2
Fundamentos

Este trabalho apresenta uma extensão para o *middleware* Kaluana original [12], que define um modelo de componentes orientado a serviços, permitindo a implementação de aplicações móveis dinamicamente adaptáveis, bem como a implantação dinâmica de componentes.

As adaptações e as implantações dinâmicas são motivadas por alterações no contexto computacional [14], com o objetivo de estender, corrigir ou otimizar o funcionamento de uma aplicação. Neste capítulo serão contextualizados os conceitos necessários para a compreensão deste trabalho.

2.1.
Componentes

Um componente de software é uma unidade binária de composição que contém interfaces contratualmente definidas e dependências explícitas. Um componente pode ser implantado separadamente e está sujeito a composições a outros componentes [15].

Para tornar possível a interação entre componentes, cada um deve prover, através de interfaces públicas, descrições de seus pontos de conexão com outros componentes (facetas ou receptáculos).

A Figura 1 mostra a representação utilizada neste trabalho da estrutura de dois componentes, provendo facetas e receptáculos. Na figura, um receptáculo do componente *C1* está conectado a uma faceta do componente *C2*. Nesta figura, pode-se notar que o componente *C1* possui duas facetas e um receptáculo, enquanto o componente *C2* possui apenas uma faceta e dois receptáculos.

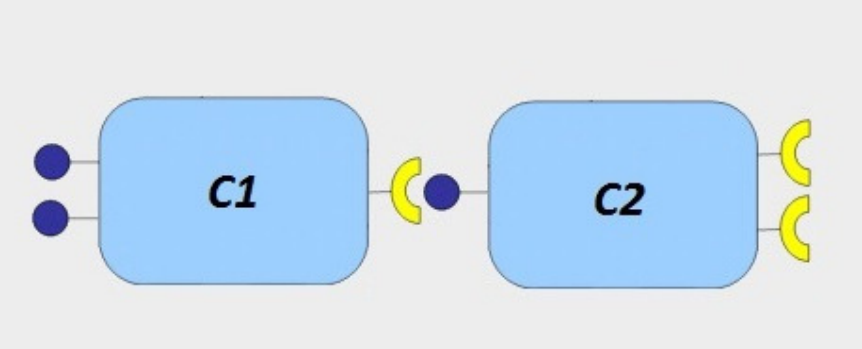

Figura 1. Representação da composição de dois componentes

A abordagem de orientação a componentes se caracteriza por uma clara separação entre o desenvolvimento de componentes e o processo de desenvolvimento de aplicações, no qual se faz somente a composição de componentes existentes. Desta maneira, o desenvolvedor de uma aplicação não precisa conhecer detalhes da implementação de um componente para usá-lo.

Existem inúmeros exemplos de sistemas de componentes, como COM [16] e Fractal [17], Corba Component Model [18], OpenCOM [19] e OSGi [20]. Em [21], há uma análise detalhada de características chaves de cada sistema de componentes, realizada a partir de uma taxonomia específica.

2.2. Ciência ao contexto

Um desafio da computação distribuída é explorar as mudanças do ambiente com novas classes de aplicações que são cientes ao contexto no qual elas executam. Muitos pesquisadores têm tentado definir o conceito de *contexto* através da enumeração de exemplos de contexto. Em [14], o conceito é dividido em três categorias:

- Contexto computacional, como conectividade de rede, custo de comunicação e largura de banda, além da proximidade de recursos como impressoras, monitores e *workstations*.
- Contexto do usuário, como o perfil do usuário, localização, proximidade a outras pessoas ou até mesmo a situação social atual.
- Contexto físico, como luminosidade, níveis de ruído, condições de tráfico e temperatura.

O tempo também pode ser considerado um contexto natural para muitas aplicações.

Alguns outros pesquisadores tentam definir o conceito de *contexto* de maneira formal. Em [22], o conceito é definido como o conhecimento sobre o usuário e o estado do dispositivo, incluindo os arredores, a situação e, de maneira menos extensa, a localização. Segundo [23], o conceito de *contexto* é representado por qualquer informação que pode ser usada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, lugar ou objeto que é considerado relevante para a interação entre um usuário e uma aplicação, incluindo o usuário e a própria aplicação.

Os sistemas cientes ao contexto se adaptam de acordo com as informações a respeito do dispositivo, além de aspectos relacionados com o usuário e com o ambiente em que ele se encontra. Um sistema com estas capacidades pode examinar o ambiente computacional e reagir a suas mudanças.

