UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Multi-MOM

Um Middleware Multi-Paradigma, Extensível e Orientado a Mensagens para Computação Móvel

Yuri Morais Bezerra

João Pessoa - PB
Agosto de 2010

Yuri Morais Bezerra

Multi-MOM

Um Middleware Multi-Paradigma, Extensível e Orientado a Mensagens para Computação Móvel

Dissertação de Mestrado submetida ao curso de Pós-Graduação em Informática da Universidade Federal da Paraíba como requisito parcial para obtenção do título de Mestre em Informática.

Orientador: Prof. Dr. Glêdson Elias da Silveira

João Pessoa - PB
Agosto de 2010

B574m Bezerra, Yuri Morais.
Multi-MOM : um middleware multi-paradigma, extensível e orientado a mensagens para computação móvel/ Yuri Morais Bezerra. - - João Pessoa: [s.n.], 2010.
116 f. : il.
Orientador: Glêdson Elias da Silveira.
Dissertação (Mestrado) – UFPB/CCEN.

1.Informática. 2.Computação Móvel. 3.Paradigmas de Comunicação. 4. Extensibilidade. 5.Linha de Produto de Software.

UFPB/BC CDU: 004(043)

Ata da Sessão Pública de Defesa de Dissertação de Mestrado do YURI MORAIS BEZERRA, candidato ao Título de Mestre em Informática na Área de Sistemas de Computação, realizada em 16 de agosto de 2010.

Aos dezesseis dias do mês de agosto do ano dois mil e dez, às nove horas, na Sala de Redes do Centro de Ciências Exatas e da Natureza da Universidade Federal da Paraíba, reuniram-se os membros da Banca Examinadora constituída para examinar o candidato ao grau de Mestre em Informática, na área de "*Sistemas de Computação*", na linha de pesquisa "*Computação Distribuída*", o Sr. Yuri Morais Bezerra. A comissão examinadora composta pelos professores doutores: Gledson Elias da Silveira (DI - UFPB), Primeiro Orientador e Presidente da Banca Examinadora, Guido Lemos de Souza Filho (DI-UFPB), como examinador interno, Carlos André Guimarães Ferraz (UFPE) como examinador externo. Dando início aos trabalhos, o Prof. Gledson Elias da Silveira, cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a palavra ao candidato para que o mesmo fizesse, oralmente, a exposição do trabalho de dissertação intitulado "*MULTI-MOM – Um Middleware Multi-Paradigma, Extensível e Orientado a Mensagens para Computação Móvel*". Concluída a exposição, o candidato foi argüido pela Banca Examinadora que emitiu o seguinte parecer: "*aprovado*". Assim sendo, deve a Universidade Federal da Paraíba expedir o respectivo diploma de Mestre em Informática na forma da lei e, para constar, a professora Tatiana Aires Tavares, Sra. Coordenadora do PPGI, lavrou a presente ata, que vai assinada por ele, e pelos membros da Banca Examinadora. João Pessoa, 16 de agosto de 2010.

Tatiana Aires Tavares

Prof. Dr. Gledson Elias da Silveira
Orientador (DI-UFPB)

Prof. Dr. Guido Lemos de Souza Filho
Examinador Interno (DI-UFPB)

Prof. Dr. Carlos André Guimarães Ferraz
Examinador Externo (UFPE)

AGRADECIMENTOS

Agradeço primeiramente a Deus, pelo dom da vida, saúde, paz e pela capacidade que me deu. A realização deste trabalho não teria sido possível sem a força que encontrei Nele durante esse tempo.

Agradeço aos meus pais, Fernando e Gisélia, por tudo que fizeram e fazem por mim, toda dedicação, amor, carinho, paciência, pelos bons exemplos que são para mim, pelo desprendimento nos investimentos que fazem em mim, e por serem os melhores pais do mundo. Amo muito vocês! Agradeço também à minha irmã, com quem eu sei que posso contar para qualquer coisa.

Agradeço à minha namorada Marília (Lila), que me compreendeu e me ajudou nos momentos mais difíceis desse mestrado, que me ajuda a ser cada dia melhor, em todos os sentidos. Nela encontro muita paz, amor, compreensão e tudo mais que procuro numa pessoa com a qual pretendo compartilhar todos os momentos da minha vida.

Agradeço ao meu orientador Prof. Glêdson, por todos os direcionamentos necessários à conclusão desse trabalho com êxito, e por tantos ensinamentos passados ao longo desse período. Agradeço também a todo corpo docente do PPGI, pelo melhoria contínua que vêm propiciando a esse curso.

Agradeço aos amigos que fiz nesse mestrado, Thaís, Talles, Vinicius, Bruno e Thiago. Essas pessoas contribuíram muito para dias mais alegres no laboratório, bem como com inúmeras discussões que me ajudaram na execução deste trabalho.

Agradeço aos meus amigos de todos os dias, Coloral, Hugo, Amilcar, Kiko, Oscar, Victor com os quais tenho convivido muito ultimamente. Nossos almoços durante a semana são essenciais para dias de trabalho mais divertidos. Agradeço também a todos os outros amigos (as), que apesar de distância de alguns, sei que posso contar e levá-los-ei sempre no coração.

Agradeço a toda minha família, em especial aos meus padrinhos, os quais serei eternamente grato por todo amor que me deram nos primeiros anos de vida.

Resumo

Os avanços nas tecnologias de comunicação sem fio e a miniaturização dos dispositivos móveis de alta capacidade estão trazendo grandes oportunidades para o desenvolvimento de aplicações que explorem essas novas fronteiras da computação. Entretanto, o desenvolvimento de aplicações nestes cenários traz novos desafios, pois estas operam em dispositivos de recursos limitados e comunicam-se através de redes sem fio, caracterizadas por conexões intermitentes. Para evitar que esses desafios tenham que ser resolvidos de forma improvisada para cada aplicação, plataformas de middleware são utilizadas, liberando os desenvolvedores de aplicações das dificuldades trazidas pela mobilidade. Devido ao seu estilo de comunicação assíncrono e fracamente acoplado, plataformas de Middleware orientado a Mensagens (MOM) têm sido comumente adotadas nestes casos. Entretanto, uma das limitações mais significantes das atuais plataformas de MOM é que elas geralmente dão suporte a um paradigma de comunicação único e predefinido (ex.: publish/subscribe). Essa restrição limita o escopo de aplicações que o middleware pode dar suporte. Para lidar com essa limitação, esta dissertação propõe um middleware para dispositivos móveis capaz de dar suporte a um conjunto extensível de paradigmas de comunicação baseados em mensagens (ex.: espaço de tuplas, filas de mensagens, publish/subscribe). Através de uma arquitetura integrada, a qual foi concebida baseada numa abordagem de Linha de Produto de Software (LPS), o middleware encapsula funcionalidades comuns para lidar com os desafios trazidos pela mobilidade, provendo componentes reusáveis e compartilhados entre os diversos paradigmas. Os resultados da avaliação mostram que o *overhead* introduzido pela abordagem multi-paradigma é mínimo, tanto em termos de espaço ocupado em memória, quanto em desempenho em tempo de execução. Por fim, com o intuito de ilustrar como aplicações móveis podem se beneficiar dessa abordagem, um cenário de aplicação é descrito.

Palavras-chave: Middleware orientado a Mensagens, Computação Móvel, Paradigmas de Comunicação, Extensibilidade, Linha de Produto de Software.

Abstract

Advances in wireless communication technologies and miniatuziation of mobile devices are leading to great opportunities in the development of applications to explore this new computing frontier. However, the development of applications for such scenarios introduces new challenges, as mobile applications run on resouce-scarce devices and communicate with each other by means of wireless networks, which are characterized by intermittent connections. In order to avoid having to deal with such issues in an ad hoc fashion for every application, middleware platforms are adopted, concealing difficulties raised by mobility from application engineers as much as possible. Due to the asynchronous and loosely coupled communication style, Message-oriented Middleware (MOM) platforms have been commonly adopted for supporting the development of networked mobile applications. However, one of the most significant limitations of current MOM for mobile platforms is that they typically support a single, predefined communication paradigm (e.g., publish/subscribe). Such a restriction limits the scope of applications supported by the middleware. In order to mitigate such a limitation, this paper presents a middleware for mobile devices capable of supporting an extensible set of message-oriented communication paradigms (e.g. tuple spaces, message queue, publish/subscribe). Supported by an integrated architecture, which has been conceived based on a Software Product Line (SPL) approach, the middleware encapsulates common features that deal with mobility issues and provides them as shared, reusable components. Evaluation results show that the overhead introduced by such a multi-paradigm approach is minimal, both in memory footprint and runtime performance. Additionally, an application scenario illustrates how mobile applications may benefit from such an approach.

Keywords: Message-oriented Middleware, Mobile Computing, Communication Paradigm, Extensibility, Software Product Line.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

3G – *Third Generation*
API – *Application Programming Interface*
CORBA – *Common Object Request Broker Architecture*
DA – *Directory Agent*
FIFO – *First in, first out*
HTTP – *Hyper Text Transfer Protocol*
ID - Identificador
IP – *Internet Protocol*
IPC – *Inter-process call*
ITS – *Interface Tuple Space*
JMS – *Java Message Service*
jSLP – *Java Service Location Protocol*
LIME – *Linda in a Mobile Environment*
LPS – Linha de Produto de Software
MANET - *Mobile Ad Hoc Networks*
MIME – *Multipurpose Internet Mail Extensions*
MOM – Middleware Orientado a Mensagens
MPI – *Message Passing Interface*
OMG – *Object Management Group*
RAM – *Random Access Memory*
RMI - *Remote Method Invocation*
RPC – *Remote Procedure Call*
SMTP – *Simple Mail Transfer Protocol*
SLP – *Service Location Protocol*
TCP – *Transmission Control Protocol*
TTL – *Time to live*
UDP - *User Datagram Protocol*
Wi-Fi – *Wireless Fidelity*

1 Introdução

Um estudo recente afirma que o mercado de aplicações para dispositivos móveis vai crescer de $1.94 bilhões de dólares registrado em 2009 para $15.65 bilhões de dólares previsto para 2013 [Research2Guidance10]. Jogos, entretenimento, aplicações multimídia e corporativas estão entre os tipos mais populares. Até pouco tempo, essas aplicações eram desenvolvidas somente pelas empresas que construíam os sistemas operacionais dos celulares e vinham geralmente pré-instaladas nos dispositivos. Todavia, recentemente, um novo modelo de negócio vem sendo utilizado nesse mercado, onde desenvolvedores e/ou empresas independentes podem desenvolver aplicações e disponibilizá-las para venda em lojas de aplicações *online*, como a *Apple App Store* [Apple10] e o *Android Market* [Android10b].

Esse crescente interesse observado no mercado de aplicações móveis é impulsionado pelo grande aumento de capacidades avançadas presentes nos dispositivos móveis e pela crescente demanda de assinantes de serviços de telefonia móvel. Ainda de acordo com o mesmo estudo [Research2Guidance10], essa tendência tem apoio também no volume de vendas de *smartphones*, que atingirá 1 bilhão de usuários até o fim de 2013. *Smartphone* é uma denominação para telefones celulares que permitem a instalação de aplicações de terceiros, tem capacidade de armazenamento medida em gigabytes e possuem capacidades avançadas de conexão, incluindo acesso a redes de dados (ex.: 3G) e a redes locais (ex.: Wi-Fi, Bluetooth).

A popularidade destes tipos de dispositivos, bem como a maturidade das tecnologias de rede sem fio está nos levando a um estágio onde podemos ter comunicação em qualquer lugar e a qualquer momento. Como conseqüência deste cenário, novos tipos de aplicações e serviços podem ser oferecidos de forma bastante dinâmica e transparente, suportando a conexão com a internet e/ou com outros dispositivos similares. Aplicações desse tipo são chamadas de aplicações móveis conectadas [Paller06].

Existe um grande conjunto de aplicações potenciais para estes cenários, mas o desenvolvimento para este domínio não é fácil, pois traz uma gama de desafios que não eram comuns no desenvolvimento de aplicações tradicionais, como a ocorrência freqüente de desconexões e a escassez de recursos dos dispositivos. Embora a escassez de memória seja bem menor atualmente, já que os novos *smartphones* possuem gigabytes de memória *flash*, esses dispositivos ainda são limitados quanto ao consumo de energia e processamento. Além disso, as redes sem fio possuem características muito dinâmicas quanto aos membros disponíveis para comunicação e compartilhamento de recursos, disponibilidade de conectividade e variação de largura de banda da rede [Hadim06]. Obviamente, resolver estes problemas para cada nova aplicação de maneira improvisada não é recomendável.

Neste sentido, o uso de tecnologias de middleware (camada de software intermediária entre sistema operacional, redes, e as aplicações) se faz altamente necessário para resolver ou minimizar as dificuldades. Tecnologias de middleware tornaram-se essenciais no suporte a computação distribuída em redes cabeadas tradicionais, amenizando os problemas de heterogeneidade de plataformas e simplificando o processo de desenvolvimento [Emmerich00]. Como resultado, muita pesquisa vem sendo realizada para examinar como sistemas de middleware podem ajudar aplicações móveis distribuídas a superar as limitações impostas pelas redes sem fio [Mascolo02]. Assim, o middleware seria responsável por prover facilidades de coordenação e comunicação entre aplicações móveis distribuídas, abstraindo ao máximo as dificuldades trazidas pela mobilidade. O middleware separa as funcionalidades relativas à comunicação em rede das funcionalidades específicas das aplicações, liberando o desenvolvedor para focar apenas na lógica de sua aplicação.

