bundle A é exportado pelo seu pacote `http.service` por algum tempo antes do bundle B, a versão B não será exportada, embora superior ao que é exportado por A.

3.5.2 Registrando e Obtendo Serviços

Se um bundle contém alguns serviços, os serviços são registrados com o serviço de registro fornecido pelo framework quando o bundle é iniciado. Registrando um serviço, possibilita para outros bundles usar o código dentro dele. Isso é também conhecido como a publicação do serviço.

```
// in the HTTP bundle's activator
public void start(BundleContext bundleContext) {
   Properties props = new Properties();
   props.put("port", new Integer(80));
   HttpServic http = new HttpServiceImpl();
   bundleContext.registerService("http.service.HttpService",
httpprops);
}
```

Este código registra o `HttpServiceImpl` instanciando `http` sob o nome da classe `http.service.HttpService` juntamente com um serviço de propriedade da porta que vai utilizar.

A estrutura que mapeia os tipos e um conjunto de propriedades de um serviço para o próprio objeto de serviço, faz com que isso seja o registro de serviço. Bundles não são obrigados a publicar serviços. Um bundle pode simplesmente exportar um conjunto comum de utilidade ou socorrer classes como biblioteca para outros bundles utilizando seus manifestos `Export-Package`; ele não precisa ter um ativador. Este tipo de bundle é chamado como biblioteca de bundles.

3.5.3 Pacotes versus Dependências de Serviços

As dependências dos pacotes são expressas pela importação e exportação de pacotes e são determinadas no tempo de desenvolvimento dos bundles.

A dependência de pacotes é estática. Quando um pacote foi exportado, permanece em vigor para todos os seus importadores, mesmo quando o bundle, inicialmente, foi exportado ou parado ou desinstalado. Por exemplo, na figura 14, a desinstalação do bundle HTTP não impede que o streaming de áudio e de diagnóstico de bundles deixe de continuar a utilizar o pacote `http.service`.

Por outro lado, a dependência de serviço é estabelecida por um bundle ao obter um serviço registrado por outro em tempo de execução. Isso acontece após o pacote de dependência entre os dois bundles já ter sido resolvido.

A dependência de serviços é dinâmica. Enquanto um bundle está ativo, ele pode se registrar ou cancelar o seu serviço a qualquer momento. Quando um serviço não está registrado, também dizemos que está retirado. Esta característica dinâmica cria desafios únicos para os chamadores de um serviço; já que, quando se tenta obter o serviço, ele pode não ter sido ainda registrado, de modo que as chamadas são deixadas sem retorno. Mesmo que os chamados tenham sucesso em obter um serviço registrado, pode ser retirado mais tarde, deixando a chamados sem nenhum serviço.

4 ANÁLISE DO ESTADO DA ARTE

Este capítulo apresenta uma análise do estado da arte no que diz respeito a arquiteturas utilizando a plataforma OSGi. A maioria dos trabalhos destacam a eficiência de suas aplicações na construção de midlewares para gateways residências.

A seguir são apresentados alguns trabalhos relevantes nesta área dentre todos que foram analisados nesta pesquisa, destacando pontos interessantes que serviram de ispiração para a realização desse trabalho. Para leitores interessados em uma análise abrangente, sugere-se a leitura (BREIER, 2008).

4.1 GATOR TECH

A proposta Gator Tech (HELAL, 2005) para ambientes pervasivos programáveis é um projeto vinculado ao laboratório de computação móvel e pervasiva da Universidade da Flórida, cujo projeto tem o objetivo de oferecer um modo escalável para o desenvolvimento de tecnologias inteligentes.

O projeto Gator Tech considera a existência de ambientes de execução e bibliotecas de software para o desenvolvimento de ambientes pervasivos programáveis. A descoberta de serviços e os protocolos de gateways integram automaticamente com componentes de sistemas utilizando um middleware genérico que suporta uma definição de serviço para cada sensor e atuador no ambiente.

Dentro da abordagem proposta no projeto Gator Tech estão destacados mecanismos para: (i) tratamento de e-mails, com percepção e notificação aos usuários; (ii) o uso de *tags* RFID para acesso e identificação dos usuários no ambiente; (iii) um simulador de direção para testar as habilidades das pessoas idosas frente ao trânsito, com a intenção de coletar dados para realização de novas propostas de pesquisa; (iv) camas inteligentes, instaladas com equipamentos para monitorar o sono das pessoas; (v) banheiros inteligentes, com exibição de mensagens importantes no espelho, sensores de papel toalha, de fluxo de água e de temperatura; (vi) refrigeradores inteligentes, smartphones, monitoramento e segurança

residencial, chamadas de emergência e diversos outros serviços, que auxiliam os usuários, contemplam o domínio do ambiente pervasivo programável.

Para criar a casa inteligente Gator Tech, foi desenvolvida uma arquitetura de referência genérica aplicável a qualquer espaço de computação pervasiva. Conforme ilustra a Figura 20, o *middleware* proposto contém diversas camadas. No projeto Gator Tech foi implementada a maior parte da arquitetura de referência, embora muito trabalho ainda necessite ser realizado na camada de conhecimento.

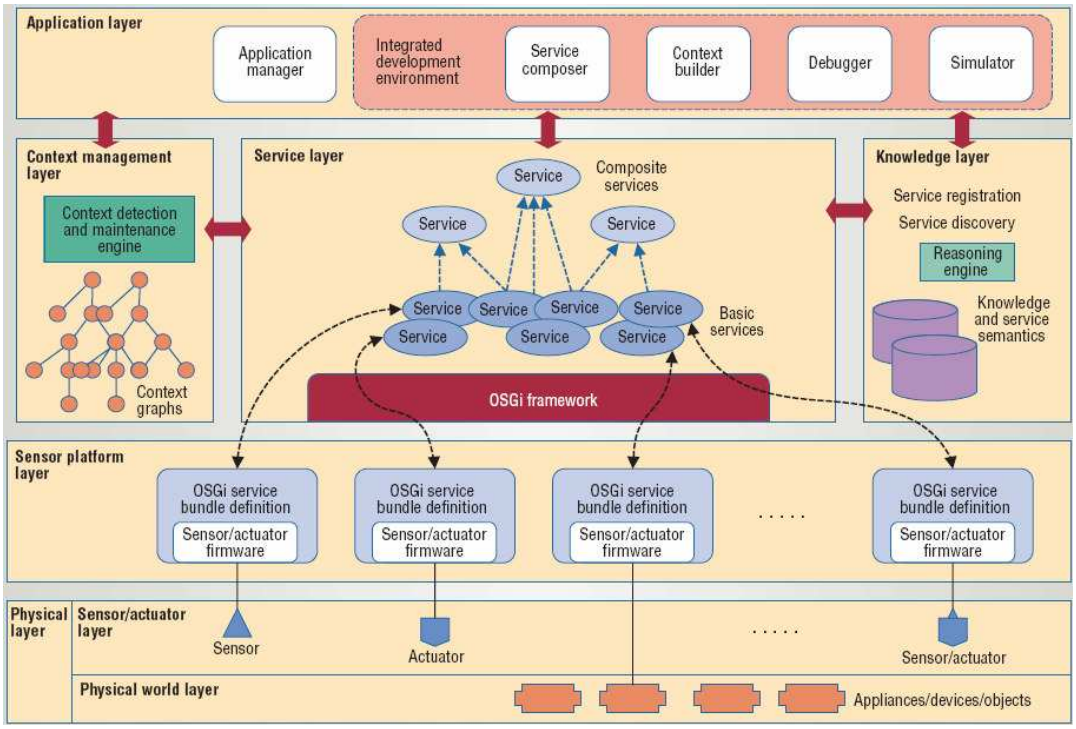

Figura 20: Arquitetura de referência genérica (HELAL, 2005).

As camadas que fazem parte da arquitetura de referência genérica são especificadas como:

Camada Física (*physical layer*): está subdividida em duas camadas a *sensor/actuator layer* e *physical world layer* que são compostas por sensores e vários dispositivos domésticos que são utilizados pelos usuários, tais como: lâmpada, televisor, *set-top box* (STB) e rádio-relógio. Também são incluídos nesta camada dispositivos como: aparelhos de ar condicionado, termostatos e outros.

Camada de Plataforma de Sensores (*Sensor paltform layer*): responsável pela comunicação com os dispositivos de aplicações domésticas, tais como: sensores e atuadores,

os representando para o resto do *middleware* de modo uniforme. A plataforma de sensores é a responsável pela conversão de sensores e atuadores da camada física para serviços de software que podem ser programados ou compostos por outros serviços. Uma vantagem é que os desenvolvedores podem definir serviços sem ter que, necessariamente, entender o mundo físico.

Camada de Serviço (*Service layer*): composta pelo framework OSGi, em que serviços básicos representam o mundo físico por meio da plataforma de sensores, consistindo em bundles de serviços para cada sensor e atuador representado no framework OSGi. Os desenvolvedores de aplicações podem criar serviços compostos utilizando um protocolo de descoberta de serviços para encontrar, e serviços que posteriormente serão utilizados.

Camada de Conhecimento (*Knowledge layer*): contém uma ontologia de vários serviços, dispositivos e aplicações domésticas conectadas ao sistema. Isso cria a possibilidade de raciocínio sobre os serviços, como, por exemplo: o sistema deve converter a saída de um sensor de temperatura antes de realizar um outro serviço. A máquina de raciocínio determina se algum serviço composto está disponível em um dado momento.

Camada de Gerenciamento de Contexto (*Context Management Layer*): permite que os desenvolvedores de aplicação criem e registrem os contextos de interesse. Cada contexto é implementado como um serviço OSGi. Um contexto pode definir ou restringir a ativação de serviços para várias aplicações, podendo, também, especificar estados em que um ambiente pervasivo não pode entrar. A máquina de contexto é responsável por detectar e, possivelmente, recuperar os estados.

Camada de Aplicação: consiste de um gerenciador de aplicação para ativar e desativar serviços, e de um ambiente de desenvolvimento integrado com várias ferramentas para auxiliar na criação dos ambientes inteligentes. Com o *context builder*, um desenvolvedor pode construir visualmente um gráfico que relaciona comportamento com contexto. É possível os desenvolvedores utilizarem agregador de serviços para procurar e descobrir os serviços ou registrar um novo serviço.

Quanto aos módulos implementados no projeto Gator Tech, vale salientar a existência de um mecanismo chamado *smart plugs*, que oferece um modo inteligente de percepção quanto aos dispositivos elétricos que são instalados no ambiente. Cada tomada de energia elétrica do ambiente é equipada com um leitor RFID conectado a um computador principal.

Já, os dispositivos que utilizam a eletricidade, tais como: lâmpadas, aparelhos de ar condicionado, dentre outros, possuem uma *tag* RFID anexada ao *plug* do dispositivo e que contém informações individuais. A idéia é: quando o usuário conectar um dispositivo elétrico na tomada, a *tag* do dispositivo é lida e as informações são enviadas para o computador central através do leitor de RFID instalado em cada local. Isso permite identificar dispositivos e controlá-los de forma transparente através dos serviços oferecidos pelo ambiente. A Figura 21 ilustra exatamente esta idéia.

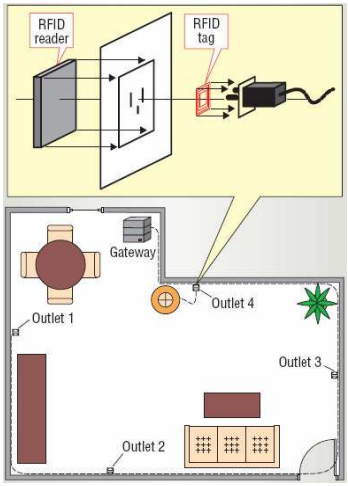

Figura 21: Smart plugs (HELAL, 2005).

4.2 OSGI/BPEL

O framework OSGi/*Business Process Execution Language (BPEL)* (REDONDO, 2007) é uma proposta que visa aumentar o suporte da composição de serviços OSGi presente em ambientes inteligentes. Estes serviços são compostos por um conjunto de serviços OSGi, em que cada um é definido para uma atividade específica. Assim, nesta proposta a idéia é encapsular as informações lógicas de composição dentro de uma aplicação OSGi, sendo que a essa composição é registrada no framework do OSGi. Quando o serviço é instanciado, a execução interpreta a descrição da composição para instanciar e gerenciar os processos que estão em execução. A proposta tenta ser transparente tanto para os serviços de composição solicitados quanto para o serviço de registro OSGi. São definidos “virtual bundles” que especificam um serviço composto utilizado a BPEL e a BPEL Engine para composição OSGi.

A BPEL Engine (servidor de aplicação que é responsável por executar as composições BPEL) é inserida na arquitetura com um conjunto de serviços OSGi.