Segundo [24], são dadas duas definições de computação ciente ao contexto:

- Ciência ao contexto ativa, onde uma aplicação se adapta automaticamente de acordo com o a mudança do contexto, alterando o comportamento da aplicação.
- Ciência ao contexto passiva, onde uma aplicação apresenta o contexto novo ou atualizado, para um usuário interessado, ou persiste para que este usuário recupere esta informação depois.

2.3. Descrição de interfaces

Um desenvolvedor pode integrar componentes em um sistema em diversos níveis. No nível mais simples, cada um lida com um único problema, ignorando outros componentes. Em um nível mais alto, os componentes compartilham algumas suposições através de informações compartilhadas para que possam funcionar em conjunto. Estes sistemas de componentes possuem quatro características principais:

1. Os sistemas são grandes
2. Eles se comunicam via estruturas de dados complexas

3. Eles podem ser escritos em linguagens de programação diferentes e podem executar em *hardwares* diferentes
4. A representação interna dos dados deve ser amarrada ao sistema através de algum tipo de linguagem de descrição.

Existem diversos tipos de descrever interfaces de componentes de maneira estruturadas. A IDL[25] provê uma solução para o problema de interconectar componentes de sistemas grandes. A solução possui três partes:

- Uma notação para descrever sistemas e as estruturas de dados através das quais eles se comunicam
- Uma organização interna dos componentes do programa destes sistemas para permitir que eles se comuniquem
- Um tradutor que analisa a descrição de um sistema descrito na notação de IDL e gera fragmentos de código (stubs) que são ligados aos códigos dos componentes.

Se dois programas desejam se comunicar, eles devem compartilhar a mesma descrição abstrata de seus tipos. Caso suas representações difiram, eles necessitam comunicar-se via algum tipo de representação compartilhada que liga as duas representações internas. Para permitir a máxima portabilidade de estrutura de dados, a IDL define um esquema para representar a estrutura de um componente.

O esquema de uma estrutura pode ser derivado diretamente da declaração desta estrutura através de um compilador. A representação externa de uma estrutura consiste na referencia para o nó raiz, seguida da sequência de nós rotulados. A representação de um nó consiste do nome do tipo do nó, seguido pela lista de pares atributos-valor. Qualquer valor pode ser rotulado, no entanto todos os rótulos devem ser únicos.

Outra forma bastante comum de descrever interfaces de componentes é através da *Extensible Markup Language* [26], abreviado como XML. Este tipo de linguagem descreve uma classe de objetos de dados chamados *XML Documents* e, parcialmente descreve o comportamento de um programa de computador que seja capaz de ler este tipo de documento. Documentos XML são feitos de unidades de armazenamento, chamadas entidades, as quais contêm tanto dados analisados como não analisados. Dados analisados são feitos de caracteres, alguns originados de *character data*, e alguns de *markups*. *Markup* codifica a descrição do de uma

estrutura lógica. O XML provê um mecanismo para impor restrições no *layout* de armazenamento e na estrutura lógica.

2.4. Compatibilidade de tipos

O eventual sucesso da indústria de softwares orientados a objetos e baseados em componentes depende da capacidade de combinar e reutilizar partes de um sistema (e.e. componentes) implementados independentemente por diferentes desenvolvedores. Assim, a noção de compatibilidade de componentes é uma preocupação fundamental, pois:

- O cliente (usuário do componente) deve fazer certas suposições sobre a maneira como um componente se comporta, a fim de usá-lo;
- O fornecedor (provedor de componentes) desejará implementar as funcionalidades que, satisfação essas expectativas

Mas como é possível garantir que os dois pontos de vista são compatíveis? Tradicionalmente a noção de tipo tem sido usada para avaliar a compatibilidade de software. Segundo [27], podemos caracterizar compatibilidade de tipos de duas maneiras fundamentais:

- Compatibilidade sintática – onde o componente fornece todas as operações previstas (nomenclatura do tipo, assinaturas das funções, interfaces);
- Compatibilidade semântica – onde as operações de todos os componentes se comportam de uma forma esperada (semântica dos estados, axiomas lógicos e provas)

Segundo [28], uma boa definição para tipos em linguagens orientadas a objetos é que o tipo de um objeto é sua interface, ou seja, sua coleção de operações. Em linguagens orientadas a objetos, os objetos possuem suas próprias operações, as quais são a única parte visível deste objeto. A única coisa que se pode fazer com um objeto é invocar um de seus métodos. Assim, se quisermos caracterizar os tipos como, por exemplo, “uma coleção de valores que partilham uma estrutura comum ou uma forma”, devemos usar apenas as operações do objeto correspondente como caracterizadores da estrutura visível do tipo.