No entanto, devido à alta dinamicidade destes cenários, onde dispositivos entram e saem da rede a todo momento, interações síncronas e baseadas no modelo cliente-servidor – geralmente utilizadas nos middlewares tradicionais (RMI, RPC) – não são adequadas para cenários de mobilidade. Ao invés disso, interações assíncronas, baseadas num modelo de rede *peer-to-peer* se adéquam melhor a computação móvel, pois estas levam a uma arquitetura com menor acoplamento entre as aplicações comunicantes. Esse estilo de comunicação é provido pelo paradigma de middleware orientado a mensagens (MOM). Nos MOMs, ao invés de invocações diretas de métodos ou procedimentos remotos, a comunicação é realizada através da produção e consumo

de mensagens, que são estruturas de dados genéricas criadas pelas aplicações para transmitir dados.

Para atender aos diversos cenários de aplicação possíveis na computação móvel, os sistemas de MOM têm sido implementados sob diversas variações, classificadas de acordo com o paradigma de comunicação que implementam [Bellavista07]. Exemplos de paradigma de comunicação incluem: publish/subscribe, espaço de tuplas, filas de mensagens e notificações ponto-a-ponto. Em todos estes, a comunicação é realizada através de trocas de mensagens. Por outro lado, o que diferencia estes paradigmas são aspectos relacionados à semântica de entrega das mensagens, que incluem: (i) consumidores são notificados ou precisam buscar as mensagens num determinado local; (ii) mensagens são endereçadas para um serviço intermediário ou diretamente para a aplicação consumidora de mensagens; (iii) mensagens são consumidas numa ordem pré-determinada ou de acordo com parâmetros especificados.

1.1 Motivação

Considerando a diversidade de paradigmas de comunicação, um dos principais problemas no desenvolvimento de aplicações móveis distribuídas é decidir qual paradigma é a melhor opção a ser adotada [Parlavantzas03]. Cada paradigma apresenta certas particularidades e é mais adequado para determinados cenários. Desta forma, de acordo com [West05] e [Costa06], plataformas de middleware não podem estar limitadas a um paradigma de comunicação específico, ao invés disso, devem ser flexíveis o suficiente para atender uma grande variedade de domínios de aplicação.

Sistemas de middleware orientado a mensagens para dispositivos móveis estão disponíveis numa grande variedade de implementações, como o LIME [Murphy06], Spontaneousware [Batista09], MOBILE MOM [Jung99], JMS for MANET [Vollset03], Pronto [Yonkei03] ToPSS e JEDI [Cugola02]. Entretanto, uma das principais limitações das atuais implementações de MOMs é que elas são projetadas para dar suporte a um paradigma de comunicação único e pré-definido (ex.: publish/subscribe). Essa restrição limita o escopo destas plataformas, no sentido de que elas não podem facilmente acomodar, ou serem estendidas para acomodar, a grande variedade de requisitos dos cenários móveis.

Por outro lado, existem alguns *frameworks* de middleware [Costa05] [Singhai98] [Apel05] que, a partir de uma estrutura genérica de alto nível, permitem derivar diversos paradigmas de comunicação. Esses *frameworks* permitem, inclusive, prover dois ou mais paradigmas de comunicação numa única instância da plataforma. Entretanto, nenhum deles provê mecanismos comuns entre os possíveis paradigmas de comunicação para tratar os requisitos da computação móvel. Isto é, esses *frameworks* permitem que sejam providos múltiplos paradigmas simultaneamente, mas não definem funcionalidades reusáveis entre esses paradigmas. Isso faz com que seja necessário tratar os problemas de comunicação em redes móveis individualmente para cada paradigma.

Essa falta de integração e reuso observada nos *frameworks* citados traz duas conseqüências indesejadas. Primeiro, perde-se em economia de recursos relacionados ao desenvolvimento, pois o esforço de programação para estender o middleware na implementação de soluções customizadas será maior, já que não serão reusadas funcionalidades comuns previamente definidas. Segundo, perde-se eficiência na utilização de recursos computacionais, pois as funcionalidades para tratar os problemas de comunicação da computação móvel terão que ser definidas individualmente para cada paradigma, resultando em redundância de funcionalidades, e consequentemente, maior quantidade de memória ocupada pelo middleware. Prover sistemas de baixa carga computacional é essencial no desenvolvimento para dispositivos de recursos escassos.

Em resumo, enquanto algumas soluções atuais provêem apenas um paradigma de comunicação pré-definido, outras provêem uma solução aberta para extensões, mas não definem uma arquitetura integrada, com funcionalidades reusáveis. Um paradigma único e predefinido impossibilita a utilização do middleware por uma grande variedade de aplicações; já a ausência de funcionalidades comuns e reusáveis, causa desperdício de recursos computacionais e torna a programação para extensão do middleware mais dispendiosa.

1.2 Objetivos

Dada as limitações das soluções e propostas existentes, este trabalho propõe um sistema de middleware orientado a mensagens para computação móvel, denominado Multi-MOM [Morais10b] [Morais10c]. O objetivo geral desse trabalho é prover uma plataforma flexível e de baixa carga computacional para lidar com problemas comuns

encontrados no desenvolvimento de aplicações para computação móvel, como as conexões intermitentes das redes sem fio e as restrições de recursos dos dispositivos móveis. Para isso, o Multi-MOM é baseado nas seguintes características:

- **Multi-paradigma:** ao invés de impor um único paradigma de comunicação para as aplicações, o middleware proposto dá suporte a múltiplos paradigmas (ex.: espaço de tuplas, publish/subscribe, filas de mensagens), tornando-o apto a atender cenários de aplicação distintos e mais propenso a ser reusado do que uma solução baseada num paradigma específico.
- **Arquitetura Integrada:** Para evitar que o tamanho da plataforma cresça excessivamente por oferecer múltiplos paradigmas, são definidas funcionalidades comuns entre paradigmas de comunicação orientados a mensagens, provendo estas funcionalidades como componentes comuns e reusáveis entre os diversos paradigmas de comunicação.
- **Projeto Extensível:** antecipar todos os requisitos de distribuição das possíveis aplicações pode não ser possível durante o desenvolvimento do middleware, assim, por meio de pontos de extensão, é possível adicionar novos paradigmas, permitindo que o middleware adquira novos comportamentos.
- **Seleção de Funcionalidades:** Além de possibilitar acrescentar novos paradigmas, o middleware proposto permite também selecionar, entre os paradigmas existentes, um subconjunto dos mesmos, de forma a otimizar o tamanho da plataforma para adequar-se a dispositivos mais limitados.

Visando atender aos requisitos supracitados, o Multi-MOM foi desenvolvido baseado no conceito de linha de produto de software (LPS). Assim, ao invés de um sistema monolítico, com funcionalidades fixas e predefinidas, o Multi-MOM define um conjunto de funcionalidades núcleo, obrigatórias, e um conjunto de funcionalidades opcionais, variáveis. O núcleo básico consiste das funcionalidades comuns entre os paradigmas de comunicação orientados a mensagens. Já as funcionalidades opcionais estão relacionadas especificamente a cada paradigma de comunicação, podendo estas estarem presentes ou não em cada instância do Multi-MOM. O middleware deixa de ser uma caixa preta e passa a ser uma composição flexível de diferentes serviços.

Para validar os conceitos apresentados, um protótipo do Multi-MOM foi implementado com suporte a cinco paradigmas de comunicação distintos. Tal implementação foi desenvolvida sobre a plataforma Android [Android10a], uma plataforma de código aberto para computação móvel amplamente utilizada por dispositivos do tipo *smartphone*, desenvolvida por um grupo de mais de 70 grandes empresas de tecnologia.

Além disso, para ilustrar os benefícios da abstração multi-paradigma no ambiente móvel, é apresentado um cenário de aplicação utilizando o Multi-MOM, destacando como as operações disponibilizadas pelo middleware proposto são naturais e suficientes para dar suporte às variadas necessidades de interação das aplicações móveis colaborativas.

Por fim, visando avaliar experimentalmente se a solução proposta atende aos objetivos e requisitos definidos inicialmente, o middleware foi avaliado sob o ponto de vista da reusabilidade das funcionalidades comuns definidas, da adequação do mesmo para implantação em dispositivos móveis e do desempenho.

1.3 Organização do Trabalho

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 introduz os conceitos básicos necessários ao entendimento deste trabalho. O Capítulo 3 apresenta as principais características do Multi-MOM, suas funcionalidades e projeto arquitetural. Posteriormente, o Capítulo 4 apresenta detalhes de implementação e um cenário de aplicação que explora as funcionalidades existentes no middleware proposto. No Capítulo 5 é apresentada uma análise do middleware proposto. No Capítulo 6 são analisados os trabalhos relacionados. Por fim, o Capítulo 7 apresenta as considerações finais, evidenciando as contribuições, limitações e trabalhos futuros.

2 Conceitos Básicos

Neste capítulo são descritos conceitos fundamentais relacionados ao desenvolvimento deste trabalho. Inicialmente são apresentados os tipos de sistemas distribuídos, alguns conceitos de middleware e os paradigmas de comunicação considerados nesta dissertação. Em seguida são apresentadas definições de Linha de Produto de Software, e a sua utilização em sistemas de middleware para computação móvel.

2.1 Tipos de sistemas distribuídos

Um sistema distribuído consiste basicamente de um conjunto de componentes, distribuídos em vários computadores (ou dispositivos) conectados através de uma rede de computadores [Tanenbaum02]. Esses componentes precisam interagir uns com os outros, para, por exemplo, trocar dados ou acessar os serviços uns dos outros. Nesta seção são introduzidos alguns conceitos necessários para entender as semelhanças e diferenças entre sistemas distribuídos tradicionais e sistemas distribuídos na computação móvel. São essas diferenças e semelhanças que guiam a definição de requisitos dos sistemas projetados para esse novo contexto [Mascolo02].

Sistemas distribuídos tradicionais. Esse tipo de sistema é formado por um conjunto de dispositivos fixos, conectados a rede de forma permanente através de redes cabeadas, com conexões estáveis e executando num ambiente estático (Figura 1).

Figura 1 - Sistema distribuído tradicional

Sistemas distribuídos móveis *ad hoc*. Consistem num conjunto de dispositivos móveis, conectados através de redes sem fio, que são caracterizadas por conexões intermitentes e taxas de transmissão baixas, quando comparadas às redes cabeadas. Esses tipos de

sistemas são executados em ambientes altamente dinâmicos, onde os dispositivos entram e saem da rede a qualquer momento. Nesse tipo de sistema não há infra-estrutura fixa pré-definida onde os dados são centralizados e coordenados. Ao contrário, clientes atuam de forma colaborativa para prover e consumir serviços. Por ser constituído apenas por clientes, é um tipo de rede que possui uma grande restrição de recursos. Entretanto permite a formação de uma rede a qualquer momento, e em qualquer lugar. A Figura 2 ilustra esse tipo de sistema distribuído.

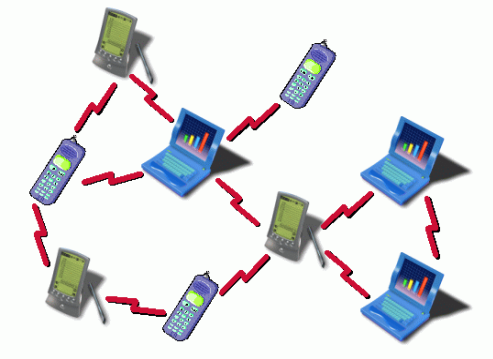

Figura 2 - Sistema distribuído móvel ad hoc

Redes *ad hoc* representam uma tendência de pesquisa em sistemas distribuídos, tanto na academia quanto na indústria. Ambos estão colocando grandes esforços na definição de middleware para estes cenários [Capra03].

Sistemas distribuídos móveis infra-estruturados. Estes podem ser considerados um passo intermediário entre as redes fixas tradicionais e as redes *ad hoc*. Sistemas móveis infra-estruturados, assim como os tradicionais, também são baseados numa infra-estrutura fixa pré-definida que centraliza e coordena a comunicação. Entretanto, os dispositivos que participam dessa rede são móveis e se conectam às estações base através de redes sem fio. Assim como nos sistemas *ad hoc*, nesse tipo de sistema é comum que os dispositivos entrem e saiam da rede a qualquer momento. A Figura 3 ilustra a organização desse tipo de sistema.

A computação móvel se enquadra nos dois últimos tipos: sistemas móveis *ad hoc* e infra-estruturados. Ela é fortemente caracterizada pela presença de dispositivos computacionais portáteis interagindo entre si e, possivelmente, com infra-estruturas fixas. A mobilidade, a escassez de recursos dos dispositivos e a fragilidade das conexões sem fio trazem novos problemas que não são levados em conta por sistemas distribuídos fixos. Assim, sistemas distribuídos desenvolvidos para ambientes de

mobilidade devem considerar primordialmente o tratamento adequado para conexões intermitentes e soluções de baixa carga computacional [Gaddah03] [Mascolo02].

Figura 3 - Sistema distribuído móvel infra-estruturado

2.2 Middleware

Na literatura existem diversas definições para *middleware*, podendo este ser entendido sob diferentes perspectivas, dependendo do contexto e dos interesses em questão. Contudo, vários autores [Linthicum01] [Emmerich00] [Geihs01] definem middleware como sendo uma camada de software que reside entre o programa aplicativo e o sistema operacional, e que desacopla as aplicações distribuídas de quaisquer dependências em termos de sistema operacional, plataforma de hardware, protocolos de comunicação e em alguns casos, também de linguagem de programação.

Uma visão mais geral seria admitir que embora tenha como um dos objetivos a promoção de transparência entre as partes comunicantes de um sistema distribuído, um middleware não é utilizado somente para essa finalidade, mas também para prover serviços, que dentre outras coisas, promovem a reusabilidade e a redução de esforço e de tempo para o desenvolvimento de aplicações de natureza distribuída.

Existem diversas tecnologias para desenvolvimento de sistemas distribuídos, como o Java/RMI, Microsoft COM, OMG CORBA, entre outros. Esses sistemas são considerados padrões ‘de fato’ e tem se mostrado eficientes no desenvolvimento de sistemas distribuídos tradicionais. Entretanto, esses sistemas de middleware foram projetados para ambientes estáticos, com forte acoplamento entre aplicações cliente e servidor, e baseados em formas de interação síncrona.