Como ilustrado na Figura 22, os componentes de execução virtual (virtual bundles), são componentes de execução (bundles) que não oferecem propriamente a execução de serviço, mas sim a especificação de um serviço composto. Um “virtual bundle”, assim como qualquer bundle, é empacotado em um padrão JAR que inclui: (i) um arquivo com a definição BPEL do serviço composto; (ii) um Bundle Activator responsável por registrar o serviço composto e notificar o BPEL Engine. Para isso, a seqüência de eventos segue como: o “Bundle Activator” obtém a localização da BPEL Engine (1) e notifica a BPEL Engine sobre sua existência, “AddBPELProcess” (2). Quando a BPEL Engine é notificada sobre um novo serviço BPEL OSGi (3) é criado um componente de execução para um “virtual bundle” que registra o serviço composto no framework OSGi (4). O componente de execução é um processo BPEL responsável por gerenciar um serviço composto quando instanciado.

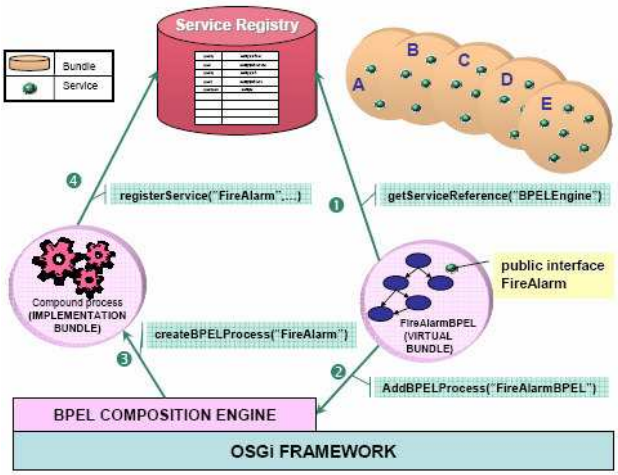

Figura 22: Registrando uma composição de serviços OSGi (REDONDO, 2007).

A proposta é implementada considerando como exemplo um cenário de ambiente inteligente, sendo instalados quatro componentes de execução (B, C, D, E) vide Figura 22 que fornecem os serviços para definir a lógica de segurança. Para o exemplo, é descrito um código BPEL para o serviço “trigAlarm” na interface “FireAlarm”, da seguinte forma:

Na primeira parte da especificação são detalhados os serviços atômicos OSGi, (actvAlarm) e os nomes das variáveis de entrada e saída desses serviços. Na outra parte da especificação são definidas as associações entre as variáveis, em que a saída de um serviço pode ser a entrada de outro.

Quando um componente de execução A precisa usar o serviço “FireAlarm”, ele obtém a referência para o componente de execução do registrador de serviços através do comando “getServiceReference” e utiliza o método “trigAlarm” da interface “FireAlarm”, conforme ilustra a Figura 23.

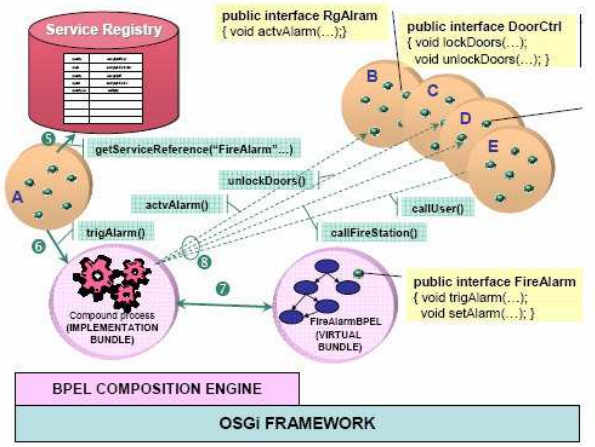

Figura 23: Instanciando um serviço OSGi composto (REDONDO, 2007).

Observa-se que o componente de execução lê a informação BPEL e solicita o serviço apropriado (componentes B, C, D e E) na ordem especificada pela descrição BPEL. Portanto, do ponto de vista de um componente de execução solicitante, não há diferença entre utilizar um serviço OSGi simples ou composto, sendo o componente de execução um componente comum.

Com o framework proposto novos serviços podem ser instalados e reconfigurados, apenas trocando as especificações BPEL, que consistem num arquivo XML.

4.3 INTEGRAÇÃO TV DIGITAL E SISTEMA DE AUTOMAÇÃO RESIDENCIAL VIA XBUNDLE

A possibilidade de controlar uma residência inteligente através da TV (CABRER, 2006) é uma proposta de ambientes cooperativos que integra a TV digital e as redes domésticas. As tecnologias adotadas na proposta são: Multimedia Home Platform (MHP), para TV Digital e OSGi como uma plataforma para configurações de gateways residenciais. Com os avanços tecnológicos que vêm ocorrendo nos últimos anos, os gateways residenciais

são fundamentais para a criação de pontes de comunicação entre os ambientes inteligentes, seus dispositivos e o mundo externo. Nesta proposta o objetivo é permitir ao usuário interagir através da TV para controlar aplicações e administrar serviços existentes no ambiente inteligente. Por isso, a necessidade em se utilizar o MHP e o OSGi, sendo que o primeiro está orientado a funções, e o segundo orientado a serviços. Como as duas tecnologias possuem arquiteturas bem diferentes, existe o desafio de integração entre elas. Esse problema foi resolvido com a criação de XbundLET, uma aplicação que permite a interação natural entre o MHP e o OSGi. Um Xbundlet não apenas define uma ponte de comunicação entre essas plataformas, como também constitui em um elemento de software híbrido que pode ser executado em ambas as arquiteturas.

Na Figura 24a é possível observar a comparação entre as diferentes arquiteturas, MHP e OSGi, respectivamente. Já a Figura 24b ilustra a solução de integração entre as diferentes plataformas, através da entidade XbundLET.

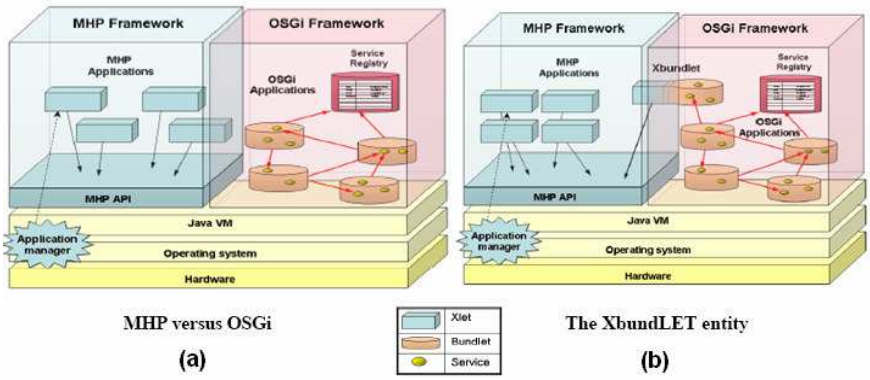

Figura 24: Arquiteturas e integração com XbundLET (CABRER, 2006).

Já a especificação MHP é um modelo com múltiplas camadas, que define uma camada entre o sistema e as aplicações MHP. As aplicações MHP (Xlets) é um conjunto de arquivos de classes Java, escritas para utilizar um conjunto de bibliotecas na API MHP. Um Xlet pode, também, ser totalmente controlado por um gerenciador de aplicações contido nos STBs, os quais são responsáveis por monitorar, iniciar ou parar um Xlet. Para a integração foi desenvolvido o XbundLET como uma aplicação híbrida, e que pode integrar-se tanto com Xlets quanto com bundles.

O ciclo de vida de um XbundLET é definido em quatro estados: (i) iniciado: estado em que o XbundLET não está pronto para executar um Xlet ou um bundle, ele é apenas iniciado; (ii) inativo: estado em que o XbundLET está pronto para executar um Xlet ou um bundle; (iii) ativo: estado em que o XBundLET é ativado; (iv) finalizado: estado para determinar que a execução do XbundLET foi finalizada.

Em relação ao método de como as aplicações MHP utilizam os serviços OSGi, pode ser visualizado (Figura 25a) e definido da seguinte forma: (1) os bundles para controle de dispositivos da residência registram seus serviços no OSGi Service Registry. Para informar o ambiente MHP da existência sobre um novo serviço OSGi, uma nova entidade chamada "Master XbundLET" está constantemente em modo de "escuta" para detectar alguma mudança na plataforma OSGi; (2) O "Service Registry" informa ao "Master XbundLET" sobre a presença de um novo serviço, através do componente "Service Listener"; (3) logo após, o "Master XbundLET" envia uma mensagem atualizada ao Inter-Xlet Communication (IXC) Registry, responsável por permitir a comunicação entre Xlets. Por essa razão, um Xlet do framework MHP pode solicitar ao "IXC Registry" para que procure os serviços OSGi disponíveis até o momento.

Para o método inverso, ou seja, de como as aplicações OSGi utilizam o MHP, está ilustrado, na Figura 25b, sendo definido como: o "Application Manager" exporta as funcionalidades da API MHP para o ambiente OSGi sob um conjunto de serviços gerenciados pelo "OSGI2MHP XbundLETs" que registra suas características dentro do "OSGI Service Registry". Após isso, na visão do framework OSGi, as características fornecidas pela API MHP podem ser utilizadas como qualquer outro serviço OSGi. Já na visão dos bundles não há diferença entre usar os serviços oferecidos por outros bundles ou oferecidos pela plataforma MHP com os "OSGi2MHP XbundLETs".

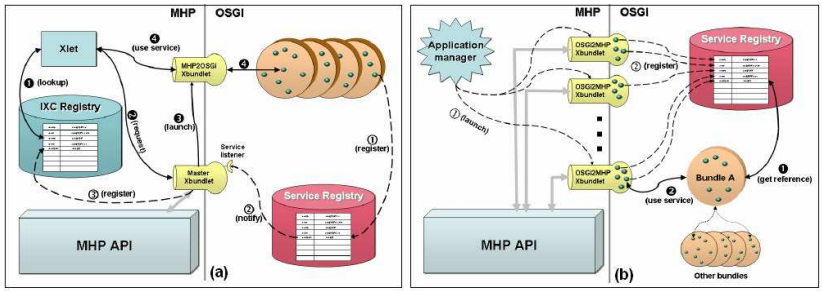

Figura 25: (a) MHP com OSGi (b) OSGi com MHP (CABRER, 2006).

A principal contribuição da proposta é a criação do XbundLET para integração das diferentes arquiteturas, que serve como uma ponte de conexão entre as camadas da arquitetura MHP com os serviços OSGi.

Desse modo, a integração entre essas tecnologias proporciona aos usuários comodidade e praticidade de gerenciamento sobre o ambiente inteligente, seus dispositivos e seus serviços disponíveis, tudo por meio da TV Digital, que oferece muitas possibilidades de integração com os sistemas de automação residencial, bem como as oferta de novos serviços.

4.4 DISCUSSÃO

Analisando os trabalhos apresentados que utilizam o framework OSGi pode-se verificar que a utilização do mesmo introduz muitas características benéficas para a descoberta e composição de serviços, resultando em maiores facilidades para os usuários, já que o framework elimina a necessidade de intermediários para a realização dos serviços.

Além disso, verifica-se também que sua utilização proporciona a facilidade de se realizar atualizações (*upgrades*) em serviços, sem a necessidade de se reiniciar o sistema.

Porém, dentre os trabalhos analisados, observa-se que todos possuem um alto custo computacional, exigindo para seus sistemas a totalidade da máquina virtual Java e computadores rodando como gateways.

Buscando uma plataforma para oferecer uma nova alternativa para o desenvolvimento de gateways customizados e que não necessitam de um alto custo computacional, foi proposta, para este trabalho, a criação de um gateway OSGi-FemtoJava, o qual será descrito no capítulo a seguir.

5 PROPOSTA DO OSGI-FEMTOJAVA

Inicialmente, para o desenvolvimento da proposta, foi realizado um estudo a fim de identificar os elementos que compõem basicamente uma arquitetura de sistema de computação pervasiva, com o objetivo de construir um gateway em uma plataforma reconfigurável. Analisado esses componentes, um estudo aprofundado dentro das APIs do OSGi foi realizado, observando em exemplos e utilizando as próprias especificações fornecidas com o framework a funcionabilidade das mesmas, verificando quais métodos e classes eram usadas e quais seriam suportadas utilizando a ferramenta SASHIMI para serem posteriormente sintetizadas na FPGA.

5.1 COMPONENTES DE UM GATEWAY DE SUPORTE À COMPUTAÇÃO PERVASIVA

A definição de um modelo que proporcione uma base de componentes para serem implementados, se faz necessária para o desenvolvimento dessa dissertação. Nas pesquisas realizadas, observou-se que os pesquisadores da universidade de Nankai desenvolveram uma arquitetura genérica, baseada em componentes e middleware, a qual contém uma série de funcionalidades para computação pervasiva (XU; XIN; LU, 2007). Dentre essas funcionalidades é possível destacar a descoberta de serviços, o controle de segurança e o gerenciamento da comunicação.