Idealmente, interfaces devem conter uma descrição formal do comportamento das operações, usando uma teoria lógica, por exemplo. As conexões do componente serão validadas então através da prova do teorema. Interfaces explícitas são úteis para esclarecer o papel de herança em linguagens de programação orientada a objetos [29].

Durante a fase de projeto, interfaces são tipicamente descrições semi-formais de como uma abstração interage com o resto do sistema. Junto com a nomenclatura adequada e convenções de documentação, elas providenciam especificações parciais. Na fase de implementação, interfaces usualmente descrevem os nomes, parâmetros e tipos resultantes de operações providas por um componente.

2.5. Descrição de requisitos

A descoberta e seleção de serviços é usada para fazer a correspondência entre as necessidades de um requisitante com as ofertas de um provedor. Ela é focada atualmente na funcionalidade dos componentes (ex. concentrando no que o serviço faz), e ocorre apenas um ambiente volátil e heterogêneo, onde há a necessidade de carregar componentes e serviços de maneira dinânica.

As questões que surgem para os requisitantes de serviços incluem a inabilidade de comparar serviços objetivamente, detalhes insuficientes sobre as propriedades funcionais e não-funcionais dos serviços, falta de linguagem comum para a descrição de serviços (i.e. provedores de serviços diferentes usam indevidamente termos de serviço relacionados) e uma falta de mecanismos para filtrar de maneira eficaz os serviços relevantes.

Essencialmente, a utilidade de um sistema é determinada por ambas suas características funcionais e não-funcionais, tais como usabilidade, flexibilidade, interoperabilidade, desempenho e segurança. No entanto, tem havido uma ênfase maior na descrição das funcionalidades de componentes, mesmo que isto apenas não seja suficiente para uma seleção apropriada de componentes, sem as características não-funcionais.

Na área de requisitos de software, o termo requisito não-funcional [30] tem sido usado para se referir a conceitos e características que não são relacionados à

funcionalidade do software, tais como utilização de recursos, confiabilidade, eficiência, etc. No entanto, diferentes autores caracterizam essa diferença nas definições de informal e desigual. Por exemplo, uma série de definições para requisitos não-funcionais é resumida em [31]:

- Descreve o aspecto não-comportamental de um sistema, capturando as propriedades e condições nas quais o sistema deve operar.
- Os atributos gerais necessários do sistema, incluindo portabilidade, confiabilidade, engenharia, testabilidade, compreensibilidade e modificabilidade.
- Requisitos que não são especificamente relacionados com a funcionalidade do sistema. Eles definem restrições no produto sendo desenvolvido e o processo de desenvolvimento. Eles especificam também as restrições que o produto deve satisfazer.
- As propriedades comportamentais que as funções especificadas devem ter, como desempenho e usabilidade.

Dito de outra forma, requisitos não-funcionais constituem as justificativas das decisões de design e restringem o modo pelo qual a funcionalidade necessária pode ser realizada [32].

2.6. Políticas de adaptação

O desenvolvimento de aplicações para computação móvel requer suporte para redes adaptativas e serviços customizados para os clientes dos sistemas sendo executados. De acordo com [33], uma política é uma regra que define uma escolha de acordo com o comportamento de um sistema. A separação entre a implementação de um sistema e as políticas que o regem permite que esta política possa ser modificada de modo a ser alterada dinamicamente para mudar a estratégia de gerenciamento do sistema e assim, modificar o comportamento deste, sem mudar sua base de execução [34].

Uma política de adaptação (ou reconfiguração) representa a especificação do comportamento de um componente de acordo com o contexto computacional do dispositivo. Por exemplo, um componente que tenha a necessidade de coordenadas de um sensor GPS, precisa de uma quantidade grande de energia para

sua execução. Este componente candidato a implantação deveria especificar em sua política de adaptação, qual é a quantidade mínima de energia necessária para que ele seja executado, evitando que o *middleware* execute uma adaptação desnecessária, desperdiçando seus recursos para o próprio carregamento e implantação do componente.