Por outro lado, cenários de mobilidade são caracterizados por conexão intermitente e alta dinamicidade, com dispositivos entrando e saindo da rede a qualquer momento. Devido a essa dinamicidade, as partes comunicantes ficam freqüentemente

fora de alcance umas das outras. Como resultado, formas de comunicação assíncrona têm sido amplamente adotadas para comunicação nestes cenários.

2.3 Middleware Orientado a Mensagens (MOM)

Devido à versatilidade e robustez, MOMs foram inicialmente propostos para interligação de sistemas de informação de grande porte (Eugster, 2003). Recentemente, devido a sua forma de comunicação assíncrona, desacoplada e baseada num modelo *peer-to-peer*, bem como seus mecanismos de tolerância a falhas, sistemas de MOM tem se mostrado a forma mais adequada de prover comunicação em ambientes móveis.

MOM é um tipo de middleware que facilita a comunicação entre aplicações distribuídas através da troca de mensagens. Assim como nos middleware tradicionais, o principal objetivo é prover uma API simples e de alto nível para programação distribuída, de forma a abstrair para as aplicações todos os detalhes das comunicações em rede. Entretanto, ao invés de invocar diretamente um método provido por uma aplicação remota, a comunicação no MOM é realizada através da produção e consumo de mensagens, que são estruturas de dados genéricas criadas pelas aplicações para transmitir dados.

Neste tipo de middleware, as mensagens são usadas como entidade principal para troca de informações, podendo carregar requisições de serviços, notificações de eventos, estado das aplicações, entre outros. Isso faz com que esse tipo de sistema seja bastante flexível, já que essas mensagens podem representar diversos tipos de informações e serem distribuídas na rede de acordo com as necessidades de cada aplicação [Sun02].

Essa generalidade dos sistemas baseados em mensagens permitiu o surgimento de diversas variações dos mesmos, sendo que cada variação define uma forma diferente de relação entre produtores e consumidores de mensagens [Eugster03]. Tais variações são brevemente descritas a seguir:

Publish/Subscribe. Neste tipo de MOM, as partes comunicantes trocam mensagens através da publicação de eventos e inscrição em determinados tipos de eventos. Um serviço intermediário, o canal de eventos, registra todas as inscrições e, ao receber eventos do publicador envia para os assinantes daquele tipo de evento. A Figura 4 ilustra a interação entre os publicadores e assinantes de um canal de eventos. A

inscrição no canal de eventos geralmente é feita de duas formas: baseada em tópicos, ou baseada em conteúdo. Nos sistemas baseados em tópicos, as inscrições são similares a grupos, isto é, publicar mensagens e inscrever-se em um tópico T pode ser visto como tornar-se membro do grupo T. Por outro lado, em sistemas baseados em conteúdo, consumidores especificam determinados filtros para os eventos, e só recebem os eventos que atendem aos filtros especificados [Eugster03].

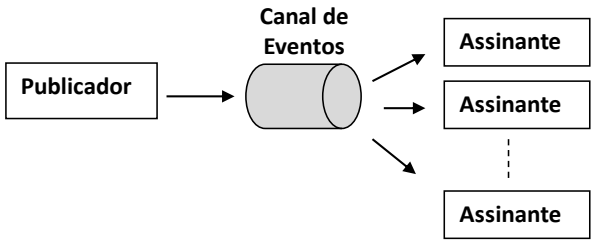

Figura 4 - Modelo Publish/Subscribe

Espaço de Tuplas. O conceito de espaço de tuplas foi inicialmente proposto no Linda [Gelernter85] como uma linguagem de coordenação para programação concorrente. Entretanto, seu estilo de comunicação oportunista e desacoplado provê facilidades importantes para ambientes de mobilidade. Um espaço de tuplas é essencialmente uma memória distribuída compartilhada que pode ser usada na comunicação entre aplicações distribuídas. Ela atua como um repositório de estruturas de dados, chamadas de tuplas, similar a idéia de mensagens anteriormente descrita. Aplicações comunicam-se inserindo, buscando e removendo tuplas do espaço de tuplas. A busca geralmente é feita por meio de associação de conteúdo, isto é, especificando um conjunto de propriedades desejadas na tupla a ser buscada. A Figura 5 ilustra a interação entre as partes comunicantes num espaço de tuplas.

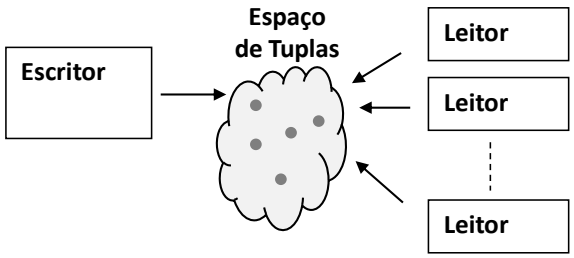

Figura 5 - Modelo Espaço de Tuplas

Fila de Mensagens. Esse paradigma de comunicação é implementado através de filas distribuídas, onde aplicações enviam mensagens para a fila e outras retiram mensagens

da fila. Em geral, apenas um consumidor vai receber cada mensagem. A dinâmica de interação entre as partes comunicantes é muito parecida com a do espaço de tuplas, diferindo basicamente no modo como as mensagens são recuperadas. No espaço de tuplas, o consumidor da mensagem especifica parâmetros de busca para a mensagem desejada, enquanto que, em filas, as mensagens são recuperadas em uma ordem predefinida, como por exemplo: "primeira a entrar, primeira a sair" (*first-in, first-out* - FIFO), ou baseada em prioridades. A Figura 6 ilustra a interação entre as partes comunicantes de uma fila de mensagens.

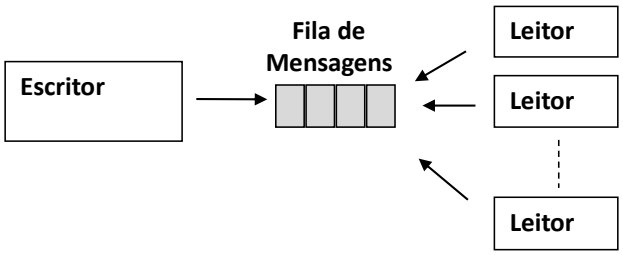

Figura 6 - Modelo Fila de Mensagens

Síncrono. Diferentemente dos paradigmas anteriores, no paradigma síncrono cada mensagem é explicitamente direcionada a uma aplicação, sem passar por um serviço intermediário, como o canal de eventos ou o espaço de tuplas. Além disso, esse paradigma adota o modelo cliente-servidor, em que as aplicações clientes enviam mensagens para a aplicação servidor, e esta envia em retorno uma mensagem para a primeira. A interação é similar a do *Remote Procedura Call* (RPC) e *Remote Method Invocation* (RMI), onde o cliente envia uma requisição e espera bloqueado pela resposta. Neste caso, a diferença é que, ao invés de invocar diretamente um procedimento ou método remoto, uma estrutura de mensagem é usada para troca de informações. A Figura 7 ilustra a interação entre cliente e servidor no modelo síncrono.

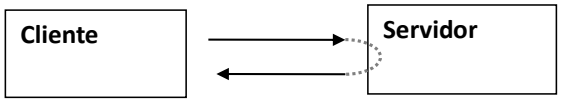

Figura 7 - Modelo Síncrono

Notificações Ponto-a-Ponto. De forma similar ao modelo síncrono, este paradigma não utiliza um serviço intermediário. Ele pode ser usado como uma alternativa ao paradigma

síncrono quando não há garantia de que haverá conexão entre as partes comunicantes no momento da produção da mensagem. Isso porque a aplicação produtora não precisa esperar bloqueada por uma resposta, se essa resposta existir. A Figura 8 ilustra a interação entre cliente e servidor no modelo notificações ponto-a-ponto.

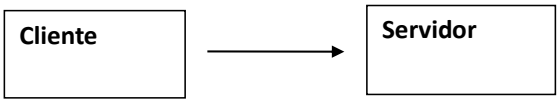

Figura 8 - Modelo Notificações Ponto-a-Ponto

Todos os paradigmas discutidos acima têm o objetivo de permitir a comunicação em rede entre duas ou mais aplicações através da troca mensagens. O que os diferencia são as **semânticas de entrega** das mensagens, isto é, propriedades relacionadas ao modo como essas mensagens são trocadas (enviadas e recebidas). Entre tais propriedades destacam-se: (i) a existência ou não de serviços intermediários; (ii) se a mensagem é entregue a apenas um consumidor, ou vários; (iii) se os consumidores das mensagens são conhecidos pelo produtor ou não; (iv) se os consumidores são notificados de novas mensagens ou precisam explicitamente buscá-las; (v) se há algum ordenamento pré-estabelecido para consumo de mensagens.

É importante ressaltar que, apenas pela modificação de algumas dessas propriedades, isto é, alterando as semânticas de entrega das mensagens, diversos outros paradigmas poderiam ser derivados. Entretanto, esses descritos acima são os mais debatidos na literatura e mais utilizados na computação móvel [Eugster03] [Mascolo02] [Aldred05].

2.4 Middleware e Linhas de Produto de Software

Como explicado anteriormente, sistemas de middleware consistem de um conjunto de serviços que facilitam o desenvolvimento de aplicações distribuídas em ambiente computacional heterogêneo. Esses serviços precisam atender a requisitos específicos de vários domínios de aplicação e de conceitos relativos à computação distribuída. Oferecer serviços para atender diversas necessidades distintas deixa o código do middleware excessivamente grande e complexo, além de implicar em alto consumo de recursos computacionais [Zhang03].

A alternativa para essa questão é customizar o middleware, isto é, possibilitar que funcionalidades possam ser selecionadas de acordo com necessidades específicas. Linhas de Produto de Software (LPS) são vistas como uma abordagem promissora para customizar sistemas de middleware, já que estas podem ser usadas para expressar as partes comuns e variáveis de uma família de sistemas de software. Assim, numa plataforma de middleware, poderia ser selecionado um subconjunto das funcionalidades providas para serem implantadas em cada dispositivo. Esta seleção poderia ser baseada nas capacidades de cada dispositivo, bem como nas necessidades das aplicações que executarão sobre o middleware [Kaul06].

Técnicas de LPS são então utilizadas em sistemas de middleware a fim de otimizar sua arquitetura. Define-se então um núcleo básico para o middleware que é estendido por meio da adição incremental de componentes. O núcleo básico é a parte comum a todos os sistemas de middleware derivados a partir da LPS, e as funcionalidades específicas formam as partes variáveis e/ou opcionais, podendo estas estarem presentes ou não em cada instância da LPS. O middleware deixa de ser uma caixa preta e passa a ser uma composição flexível de diferentes serviços.

Assim, funcionalidades comuns, também chamadas núcleo, são implementadas apenas uma vez e reaproveitadas para todos os sistemas que fazem parte da LPS. A derivação de novos produtos a partir dessa LPS utilizará essas funcionalidades comuns, e acrescentará algumas funcionalidades específicas a esse novo sistema. Essa abordagem favorece o reuso e, consequentemente, diminui o tempo de desenvolvimento de novos sistemas de middleware.

3 Multi-MOM

Este capítulo apresenta o middleware proposto, denominado Multi-MOM. Inicialmente é apresentada uma visão geral do Multi-MOM, seguida das suas principais características e funcionalidades. Em seguida é apresentada a arquitetura do mesmo, explicando cada um dos seus componentes. Além disso, o projeto do middleware é detalhado, tanto numa visão estática estrutural, como numa visão dinâmica comportamental.

3.1 Visão geral

Levando em consideração a diversidade de paradigmas de comunicação, um dos principais problemas ao desenvolver-se middleware para aplicações móveis distribuídas é decidir qual paradigma é o melhor a ser adotado. Cada paradigma apresenta particularidades e atende a diferentes cenários. Em [Wuest05] algumas recomendações são dadas:

- Publish/Subscribe deve ser usado quando a notificação de eventos é necessária, mas não é essencial saber quem são os consumidores destes eventos;
- Espaço de Tuplas deve ser usado para trocar mensagens sem a notificação de que novas mensagens estão disponíveis;
- O paradigma Síncrono deve ser usado quando há garantia de que a conexão entre dois dispositivos em comunicação será mantida até que, requisição e resposta, sejam trocadas;
- A Notificação Ponto-a-Ponto deve ser usada quando não pode ser garantido que haverá conexão entre duas aplicações comunicantes no momento da produção da mensagem;
- As Filas de Mensagens devem ser usadas quando cada mensagem deve ser processada por apenas um consumidor, independente de qual seja.

Apesar dessas particularidades, a maioria das soluções de middleware atuais impõe um paradigma de comunicação para as aplicações, desconsiderando o fato de que os cenários na computação móvel são extremamente variados, e, portanto, sistemas de middleware devem estar aptos a atender aos diversos requisitos desses cenários. Desta forma, ao invés de impor um único paradigma de comunicação, o middleware proposto oferece às aplicações um conjunto configurável e extensível de paradigmas de comunicação.

De forma a diminuir o espaço ocupado em memória, e, assim, evitar que o middleware torne-se incompatível com dispositivos de capacidades limitadas, como os celulares e *smartphones*, são explorados aspectos comuns entre os paradigmas de comunicação para prover um conjunto base de componentes de software que são compartilhados e reusados por todos os paradigmas descritos anteriormente.

Neste sentido, a arquitetura do middleware proposto foi concebida baseada na abordagem de Linha de Produto de Software (LPS). De acordo com [Clements02], uma linha de produto de software é um conjunto de sistemas de software que compartilham um conjunto de características comuns e gerenciadas, que satisfazem às necessidades específicas de um segmento de mercado e que são desenvolvidos a partir de um conjunto comum de artefatos de maneira pré-estabelecida.