A arquitetura é composta, essencialmente, de serviços centrais que são implementados através de componentes e middleware. Os objetos da aplicação podem se comunicar com o "*Service Manager*", que é um dos componentes centrais e responsável por fornecer interfaces entre diversos componentes de serviços para os objetos da aplicação. Há um agente de gerenciamento responsável pela comunicação com a infra-estrutura ou com os protocolos de rede, que podem ser, freqüentemente, substituídos em ambientes de computação pervasiva.

Todos os componentes da arquitetura podem ser encapsulados dentro de diferentes categorias de middleware, tais como: descoberta de serviços, contexto, controle de segurança e gerenciamento. Assim, os vários serviços de middlewares são integrados de modo a construir uma arquitetura de sistema completa para computação pervasiva, conforme pode ser observado na Figura 26.

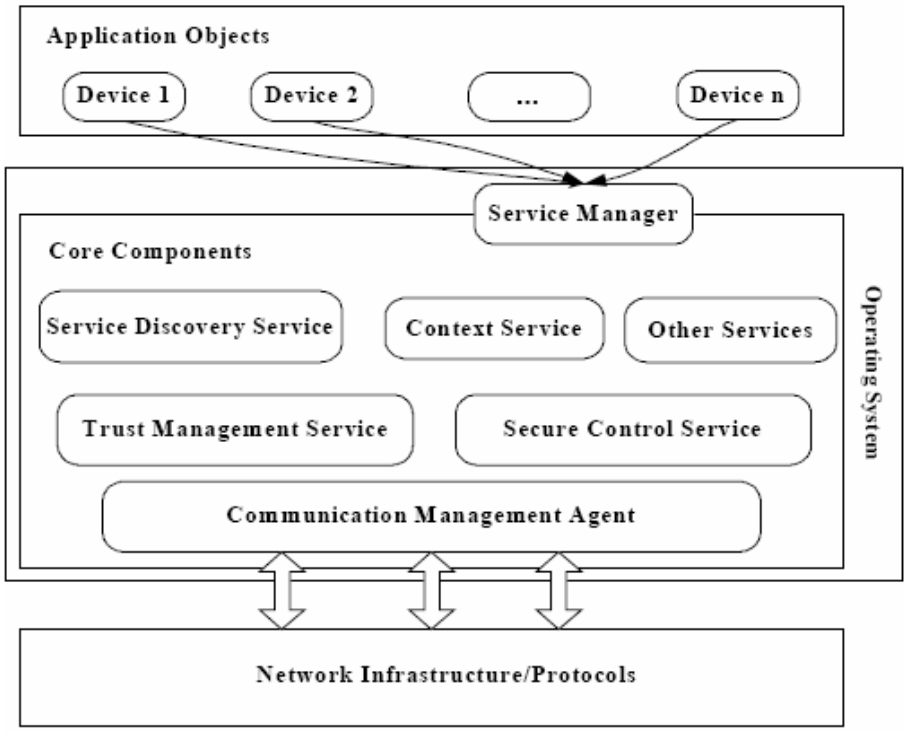

Figura 26 Arquitetura de sistema para computação pervasiva (XU; XIN; LU, 2007).

Nessa arquitetura, novos grupos de middlewares que ofereçam outros serviços podem ser facilmente adicionados, tornado-a flexível e extensível. Os componentes centrais da arquitetura proposta são definidos em:

Serviço de Descoberta de Serviços (*Service Discovery Service)*: consiste num algoritmo seguro e tolerante a falhas, implementado em linguagem Java, e que realizada funções para garantir o balanceamento de carga do sistema, sem considerar a variação que o ambiente possa ter.

Serviço de Contexto (*Context Service)*: responsável por coletar informações estáticas ou dinâmicas sobre os usuários e os recursos no ambiente de computação pervasiva. Ele pode fornecer as informações obtidas para os componentes "*Service Discovery*" e "*Trust Management*" para auxiliar na tomada de decisões.

Serviço de Controle de Segurança (*Security Control Service)*: devido às características dinâmicas do ambiente pervasivo, as quais são alteradas com freqüência, esse componente é responsável pela segurança no ambiente.

Serviço de Gerenciamento com Base no Histórico de Serviços Executados (*Trust Management Service)*: responsável por realizar uma conexão segura entre os componentes, baseada em informações de históricos (antigas interações), de tempo e de outros parâmetros. Logo, o que determina se uma nova conexão poderá, ou não, ser realizada são as informações obtidas.

Gerenciamento de Comunicação (*Communication Management Agent):* consiste em um agente implementado em linguagem Java, e é responsável por solucionar os problemas de comunicação para o ambiente de computação pervasiva.

Gerenciador de Serviços (*Service Manager):* é responsável pela comunicação entre os componentes centrais e os objetos da aplicação, fornecendo interface para vários componentes da arquitetura. Através deste componente, os objetos das aplicações podem obter acesso e solicitar serviços.

De modo geral, a arquitetura proposta pode integrar aplicações e serviços em ambientes de computação pervasiva. Neste contexto observou-se que, em se utilizando o framework OSGi, seriam necessários para o desenvolvimento dessa dissertação, alguns componentes para a descoberta de serviços, gerenciamento da comunicação e módulos de interação com o meio, os quais podem ser visualizados em destaque na Figura 27.

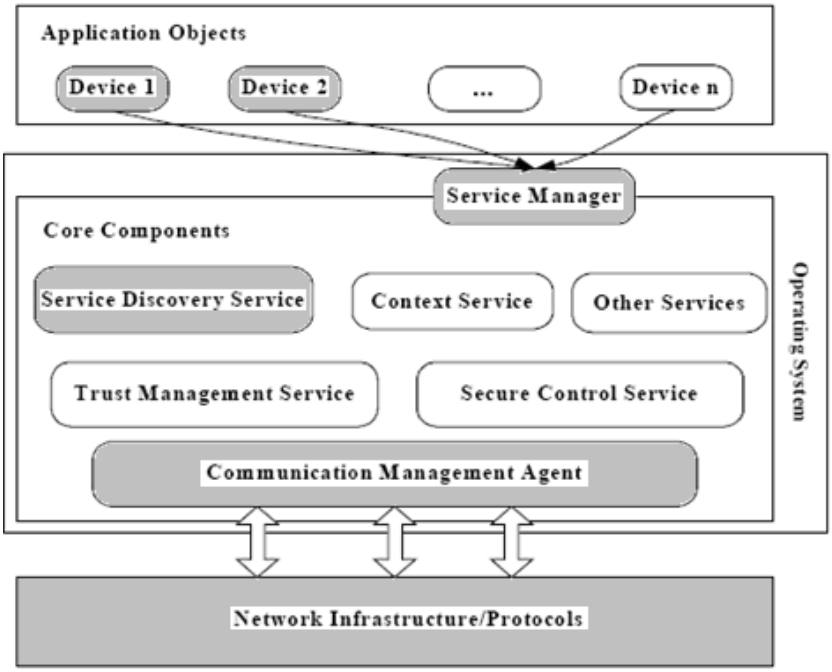

Figura 27: Componentes implementados na presente proposta.

5.2 OSGI-FEMTOJAVA

Neste item será comentado como são desenvolvidos os bundles nesse trabalho. Durante o processo de desenvolvimento de bundles, serão apresentadas as APIs usadas do framework OSGi e testadas no SASHIMI. A especificação completa das APIs utilizadas pode ser encontrada no Apêndice A.

5.2.1 Análise do *bytecodes* do OSGi no SASHIMI

Inicialmente tentou-se realizar a portabilidade do OSGi framework plenamente para o microcontrolador RT-Femtojava através da ferramenta SASHIMI. Depois de alguns experimentos, verificou-se e conclui-se que nem todas as funcionalidades que o framework OSGi oferece são de possível implementação e execução com o RT-Femtojava. Além das limitações de memória que o RT-Femtojava apresenta, algumas classes que o OSGi oferece

necessitam de recursos que só podem ser executadas em máquinas com a completa máquina virtual Java executando.

Observado isso, iniciou-se um estudo das APIs do OSGi a fim de se utilizar apenas o necessário das classes que o framework oferece, com o objetivo de executar uma aplicação focada num gateway de serviços utilizando o RT-Femtojava, ou seja, a adaptação do framework utilizado foi feita nas etapas de geração do software da aplicação a fim de que o SASHIMI pudesse gerar a síntese da aplicação em VHDL.

Com a realização, então, desse estudo, observou-se que um serviço no OSGi deve fornecer uma interface pública que especifica o que faz e as classes de execução com que é feito. O serviço deve ser registrado com o framework e ser útil para outros bundles no gateway.

O processo geral de desenvolvimento de um bundle de serviço no OSGi, de uma forma simplificada, é desenvolvido em algumas etapas:

1 – Escrita da interface de serviço;

2 – Implementação do serviço proposto;

3 – Escrita do bundle ativador;

4 – Declaração dos pacotes a serem exportados;

5 – Compilação das classes dentro do arquivo JAR com o arquivo manifesto junto.

Visto esse desenvolvimento verificou-se que algumas APIs, as quais são descritas nas próximas seções, são fundamentais para a implementação do OSGi.

5.2.2 Relevantes APIs do org.osgi.framework

Algumas das APIs e sua interfaces de serviços são comentadas nos seguintes itens. A ênfase é sobre como usar essas APIs através de exemplos. A formal e completa especificação da API pode ser encontrada no Apêndice A.

5.2.2.1 `BundleContext` Interface

A API `BundleContext` permite que serviços sejam registrados com o framework, a consulta de serviços requisitados, bem como para obter serviços a partir do registro. Uma instância do `BundleContext` é criada pelo framework e é passada para um bundle como um argumento do método `START` e `STOP` do Bundle ATIVADOR. Normalmente, um bundle mantém o `BundleContext` recebido no seu ativador, assim, qualquer serviço obtido através do chamado `getService` sobre `BundleContext` é atribuído a este pacote.

public ServiceRegistration registerService(String Sensor, Object service_temperatura)

O parâmetro `sensor` é o nome da classe do serviço, e o `service_temperatura` é o próprio objeto de serviço. Se um objeto serviço é uma instância de vários tipos, ele pode ser registrado usando as seguintes alternativas, que tem um array como argumento.

public ServiceRegistration registerService(String[] classes, Object service)

O framework garante que este serviço seja realmente do tipo especificado nos parâmetros das classes. Os nomes das classes devem ser totalmente qualificados e devem especificar os nomes das interfaces de serviço (por exemplo, `com.acme.service.temperatura.Temperatura_Service`), e não os nomes das classes execução (por exemplo, `com.acme.impl.temperatura.sensor`).

Além disso, um conjunto de propriedades pode ser registrado juntamente com o serviço. O que é passado junto com essas propriedades é a aplicação específica, e este parâmetro pode ser `null`, se não houver registro de propriedades. O nome da propriedade deve ser do tipo `java.lang.String`; o valor pode ser qualquer objeto. Alguns exemplos podem ser observados a seguir.

```
bundleContext.registerService(
```

```
            "com.acme.service.temperatura.TemperaturaService",
temperaturaService, null);
```

O código a seguir registra um serviço, e registra um conjunto de propriedades que descreve as suas características:

```
Properties props = new Properties();
props.put("location", "Sala 01");
props.put("Sensor", Boolean.TRUE);
props.put("capability",
   new String[] {"Celcius", "Fahrenheit", "Kelvin"});
bundleContext.registerService(
      "com.acme.service.temperatura.TemperaturaService",
TemperaturaService);
```

O código a seguir registra um serviço com várias classes:

```
String[]classes= {"com.acme.service.temperatura.TemperaturaService",
   "com.acme.service.humidade.HumidadeService",
   "com.acme.service.pressao.PressaoService" };
bundleContext.registerService(classes, temperaturaService);
```

Neste caso, `temperaturaService` obviamente deve ser uma instância de `com.acme.service.temperatura.TemperaturaService`, ou `com.acme.service.humidade. HumidadeService`, ou `com.acme.service.pressao.PressaoService`. O framework realiza um tipo de verificação e, se `temperaturaService` não é uma instância de qualquer uma das referidas classes, o serviço de registro falhou.

public ServiceReference getServiceReference(String sensor)

O parâmetro `sensor` é o nome da interface de serviço a ser usado. Se nenhum serviço for encontrado, este método retorna null.

public ServiceReference[] getServiceReferences(String sensor, String filter) throws InvalidSyntaxException

A sintaxe usada para expressar condições no filtro argumento é baseada neste comando.

public Object getService(ServiceReference ref)

Este método recebe o serviço referenciado pelo objeto `ServiceReference` especificado. O chamado é delegado a um método de serviço `getService` se `ref` representa um serviço registrado, e não apenas a devolução dos registrados do próprio serviço.

5.2.2.2 `ServiceReference` Interface

`ServiceReference` serve como um intermediário enquanto o bundle cliente examina várias propriedades do serviço antes dele ser usado. Ele é retornado pela API de `getServiceReference` na interface de `BundleContext` ou pelo método `getReference` na interface `ServiceRegistration`.

public Object getProperty(String key)

Este método retorna o valor específico na propriedade do serviço.

public Bundle getBundle()

Este método retorna o bundle que foi registrado no serviço.