Para este trabalho foram especificadas nas políticas de adaptação dos componentes apenas a informação sobre as restrições de energia, conectividade e de memória. No entanto é possível definir outras informações nestas políticas como: políticas de segurança para controle de acesso, restrições de *hardware* (para componentes que necessitam de algum tipo de *hardware* específico como câmera, por exemplo), etc. Isto pode ser feito apenas criando novos tipos de restrições de execução no *middleware* Kaluana para que sejam usados pelos componentes que implementam algum tipo de política de adaptação.

No novo Kaluana, as políticas de adaptação estão definidas no contrato de reconfiguração dos componentes, o qual define quais os serviços são disponibilizados, quais receptáculos são requisitados para sua execução e quais são as restrições computacionais necessárias para que ele possa ser executado. O contrato de reconfiguração é descrito de maneira mais aprofundada na seção 5.2 desta dissertação.

Existem inúmeros grupos trabalhando em diferentes estratégias de especificação de políticas de adaptação, como [35-37].

2.7. Adaptação dinâmica

A necessidade de adaptação dinâmica de um sistema pode ter vários motivos, dentre eles, a capacidade de compensar ou reagir a mudanças no seu ambiente de execução ou no contexto do (s) usuário (s). Estas aplicações podem modificar a funcionalidade ou a estrutura de seus componentes a fim de melhor se adequar às novas condições de operação ou uso (i.e. ao novo contexto).

O momento de uma adaptação é definido segundo critérios baseados na motivação para sua realização. Segundo [38], adaptações realizadas devido a mudanças no ambiente de execução de uma aplicação, como no estado interno de outras aplicações são classificadas como **adaptações reativas**. Por esta

classificação, a adaptação dinâmica ocorre para que a aplicação possa continuar atendendo seus parâmetros de qualidade de serviço mesmo em diferentes condições de execução.

Adaptações corretivas ocorrem para corrigir partes defeituosas do sistema, as quais são substituídas, mantendo-se inalteradas as funcionalidades do sistema.

Outro tipo de adaptação dinâmica conhecido é a **adaptação de extensão**, que ocorre quando é necessária a adição de novas funcionalidades do sistema, não contempladas em tempo de desenvolvimento, porém mais tarde necessárias ao usuário.

A **adaptação de otimização** é necessária quando melhorias em implementações ocorrem, substituindo componentes para melhorar o desempenho do sistema.

Devido ao alto grau de variação nas condições de execução de dispositivos móveis, como a qualidade da conectividade sem fio, o conjunto de sensores ativados, o modo corrente de uso (por exemplo, em chamada telefônica versus acesso a internet) e o nível de energia residual, bem como no contexto do usuário, torna-se interessante que *frameworks* para desenvolvimento de aplicações móveis ofereçam mecanismos eficientes, para que desenvolvedores possam implementar em suas aplicações, comportamentos reativos a mudanças preemptivas nas condições de contexto.

Além de diferenças no objetivo e na motivação para uma adaptação dinâmica, outros dois aspectos são fundamentais para classificá-la: a granularidade e o objeto alvo da adaptação. Duas abordagens são comuns quanto à granularidade de adaptações dinâmicas: adaptações **em parâmetros** e adaptações **na composição** [39]. Na primeira, ocorrem apenas mudanças em variáveis que determinam o comportamento da aplicação. Segundo os autores, um exemplo clássico é a variação dos valores de controle de retransmissão e do tamanho das janelas no protocolo TCP. Neste caso, a implementação se restringe a algoritmos e componentes que substituirão os originais para reagir às mudanças na aplicação, sem modificar seu contexto de execução. A Figura 2 mostra os tipos de adaptação de acordo com as três classificações apresentadas.