Baseado neste conceito de LPS, as funcionalidades comuns para comunicação através de mensagens são compartilhadas entre os paradigmas de comunicação do middleware proposto, formando o núcleo da linha de produto. Por outro lado, as semânticas de entrega de mensagens específicas de cada paradigma de comunicação são modeladas como funcionalidades opcionais. Desta forma, essas funcionalidades opcionais podem ser selecionadas para fazer parte ou não de cada instância do middleware derivada da linha de produto.

3.2 Contexto e Ambiente de Utilização

Como forma de melhor situar o contexto em que os serviços de middleware abordados nesse trabalho se encaixam, a Figura 9 apresenta a estrutura de middleware definida por [Schantz01], na qual os serviços são divididos em camadas, de acordo com as funcionalidades providas. Seguindo essa estrutura, os serviços providos pelo middleware proposto são relativos a “serviços de distribuição”, os quais definem

modelos de programação distribuída de alto nível, provendo interfaces de programação (*Application Programming Interafaces* – API) e componentes para automatizar e estender as capacidades de rede nativas do sistema operacional, encapsuladas pela camada de middleware de infra-estrutura.

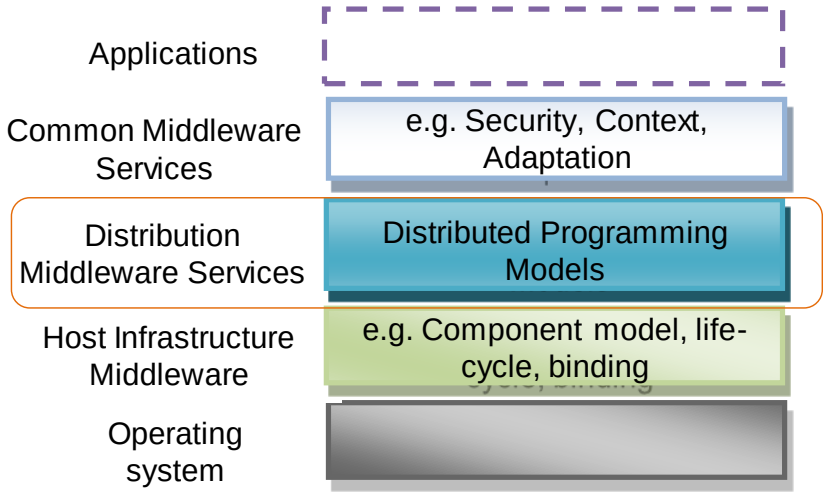

Figura 9 - Estrutura de middleware em camadas [Schantz01]

Seguindo essa estrutura em camadas, aplicações móveis podem beneficiar-se diretamente desses serviços de distribuição, de forma que estas podem ser implementadas sem a necessidade de lidar com protocolos de comunicação de baixo nível e problemas com heterogeneidade de hardware, plataforma e sistema operacional. Além disso, outra camada de middleware pode ser adicionada acima da camada de distribuição, reusando tais serviços e provendo outros serviços comuns para aplicações móveis, tais como ciência de contexto, serviços relacionados a segurança, gerenciamento de grupos de dispositivos e adaptação dinâmica.

Neste sentido, os serviços oferecidos pelo middleware proposto são projetados para dar suporte a aplicações móveis que precisam se comunicar com outras aplicações numa rede móvel local, caracterizando assim um sistema distribuído móvel *ad hoc*, ou móvel infra-estruturado, conforme definido na Seção 2.1. Exemplos de tais aplicações incluem: jogos colaborativos, compartilhamento de mídias, aplicações para campo de batalha militar ou resgates em ambientes remotos. A Figura 10 ilustra o ambiente em que o middleware deve operar.

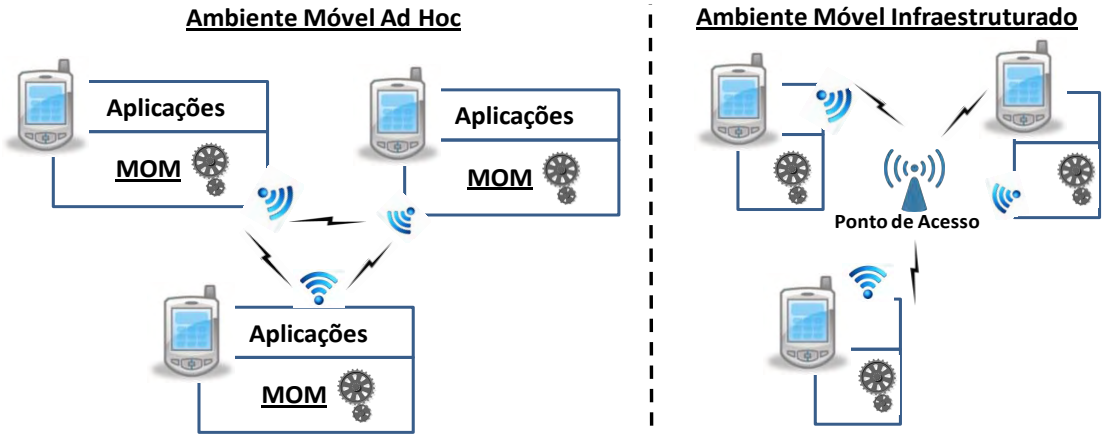

Figura 10 - Ambiente de operação do middleware

Note que na Figura 10, os dispositivos se comunicam diretamente, ou no máximo, através de um ponto de acesso sem fio. Não há necessidade de servidores de grande porte para viabilizar a operação do middleware proposto. Isso porque nem sempre pode ser garantida a existência de uma infra-estrutura fixa, pré-definida e baseada em servidores. Além disso, aplicações móveis são muito diversificadas, e assim, mesmo que haja um ponto de acesso para interligar os dispositivos, a variedade de aplicações que esses dispositivos podem carregar é ilimitada, pois diferentes dispositivos entram e saem dessas redes a todo momento. Assim, é impraticável prever quais tipos de aplicações utilizarão estas redes e disponibilizar servidores, com serviços específicos, para dar suporte a tais aplicações.

Nesse contexto, ao invés do tradicional modelo cliente-servidor, o Multi-MOM inclui, localmente, em cada dispositivo, um conjunto conciso de funcionalidades típicas de servidor (ex.: persistência, descoberta de serviços, encaminhamento de mensagens). Desta forma, o objetivo é dar suporte a um modelo *peer-to-peer*, no qual todos os dispositivos têm papéis similares. Isto é, cada dispositivo pode atuar como produtor de mensagens e/ou consumidor. Não há distinção entre clientes e servidores neste modelo, pois cada dispositivo já inclui funcionalidades típicas de servidor.

Não obstante, há casos em que é desejável a existência de um servidor para dar suporte a um serviço específico, geralmente provido por alguma organização. Exemplos de serviços como esses incluem: guias eletrônicos para ambientes fechados (*indoor*), como museus; disseminação de informações num piso de um prédio, como anúncios dentro de um supermercado, *shopping*, ou atualizações da bolsa de valores. Para casos como esse, o middleware proposto teria que ser implantado num servidor. Embora esse

trabalho não aborde questões específicas para implantação do middleware num servidor de grande porte, a princípio ele pode ser implantado nesses tipos de computadores sem modificações significativas. Nesse caso, para os dispositivos se comunicarem com servidores, o middleware instalado nos dispositivos não precisaria ser modificado, pois a localização física dos serviços providos pelo middleware é transparente, como explicado mais adiante.

3.3 Funcionalidades Comuns entre Paradigmas Baseados em Mensagens

O provimento de múltiplos paradigmas de comunicação permite atender a diversos tipos de aplicações diferentes, permitindo que desenvolvedores escolham o paradigma que mais se adéque ao estilo de comunicação da aplicação em desenvolvimento. Além disso, a abordagem multi-paradigma permite que uma mesma aplicação possa utilizar diversas formas de interação simultaneamente, utilizando diferentes paradigmas de comunicação para as diferentes atividades que ela venha a desempenhar.

Apesar dessas facilidades, a abordagem multi-paradigma levanta duas questões que devem ser consideradas num trabalho que se proponha a utilizá-la: (i) o tamanho do middleware pode crescer demais por oferecer diversos paradigmas de comunicação diferentes; e (ii) o que fazer caso o desenvolvedor de aplicações necessite de um conjunto de paradigmas diferentes dos providos por padrão pelo Multi-MOM.

Visando projetar uma solução que trate as questões levantadas acima, o Multi-MOM foi desenvolvido baseado no conceito de linha de produto de software (LPS). Assim, ao invés de um sistema monolítico, com funcionalidades fixas e predefinidas, o Multi-MOM define um conjunto de funcionalidades núcleo, obrigatórias, e um conjunto de funcionalidades opcionais, variáveis.

As funcionalidades núcleo estão relacionadas às questões que são comuns entre paradigmas de comunicação baseados em mensagens, isto é, funcionalidades que viabilizem a comunicação através de mensagens no contexto da computação móvel. Já as funcionalidades variáveis referem-se aos diferentes paradigmas de comunicação baseados em mensagens que podem ser implementados.

A Figura 11 apresenta o modelo de *features*[1] do Multi-MOM. Este modelo utiliza diagramas de classes da UML e adapta-os para representar partes comuns e variáveis em linhas de produto. Algumas *features* são agrupadas de acordo com seus objetivos. Um estereótipo é utilizado em cada uma das caixas para definir se a *feature* é obrigatória, opcional, ou um agrupamento de funcionalidades. As linhas com um losango no final, que na UML significam composição de classes, nesse modelo significam dizer que uma determinada *feature* faz parte de um grupo de *features*. Mais detalhes sobre a notação utilizada para esse tipo de modelo é descrita em [Gomaa04].

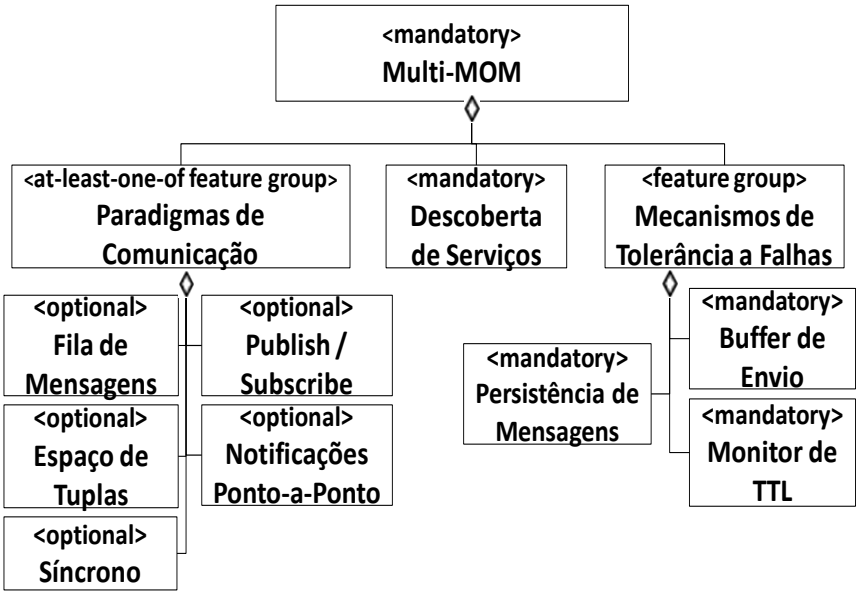

Figura 11 - Modelo de *features* do middleware

O restante desta seção apresenta em detalhes as *features* da Figura 11 que são comuns entre os paradigmas de comunicação, caracterizando o núcleo da LPS. Algumas dessas funcionalidades são utilizadas em MOMs tradicionais, entretanto, aqui é discutido como estas funcionalidades foram adaptadas para lidar com os problemas encontrados no contexto das redes sem fio e dispositivos móveis, pois, como afirma [Mascolo02], implementações tradicionais de MOMs não são suficientes nestes tipos de cenários.

É importante destacar que essas funcionalidades definem um conjunto de facilidades e conceitos comuns que são reusados pelos paradigmas de comunicação

[1] *Feature* é definido como requisito, característica ou função perceptível pelo usuário do sistema e que tem importância direta para este. No contexto deste trabalho, a palavra encontrada na língua portuguesa que mais se adéqua a esse conceito é "funcionalidade".

providos pelo Multi-MOM. Já as funcionalidades relativas especificamente aos paradigmas de comunicação serão descritas na Seção 3.4.

3.3.1 Buffer de Envio

Considerando a baixa largura de banda, alta latência e as freqüentes desconexões das redes sem fio, é essencial a definição de um mecanismo que permita às aplicações continuarem operando mesmo em momentos em que não haja conexão. Nestes casos, as mensagens enviadas ficam temporariamente armazenadas num *buffer* local ao dispositivo, enquanto aguardam serem transmitidas quando a conexão estiver novamente disponível. Entretanto, se houver conexão, a mensagem é entregue imediatamente. Este *buffer* atua como um mecanismo de tolerância a falhas, garantindo assincronismo na transmissão e processamento das mensagens.

O funcionamento deste *buffer* dá-se através de uma estrutura de dados do tipo fila, e de uma *thread* que fica varrendo esta fila. Primeiro, há uma tentativa de envio da mensagem, se não for possível enviar essa mensagem, a mesma é colocada no final da fila do *buffer*. Em paralelo a isso, existe uma *thread* que fica retirando as mensagens da cabeça da fila e tentando enviá-las, e, caso não haja conexão com o destinatário da mensagem, a mesma é colocada novamente no final da fila. Note que, para a aplicação que enviou a mensagem através do middleware, é indiferente se a mensagem foi enviada imediatamente ou colocada no *buffer*.

Esse mecanismo aumenta a confiabilidade da entrega de mensagens, pois permite que aplicações que produzem mensagens possam "disparar e esquecer" (*fire-and-forget*) as mesmas, confiando no middleware para tratar da entrega. Sem a utilização de um mecanismo como esse, a confiabilidade da entrega de mensagens seria caracterizada como *at-most-once*, ou seja, caso não houvesse conexão, a mensagem seria descartada pelo middleware. Já com a utilização do *buffer* de envio, a confiabilidade de entrega é caracterizada como *exactly-once*, ou seja, a mensagem permanece no middleware até que o seu recebimento seja confirmado.