5.2.2.3 `ServiceRegistration` Interface

A API `registerService` na interface `BundleContext` retorna um objeto `ServiceRegistration`.

public ServiceReference getReference()

Este método retorna a referência do serviço para este registro. O serviço referente obtido por diferentes instâncias do `ServiceRegistration` podem ser todos diferentes, mesmo se o objeto de serviço for o mesmo.

Através de experimentos com esses métodos implementados nas classes do OSGi com a ferramenta SASHIMI, observou-se que esses arquivos binários (*Java classfiles*) gerados, foram analisados e os seus *opcodes* utilizados são identificados de forma que a máquina de controle do processador RT-Femtojava seja gerada dando suporte apenas a esses *opcodes*, reduzindo o tamanho do processador.

5.2.3 Adaptação no FemtoJava

O gerenciamento das escritas e leituras do sistema de entrada e saída do microcontrolador Java são de responsabilidade do arbritrador do barramento. Ele considera o endereço contido no barramento para determinar sobre qual porta o microcontrolador deve operar. A escrita dos dados em um periférico ou em uma porta, devem ser primeiro armazenados na pilha juntamente com o endereço correspondente. A mesma estrutura é utilizada para a leitura, onde o dado obtido na porta é armazenado na pilha de operandos. A memória do FemtoJava está dividida em duas partes, a memória de dados e a memória de programa. Na memória de dados, os endereços iniciais, da posição 00H até a posição 10H, contém alguns registradores mapeados em memória, que tem a finalidade de possibilitar a programação das interrupções e dos temporizadores assim como permitir as operações de I/O.

A especificação da máquina virtual Java não define mecanismos para manipular interrupções, pois visa a portabilidade entre diferentes plataformas de hardware. Entretanto o sistema de tratamento de interrupção é um componente importante a ser incluído em um microcontrolador, para facilitar a comunicação entre o processador e os seus periféricos, liberando o processador de ficar esperando a requisição do periférico. Assim o

microcontrolador FemtoJava implementou cinco interrupções distintas com dois níveis de prioridade para facilitar a requisição de periféricos ao processador.

O microcontrolador FemtoJava possui quatro barramentos em sua arquitetura para comunicação com os periféricos, sendo barramentos distintos para o acesso e endereçamento das memórias de programa e dados. Esses barramentos são parametrizáveis, sendo que para a implementação do FemtoOSGi foram utilizados os barramentos de endereços e dados. O FemtoOSGi utiliza esses barramentos para comunicar-se entre o microcontrolador e os dispositivos implementados.

Na Figura 28 estão representadas as ligações entre o barramento do FemtoJava e seus componentes nessa proposta.

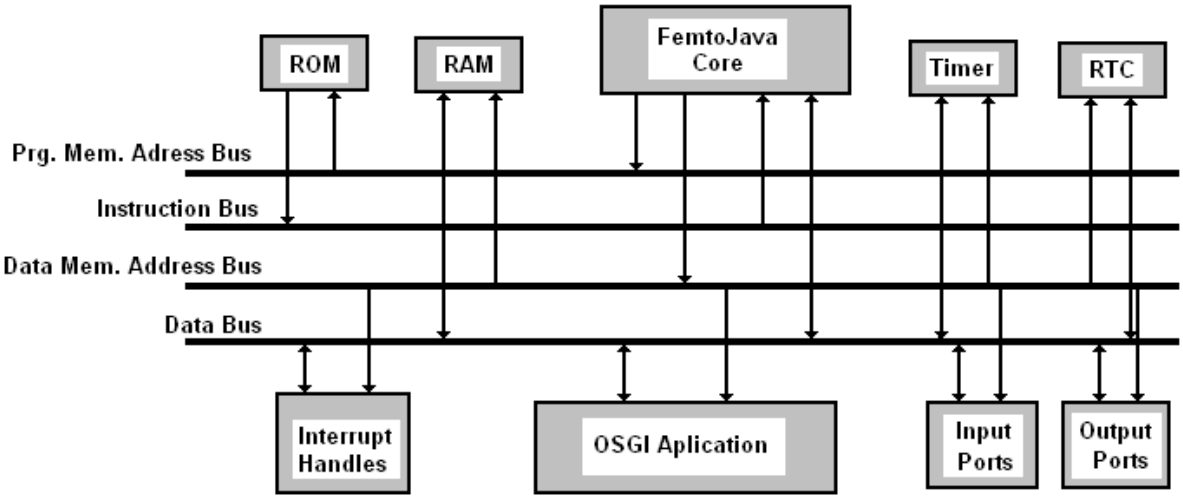

Figura 28: Arquitetura de hardware do FemtoOSGi.

Como o FemtoOSGi realiza a comunicação com o microcontrolador através dos barramentos de endereços e dados, foram introduzidos dois novos endereços como buffers para a realização das escritas e leituras realizadas pelos módulos que realizam a comunicação com o meio. Assim, esses dois novos endereços receberam suas posições na pilha de memória de dados, ocupando os endereços da posição 13H e 14H conforme representado na figura 29.

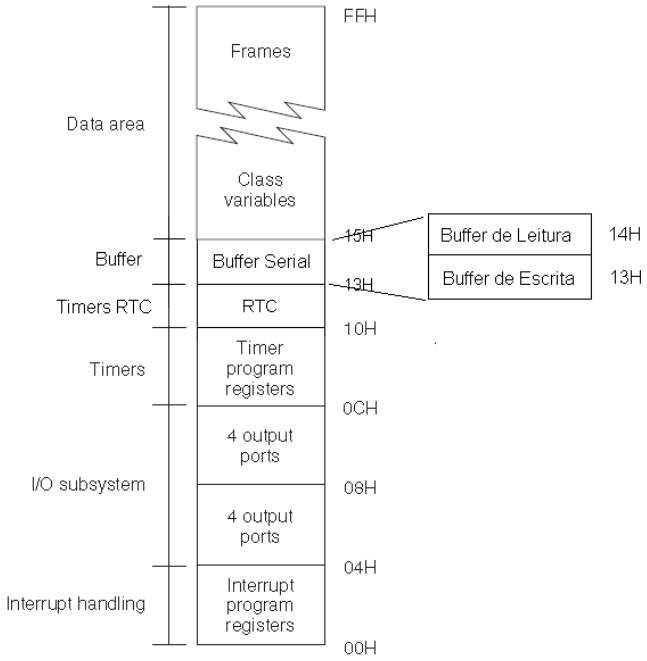

Figura 29: Organização da memória de dados do FemtoOSGi.

Os dispositivos implementados utilizam as interrupções do microcontrolador RT-FemtoJava para realizarem a requisição de comunicações com o processador.

5.3 IMPLEMENTAÇÃO

Para a implementação do sistema foi utilizado o framework do OSGi e desenvolvido bundles para a descoberta dos dispositivos presentes, e bundles de driver para a comunicação além dos bundles para a criação de um sistema de controle de temperatura como proposta para validação da arquitetura.

Para a comunicação com os dispositivos, foi constituído um serial service bundle e um *driver* bundle, conforme a Figura 30.

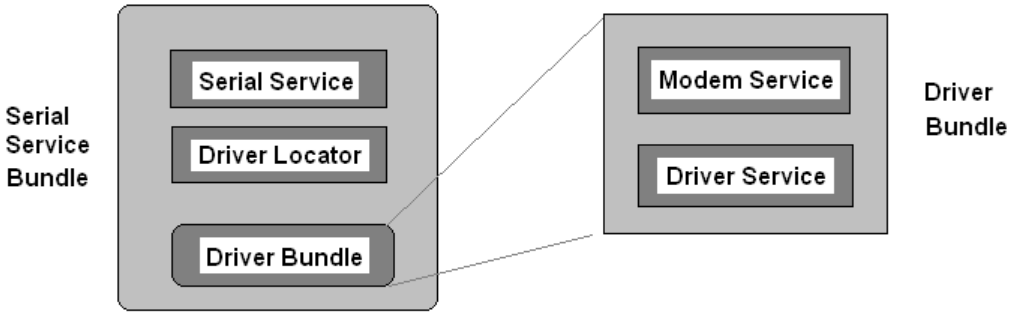

Figura 30: Serial Server Bundle.

Primeiramente, foi desenvolvido um bundle que contém um `device service` que representa o serviço de porta serial. Como qualquer serviço no framework, foi definido primeiramente a interface do serviço conforme Figura 31.

```
import saito.sashimi.*;
import org.osgi.service.device.Device;

public interface SerialService extends Device {
   /**
    * Gets the SerialPort object for the port.
    */
   public javax.comm.SerialPort getPort();

   /**
    * Adds an event listener for SerialPortEvents.
    */
   public void addEventListener(
      javax.comm.SerialPortEventListener l);

   /**
    * Removes the event listener for SerialPortEvents.
    */
   public void removeEventListener(
      javax.comm.SerialPortEventListener l);
}
```

Figura 31: Interface do serviço.

Como especificado, `SerialService` estende a interface `org.osgi.service.device.Device`. Uma instância deste serviço representa uma porta serial disponível para os bundles no framework. Cada porta serial adicional corresponde a uma instância do serviço. O método `getPort` dá acesso à porta serial, enquanto os outros dois métodos permitem adicionar e remover os *listeners* para os eventos na porta serial.

A API `Javax.comm.SerialPort` permite que seja adicionado apenas um *listener* por serial, sendo ele usado para a detecção de dispositivos.

O método ativador enumera todas as portas seriais disponíveis na plataforma. Para cada porta disponível, foi adicionado um *listener* de eventos na serial conforme Figura 32.

```
import saito.sashimi.*;
import javax.comm.CommPortIdentifier;
import javax.comm.SerialPort;


public class Activator implements BundleActivator {
   private Vector ports = new Vector(4);

   public void start(BundleContext ctxt)
      throws Exception
   {
      for (Enumeration enum =
         CommPortIdentifier.getPortIdentifiers();
         enum.hasMoreElements(); )
      {
         CommPortIdentifier portId =
            (CommPortIdentifier) enum.nextElement();
         if (portId.getPortType() ==
            CommPortIdentifier.PORT_SERIAL) {
            try {
               SerialPort port = (SerialPort)
                  portId.open("SerialService", 2000);
               port.notifyOnCTS(true);
               SerialListener serialListener =
                  new SerialListener(ctxt);
               port.addEventListener(serialListener);
               ports.addElement(port);
            } catch (PortInUseException e) {
                  // the serial port has been occupied;
            } catch (TooManyListenersException e) {
                  // can't happen
            }
         }
      }
   }
   public void stop(BundleContext ctxt)
      throws Exception
   {
      for (int i=0; i<ports.size(); i++) {
            SerialPort port = (SerialPort) ports.elementAt(i);
            port.removeEventListener();
```

```
            port.close();
      }
   }
}
```

Figura 32: Método Ativador.

O *listener* de eventos da serial representa o papel principal na detecção dos dispositivos e é implementado da seguinte maneira, conforme a Figura 33.

```
import saito.sashimi.*;
import org.osgi.framework.*;
class SerialListener implements SerialPortEventListener {
   private BundleContext context;
   private ServiceRegistration serialReg = null;
   private SerialServiceImpl serialService = null;
   private String[] deviceClazzes = {
      "com.acme.service.device.serial.SerialService",
      "org.osgi.service.device.Device" };

   SerialListener(BundleContext ctxt) {
      this.context = ctxt;
   }

   public synchronized void serialEvent(SerialPortEvent e) {
      SerialPort port = (SerialPort) e.getSource();
      int t = e.getEventType();
      if (t == SerialPortEvent.CTS) {
            // listen to CTS pin of the serial interface
            boolean plugged = e.getNewValue();
            if (plugged) {
               // register a serial device service when a
               // device is connected to the serial port
               Properties props = new Properties();
               props.put("DEVICE_CATEGORY", "serial");
               props.put("DEVICE_MAKE", "Acme");
               props.put("Port", port.getName());
               serialService = new SerialServiceImpl(port);
               serialReg = context.registerService(deviceClazzes,
                  serialService, props);
            } else {
               // unregister the serial device service when
               // the device is disconnected from the serial
               // port
               if (serialReg != null) {
                  serialReg.unregister();
                  serialReg = null;
                  serialService = null;
            }
         }
      }
   }
```

Figura 33: *Listener* de Eventos.

O método `serialEvent` de `SerialListener` é chamado se um evento ocorre na porta serial. Se o pino CTS está em alto (`getNewValue` retorna verdadeiro), um dispositivo está conectado à porta, se não (`getNewValue` retorna falso), não há ou foi desconectado da porta.

Uma vez descoberto que o dispositivo está conectado na porta serial, é registrado uma instância de `SerialService` para representar a porta, juntamente com as propriedades do serviço `DEVICE_CATEGORY`, `DEVICE_MAKE`, e `Port`.