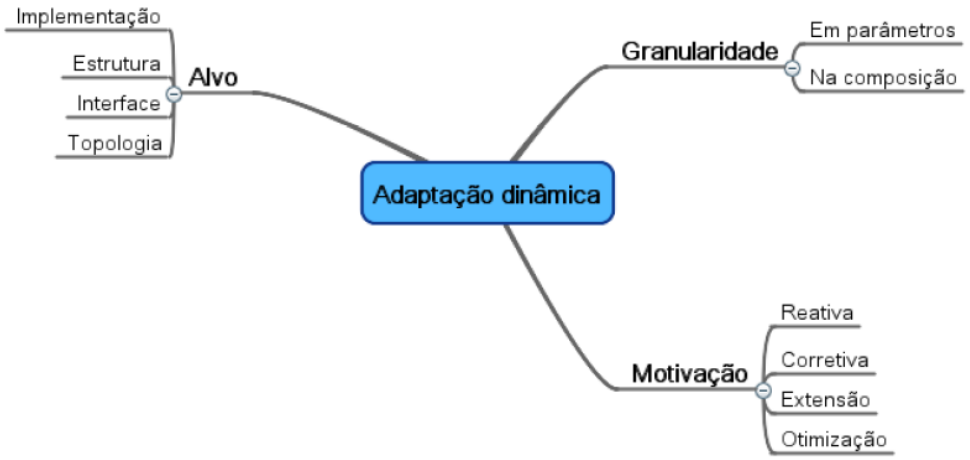

Figura 2: Classificação dos tipos de adaptação dinâmica, segundo [12]

Ao contrário da adaptação paramétrica, a adaptação na composição permite a adoção de algoritmos e componentes desenvolvidos após a implantação da aplicação, oferecendo maior flexibilidade ao desenvolvedor e ao usuário. Este tipo de adaptação é necessário quando existe limitação quanto ao número de componentes instanciados, como em dispositivos com pouca memória, ou quando novas implementações são necessárias para acomodar condições ou requisitos não previstos inicialmente.

Existem quatro estratégias que classificam o objeto de uma adaptação dinâmica na composição de uma aplicação [38]: adaptação **na implementação** dos componentes, adaptação **na estrutura** da aplicação, adaptação **na interface** de um componente ou adaptação **na distribuição** da aplicação.

A adaptação na implementação é mais adequada a adaptações corretivas, aumento de desempenho e mudanças no ambiente, não previstas em tempo de desenvolvimento. Nela um componente é substituído por outro, mantendo-se suas interfaces e funcionalidades inalteradas. Por exemplo, um componente de compactação de vídeo pode ser substituído por outro algoritmo de acordo com a taxa de transferência com que o vídeo é transmitido, ou um protocolo de comunicação podem ser trocados, baseado na taxa de erros da comunicação sem fio.

O segundo tipo de adaptação pode ser feito pela introdução de um novo componente, remoção de um componente existente ou modificação na conexão entre componentes. Por exemplo, uma aplicação que controla o acesso de um dispositivo a uma rede sem fio, pode chavear a conectividade para uma interface

mais lenta, caso o nível de energia do dispositivo se mostre muito baixo, substituindo este componente por outro que não tenha um consumo tão alto (alterando de *wifi* para GPRS, por exemplo).

A adaptação na interface de um componente significa a introdução ou remoção de uma interface, modificando a lista de serviços provida pelo componente. Neste caso, as funcionalidades providas por um componente são modificadas. Um exemplo típico é uma adaptação de extensão, onde uma nova funcionalidade já implementada por um componente, passa a ser disponibilizada para uso por novas versões das aplicações móveis.

Por fim, a adaptação na distribuição da aplicação, se dá, geralmente, por questões de desempenho, como balanceamento de carga. Ela consiste na divisão de processamento ou armazenamento entre diferentes dispositivos físicos. Um exemplo para este tipo de adaptação é [38]

2.8. Arquiteturas orientadas a serviço

Uma parte dos avanços tecnológicos na integração de *software*, mais notavelmente, a *Service-Oriented Architecture* (SOA), originou-se da emergência do desenvolvimento de *WebService* e de padrões para dar suporte à integração de negócios automatizados.

Em uma SOA, os recursos do *software* são empacotados como "serviços", os quais são módulos auto-contidos bem definidos, que provêm padrões de funcionalidades do negócio e são independentes de estado ou contexto de outros serviços [40]. Segundo [41], estes módulos são descritos em uma linguagem padrão, possuem uma interface pública e comunicam entre si requisitando execução de outras operações a fim de apoiar uma tarefa comum de um negócio ou processo.

Uma SOA é concebida para permitir que os desenvolvedores possam superar muitos desafios de computação distribuída como integração de aplicações, gerenciamento de transações ou políticas de segurança, enquanto possibilita que múltiplas plataformas e protocolos de acesso provenham numerosos acessos a

dispositivos e sistemas. Ela provê uma arquitetura flexível, que unifica processos de um sistema, modularizando grandes aplicações em serviços.

O objetivo principal de uma SOA é eliminar barreiras para que aplicações possam executar de maneira integrada e consistente. Desta maneira, uma SOA pode gerar a flexibilidade e agilidade requisitadas pelos usuários do sistema. Assim ela define serviços de alta granularidade, os quais podem ser agregados e reutilizados para facilitar as necessidades das mudanças de um sistema em execução.