3.3.2 Monitor de Tempo de Vida das Mensagens

Apesar do *buffer* de envio mostrar-se uma maneira eficiente de lidar com as desconexões, nem todo tipo de aplicação é compatível com este modo de tratamento de desconexões. Algumas aplicações podem disseminar mensagens que não tem mais valor

depois de algum tempo. Por exemplo, numa aplicação que envia mensagens com informações de contexto referentes à localização do dispositivo, depois de algum tempo, os usuários podem ter se movido e a informação de localização pode não ter mais valor. Além disso, períodos longos de desconexão podem sobrecarregar a memória do dispositivo móvel, que já é normalmente bastante limitada. Deste modo, o monitor de tempo de vida (*time-to-live* – TTL) das mensagens é o mecanismo responsável por aliviar tais problemas.

Neste sentido, aplicações devem especificar um tempo de vida para as mensagens, definindo o tempo que uma mensagem pode ser mantida pelo middleware na memória do dispositivo enquanto espera ser transmitida. Se o TTL expirar, a mensagem deve ser descartada. Além disso, a definição do TTL previne uma possível utilização descontrolada da memória do dispositivo, causada por mensagens que ficam esperando indefinidamente para serem transmitidas para um dispositivo que saiu da rede e talvez não volte mais.

3.3.3 Persistência de Mensagens

Como dito na Seção 3.3.1, o *buffer* de envio provê uma entrega de mensagem com garantia *exactly-once*. Entretanto, em casos de condições atípicas, como exaustão de recursos do dispositivo, falha de processamento ou dispositivo ser desligado devido ao baixo nível de bateria, mensagens mantidas pelo dispositivo podem ser perdidas. O middleware proposto suaviza este tipo de problema oferecendo a possibilidade das mensagens serem marcadas como persistentes, e, assim, serem armazenadas em memória não-volátil.

É importante destacar que é necessário um esforço extra para garantir que tais mensagens sejam conservadas nesses casos de falha. Enquanto ganha-se em confiabilidade, perde-se em desempenho. Assim, é importante que a escolha de marcar as mensagens como persistente considere o TTL das mesmas, pois a persistência é mais adequada em casos de mensagens com um TTL longo ou infinito, como geralmente são as mensagens dos espaços de tuplas, que são mantidas pelo middleware até que alguma aplicação venha buscá-la e remova-a do espaço de tuplas.

As mensagens marcadas como persistentes são mantidas na memória principal e na memória não-volátil do dispositivo simultaneamente. Para obter um melhor desempenho, as mensagens são acessadas diretamente da memória principal sempre que

possível. Assim, a memória não-volátil é acessada somente quando é necessário sincronizar alguma mudança ocorrida na mensagem da memória RAM, como remoção, ou alteração.

3.3.4 Localização de Serviços

Uma aplicação que deseja publicar eventos precisa saber onde está o canal de eventos, ou, de forma similar, para escrever em um espaço de tuplas remoto é necessário saber o endereço de rede deste serviço. Em sistemas tradicionais de MOM, tal como Java Message Service (JMS) [Sun02], um serviço de diretório estático é previamente conhecido por todas as aplicações da rede. Nestes casos, as aplicações registram seus serviços nessa entidade central, e, sempre que uma aplicação está procurando serviços, acessa o serviço de diretório que mantém o registro de todos os serviços de mensagens da rede. A Figura 12 ilustra o registro de serviços centralizado.

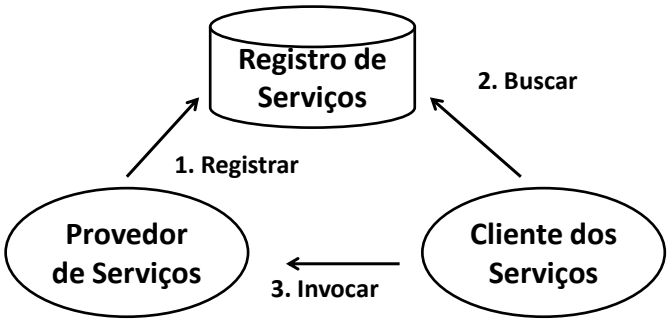

Figura 12 - Registro de serviços centralizado

Entretanto, em redes móveis, não se pode assumir que os dispositivos encontrar-se-ão numa rede que possui um servidor central que mantém um registro de serviços. Por esta razão, os serviços de mensagens providos pelo middleware são anunciados na rede através de um protocolo de descoberta de serviços, similar ao proposto em [Rellermeyer07]. Este protocolo funciona como um registro de serviços distribuído, pois cada dispositivo da rede mantém o registro dos seus próprios serviços.

Para buscar serviços, o middleware, a pedido da aplicação, envia uma mensagem em *multicast* para a rede. Esse endereço *multicast* é padrão para todos que utilizam o protocolo de descoberta de serviços. Dispositivos que oferecem serviços de mensagens estarão escutando naquele endereço *multicast*, e, ao receber uma requisição, responderão em *unicast* indicando os endereços dos seus serviços. A Figura 13 ilustra o registro de serviços descentralizado, utilizado pelo Multi-MOM.

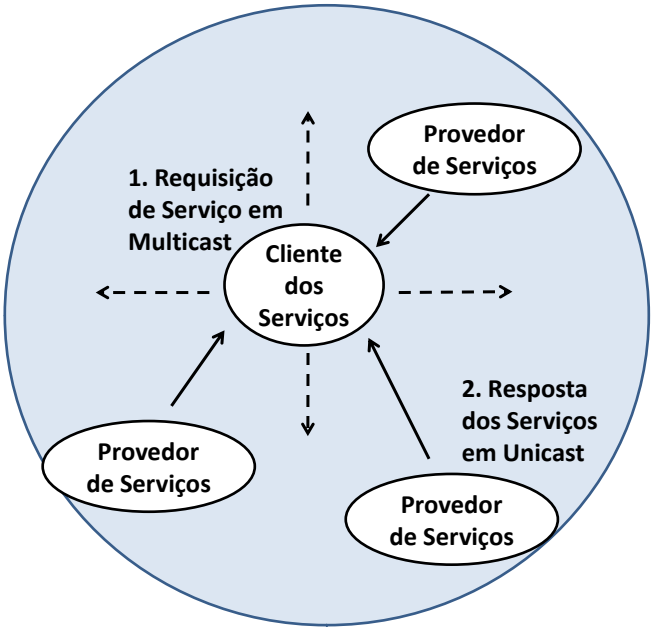

Figura 13 - Registro de serviços descentralizado

Este modelo de descoberta de serviços permite que aplicações encontrem serviços na rede sem nenhum conhecimento prévio dos dispositivos ou serviços disponíveis naquela rede. Além disso, o registro descentralizado garante refletir a real disponibilidade dos serviços da rede a qualquer momento.

3.4 Paradigmas de Comunicação Orientados a Mensagens

As funcionalidades apresentadas na Seção 3.3 representam facilidades comuns para lidar com restrições de cenários de mobilidade. Essas funcionalidades formam a base para possibilitar comunicação através de mensagens em diversos paradigmas de comunicação orientados a mensagens. Assim, com o objetivo de reduzir o espaço ocupado em memória e favorecer o reuso de código, essas funcionalidades comuns de sistemas de mensagens são integradas e reusadas entre os paradigmas de comunicação oferecidos.

Desta forma, os componentes que irão prover as funcionalidades específicas de cada paradigma de comunicação apóiam-se nessas funcionalidades comuns e acrescentam apenas o código relacionado à semântica de troca de mensagens específica de cada paradigma. Nesta seção são especificados os paradigmas de comunicação considerados pelo middleware proposto, juntamente com uma discussão das principais decisões de projeto relacionadas aos mesmos.

3.4.1 Fila de Mensagens

Em implementações tradicionais de filas de mensagens, as filas são criadas de forma administrativa, isto é, o administrador do sistema precisa explicitamente criar cada fila. Isso é adequado em redes fixas, onde sabe-se previamente quais aplicações estarão executando naquela rede específica. Em situações de alta dinamicidade, como são os cenários de mobilidade, é comum que dispositivos criem redes temporárias, ou utilizem alguma rede desconhecida para executar variados tipos de aplicações. Neste sentido, as filas de mensagens do middleware proposto são criadas de forma dinâmica e sob demanda pelas aplicações que utilizam o middleware. Para que o middleware possa dar suporte a diversas filas de uma única vez, cada fila criada recebe um nome específico definido pela aplicação. Assim, uma aplicação pode fazer uso de várias filas, ou mesmo pode haver diversas aplicações distintas na rede utilizando serviços de filas distintos.

As filas atuam como um serviço intermediário para comunicação entre aplicações distribuídas, podendo estar executando em qualquer dispositivo da rede. A Figura 14 ilustra as possibilidades de interação utilizando filas de mensagens, mostrando um exemplo de cenário onde vários dispositivos interagem lendo e escrevendo em diferentes filas de mensagens. As aplicações podem acessar filas, isto é, ler e escrever nas filas, localizadas no próprio dispositivo ou em algum outro dispositivo da rede. A localização é transparente para as aplicações, pois a referência a tais serviços é obtida através do protocolo de localização de serviços, discutido anteriormente.

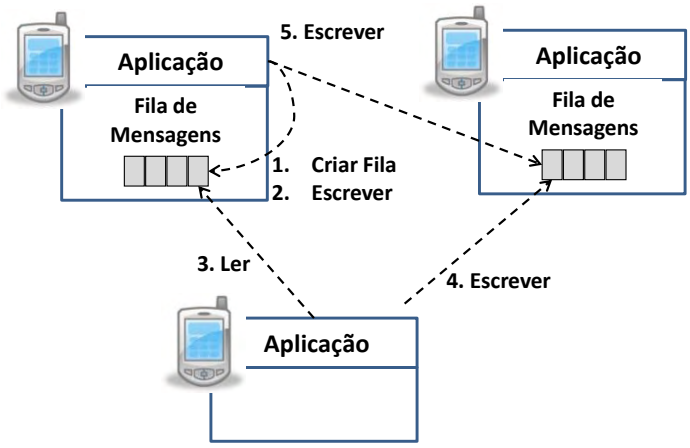

Figura 14 - Interações utilizando Filas de Mensagens

Além disso, um mesmo dispositivo pode prover mais de um serviço de filas simultaneamente. Entretanto, com intuito de minimizar a utilização de memória RAM e

processamento, internamente, cada instância do middleware utiliza uma única estrutura de dados simulando o comportamento de várias filas. A Figura 15 ilustra essa estrutura de dados. Neste exemplo, o middleware provê dois serviços de filas simultaneamente. Quando uma aplicação requisita ao middleware para obter uma mensagem de uma determinada fila, neste exemplo a fila "Chat-Messenger", o middleware busca na estrutura de dados a primeira mensagem que aparece pertencendo àquela fila especificada pela aplicação.

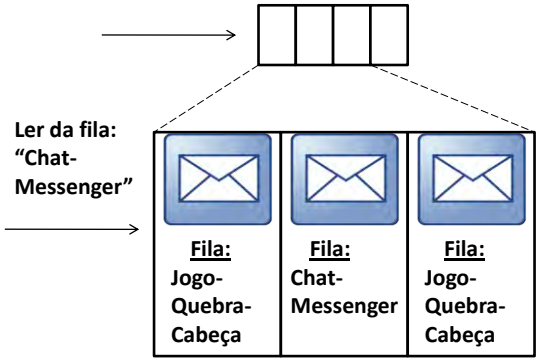

Figura 15 - Estrutura de dados simulando várias filas

Por fim, devido à escassez de recursos dos dispositivos móveis, o intervalo de tempo que as mensagens são mantidas pelas filas é também controlado para evitar exaustão de recursos. Esse controle é feito através do TTL das mensagens, que especifica quanto tempo uma mensagem vai passar na fila até ser lida, ou então descartada.

3.4.2 Espaço de Tuplas

Espaços de tuplas podem ser vistos como uma memória compartilhada possível de ser acessada por várias aplicações em rede. Cada tupla tem um conjunto de propriedades que diferencia umas das outras. Essas propriedades são definidas pelas aplicações que as inserem no espaço de tuplas. As aplicações que buscam tuplas passam como parâmetro um conjunto de propriedades desejadas.

Um espaço de tuplas pode ser criado em qualquer dispositivo da rede e servir como um meio de comunicação entre as aplicações distribuídas. De forma similar às filas de mensagens, espaços de tuplas funcionam como serviços intermediários, e são criados sob demanda pelas aplicações, podendo um mesmo dispositivo prover mais de um espaço de tuplas. Cada espaço tem seu nome definido pela aplicação que o criou,

assim duas aplicações distintas poderão utilizar dois serviços de espaço de tuplas distintos, sem que as mensagens de uma aplicação interfiram em outra aplicação que não tenha relação alguma com a primeira. É importante destacar que a associação de mensagens a nomes de serviços específicos é válida também para todos os outros paradigmas discutidos neste trabalho.

As aplicações podem utilizar serviços de espaço de tuplas executando localmente, no mesmo dispositivo, ou executando num dispositivo remoto, de forma similar às filas mensagens. Assim, as possibilidades de interação através da rede entre aplicações e serviços de espaços de tuplas são similares às de filas de mensagens, ilustradas anteriormente, na Figura 14. A diferença está na semântica utilizada para escolha das mensagens, enquanto os serviços de filas retornam as mensagens na cabeça da fila, os espaços de tuplas permitem que seja especificado um conjunto de propriedades para obtenção da mensagem desejada.

Com relação à permanência das mensagens no espaço de tuplas, é comum em sistemas de espaços de tuplas tradicionais que as tuplas fiquem armazenadas indefinidamente, isto é, uma vez que a tupla é escrita no espaço de tuplas, ela permanece armazenada nesse espaço até que alguma aplicação retire-a. Entretanto, nos espaços de tuplas do Multi-MOM, o intervalo de tempo que a mensagem permanece no espaço de tuplas é controlado pelo TTL da mensagem.