Após isso, foi adicionada a implementação de `DriverLocator` no serial bundle. O serviço `DriverLocator` sabe onde achar e baixar drivers refinando o `SerialService`. Essa implementação pode ser observada na Figura 34.

```
import saito.sashimi.*;
import osgi.service.device.*;
public class DriverLocatorImpl implements DriverLocator {

   // map a device category to IDs of refining drivers
   private Hashtable categoryMap;
   // map driver IDs to their URLs
   private Hashtable driverMap;
   static final String DRIVER_ID_PREFIX =
      "com.acme.device.drivers";

   public String[] findDrivers(Dictionary props) {
      String make = (String) props.get("DEVICE_MAKE");
      String category = (String) props.get("DEVICE_CATEGORY");
      if (!"Acme".equalsIgnoreCase(make))
      return null;
      String[] ids = (String[]) categoryMap.get(category);
      return ids;
   }

   public InputStream loadDriver(String id) throws IOExcep-
   tion {
      URL u = (URL) driverMap.get(id);
      if (u != null) {
            return u.openStream();
      }
      return null;
   }

   public DriverLocatorImpl() {
      categoryMap = new Hashtable(3);
      categoryMap.put("serial", new String[] {
         DRIVER_ID_PREFIX + ".serial.modem.0" });
      driverMap = new Hashtable(3);
      driverMap.put(DRIVER_ID_PREFIX + ".serial.modem.0",
        this.getClass().getResource("/drivers/driver.jar"));
   }
}
```

Figura 34: Driver Locator.

Muitos drivers podem estar disponíveis. Através do método `match` é selecionado o driver mais apropriado. Uma vez escolhido, através do método `attach` é registrado o serviço do dispositivo.

A implementação do driver service foi feita utilizando esses dois métodos.

```
import saito.sashimi.*;
import org.osgi.framework.*;

class DriverImpl implements Driver {
   private BundleContext context;
   private ServiceRegistration reg = null;

   public int match(ServiceReference sr) throws Exception {
      String category =
         (String) sr.getProperty("DEVICE_CATEGORY");
      if ("serial".equals(category))
            return SerialService.MATCH_OK;
      return Device.MATCH_NONE;
   }

   public String attach(ServiceReference sr) throws Exception {
      String[] deviceClazzes = new String[] {
            "org.osgi.service.device.Device",
            "com.acme.service.device.modem.ModemService" };
      Properties props = new Properties();
      props.put("DEVICE_CATEGORY", "modem");
      props.put("DEVICE_MAKE", "Acme");
      reg = context.registerService(deviceClazzes,
         new ModemServiceImpl(), props);
      return null;
   }

   DriverImpl(BundleContext ctxt) {
      this.context = ctxt;
   }
}
```

Figura 35: A implementação do driver service.

Após retornar o valor do driver, o gerenciador de dispositivo seleciona e chama o método `attach`. Feito isso, é registrado `ModemService` na categoria de dispositivos “modem” e está disponível para o `SerialService`.

```
import saito.sashimi.*;
import java.io.*;
import javax.comm.*;
import com.acme.service.device.modem.ModemService;
```

```
import com.acme.service.device.serial.SerialService;
import org.osgi.framework.*;

public class ModemServiceImpl implements ModemService {
   private SerialPort port;

   public void noDriverFound() {
   }

   public ModemService.DataConnection dialup(String phoneNumber)
      throws IOException
   {
      String rc = sendCommand("ATDT"+phoneNumber+"\r\n");
      if (rc.indexOf("CONNECT") != -1) {
         return this.new DataConnectionImpl();
      }
      return null;
   }

   public void hangup() throws IOException {
      sendCommand("ATH\r\n");
   }

   public String getInfo() throws IOException {
      String rc = sendCommand("ATI4\r\n");
      int pos = rc.indexOf("\r\n", 2);
      if (pos != -1)
         rc = rc.substring(2, pos);
      return rc;
   }
   public void configureSpeaker(int mode) throws IOException {
      sendCommand("ATM" + mode + "\r\n");
   }

   public byte getSRegister(int n) throws IOException {
      String rc = sendCommand("ATS" + n + "?\r\n");
      int pos = rc.indexOf("\r\n", 2);
      if (pos != -1)
         rc = rc.substring(2, pos);
      return Byte.parseByte(rc);
   }

   public void setSRegister(int n, byte value) throws IOException {
      String r = sendCommand("ATS" + n + "=" + value);
   }

   public ModemServiceImpl(SerialService ss)
      throws UnsupportedCommOperationException
   {
      this.port = ss.getPort();
      port.enableReceiveTimeout(5000);
   }

   private String sendCommand(String cmd) throws IOException {
      InputStream in = port.getInputStream();
      int byteRead;
      int off = 0;
      byte[] buf = new byte[1024];
      StringBuffer sb = new StringBuffer();
      while (true) {
         byteRead = in.read(buf);
```

```
         if (byteRead == 0)
            break;
         String s = new String(buf, 0, byteRead);
         sb.append(s);
         if (s.indexOf("\r\nOK\r\n") >= 0)
            break;
         else if (s.indexOf("\r\nERROR\r\n") >=0)
            throw new IOException(sb.toString());
      }
         return sb.toString();
   }

   class DataConnectionImpl
      implements ModemService.DataConnection
   {
      public OutputStream getOutputStream() throws IOException {
         return port.getOutputStream();
      }

      public InputStream getInputStream() throws IOException {
         return port.getInputStream();
      }

      public void returnToCommandMode() {
         try {
            getOutputStream().write("+++".getBytes());
         } catch (IOException e) {
         }
      }
   }
}
```

Figura 36: Modem service.

O ativador do driver bundle desempenha a atividade de registro no serviço de driver quando o bundle é iniciado, e exclui o registro quando o bundle é parado, conforme código na Figura 37.

```
import saito.sashimi.*;
import org.osgi.framework.*;

public class Activator implements BundleActivator {
   private ServiceRegistration reg;

   public void start(BundleContext ctxt) {
      Properties props = new Properties();
      props.put("DRIVER_ID",
         "com.acme.device.drivers.serial.modem.0");
      props.put("description", "The driver that refines " +
         "a serial device to a modem device.");
      reg = ctxt.registerService("org.osgi.service.device.Driver",
               new DriverImpl(ctxt), props);
   }
   public void stop(BundleContext ctxt) {
      if (reg != null)
            reg.unregister();
   }
}
```

Figura 37: Ativador do Bundle.

5.3.1 O Protótipo

A arquitetura do gateway é fundamental para o desenvolvimento de uma aplicação eficiente e para que os serviços sejam concluídos com sucesso. Deseja-se ter um gateway de processamento dedicado e customizado para as aplicações desenvolvidas. O gateway é constituído por um dispositivo contendo um microcontrolador FemtoJava, sintetizado a partir da ferramenta SASHIMI, e a ele estão adicionados dois módulos um para leitura dos dados (SENSOR) e outro para atuar no meio (ATUADOR). Sendo o FemtoJava customizável, o seu código é adaptado às necessidades da aplicação, sendo que somente o hardware necessário é sintetizado. A comunicação entre o gateway e os módulos, neste protótipo, é realizada através da porta serial com protocolo RS232.

O protótipo do gateway foi mapeado para uma FPGA da família Xilinx (SPARTAN AND SPARTAN-XL FAMILIES FIELD PROGRAMMABLE GATE ARRAYS: PRODUCT SPECIFICATION, 2007) com o auxílio da ferramenta de síntese ISE da própria Xilinx. A

versão utilizada foi da Spartan 3-AN. A família Spartan-3AN possui um custo menor e apresenta um número reduzido de elementos lógicos em relação à família Virtex.

Dois módulos foram propostos e desenvolvidos para interagir com o meio utilizando um microcontrolador UART conforme (SMITH, ROE, KNUDSEN, 2002). Estes módulos têm a função de enviar para o meio e receber os dados do ambiente. Isso permite ao gateway operar com esses módulos como objetos. Conforme os blocos na Figura 38.

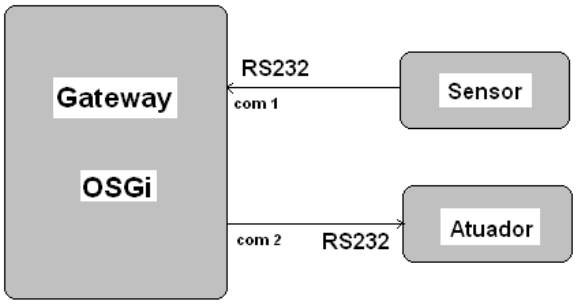

Figura 38: Blocos do protótipo.

Para acessar dispositivos físicos, aplicações escritas na linguagem de programação Java necessitam passar por algumas camadas. As aplicações podem se comunicar com o sistema operacional, com o qual pode se comunicar com o hardware através de drivers de dispositivos. Um exemplo típico deste tipo de biblioteca é o padrão Java Communications API definido no pacote javax.comm que permite que os programas acessem as portas serial e paralela.

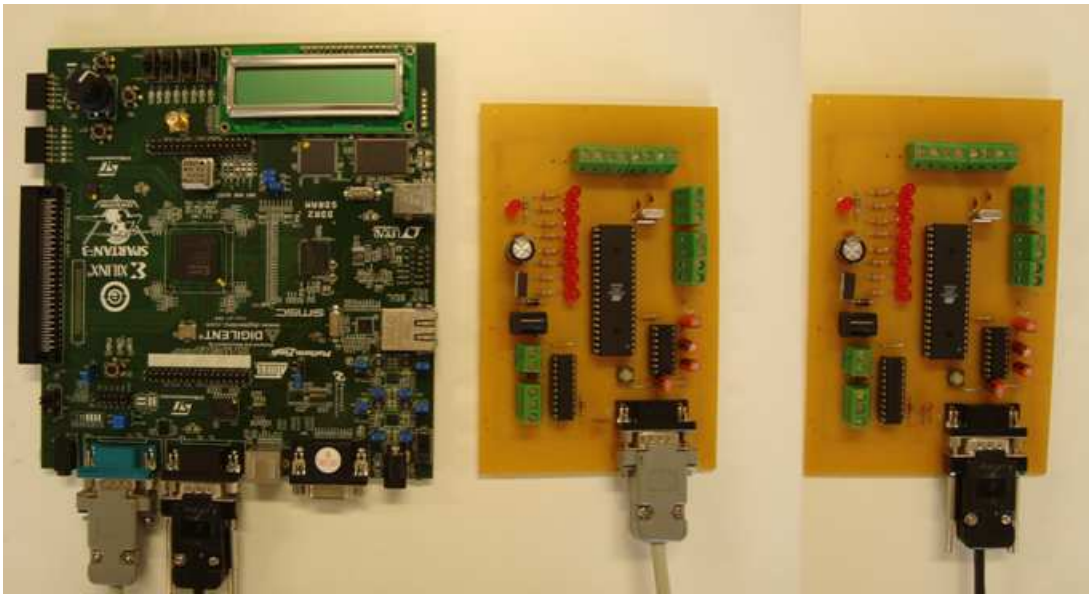

Figura 39: Protótipo usado.

5.4 ESTUDO DE CASO

Como estudo de caso para essa proposta foi desenvolvida uma aplicação para o controle de temperatura de uma residência.

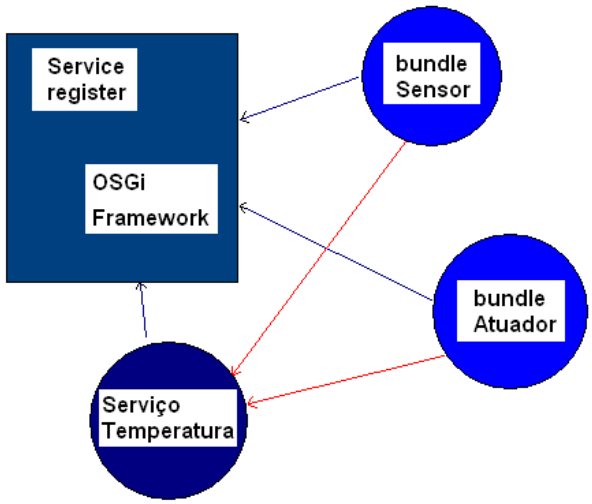

Figura 40: Estudo de caso para validação.

Para a criação do bundle do sensor, criou-se inicialmente um arquivo manifesto como o exposto na Figura 41.

```
Manifest-Version: 1.0
Bundle-Name: Sensorbundle
Bundle-SymbolicName: Sensorbundle
Bundle-Version: 1.0.0
Bundle-Description: Sensor Bundle
Bundle-Vendor: Guilherme Breier
Bundle-Activator: de.vpe.sensor.impl.Activator
Bundle-Category: example
Import-Package: org.osgi.framework
```

Figura 41: Manifesto do Sensor Bundle.

As mais importantes propriedades aqui são o Bundle-Activator e o Import-Package, pois o bundle-activator informa ao framework qual classe é sua ativadora. A propriedade Import-Package informa ao framework que o bundle necessita ter acesso a todas as classes contidas dentro do pacote org.osgi.framework. Como comentado em capítulos anteriores, cada bundle criado tem acesso as essas classes no Framework OSGi.