As arquiteturas de *software* convencionais organizam seus sistemas em (sub) sistemas e inter-relações. Em contraste, a SOA cria, através de interfaces públicas, uma maneira lógica de planejar um sistema de *software* para prover serviços para aplicações finais ou mesmo para outros serviços distribuídos através de uma rede.

Uma SOA é independente de uma tecnologia específica, como *WebServices* ou *J2EE enterprise beans*. Isto é alcançado limitando o número de restrições de implementação ao nível da interface de serviço. Para isto, ela requer que os serviços sejam definidos através de uma linguagem de descrição (*Interface Description Language* [42], WSDL [43] , etc.

A intenção fundamental de um serviço em uma SOA é representar uma unidade reutilizável de um sistema. Um serviço é exposto como uma peça de uma funcionalidade com três propriedades essenciais:

1. Ele deve ser auto-contido, i.e., mantém seu próprio estado de execução.
2. Serviços são independentes de plataformas, Isto implica no fato da interface de contrato dos serviços ser limitada a asserções independentes de plataformas.
3. A SOA assume que serviços podem ser dinamicamente alocados, invocados e (re-)combinados.

Logicamente, um serviço é um par interface-implementação, onde a interface define sua identidade e suas logísticas. No entanto, a implementação define o trabalho que o serviço é designado a fazer.

Como as interfaces são independentes de plataforma, um cliente pode usar o serviço de qualquer dispositivo de comunicação, usando qualquer plataforma computacional, sistema operacional ou linguagem de programação. Estas duas

facetas do serviço são criadas e mantidas como itens distintos, no entanto suas existências são altamente inter-relacionadas.

A pesar de ser implementado usando o conceito de SOA, de acordo com as as três propriedades essenciais definidas nesta seção, o Kaluana provê um modelo de componentes orientado a serviços. Herdando a flexibilidade do modelo orientado a serviços e as capacidades de implantação, composição e reutilização de *software* do modelo de componentes, o modelo implementado permite a composição de aplicações em tempo de execução e a realização de adaptações por meio da substituição de componentes e de serviços.

Desta maneira ele, por um lado, agiliza o desenvolvimento de componentes, pois realiza a composição dos componentes através de reflexão computacional além de utilizar da orientação a serviços para prover abstrações ao desenvolvedor. Assim a tarefa de implementação de aplicações adaptáveis torna-se mais simples por utilizar-se de componentes reutilizáveis de *software*.

Alguns exemplos de sistemas orientados a serviço são OSGi [20] e Web Services Architectures [44].

2.9. Gerenciamento de serviços

O gerenciamento de serviços em sistemas distribuídos dinamicamente adaptáveis é uma tarefa bastante complexa. O reuso e a interoperabilidade de *software* é tipicamente suportado pelo uso de regulações implicitamente declaradas para nomear e definir o tipo de serviços em hierarquias.

Em sistemas dinamicamente adaptáveis, as aplicações são compostas por serviços oferecidos por provedores autônomos. Novos serviços são constantemente adicionados ao sistema enquanto antigos são removidos. É necessário o gerenciamento do tipo de serviços, para prover um controle das características apresentadas por cada serviço disponível para uso. A interoperabilidade entre os negócios das aplicações adaptadas deve ser garantida apesar da heterogeneidade dos serviços utilizados.

Através do gerenciamento de serviços, cada tecnologia possui uma abordagem específica para descrever, catalogar e selecionar os serviços disponíveis para utilização, categorizando equivalências entre eles. A equivalência

entre serviços é feita através de uma camada mediadora que agrega serviços semanticamente distintos, provendo uma visão homogênea de serviços heterogêneos [45]. Semanticamente, serviços equivalentes fornecem diferentes descrições, mas oferecem a mesma funcionalidade. Caso serviços semanticamente equivalentes possuam comportamentos transacionais diferentes, o mediador usará o comportamento menos restritivo disponível para agregar os serviços remotos.

Diferentes abordagens podem ser encontradas para realizar o gerenciamento de serviços em sistemas adaptáveis. Em [46] é apresentada uma análise dos requisitos necessários para o gerenciamento de tipos baseado no paradigma de computação orientada a serviços (*Service Oriented Computing* [47]), usando a tecnologia de *WebServices* [44]. Em [48], é feita uma comparação entre algumas características do gerenciamento de serviços em *WebServices* e CORBA [18].