Considerando a alta dinamicidade dos cenários de mobilidade, tal controle é importante pelo fato de que, dispositivos podem entrar na rede, escrever suas tuplas no espaço compartilhado e então sair da rede. Assim, se outras aplicações participantes da rede não souberem da existência dessas tuplas, elas podem ficar esquecidas ocupando espaço na memória do dispositivo. Se o TTL das mensagens não fosse monitorado pelo middleware, essas mensagens ficariam armazenadas indefinidamente, consumindo uma memória que já é bastante restrita.

3.4.3 Publish/Subscribe

Este paradigma provê troca de mensagens através da publicação de eventos e do registro de interesse em determinados tipos de eventos. Quando uma aplicação cria um canal de eventos, o middleware fica responsável por anunciar o mesmo na rede, manter o registro de aplicações interessadas e encaminhar as mensagens para os interessados assim que as mensagens são publicadas.

Como explicado na Seção 2.3, sistemas publish/susbcribe podem adotar duas formas de inscrição no canal de eventos: baseada em tópicos ou baseada em conteúdo. As inscrições baseadas em conteúdo são normalmente mais flexíveis, já que cada assinante especifica um conjunto de filtros específicos, de acordo com seus interesses. Entretanto essa flexibilidade gera uma maior carga computacional, pois é necessário guardar o conjunto de filtros especificados por cada assinante, bem como, para cada evento publicado, é necessário verificar correspondência entre o evento em questão e cada uma das inscrições registradas.

Por outro lado, nas inscrições baseadas em tópicos, uma simples tabela com a referência para os assinantes é suficiente. Por esta razão, considerando as restrições dos dispositivos móveis, decidiu-se dar suporte somente às inscrições baseadas em tópicos. Apesar dessa restrição, o projeto extensível do Multi-MOM permite que novos comportamentos, como a inscrição baseado em conteúdo, por exemplo, sejam facilmente acrescentados, como será detalhado mais a frente neste trabalho.

Seguindo o modelo dos paradigmas anteriores, aplicações podem publicar e inscrever-se em canais de eventos executando no seu próprio dispositivo ou em algum dispositivo da rede. A localização de tais serviços é sempre obtida através do protocolo de localização de serviços. Assim, para publicar mensagens, aplicações podem criar seu próprio canal de eventos ou utilizar algum canal existente na rede. A Figura 16 ilustra exemplos de interações entre aplicações e canais de eventos numa rede, mostrando que o canal de eventos é criado por uma aplicação, mas a partir daí, outras aplicações remotas podem utilizá-lo tanto para enviar, como para receber mensagens.

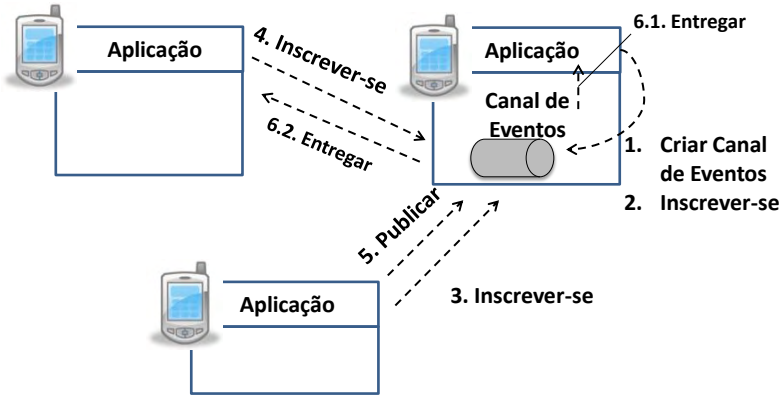

Figura 16 - Interações entre aplicações e canais de eventos

Além disso, quando um canal de eventos não tem conexão com um determinado assinante, o middleware fica responsável por retransmitir aquela mensagem num momento oportuno. Desta forma, o *buffer* de envio dá suporte ao paradigma publish/subscribe em dois momentos. Primeiro, quando uma aplicação deseja publicar uma mensagem num canal de eventos remoto. Segundo, quando o canal de eventos, atuando como serviço intermediário, recebe uma mensagem para ser encaminhada aos seus assinantes, e um ou mais desses assinantes está fora de alcance.

Por fim, o TTL das mensagens é também monitorado dentro do canal de eventos, pois caso isso não acontecesse, a memória do dispositivo executando o canal de eventos ficaria rapidamente inundada de mensagens não entregues a determinados assinantes naquele canal, devido aos mesmos poderem sair da rede a qualquer momento.

3.4.4 Mensagens Síncronas

Este paradigma utiliza um modelo similar ao RMI e RPC, só que ao invés de fazer invocações diretas aos métodos de uma aplicação remota, é enviada uma mensagem de requisição, e retornada uma outra mensagem como resposta. Comparado aos outros paradigmas, este modelo é o que utiliza uma semântica de comunicação mais parecida com as chamadas de operações dentro de uma mesma aplicação, pois permite enviar uma requisição e receber um retorno imediatamente.

No entanto, é também o paradigma que gera maior acoplamento entre as partes comunicantes. Enquanto ele oferece a vantagem de, ao enviar uma mensagem, obter uma resposta instantânea, perde-se na questão de tolerância a falhas. Isto acontece porque as mensagens não passam pelo *buffer* de envio, já que a comunicação deve ser instantânea. Assim, se não houver conexão com a outra parte comunicante, a aplicação recebe uma exceção indicando o ocorrido. Devido a esse estilo de comunicação síncrona, esse paradigma não utiliza os recursos de persistência e TTL das mensagens, pois a mensagem nunca precisa ficar armazenada no middleware. Note que este paradigma é o único que não utiliza os mecanismos de tolerância a falhas (*buffer* de envio, TTL e persitência).

Para utilizar este paradigma, uma aplicação “servidora” deve criar o serviço síncrono junto ao middleware, passando como parâmetro o nome do serviço e uma referência de *callback* para que o middleware contate a aplicação ao receber mensagens de outros dispositivos. Diferentemente dos paradigmas discutidos anteriormente nesta

seção, em que os serviços eram agentes intermediários utilizados para comunicação entre diversas aplicações distribuídas, os serviços de mensagens síncronas são ligados diretamente a uma aplicação específica. Assim, uma mensagem enviada a um serviço de mensagem síncrona é sempre recebida pela mesma aplicação, isto é, pela aplicação que criou o serviço. A Figura 17 ilustra as interações entre a aplicação criadora do serviço, e as outras aplicações que enviam requisições àquele serviço.

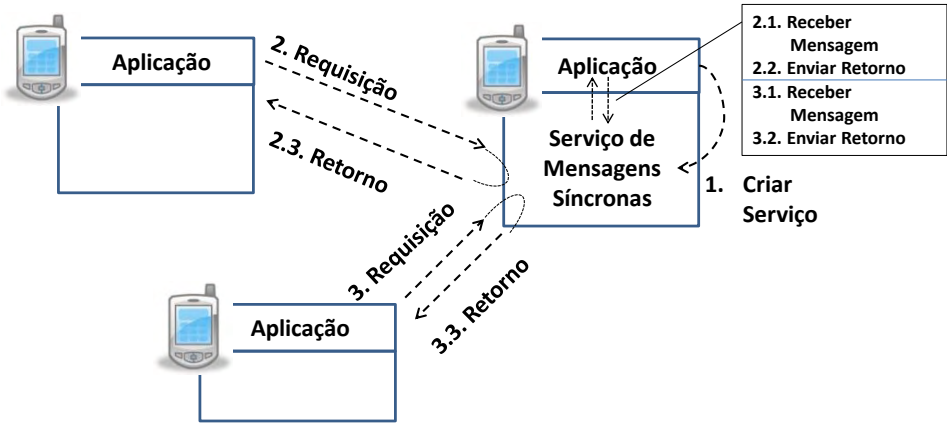

Figura 17 - Interações utilizando mensagens síncronas

Quando aplicações remotas enviarem mensagens para esse serviço, o middleware, através da referência de *callback*, repassa a mensagem diretamente para aquela aplicação, que extrai o conteúdo da mensagem e retorna uma outra mensagem. Note que, esse paradigma pode facilmente simular a invocação de operações remotas providas pelo RMI, onde o conteúdo da mensagem especificaria o nome do método a ser invocado e os parâmetros necessários, enquanto a mensagem de retorno carregaria o objeto retornado pela operação.

3.4.5 Notificações Ponto-a-Ponto

De forma similar ao paradigma síncrono, neste, as aplicações que produzem mensagens sabem exatamente quem vai consumir aquela mensagem, pois cada serviço de comunicação destes paradigmas é associado somente a uma aplicação, não havendo um serviço intermediário que de alguma forma redistribui as mensagens. Entretanto, em relação ao síncrono, este paradigma oferece a vantagem de que, se não houver conexão com a outra parte, a mensagem fica no *buffer* de envio para reenvio posterior, e a aplicação que produziu a mensagem continua o seu fluxo de execução normalmente.

Note que, diferente do paradigma síncrono, neste é possível utilizar os recursos de tolerância a falhas (*buffer* de envio, TTL e persistência), pois a aplicação não espera bloqueada até que a mensagem seja enviada, ela passa a mensagem para o middleware e espera que este possa enviar a mensagem assim que possível, continuando seu fluxo normal de execução.

O esquema de interação entre aplicações e serviços de notificações ponto-a-ponto é similar ao ilustrado para mensagens síncronas, na Figura 17. A diferença é que não existe uma mensagem de retorno, apenas a mensagem de requisição. Desta forma, esse paradigma pode ser utilizado para dividir uma invocação síncrona em duas invocações assíncronas: a primeira enviada do cliente para a aplicação servidora – acompanhada dos parâmetros de invocação e uma referência remota para a aplicação cliente – e a segunda enviada a partir da aplicação servidora em resposta a primeira invocação do cliente. Opcionalmente, esse esquema pode facilmente ser alterado para fazer com que o lado servidor retorne diversas mensagens em resposta a uma única requisição, como descrito em [Eugster03].

3.5 Arquitetura Integrada

Ao invés de uma arquitetura monolítica, a arquitetura do Multi-MOM é baseada no conceito de linha de produto de software (LPS). Neste sentido, a arquitetura do Multi-MOM explora as funcionalidades comuns dos paradigmas de comunicação baseados em mensagens e representa-as como "componentes núcleo". Por outro lado, toda funcionalidade exclusivamente relacionada a algum paradigma de comunicação é modelada em componentes opcionais/variáveis. A Figura 18 ilustra a arquitetura projetada para dar suporte à solução proposta. Os componentes opcionais estão representados dentro do quadrado pontilhado[2].

Por meio desta arquitetura, é possível que dispositivos de recursos limitados, como celulares e *smartphones*, possam carregar dois ou mais desses paradigmas de comunicação utilizando bem menos recursos do que se tivessem que executar plataformas de middleware separadas para cada paradigma. Os benefícios dessa abordagem incluem:

[2] No decorrer do texto, bem como nas figuras, alguns termos encontram-se em língua inglesa devido à especificação e implementação do middleware terem sido escritos nesta língua.

- Uma única plataforma de middleware capaz de dar suporte a diversos tipos de aplicação da computação móvel;
- O middleware tende a tornar-se mais reusável do que uma solução baseada num único paradigma de comunicação;
- Desenvolvedores de aplicações têm a possibilidade de escolher qual o paradigma mais adequado para suas aplicações, podendo inclusive utilizar dois ou mais desses paradigmas numa única aplicação sem um custo adicional considerável em termos de recursos computacionais;

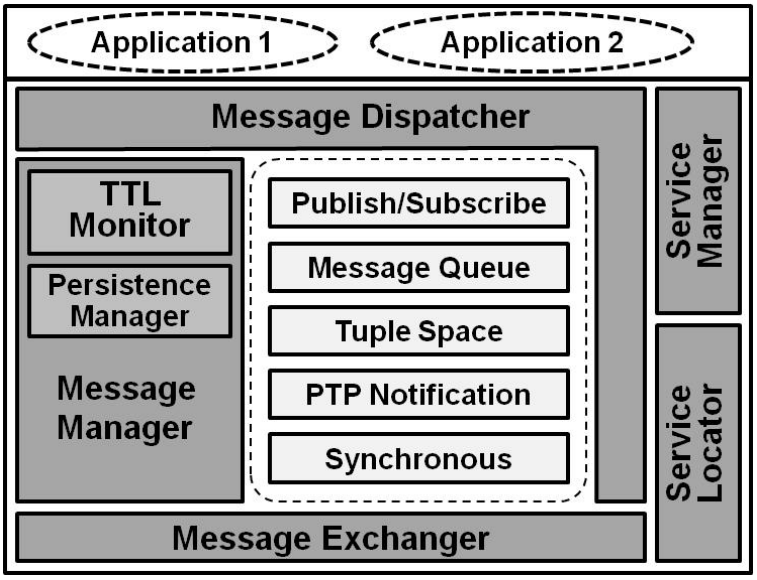

Figura 18 - Arquitetura integrada do Multi-MOM

Na Figura 18, o componente *service manager* oferece uma API para as aplicações que utilizam o middleware que possibilita a criação e busca de serviços na rede. Os métodos dessa API, bem como de toda API oferecida pelo middleware serão especificados mais adiante. A criação de serviços é registrada tanto no componente que implementa o paradigma de comunicação desejado, quanto no componente *service locator*. Os componentes dos paradigmas de comunicação mantêm os registros dos serviços criados pelas aplicações para controle interno. Por outro lado, o componente *service locator* fica responsável por anunciar tal serviço na rede.

Além de anunciar na rede os serviços criados pelas aplicações locais, o *service locator* também é responsável por descobrir serviços providos por outros dispositivos da rede. Note que esses dois componentes, em conjunto, implementam a funcionalidade de localização de serviços, mostrada no modelo de *features* da Figura 11.