A maioria dos bundles tem uma classe ativadora, especificada no arquivo de manifesto do bundle. A classe ativadora implementa os dois métodos, start() e stop(), com as quais são usadas pelo framework para gerenciar o bundle. A implementação e o serviço são mostrados a seguir.

```
import saito.sashimi.*;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
	public static BundleContext bc = null;
	private SensorThread thread = null;
	public void start(BundleContext bc) throws Exception {

		Activator.bc = bc;
		this.thread = SensorThread();
		this.thread.start();
	}
	public void stop(BundleContext bc) throws Exception {

		this.thread.stopThread();
		this.thread.join();
		Activator.bc = null;
	}
}
```

Figura 42: Bundle Ativador do Sensor.

```
import saito.sashimi.*;

public class SensorThread extends Thread {
      private boolean running = true;
      public SensorThread() {
      }
      public void run() {
      portId1 = CommPortIdentifier.getPortIdentifier("COM1");
      while (running) {
        serialPort1.setSerialPortParams(9600,
                SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE);

         switch(event.getEventType()) {
        case SerialPortEvent.DATA_AVAILABLE:
             StringBuffer readBuffer = new StringBuffer();
             int c;
             try {
                  while ((c=inputStream.read()) != 10){
                    if(c!=13)  readBuffer.append((char) c);
                  }
                  String scannedInput = readBuffer.toString();
                  TimeStamp = new java.util.Date().toString();
                  inputStream.close();


      try {
      } catch (InterruptedException e) {
      }
      }
      }
      public void stopThread() {
      this.running = false;
      }
}
```

Figura 43: Thread do sensor.

Da mesma maneira se implementou o bundle para o atuador criando novamente um arquivo manifesto como:

```
Manifest-Version: 1.0
Bundle-Name: Atuadorbundle
Bundle-SymbolicName: Atuadorbundle
Bundle-Version: 1.0.0
Bundle-Description: Atuador Bundle
Bundle-Vendor: Guilherme Breier
Bundle-Activator: de.vpe.sensor.impl.Activator
Bundle-Category: example
Import-Package: org.osgi.framework
```

Figura 44: Manifesto do Atuador.

A classe ativadora do bundle Atuador também implementa os dois métodos, start() e stop(), os quais são usados pelo framework para gerenciar o bundle. A implementação e o serviço são mostrados a seguir.

```
import saito.sashimi.*;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
      public static BundleContext bc = null;
      private AtuadorThread thread = null;
      public void start(BundleContext bc) throws Exception {

            Activator.bc = bc;
            this.thread = AtuadorThread();
            this.thread.start();
      }
      public void stop(BundleContext bc) throws Exception {

            this.thread.stopThread();
            this.thread.join();
            Activator.bc = null;
      }
}
```

Figura 45: Bundle Ativador do Atuador.

```
import saito.sashimi.*;
public class AtuadorThread extends Thread {
      private boolean running = true;
      public AtuadorThread() {
      }
      public void run() {
      portId2 = CommPortIdentifier.getPortIdentifier("COM2");
      while (running) {
      serialPort1.setSerialPortParams(9600,
                SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE);
        switch(event.getEventType()) {
        case SerialPortEvent.DATA_AVAILABLE:
            outputStream = serialPort2.getOutputStream();
                  outputStream.write(divertCode.getBytes());
                  outputStream.close();

      try {
      } catch (InterruptedException e) {

      }
      }
      }
      public void stopThread() {
      this.running = false;
      }
}
```

Figura 46: Thread do Atuador.

O conjunto de classes dessa aplicação elaborada por um compilador Java padrão, foi submetido a uma ferramenta de pós-compilação, chamada SASHIMI. Esta ferramenta lê todas as classes da aplicação e gera o código final, que pode ser usado tanto para simulação quanto para síntese lógica. Este código final é um conjunto de arquivos que inclui a descrição VHDL do processador (cujo conjunto de instruções contém somente aquelas usadas pela aplicação) a as memórias ROM (métodos) e RAM (atributos). O SASHIMI ainda remove do código final, os métodos e atributos não alcançados, ou seja, aqueles que não são utilizados pela aplicação em nenhum momento da execução. Assim, o SASHIMI é também uma ferramenta de otimização de JVM.

No desenvolvimento de aplicações sobre a plataforma SASHIMI-FemtoJava os objetos da aplicação, estão alocados estaticamente em tempo de projeto. Em outras palavras, todos os objetos no sistema são construídos *a priori* (pelo SASHIMI), permitindo que seja determinada a quantidade total de memória RAM para comportar a aplicação.

Finalizando o processo de síntese da aplicação, o código orientado a objetos da aplicação é gerado no arquivo ROM.MIF. Então os arquivos VHDL gerados pelo SASHIMI servem como entrada para um ferramenta de síntese de FPGAs, que sintetiza o processador FemtoJava customizado para a aplicação específica.

A tabela 1 foi construída para oferecer uma referência de uso de memória quando as aplicações são construídas sobre a plataforma SASHIMI-FemtoJava. Foram coletados os dados de memória de código (ROM) e de atributos (RAM).

Tabela 1: Resultado da Implementação usando FemtoJava

Aplicação	***ROM***	***RAM***
OSGi-FemtoJava	4833 bytes	238 bytes
OSGi-FemtoJava + Bundles de comunicação	5047 bytes	302 bytes
OSGi-FemtoJava + Bundles + Aplicação de estudo	5102 bytes	317 bytes

A arquitetura resultante do FemtoOSGi pode ser então visualizada na figura 47, onde se pode verificar que aplicações para o framework OSGi podem ser desenvolvidas e que ambas podem serem executadas sobre o FemtoJava de forma customizadas.

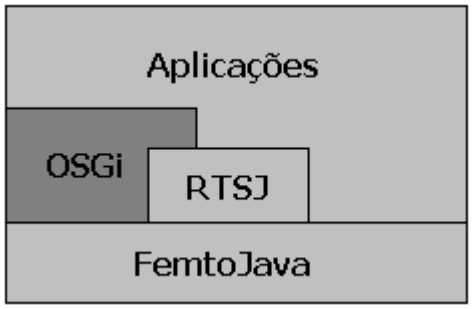

Figura 47: Arquitetura FemtoOSGi.

Avaliando o custo da implementação do estudo de caso gerou-se um teste apenas com a classe *main* no sashimi para verificar o custo do OSGi com seus bundles de serviços. O resultado desse teste pode ser visualizado na tabela 2.

Tabela 2: Femtojava

	ROM	*RAM*
FEMTOJAVA {main}	3bytes	1byte

O resultado observado se deve à otimização que o sashimi proporciona, pois as instruções em Java são transformadas em *opcodes* e o microprocessador femtojava executa apenas os *opcodes* necessários à aplicação.

Em nível de comparação, pode-se realizar outra análise referenciando o trabalho de (ALLGAYER, 2009). Nessa aplicação tem-se um sistema em uma rede de sensores e atuadores, o resultado dessa aplicação esta exposto na tabela 3.

Tabela 3: FEMTONODE

	ROM	*RAM*
FemtoNODE (módulo controlador)	5003bytes	282bytes

Comparativamente não está sendo agregado muito mais no custo computacional, pois a aplicação de estudo em (ALLGAYER, 2009) não oferece descoberta de serviços e carga dinâmica, mas podemos observar que a aplicação com o OSGi não causa um overhead muito grande comparado com a outra, entretanto agrega muito mais flexibilidade a uma aplicação.

6 CONCLUSÃO

Na abordagem orientada a serviços, os provedores oferecem aos clientes seus serviços. Uma descrição de serviços representa um contrato entre um provedor de serviços e um solicitante de serviço do serviço provido. Serviços são descobertos usando suas descrições de serviços em tempo de execução por solicitante de serviços consultando um registro de serviços, onde os provedores publicam as descrições dos serviços. Dois fatores são avaliados: Dinamismo e Capacidade de mudança. No dinamismo, provedores de serviços podem oferecer ou retirar serviços a qualquer hora. Capacidade de mudança é derivada do fato de que uma descrição de serviço representa um contrato. Como resultado, participação em uma aplicação orientada a serviços é aberta para qualquer serviço que deseje por um contrato requerido, e um serviço particular.

Com este tipo de arquitetura, um usuário pode compor aplicações com diferentes arquiteturas ou mesmo estilos diferentes de arquitetura. Esta nova abordagem permite que novos serviços sejam adicionados ao sistema sem parar a aplicação. Uma nova aplicação pode ser composta com serviços existentes mudando a especificação de workflow sem mudar o framework básico. Isto não é possível na arquitetura tradicional de software. A segunda característica da arquitetura orientada a serviços é que o ciclo de vida pode ser embarcado na infra-estrutura de operação para facilitar composição de software dinâmico. No projeto de software tradicional, a arquitetura não pode ser mudada depois de sua distribuição. O projetista tem que fazer a análise correta antes de construir a aplicação, de forma que qualquer problema ocorrido poderá ocasionar a inviabilidade da arquitetura. Isso ocorre porque a arquitetura tradicional consiste na construção de componentes acoplados, sendo de difícil desagregação.

A comunicação entre os serviços é controlada por um centro de controle. O centro de controle normalmente tem um gerenciador de composição via uma especificação de workflow. Esta especificação define como os serviços são conectados juntos para entregar a aplicação desejada e como as mensagens são transferidas entre serviços. Cada fabricante implementa seu dispositivo e a comunicação entre esses dispositivos em uma rede residencial seria gerenciada pelo framework. A arquitetura tradicional de software, por ser estática, consiste em projetar e codificar a aplicação.

Esta grande diferença entre as arquiteturas, a primeira por ser dinâmica e baseada em serviços, e a outra, estática e fortemente acoplada por componentes, torna a arquitetura orientada a serviços mais flexível para tratar a questão da interoperabilidade entre aplicações de vários fabricantes. Cada fabricante fornece o seu padrão, o seu protocolo, e é necessária uma arquitetura que saiba unificar diferentes implementações em uma única solução, completa e gerenciável.

Três fatores são fundamentais para destacar a arquitetura orientada a serviços como a arquitetura ideal para casas inteligentes:

• Composição dinâmica via descoberta - provê uma nova maneira de desenvolvimento de aplicação por usar serviços já descobertos. Embora a composição e a descoberta possam ser descarregadas em tempo de execução;

• Administração dinâmica - a execução de serviços precisa ser controlada e vários mecanismos estão disponíveis. Um é o serviço de governança por política. Políticas podem ser especificadas, verificadas, e reforçadas durante o tempo de desenvolvimento e tempo de execução;

• Sincronismo – o processo de execução será coordenado por um controlador central e é responsável por agendar a execução de serviços que podem ser distribuídos em torno da conectividade da rede.

OSGi dispõe de uma plataforma rica de gerenciamento de dispositivos e serviços encontrados em uma casa, o que é de fato, de grande importância para o desenvolvimento de serviços pelas grandes indústrias. Um ambiente interoperativo é fundamental para que novos agentes possam desenvolver suas soluções sem se preocupar com o fator integração, o que torna o mercado muito mais competitivo e aberto.

A orientação a serviços envolve algumas questões complexas, como segurança, descrição semântica e serviços dinâmicos. Todavia, muitas dessas questões foram resolvidas na platafoma OSGi. Um dos maiores desafios da plataforma OSGi é o seu gerenciamento de dependência dos serviços, causado pela disponibilidade dinâmica dos serviços.

Existem quatro fatores a serem considerados: Dinâmica, Ambigüidade, Arquitetura e Descoberta de recursos.

• Dinâmica - Um bundle é estabelecido pelo desenvolvedor da solicitação de serviços para constantemente monitorar e adaptar as mudanças na disponibilidade do serviço. Este tipo

de código é complexo e propício a erros desde que trate com questões de concorrência e requer que o desenvolvedor se preocupe com aplicações e lógica de adaptação simultanemante. Para OSGi, isto apresenta questões de tolerância a falhas e confiabilidade que não são comuns em aplicações de simples processos como as que rodam nas plataformas de serviço;

• Ambigüidade - É complexa e afeta todas as tecnologias orientadas a serviços. Surge quando existem múltiplos serviços candidatos disponíveis, que podem resolver uma dada dependência de serviços. É difícil saber qual dos serviços candidatos é o melhor a ser escolhido.

• Arquitetura - É relacionada à estrutura global de uma aplicação. OSGi e orientação a serviços não possuem uma visão arquitetural de aplicação global. A composição de serviços orientados a processos não constitui uma visão arquitetural. Por isso é difícil expressar relacionamentos estruturais para resolver dependência de serviços. Surge no desejo de eliminar algumas formas de ambigüidade, como o desejo de usar um serviço compartilhado para resolver múltiplas dependências de serviços.

• Descoberta de recursos – O OSGi somente define mecanismos para lidar com a descoberta de recursos depois dos bundles proverem os pacotes necessários e os serviços serem localmente instalados e ativados no framework. A especificação não endereça qualquer mecanismo padrão para descobrir e distribuir pacotes requeridos ou serviços de origem externa. Este mecanismo pode ser encontrado em ferramentas de terceiros ou em métodos ad hocs.