2.10. Mobilis

A plataforma Mobilis [49, 50] é um *middleware* para desenvolvimento de aplicações móveis colaborativas e sensíveis ao contexto, que possibilita aos desenvolvedores se concentrarem somente nos aspectos restritos às suas aplicações. O *middleware* é orientado a serviços, para oferecer extensibilidade às aplicações desenvolvidas, de forma a facilitar a integração entre elas.

A arquitetura do Mobilis é composta por uma camada de *middleware* e uma camada de aplicação, como mostrado na Figura 3. De uma forma geral, a plataforma Mobilis é composta de componentes de software que gerenciam a aplicação e seu contexto a partir de serviços essenciais. Cada um desses serviços implementam alguma função básica de comunicação ou compartilhamento de dados. O ***Component Manager*** é responsável pela descoberta, escolha e ativação dos componentes usados pela aplicação, enquanto o ***Adaptation Manager*** inicia a adaptação dinâmica em relação aos componentes utilizados pela aplicação. O ***MD-ECI*** representa um sistema *Publish-Subscribe* baseado no SIP [51], que dá suporte a distribuição remota, entre dispositivos móveis de publicações, que podem ser objeto de dados ou eventos.

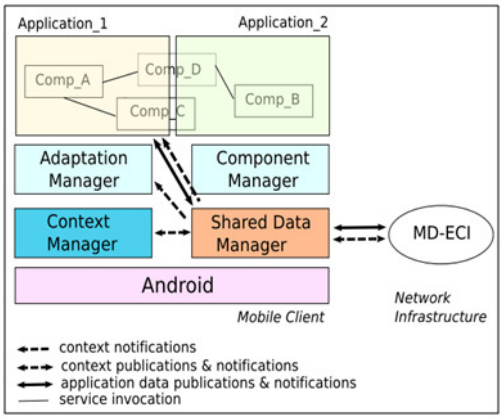

Figura 3. Arquitetura em camadas, da plataforma Mobilis

Os principais serviços da plataforma Mobilis que foram usados neste trabalho são o ***Shared Data Manager*****,** que provê uma API para a comunicação da aplicação com os componentes Mobilis, e o ***Context Manager***, que é responsável pela busca, escolha e ativação de provedores de contexto do dispositivo. Estes serviços serão melhor descritos nas seções 2.10.1 e 2.10.2.

2.10.1. Shared Data Manager

O *Shared Data Manager*, ou SDM, é o serviço da plataforma Mobilis que implementa um mecanismo de *Publish-Subscribe* que é utilizado tanto pelas aplicações quanto pelos serviços do *middleware*. Ele é utilizado principalmente para trocar dados e eventos localmente, isto é, para comunicação assíncrona entre os componentes ativos em um mesmo dispositivo, ou interação remota entre componentes e aplicações executando em dispositivos distintos.

Para receber notificações de um determinado assunto, uma aplicação ou serviço de *middleware* deve realizar uma assinatura junto ao SDM, registrando um ouvinte (*listener*) e, opcionalmente, informar uma expressão de filtro sobre as propriedades dos dados ou eventos a serem publicados. Assim, sempre que uma nova publicação no assunto desejado ocorrer, o SDM irá notificar todos os assinantes desse assunto, desde que a expressão registrada por esses esteja satisfeita de acordo com as propriedades da publicação.

2.10.2. Context Management Service

O *Context Management Service* (CMS) é o serviço responsável por gerenciar a coleta, processamento e distribuição das informações de contexto na arquitetura do Mobilis.

A coleta das informações é realizada através de componentes provedores de contexto gerenciados pelo CMS, que alimentam o serviço com as informações encontradas. Cada *ContextProvider* é responsável por informações de contexto específicas. É função do CMS, gerenciar o ciclo de vida dos provedores de contexto, carregando e ativando apenas os componentes de interesse das aplicações que estão em execução no dispositivo. *ContextProviders* também podem consumir informações de contexto de outros provedores, nesse caso o provedor de contexto deve requisitar a informação ao CMS de maneira análoga ao modo que é feito por uma aplicação.

As informações de contexto produzidas nos *ContextProviders* são repassadas às aplicações interessadas utilizando o serviço SDM. É tarefa do CMS, fazer a recepção das informações produzidas nos provedores, encapsulá-las e publicá-las no SDM.