O componente *message dispatcher* é responsável por desacoplar os componentes núcleo do middleware, dos componentes opcionais, que são específicos de cada paradigma de comunicação. Para isso, ele utiliza mecanismos de reflexão para invocar operações nos componentes opcionais, sem ter que estar diretamente acoplado a estes. Esse componente será detalhado na próxima seção.

Ainda com relação à Figura 18, a entidade central desta arquitetura é o componente *message manager*. Internamente, ele implementa uma única estrutura de dados que mantém todas as mensagens gerenciadas pelo middleware. Para cada mensagem, o *message manager* guarda também o paradigma associado àquela mensagem. Assim, os componentes que implementam os paradigmas de comunicação (ex.: componente *espaço de tuplas*) não precisam implementar uma estrutura de dados para gerenciar as mensagens que eles manipulam. Ao invés disso, quando esses componentes precisam acessar suas mensagens, eles invocam o *message manager* passando como parâmetro o paradigma de comunicação e o nome do serviço desejado.

Essa entidade central é importante para evitar que seja necessário gerenciar diferentes estruturas de dados para cada paradigma, ou possivelmente para cada serviço criado pelas aplicações. Assim, cada componente que dá suporte a um determinado paradigma de comunicação tem que incluir apenas o código necessário para lidar especificamente com a sua semântica de entrega de mensagens, e não código relativo ao armazenamento das mensagens.

Como ilustrado na Figura 18, o *message manager* é um componente composto por outros dois componentes. Quando uma mensagem é marcada como persistente, ele utiliza o *persistence manager* para armazenar a mensagem num meio de armazenamento não-volátil. A cópia da mensagem que é armazenada de forma persistente sempre fica em sincronia com a mensagem da memória RAM, assim, se a mensagem for modificada ou apagada, o mesmo acontecerá com a cópia que está na memória não-volátil. Note que o componente *persistence manager* é responsável por implementar a funcionalidade “persistencia de mensagens”, mostrada no modelo de *features* da Figura 11.

Além disso, o *TTL monitor* implementa a *feature* relativa ao monitoramento de tempo de vida das mensagens, mostrada no modelo de *features* da Figura 11. Esse componente verifica se as mensagens ainda estão válidas ou devem ser descartadas.

Esta verificação acontece de duas maneiras. Primeiro, sempre que algum componente requisita uma mensagem ao *message manager*, antes de entregar a mensagem, ele verifica se a mesma ainda é válida. Se não for, a mensagem não fará parte do conjunto de mensagens retornadas. A segunda forma de verificação é periodicamente checar todas as mensagens armazenadas.

Essa verificação periódica é implementada numa *thread* separada, que faz a verificação e depois fica em estado de espera por um intervalo de tempo. Esse tempo precisa ser curto o suficiente para não deixar acumular uma grande quantidade de mensagens expiradas, mas por outro lado precisa ser longo o suficiente para não desperdiçar processamento e bateria do dispositivo fazendo essa checagem a todo momento. Neste sentido, decidiu-se que um intervalo de tempo adequado para essa verificação é de 5 em 5 minutos. Assim, as mensagens expiradas que não forem descartadas a tempo por essa *thread* de checagem, serão descartadas se precisarem ser recuperadas neste espaço de tempo entre a sua expiração e o momento da nova checagem.

Por fim, os componentes da parte de baixo da Figura 18 (*service locator* e *message exchanger*) são os únicos que interagem diretamente com funcionalidades específicas de rede, tal como conexões de sockets. O *message exchanger* utiliza sockets para implementar a comunicação nas trocas de mensagens entre dispositivos.

A Tabela 1 mostra o mapeamento entre as funcionalidades especificadas no modelo de *features* da Figura 11 com os componentes especificados na arquitetura de alto nível da Figura 18.

Tabela 1 - Mapeamento de *features* para componentes

Funcionalidade do Modelo de *Features*	**Componente da Arquitetura**
Localização de Serviços	*Service manager / Service Locator*
Monitor de TTL	TTL Monitor
Persistência de Mensagens	Persistence Manager
Buffer de Envio	Message Manager
Paradigmas de comunicação	Um componente para cada paradigma de comunicação

3.6 Estrutura das Mensagens

O middleware proposto utiliza o conceito de mensagens como elemento fundamental, o que leva a aplicações que são estruturadas baseadas em seqüências de trocas de mensagens. Como explicado no Capítulo 2, mensagens podem carregar diversos tipos de informações, como estado das aplicações, informações de contexto, requisições de serviço, notificação de eventos, entre outros. A estrutura das mensagens definidas no Multi-MOM tem por objetivo principal possibilitar a criação de mensagens que possam transportar tipos de dados heterogêneos, de acordo com as necessidades de cada aplicação, bem como definir uma estrutura que seja reusável entre diversos paradigmas de comunicação baseados em mensagens.

Assim, as mensagens do Multi-MOM são entidades de leve carga computacional constituídas de um cabeçalho, um conjunto de propriedades e um corpo. O Multi-MOM define uma estrutura geral para as mensagens, e mais dois subtipos para esta estrutura: um para mensagens síncronas, e outro para os outros paradigmas definidos, que são considerados assíncronos. A Figura 19 ilustra essas estruturas.

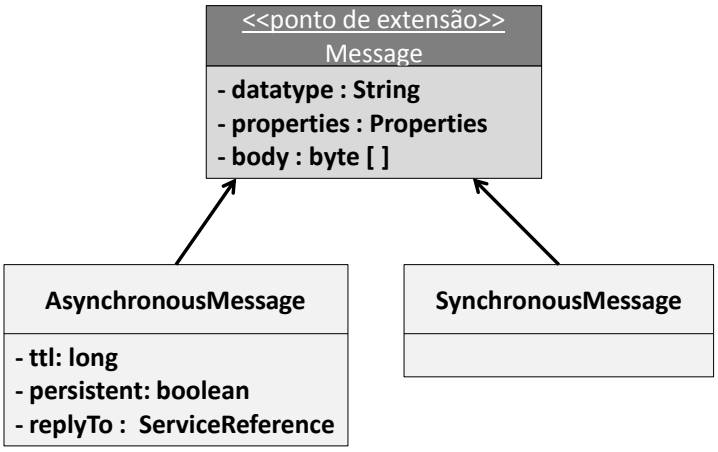

Figura 19 - Estrutura das mensagens

Além desses tipos, um novo paradigma de comunicação que venha a ser adicionado posteriormente ao middleware pode definir o seu próprio subtipo de mensagem, que pode herdar diretamente do objeto *Message*, ou a partir de algum dos subtipos já definidos. A mensagem básica (*message*) define os seguintes campos:

Datatype. Mensagens podem carregar vários tipos de dados, de acordo com as necessidades de cada aplicação. Assim, esse campo especifica que tipo de dado será encontrado no corpo da mensagem. Esse campo é baseado no *MIME type* encontrado no

cabeçalho de vários protocolos da internet, como o HTTP e o SMTP. Exemplos de tipos de dados incluem: image/jpeg, audio/mpeg, text/xml. Além disso, se a aplicação estiver passando um objeto serializado no corpo da mensagem, ela pode escrever nesse campo o nome completo da classe associada ao objeto, para que a aplicação consumidora saiba exatamente como tratá-la.

Proprierties. Esse campo é comum em sistemas de MOM, como em [Sun02]. Ele provê um mecanismo para que sejam adicionados campos de cabeçalho específicos de cada aplicação. No caso de comunicação em grupo, como no paradigma publish/subscribe, as propriedades podem ser usadas para que as aplicações possam filtrar as mensagens que recebem sem precisar ler o corpo inteiro da mensagem. Além disso, o paradigma espaço de tuplas utiliza esse campo para diferenciar as mensagens umas das outras no momento da recuperação de mensagens do espaço de tuplas. A aplicação que produz a mensagem define um conjunto de propriedades, e a aplicação consumidora busca as tuplas especificando as propriedades desejadas.

Body. Este campo corresponde ao corpo da mensagem, ou seja, os dados sendo enviados de uma aplicação a outra. O middleware é indiferente aos tipos de dados sendo transportado no corpo da mensagem, tratando esses dados simplesmente como um arranjo de bytes genérico. A interpretação dos dados no corpo da mensagem é feita pelas aplicações comunicantes, as quais podem utilizar os dois campos descritos anteriormente para facilitar essa interpretação.

Os paradigmas de comunicação não podem utilizar diretamente o objeto *Message*, pois esse é um tipo abstrato. Ao invés disso, os paradigmas de comunicação devem utilizar algum dos subtipos já definido no Multi-MOM, ou, opcionalmente, criar seu próprio subtipo de *Message*. Assim, por padrão o Multi-MOM define duas estruturas que estendem o objeto mensagem genérico:

- *AsynchronousMessage:* Define mais três campos: (i) TTL da mensagem; (ii) um valor booleano indicando se a mensagem deve ser tratada como persistente; e (iii) uma referência para um outro serviço para onde a aplicação que consumiu a mensagem pode enviar sua resposta, caso seja necessário, como em [Sun02]. Esse último campo é importante já que na comunicação assíncrona, diferentemente da comunicação síncrona, não é possível enviar uma resposta utilizando a mesma conexão.

- *SynchronousMessage:* Utilizado pelo paradigma Síncrono, não define campos extras, além dos já definidos em *Message*. Entretanto, a definição desse tipo é necessária para prover um subtipo concreto de mensagem para o paradigma síncrono, já que *Message* é um tipo abstrato. Este subtipo de mensagem não precisa de nenhum dos campos definidos em *AsynchronousMessage*, pois mensagens síncronas não ficam armazenadas no middleware, já que a comunicação síncrona é instantânea, e caso não haja conexão, é retornada uma exceção imediatamente, ficando a cargo da aplicação tentar retransmitir mais tarde.

É importante destacar que esses tipos de mensagens não são instanciados diretamente pelas aplicações clientes, ao invés disso, estas chamam a operação *createMessage()* nos componentes específicos de cada paradigma de comunicação, os quais automaticamente retornam o tipo de mensagem utilizado por aquele paradigma. Por exemplo, instâncias de *AsynchronousMessage* e *SynchronousMessage* são retornadas respectivamente a partir dos componentes de filas de mensagens e paradigma síncrono.

3.7 Referências aos Serviços

Como dito anteriormente, a localização dos serviços providos pelo Multi-MOM é transparente às aplicações que utilizam o middleware, pois as mesmas obtêm referências a tais serviços através da funcionalidade de localização de serviços do middleware. Para isso, o middleware define a classe *ServiceReference*, que encapsula todas as informações necessárias para se obter acesso a um determinado serviço. A Figura 20 mostra os campos que constituem a classe de referência a serviços.

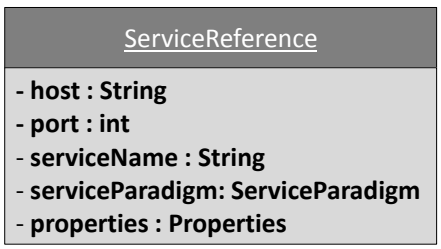

Figura 20 - Classe ServiceReference

O campo *host* indica o endereço do dispositivo na rede, e o campo *port* indica o número da porta à qual o middleware está associado. Note que as aplicações não têm

como escolher os valores desses dois campos, pois o *host* é atribuído de acordo com a rede que o dispositivo se encontra. Já a porta utilizada pelo middleware por padrão é a 5000. Por outro lado, os outros campos são todos escolhidos pela aplicação que cria o serviço. O campo *serviceName* indica o nome escolhido pela aplicação para o serviço criado. Já o campo *serviceParadigm* indica o paradigma de comunicação associado àquele serviço. Observe que o paradigma é indicado pelo tipo *ServiceParadigm*, que será detalhado mais adiante.

Por sua vez, o campo *properties* permite que seja atribuído um conjunto de propriedades aos serviços, de forma que facilite sua identificação. Esse campo é mais importante para casos em que os nomes dos serviços coincidam, seja intencionalmente ou não. Nesses casos, o campo *properties* pode dar mais informações a respeito de um determinado serviço, facilitando a seleção do serviço mais adequado.

Por exemplo, imagine que uma aplicação de jogo *multi-player* é criada na rede. O nome dado ao serviço que distribui as mensagens dessa aplicação pode ser o próprio nome da aplicação, para facilitar sua busca. Entretanto, se houver outro grupo de pessoas jogando o mesmo jogo na rede, é necessário diferenciar os serviços, para que os jogadores, ao entrar numa partida, saibam em qual grupo eles estão ingressando. Assim, além do nome do serviço, um conjunto de propriedades é definido para fornecer informações adicionais sobre cada serviço. Assim, torna-se possível diferenciar dois serviços distintos que estão dando suporte a aplicações do mesmo tipo.

É importante destacar que, sempre que uma aplicação busca por serviços na rede, ela recebe uma referência, ou um arranjo de referências do tipo *ServiceReference*. O conjunto de referências retornado dependerá dos parâmetros especificados pela aplicação na busca por serviços. Esses parâmetros podem: não especificar nada, indicando que devem ser retornados todos os serviços disponíveis; especificar um, ou mais de um dos campos *serviceName*, *serviceParadigm* e *properties*.

Os objetos do tipo *ServiceReference* são também utilizados sempre que uma aplicação vai se comunicar com um serviço remoto, seja para enviar uma mensagem àquele serviço, inscrever-se num canal de eventos ou buscar uma mensagem numa fila de mensagens, por exemplo. Assim, a aplicação ao invocar o middleware para comunicação com serviço remoto, sempre vai passar, como um dos parâmetros, um

objeto do tipo *ServiceReference*, para que o middleware possa identificar o destinatário daquela comunicação.