A aplicação desenvolvida e sintetizada para o FemtoJava se mostra inovadora pois todas as aplicações disponíveis no mercado são desenvolvidas e executadas em cima de computadores pessoais, o que torna muito mais fácil a implementação de serviços usando outros dispositivos e protocolos como por exemplo Ethernet, Bluetooth entre outros. Mas a possibilidade de uso de um protocolo serial wifi é um próximo passo para o desenvolvimento desse trabalho. O OSGi está voltado para aplicações atualmente muito mais multimídias e de comunicações, por isso a implementação de um gateway para atender essas necessidades usando o FemtoJava, não se mostra a mais apropriada. Para outras, que estão sendo desenvolvidas dentro do grupo de pesquisa do GCAR em descoberta de dispositivos e para nós, em redes sem fio, todo o trabalho desenvolvido se mostra bastante útil.

REFERÊNCIAS

ALLGAYER R. FemtoNode Arquitetura de Nó-Sensor Reconfigurável e Customizável para Rede de Sensores sem Fio, 2009. 89p. Dissertação (Mestrado em Engenharia) — Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Porto Alegre.

ARK, W. S.; SELKER, T. A Look at Human Interaction with Pervasive Computers. In: IBM Systems Journal, v. 38, n. 9, p. 504 – 507, 1999.

ARTS, E. Ambient intelligence: a multimedia perspective. In: Multimedia, IEEE. Vol. 11. Issue 1, Jan-Mar, pp. 12 – 19, 2004.

BREIER, G. B. Descoberta de Serviços usando o framework OSGi para o desenvolvimento de um gateway residencial. 2008, 84p Trabalho Individual (Mestrado em Engenharia Elétrica) – Escola de Engenharia, UFRGS, Porto Alegre.

CABRER, M. R. et al. Controlling the Smart Home form TV. In: INTERNATIONAL CONFERENCE ON CONSUMER ELECTRONICS, 2006, Las Vegas: Proceedings… NewYork: IEEE, p. 421 - 429, 2006.

COMPTON, K; HAUCK, S. Reconfigurable Computing: a survey of systems and software. ACM Computing Surveys, New York, NY, USA, v.34 n.2, p171-210, 2002.

DUCATEL, K. et al. Scenarios for Ambient Intelligence (ISTAG Report), Institute for Prospective Technological Studies (European Commission), Seville, 2001.

EDWARDS, W. K. Discovery systems in ubiquitous computing. In: Pervasive Computing, IEEE. Vol. 5, Issue 2, April-June, pp. 70-77, 2006.

GARCIA, P.; COMPTON, K.; SCHULTE, M.; BLEM, E.; FU, W. An overview of reconfigurable hardware in embedded systems. EURASIP J. Embedded Syst., New York, NY, United States, v.2006, n.1, p.13–13, 2006.

GONG, L. A software architecture for Open Service Gateways. In Embedded Systems IEEE. Vol.2, Issue 1, January-February, 2001.

GÖTZ, M. Run-time Reconfigurable RTOS for Reconfigurable Systems-on-Chip. 2007. 135p. Tese (Doutorado em engenharia) — Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Porto Alegre.

HALL, R.; CERVANTES, H. An OSGi implementation and experience report. Consumer Communications and Networking Conference, 2004. CCNC 2004. First IEEE, [S.l.], p.394–399, Jan. 2004.

HALL, R.; CERVANTES, H. Challenges in building service-oriented applications for OSGi. Communications Magazine, IEEE, [S.l.], v.42, n.5, p.144–149, May 2004.

HELAL, A.; HAMMER, J. UbiData: Requirements and Architecture for Ubiquitous Data Access. In: SIGMOD Rec Journal… New York: ACM, p. 71 – 76, 2004.

HELAL, S. et al.The Gator Tech Smart House: A Programmable Pervasive Space. In: IEEE Pervasive Computing, p 64 - 74, 2005.

HSU, J. M.; WU, W. J.; CHANG, I. R. Ubiquitous Multimedia Information Delivering Service for Smart Home. In: INTERNATIONAL CONFERENCE ON MULTIMEDIA AND UBIQUITOUS ENGINEERING, 2007, Seoul: Proceedings… New York: IEEE Computer Society, p. 341 – 346, 2007.

HOFRICHTER, K. The residential gateway as service platform. Consumer Electronics, 2001. ICCE. International Conference on, [S.l.], p.304–305, 2001.

HWANG, T.; PARK, H.; CHUNG, J. W. Design and implementation of the home service delivery and management system based on OSGi service platform. Consumer Electronics, 2006. ICCE ’06. 2006 Digest of Technical Papers. International Conference on, [S.l.], p.189–190, Jan. 2006.

ITO, S. A.; CARRO, L.; JACOBI, R. P. Making Java Work for Microcontroller Applications. IEEE Des. Test, Los Alamitos, CA, USA, v.18, n.5, p.100–110, 2001.

LI, X.; ZHANG, W. The design and implementation of home network system using OSGi compliant middleware. Consumer Electronics, IEEE Transactions on, [S.l.], v.50, n.2, p.528–534, May 2004.

LIANG, L.;WANG, A. Bundle dependency in open service gateway initiative framework initialization. Networked Appliances, 2002. Liverpool. Proceedings. 2002 IEEE 5th International Workshop on, [S.l.], p.122–126, Oct. 2002.

MARPLES, D.; KRIENS, P. The Open Services Gateway Initiative: an introductory overview. Communications Magazine, IEEE, [S.l.], v.39, n.12, p.110–114, Dec 2001.

NAKAMURA, M.; IGAKI, H.; TAMADA, H.; MATSUMOTO, K. ichi. Implementing integrated services of networked home appliances using service oriented architecture. In: ICSOC '04: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON SERVICE ORIENTED COMPUTING, 2004, New York, NY, USA. Anais. . . ACM, 2004. p.269–278.

OSGI. http://www.osgi.org. <http://www.osgi.org> [Online; Accessed 10-Mar-2008].

REDONDO, R. P. D. *et al*. Enhancing Residential Gateways: OSGi Services Composition. In: INTERNATIONAL CONFERENCE ON CONSUMER ELECTRONICS, 2007, Las Vegas: Proceedings… New York: IEEE, p. 1 - 2, 2007.

SMITH, L; ROE, C; KNUDSEN, K. S. A jini lookup service for resource-constrained device. Networked appliances, Gaithersburg Proceedings 2002. IEEE 4th international workshop 2002. Pages 135-144.

SPARTAN and Spartan-XL Families Field Programmable Gate Arrays: product specification. [S.l.]: XILINX, 2007. 82p. Disponível em: <http://direct.xilinx.com/ bvdocs/publications/ds060.pdf>. Acesso em: 18 jul. 2008.

SUN. http://www.sun.com. <http://www.sun.com> [Online; accessed 15-jun-2008].

TAN, Y.; YAU, C.; LO, K.; YU,W.; MOK, P.; FONG, A. Design and implementation of a Java processor. Computers and Digital Techniques, IEEE Proceedings -, [S.l.], v.153,n.1, p.20–30, Jan. 2006.

TSAI, W. T.; FAN, C.; CHEN, Y.; PAUL, R.; CHUNG, J.-Y. Architecture Classification for SOA-Based Applications. In: ISORC '06: PROCEEDINGS OF THE NINTH IEEE INTERNATIONAL SYMPOSIUM ON OBJECT AND COMPONENTORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2006, Washington, DC, USA. Anais. . . IEEE Computer Society, 2006. p.295–302.

TODMAN, T.; CONSTANTINIDES, G.; WILTON, S.; MENCER, O.; LUK, W.; CHEUNG, P. Reconfigurable computing: architectures and design methods. Computers and Digital Techniques, IEEE Proceedings -, [S.l.], v.152, n.2, p.193–207, Mar 2005.

UFRGS. SASHIMI - Manual do Usuário. [S.l.]: Universidade Federal do Rio Grande do Sul (UFRGS), 2006.

VALTCHEV, D; FRANKOV, L. Service gateway architecture for a smart home. In: Communications Magazine, IEEE, issue 4, v.40, p.126–132, April 2002.

XU, W.; XIN, Y.; LU, G. A System Architecture for Pervasive Computing. In: INTERNACIONAL CONFERENCE ON NATURAL COMPUTATION, 2007, Haikou. Proceedings… New York: IEEE Computer Society, p. 772 - 776, 2007.

WEHRMEISTER, M. A.; PEREIRA, C. E.; BECKER, L. B. Optimizing the generation of object-oriented real-time embedded applications based on the real-time specification for Java. In: DATE ’06: PROCEEDINGS OF THE CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2006, 3001 Leuven, Belgium, Belgium. Anais. . . European Design and Automation Association, 2006. p.806–811.

ZHANG, H.; WANG, F.-Y.; AI, Y. An OSGi and agent based control system architecture for smart home. Networking, Sensing and Control, 2005. Proceedings. 2005 IEEE, [S.l.], p.13–18, March 2005.

ANEXO A – PRINCIPAIS APIS OSGI

ORG.OSGI.FRAMEWORK

INTERFACE BUNDLE

---

public interface **Bundle**

An installed bundle in the Framework.

A `Bundle` object is the access point to define the lifecycle of an installed bundle. Each bundle installed in the OSGi environment must have an associated `Bundle` object.

A bundle must have a unique identity, a `long`, chosen by the Framework. This identity must not change during the lifecycle of a bundle, even when the bundle is updated. Uninstalling and then reinstalling the bundle must create a new unique identity.

A bundle can be in one of six states:

- `UNINSTALLED`
- `INSTALLED`
- `RESOLVED`
- `STARTING`
- `STOPPING`
- `ACTIVE`

Values assigned to these states have no specified ordering; they represent bit values that may be ORed together to determine if a bundle is in one of the valid states.

A bundle should only execute code when its state is one of `STARTING`,`ACTIVE`, or `STOPPING`. An `UNINSTALLED` bundle can not be set to another state; it is a zombie and can only be reached because references are kept somewhere.

The Framework is the only entity that is allowed to create `Bundle` objects, and these objects are only valid within the Framework that created them.

ThreadSafe

Field Summary

`static int`	**`ACTIVE`** The bundle is now running.
`static int`	**`INSTALLED`**

	The bundle is installed but not yet resolved.
static int	**RESOLVED** The bundle is resolved and is able to be started.
static int	**START_ACTIVATION_POLICY** The bundle start operation must activate the bundle according to the bundle's declared activation policy.
static int	**START_TRANSIENT** The bundle start operation is transient and the persistent autostart setting of the bundle is not modified.
static int	**STARTING** The bundle is in the process of starting.
static int	**STOP_TRANSIENT** The bundle stop is transient and the persistent autostart setting of the bundle is not modified.
static int	**STOPPING** The bundle is in the process of stopping.
static int	**UNINSTALLED** The bundle is uninstalled and may not be used.

Method Summary

java.util.Enumeration	**findEntries**(java.lang.String path, java.lang.String filePattern, boolean recurse) Returns entries in this bundle and its attached fragments.
BundleContext	**getBundleContext**() Returns this bundle's BundleContext.
long	**getBundleId**() Returns this bundle's unique identifier.
java.net.URL	**getEntry**(java.lang.String path) Returns a URL to the entry at the specified path in this bundle.
java.util.Enumeration	**getEntryPaths**(java.lang.String path) Returns an Enumeration of all the paths (String objects) to entries within this bundle whose longest sub-path matches the specified path.
java.util.Dictionary	**getHeaders**() Returns this bundle's Manifest headers and values.
java.util.Dictionary	**getHeaders**(java.lang.String locale) Returns this bundle's Manifest headers and values localized to the specified locale.
long	**getLastModified**() Returns the time when this bundle was last modified.
java.lang.String	**getLocation**() Returns this bundle's location identifier.
ServiceReference[]	**getRegisteredServices**() Returns this bundle's ServiceReference list for all services it has registered or null if this bundle has no registered services.
java.net.URL	**getResource**(java.lang.String name) Find the specified resource from this bundle.
java.util.Enumeration	**getResources**(java.lang.String name)

	Find the specified resources from this bundle.
ServiceReference[]	**getServicesInUse**() Returns this bundle's ServiceReference list for all services it is using or returns null if this bundle is not using any services.
int	**getState**() Returns this bundle's current state.
java.lang.String	**getSymbolicName**() Returns the symbolic name of this bundle as specified by its Bundle-SymbolicName manifest header.
boolean	**hasPermission**(java.lang.Object permission) Determines if this bundle has the specified permissions.
java.lang.Class	**loadClass**(java.lang.String name) Loads the specified class using this bundle's classloader.
void	**start**() Starts this bundle with no options.
void	**start**(int options) Starts this bundle.
void	**stop**() Stops this bundle with no options.
void	**stop**(int options) Stops this bundle.
void	**uninstall**() Uninstalls this bundle.
void	**update**() Updates this bundle.
void	**update**(java.io.InputStream in) Updates this bundle from an InputStream.

ORG.OSGI.FRAMEWORK

INTERFACE BUNDLEACTIVATOR

All Known Implementing Classes:
XMLParserActivator

---

public interface **BundleActivator**

Customizes the starting and stopping of a bundle.

BundleActivator is an interface that may be implemented when a bundle is started or stopped. The Framework can create instances of a bundle's BundleActivator as required. If an instance's BundleActivator.start method executes successfully, it is guaranteed that the same instance's BundleActivator.stop method will be called when the bundle is to be stopped. The Framework must not concurrently call a BundleActivator object.

BundleActivator is specified through the Bundle-Activator Manifest header. A bundle can only specify a single BundleActivator in the Manifest file. Fragment bundles must not have a BundleActivator. The form of the Manifest header is:

`Bundle-Activator: class-name`

where *`class-name`* is a fully qualified Java classname.

The specified `BundleActivator` class must have a public constructor that takes no parameters so that a `BundleActivator` object can be created by `Class.newInstance()`.

NotThreadSafe

Method Summary

`void`	**`start`**`(BundleContext context)` Called when this bundle is started so the Framework can perform the bundle-specific activities necessary to start this bundle.
`void`	**`stop`**`(BundleContext context)` Called when this bundle is stopped so the Framework can perform the bundle-specific activities necessary to stop the bundle.

ORG.OSGI.FRAMEWORK

INTERFACE BUNDLECONTEXT

---

public interface **BundleContext**

A bundle's execution context within the Framework. The context is used to grant access to other methods so that this bundle can interact with the Framework.

`BundleContext` methods allow a bundle to:

- Subscribe to events published by the Framework.
- Register service objects with the Framework service registry.
- Retrieve `ServiceReferences` from the Framework service registry.
- Get and release service objects for a referenced service.
- Install new bundles in the Framework.
- Get the list of bundles installed in the Framework.
- Get the `Bundle` object for a bundle.
- Create `File` objects for files in a persistent storage area provided for the bundle by the Framework.

A `BundleContext` object will be created and provided to the bundle associated with this context when it is started using the `BundleActivator.start(org.osgi.framework.BundleContext)` method. The same `BundleContext` object will be passed to the bundle associated with this context when it is stopped using the `BundleActivator.stop(org.osgi.framework.BundleContext)` method.

A BundleContext object is generally for the private use of its associated bundle and is not meant to be shared with other bundles in the OSGi environment.

The Bundle object associated with a BundleContext object is called the *context bundle*.

The BundleContext object is only valid during the execution of its context bundle; that is, during the period from when the context bundle is in the STARTING, STOPPING, and ACTIVE bundle states. If the BundleContext object is used subsequently, an IllegalStateException must be thrown. The BundleContext object must never be reused after its context bundle is stopped.

The Framework is the only entity that can create BundleContext objects and they are only valid within the Framework that created them.

ThreadSafe

Method Summary

void	**addBundleListener**(BundleListener listener) Adds the specified BundleListener object to the context bundle's list of listeners if not already present.
void	**addFrameworkListener**(FrameworkListener listener) Adds the specified FrameworkListener object to the context bundle's list of listeners if not already present.
void	**addServiceListener**(ServiceListener listener) Adds the specified ServiceListener object to the context bundle's list of listeners.
void	**addServiceListener**(ServiceListener listener, java.lang.String filter) Adds the specified ServiceListener object with the specified filter to the context bundle's list of listeners.
Filter	**createFilter**(java.lang.String filter) Creates a Filter object.
ServiceReference[]	**getAllServiceReferences**(java.lang.String clazz, java.lang.String filter) Returns an array of ServiceReference objects.
Bundle	**getBundle**() Returns the Bundle object associated with this BundleContext.
Bundle	**getBundle**(long id) Returns the bundle with the specified identifier.
Bundle[]	**getBundles**() Returns a list of all installed bundles.
java.io.File	**getDataFile**(java.lang.String filename) Creates a File object for a file in the persistent storage area provided for the bundle by the Framework.
java.lang.String	**getProperty**(java.lang.String key) Returns the value of the specified property.
java.lang.Object	**getService**(ServiceReference reference) Returns the specified service object for a service.
ServiceReference	**getServiceReference**(java.lang.String clazz) Returns a ServiceReference object for a service that implements and

	was registered under the specified class.
ServiceReference[]	**getServiceReferences**(java.lang.String clazz, java.lang.String filter) Returns an array of ServiceReference objects.
Bundle	**installBundle**(java.lang.String location) Installs a bundle from the specified location string.
Bundle	**installBundle**(java.lang.String location, java.io.InputStream input) Installs a bundle from the specified InputStream object.
ServiceRegistration	**registerService**(java.lang.String[] clazzes, java.lang.Object service, java.util.Dictionary properties) Registers the specified service object with the specified properties under the specified class names into the Framework.
ServiceRegistration	**registerService**(java.lang.String clazz, java.lang.Object service, java.util.Dictionary properties) Registers the specified service object with the specified properties under the specified class name with the Framework.
void	**removeBundleListener**(BundleListener listener) Removes the specified BundleListener object from the context bundle's list of listeners.
void	**removeFrameworkListener**(FrameworkListener listener) Removes the specified FrameworkListener object from the context bundle's list of listeners.
void	**removeServiceListener**(ServiceListener listener) Removes the specified ServiceListener object from the context bundle's list of listeners.
boolean	**ungetService**(ServiceReference reference) Releases the service object referenced by the specified ServiceReference object.

ORG.OSGI.FRAMEWORK

INTERFACE BUNDLELISTENER

All Superinterfaces:
java.util.EventListener
All Known Subinterfaces:
SynchronousBundleListener

---

public interface **BundleListener**
extends java.util.EventListener

A BundleEvent listener. BundleListener is a listener interface that may be implemented by a bundle developer. When a BundleEvent is fired, it is asynchronously delivered to a BundleListener. The Framework delivers BundleEvent objects to a BundleListener in order and must not concurrently call a BundleListener.

A `BundleListener` object is registered with the Framework using the `BundleContext.addBundleListener(org.osgi.framework.BundleListener)` method. `BundleListeners` are called with a `BundleEvent` object when a bundle has been installed, resolved, started, stopped, updated, unresolved, or uninstalled.

See Also:
`BundleEvent`
NotThreadSafe

Method Summary

`void`	**`bundleChanged`**`(BundleEvent event)` Receives notification that a bundle has had a lifecycle change.

ORG.OSGI.FRAMEWORK

INTERFACE FRAMEWORKLISTENER

All Superinterfaces:
java.util.EventListener

---

public interface **FrameworkListener**
extends java.util.EventListener

A `FrameworkEvent` listener. `FrameworkListener` is a listener interface that may be implemented by a bundle developer. When a `FrameworkEvent` is fired, it is asynchronously delivered to a `FrameworkListener`. The Framework delivers `FrameworkEvent` objects to a `FrameworkListener` in order and must not concurrently call a `FrameworkListener`.

A `FrameworkListener` object is registered with the Framework using the `BundleContext.addFrameworkListener(org.osgi.framework.FrameworkListener)` method. `FrameworkListener` objects are called with a `FrameworkEvent` objects when the Framework starts and when asynchronous errors occur.

See Also:
`FrameworkEvent`
NotThreadSafe

Method Summary

`void`	**`frameworkEvent`**`(FrameworkEvent event)` Receives notification of a general `FrameworkEvent` object.

ORG.OSGI.FRAMEWORK

INTERFACE SERVICELISTENER

All Superinterfaces:
java.util.EventListener
All Known Subinterfaces:
AllServiceListener

---

public interface **ServiceListener**
extends java.util.EventListener

A `ServiceEvent` listener. `ServiceListener` is a listener interface that may be implemented by a bundle developer. When a `ServiceEvent` is fired, it is synchronously delivered to a `ServiceListener`. The Framework may deliver `ServiceEvent` objects to a `ServiceListener` out of order and may concurrently call and/or reenter a `ServiceListener`.

A `ServiceListener` object is registered with the Framework using the `BundleContext.addServiceListener` method. `ServiceListener` objects are called with a `ServiceEvent` object when a service is registered, modified, or is in the process of unregistering.

`ServiceEvent` object delivery to `ServiceListener` objects is filtered by the filter specified when the listener was registered. If the Java Runtime Environment supports permissions, then additional filtering is done. `ServiceEvent` objects are only delivered to the listener if the bundle which defines the listener object's class has the appropriate `ServicePermission` to get the service using at least one of the named classes under which the service was registered.

`ServiceEvent` object delivery to `ServiceListener` objects is further filtered according to package sources as defined in `ServiceReference.isAssignableTo(Bundle, String)`.

See Also:
`ServiceEvent`, `ServicePermission`
ThreadSafe

Method Summary

`void`	**`serviceChanged`**`(ServiceEvent event)` Receives notification that a service has had a lifecycle change.

ORG.OSGI.FRAMEWORK

INTERFACE SERVICEREFERENCE

All Superinterfaces:
java.lang.Comparable

---

public interface **ServiceReference**

extends java.lang.Comparable

A reference to a service.

The Framework returns `ServiceReference` objects from the `BundleContext.getServiceReference` and `BundleContext.getServiceReferences` methods.

A `ServiceReference` object may be shared between bundles and can be used to examine the properties of the service and to get the service object.

Every service registered in the Framework has a unique `ServiceRegistration` object and may have multiple, distinct `ServiceReference` objects referring to it. `ServiceReference` objects associated with a `ServiceRegistration` object have the same `hashCode` and are considered equal (more specifically, their `equals()` method will return `true` when compared).

If the same service object is registered multiple times, `ServiceReference` objects associated with different `ServiceRegistration` objects are not equal.

See Also:

`BundleContext.getServiceReference(java.lang.String)`, `BundleContext.getServiceReferences(java.lang.String, java.lang.String)`, `BundleContext.getService(org.osgi.framework.ServiceReference)`

ThreadSafe

Method Summary

int	**`compareTo`**`(java.lang.Object reference)` Compares this `ServiceReference` with the specified `ServiceReference` for order.
Bundle	**`getBundle`**`()` Returns the bundle that registered the service referenced by this `ServiceReference` object.
java.lang.Object	**`getProperty`**`(java.lang.String key)` Returns the property value to which the specified property key is mapped in the properties `Dictionary` object of the service referenced by this `ServiceReference` object.
java.lang.String[]	**`getPropertyKeys`**`()` Returns an array of the keys in the properties `Dictionary` object of the service referenced by this `ServiceReference` object.
Bundle[]	**`getUsingBundles`**`()` Returns the bundles that are using the service referenced by this `ServiceReference` object.
boolean	**`isAssignableTo`**`(Bundle bundle, java.lang.String className)` Tests if the bundle that registered the service referenced by this `ServiceReference` and the specified bundle use the same source for the package of the specified class name.

ORG.OSGI.FRAMEWORK

INTERFACE SERVICEREGISTRATION

---

public interface **ServiceRegistration**

A registered service.

The Framework returns a `ServiceRegistration` object when a `BundleContext.registerService` method invocation is successful. The `ServiceRegistration` object is for the private use of the registering bundle and should not be shared with other bundles.

The `ServiceRegistration` object may be used to update the properties of the service or to unregister the service.

See Also:
`BundleContext.registerService(String[],Object,Dictionary)`
ThreadSafe

Method Summary

`ServiceReference`	**`getReference`**`()` Returns a `ServiceReference` object for a service being registered.
`void`	**`setProperties`**`(java.util.Dictionary properties)` Updates the properties associated with a service.
`void`	**`unregister`**`()` Unregisters a service.

ORG.OSGI.SERVICE.DEVICE

INTERFACE DEVICE

---

public interface **Device**

Interface for identifying device services.

A service must implement this interface or use the `Constants.DEVICE_CATEGORY` registration property to indicate that it is a device. Any services implementing this interface or registered with the `DEVICE_CATEGORY` property will be discovered by the device manager.

Device services implementing this interface give the device manager the opportunity to indicate to the device that no drivers were found that could (further) refine it. In this case, the device manager calls the `noDriverFound()` method on the `Device` object.

Specialized device implementations will extend this interface by adding methods appropriate to their device category to it.

See Also:
`Driver`

Field Summary

`static int`	**`MATCH_NONE`** Return value from `Driver.match(org.osgi.framework.ServiceReference)` indicating that the driver cannot refine the device presented to it by the device manager.

Method Summary

`void`	**`noDriverFound`**`()` Indicates to this `Device` object that the device manager has failed to attach any drivers to it.

Field Detail

MATCH_NONE

`public static final int` **`MATCH_NONE`**

Return value from `Driver.match(org.osgi.framework.ServiceReference)` indicating that the driver cannot refine the device presented to it by the device manager. The value is zero.

See Also:
Constant Field Values

Method Detail

noDriverFound

`public void` **`noDriverFound`**`()`

Indicates to this `Device` object that the device manager has failed to attach any drivers to it.

If this `Device` object can be configured differently, the driver that registered this `Device` object may unregister it and register a different Device service instead.