3.8 Projeto Estrutural

Com a arquitetura definida na Seção 3.5, o próximo passo no desenvolvimento do Multi-MOM foi a especificação do projeto estrutural. De acordo com [Gomaa04], um projeto baseado em linha de produto proporciona a reusabilidade do projeto ou do sistema ou subsistema implementado através de uma coleção de classes concretas e abstratas e suas colaborações. Ele provê uma estrutura geral, a partir da qual podem ser derivadas soluções customizadas, de acordo com as necessidades das aplicações.

Assim, o projeto especificado nesta seção encontra-se documentado em UML [Booch05]. Os diagramas estruturais existem para visualizar, especificar, construir e documentar os aspectos estáticos de um sistema, ou seja, a representação de seu esqueleto e estruturas relativamente estáveis.

As estruturas das classes do Multi-MOM são descritas de acordo com suas responsabilidades, na seguinte ordem: classes da API, classes de transporte de dados, classes internas, classes de interação com a rede e, por fim, as classes de ponto de extensão.

As classes de interação com a rede são *ServiceLocator* e *MessageExchanger*, as quais foram mapeadas diretamente de dois componentes com esse mesmo nome da arquitetura de alto nível, especificada na Seção 3.5. Já que tais classes correspondem exatamente a esses dois componentes que tiveram sua função definida na Seção 3.5, elas não serão descritas novamente nesta seção. Já as outras classes são descritas a seguir.

3.8.1 Classes da API

As classes que definem a API do Multi-MOM estão especificadas na Figura 21. Note que a API do middleware está dividida em classes que pertencem ao lado da aplicação, e classes que pertencem ao lado do middleware.

É importante destacar que as aplicações e o middleware executam em processos distintos do sistema operacional. A execução do middleware num processo independente traz duas vantagens: (i) a possibilidade de adotar uma única instância do

middleware para atender a várias aplicações executando em um mesmo dispositivo; (ii) o isolamento do middleware em relação às aplicações, evitando que erros ou falhas em uma aplicação interrompam o funcionamento do middleware (e vice-versa), que pode estar servindo a outras aplicações em paralelo.

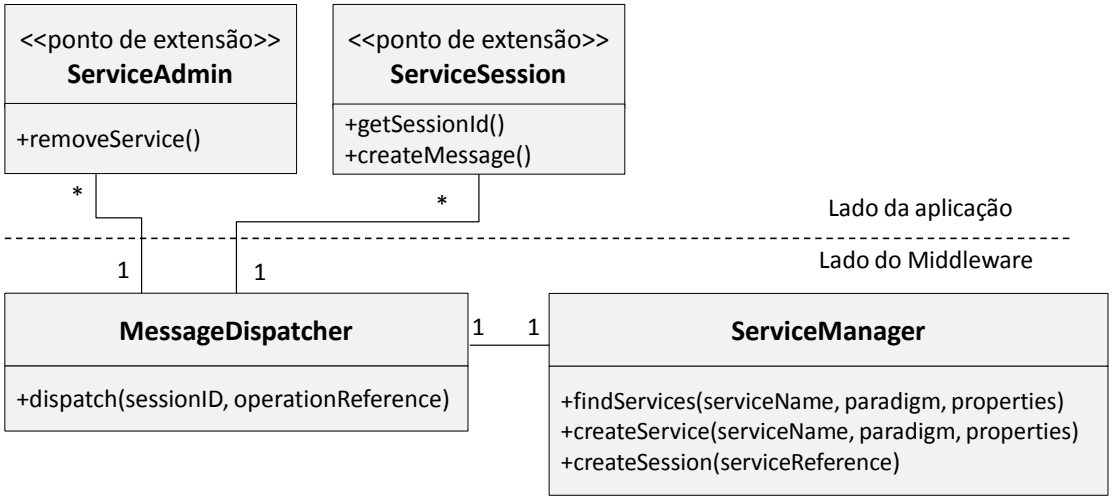

Figura 21 - Classes da API do middleware

A desvantagem é o processamento e *overhead* extra introduzido pelas chamadas entre processos (*inter-process call* - IPC). Como meio de minimizar este *overhead*, todo código relacionado a IPC está concentrado nas duas classes do lado do middleware especificadas acima: *service manager* e *message dispatcher*. O primeiro é chamado diretamente pelas aplicações para obter referências aos serviços de mensagens. O segundo é invocado indiretamente pelas aplicações, através das classes *service session* e *service admin*.

A classe *ServiceManager* oferece uma API para as aplicações que utilizam o middleware que possibilita a criação e busca de serviços na rede, bem como a criação de sessões para iniciar a comunicação com serviços remotos. Os métodos oferecidos por essa classe são:

- public List<ServiceReference> findServices(String serviceName, ServiceParadigm paradigm, Properties properties): Este método dispara uma busca por serviços disponíveis na rede. Os parâmetros podem ser especificados ou não. Caso seja especificado um ou mais parâmetros, os resultados da busca serão restritos aos que estiverem de acordo com os parâmetros especificados. O

tipo *ServiceParadigm*, que especifica o paradigma utilizado, será detalhado na Seção 3.8.3, que discute as classes de transporte de dados;

- public ServiceAdmin createService(String serviceName, ServiceParadigm paradigm, Properties properties): Este método permite a criação de um serviço de mensagens. A aplicação que cria o serviço fica responsável por exercer tarefas administrativas sobre o mesmo, através da referência do tipo *ServiceAdmin* que é retornada para ela.
- public ServiceSession createSession(ServiceReferece serviceReference): Após obter a referência ao serviço desejado, através do método *findServices()*, este método precisa ser chamado para que seja retornada um objeto do tipo *ServiceSession*, que possibilita interagir diretamente com o serviço desejado.

Como mostrado nos métodos acima, as classes *ServiceAdmin* e *ServiceSession* são retornadas, respectivamente, quando um serviço é criado, e quando é criada uma sessão para interagir com um serviço da rede. A função dessas classes é oferecer às aplicações acesso às operações dos serviços de mensagens providos pelo middleware, seja para utilizar o serviço (*ServiceSession*), ou para gerenciá-lo (*ServiceAdmin*).

É importante observar que, cada paradigma de comunicação define um conjunto distinto de operações. Desta forma, se os métodos de todos os paradigmas fossem definidos nessas classes exclusivamente, o middleware não seria extensível, pois a cada novo paradigma adicionado, seria necessário alterar os métodos dessas classes. Assim, essas duas classes, *ServiceSession* e *ServiceAdmin,* são providas na forma de pontos de extensão. Isso quer dizer que, cada paradigma deve prover uma extensão dessas classes, definindo suas próprias operações.

Para os paradigmas providos pelo Multi-MOM, as extensões dessas classes são especificadas abaixo. É importante destacar que na utilização de um serviço remoto, através de *ServiceSession*, a referência ao serviço é passada no momento da criação da sessão. Assim, nas chamadas aos métodos que acessam os serviços, não é necessário que seja passado novamente a referência que indica o endereço do serviço destinatário, pois cada sessão está associada a um e somente um serviço. Da mesma forma acontece para *ServiceAdmin*, já que esse objeto é obtido na criação de um serviço. Logo, todas as chamadas utilizando um objeto *ServiceAdmin* serão direcionadas ao serviço que foi criado na operação em que se obteve aquele objeto *ServiceAdmin*.

A Figura 22 e a Figura 23 ilustram as classes que são derivadas a partir de *ServiceSession* e *ServiceAdmin*. Esse conjunto de classes e métodos representa as operações disponibilizadas para utilizar cada uma dos paradigmas do Multi-MOM. Em seguida, cada uma dessas operações é descrita.

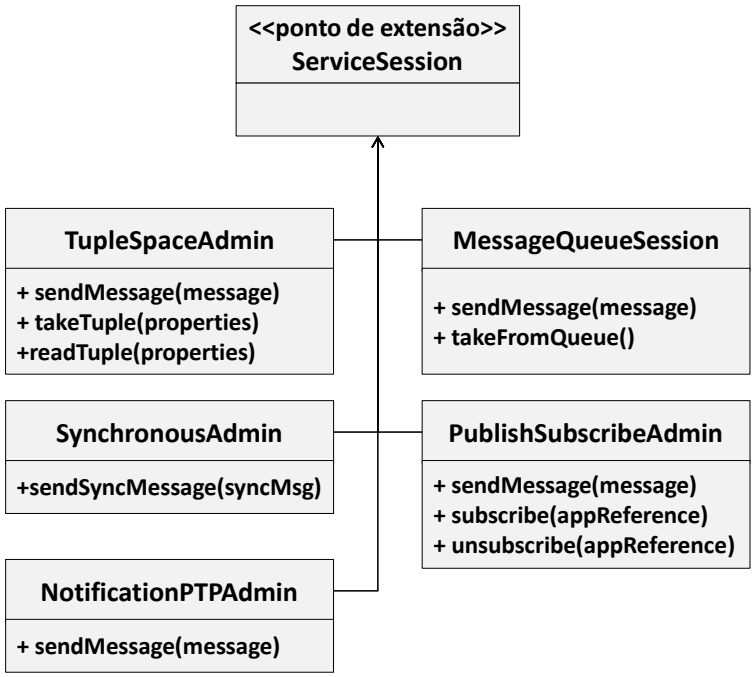

Figura 22 - Classes derivadas de ServiceSession

Fila de Mensagens

- MessageQueueSession
    - void sendMessage(asynchronousMessage): enviar mensagem para uma fila de mensagens;
    - AsynchronousMessage takeFromQueue(): obter uma mensagem da cabeça de uma fila de mensagens;
- MessageQueueAdmin
    - List<Message> getMessagesFromQueue(): obtém todas as mensagens existentes na fila, sem removê-las;
    - void cleanQueue(): remove todas as mensagens da fila;
    - int queueLenght(): obtém a quantidade de mensagens na fila;

- void removeService(): desativa o serviço, descartando as mensagens existentes;
- void inactivateService(): bloqueia a inserção de novas mensagens na fila, permitindo apenas que sejam retiradas as mensagens já existentes;

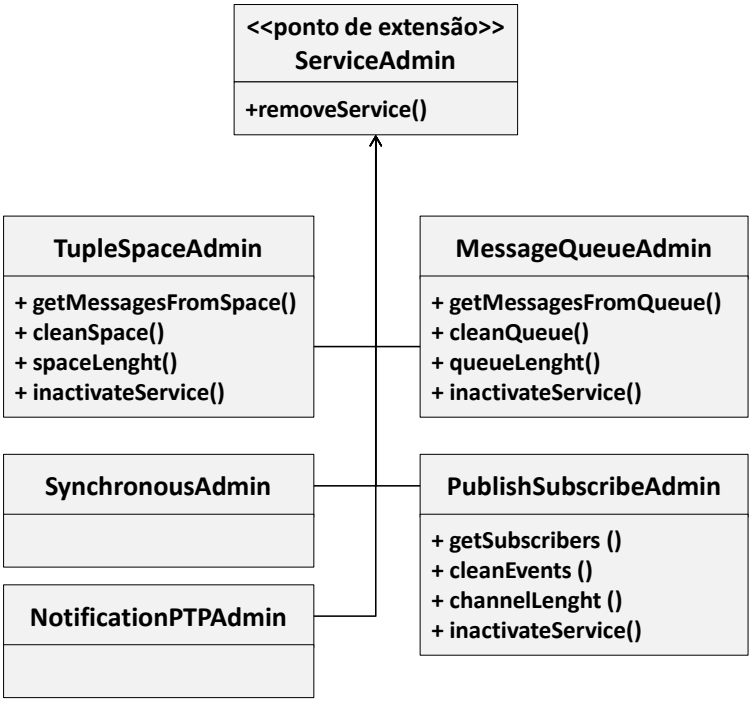

Figura 23 - Classes derivadas de ServiceAdmin

Espaço de Tuplas

- TupleSpaceSession
  - void sendMessage(asynchronousMessage): enviar mensagem para um espaço de tuplas;
  - AsynchronousMessage readTuple(properties); ler do espaço de tuplas, sem apagar, uma mensagem que contenha as propriedades especificadas no parâmetro;
  - AsynchronousMessage takeTuple(properties); obter e remover do espaço de tuplas uma mensagem que contenha as propriedades especificadas no parâmetro;
- TupleSpaceAdmin

- List<Message> getMessagesFromSpace(): obtém todas as mensagens existentes no espaço de tuplas, sem removê-las;
- void cleanSpace(): remove todas as mensagens do espaço de tuplas;
- int spaceLenght(): obtém a quantidade de mensagens no espaço de tuplas;
- void removeService(): desativa o serviço, descartando as mensagens existentes;
- void inactivateService(): bloqueia a inserção de novas mensagens no espaço de tuplas, permitindo apenas que sejam retiradas as mensagens já existentes;

Publish/Subscribe

- PublishSubscribeSession
  - void sendMessage(asynchronousMessage): enviar mensagem para um canal de eventos;
  - void subscribe(applicationReference): inscreve-se num canal de eventos. O parâmetro applicationReference provê uma referência de *callback* para que o middleware possa, ao receber uma mensagem do canal de eventos, repassá-la para a aplicação;
  - void unsubscribe(applicationReference): cancela inscrição num canal de eventos.
- PublishSubscribeAdmin
  - List<ServiceReference> getSubscribers(): Obtém informações sobre os inscritos no canal de eventos
  - void cleanEvents(): remove todas as mensagens do canal de eventos, isto é, aquelas que esperavam para ser entregues aos assinantes;
  - int channelLenght(): obtém a quantidade de mensagens no canal de eventos, isto é, mensagens que foram enviadas ao canal mas ainda não foram entregues aos assinantes;
  - void removeService(): desativa o serviço, descartando as mensagens existentes;

- void inactivateService(): bloqueia o envio de novas mensagens para o canal de eventos, permitindo apenas que sejam entregues as mensagens que já estavam naquele canal, esperando serem entregues;

Síncrono

- SyncrhonousSession
  - SynchronousMessage sendSyncMessage(synchronousMessage): enviar mensagem para um serviço síncrono, e obter mensagem de retorno;
- SyncrhonousAdmin
  - void removeService(): desativa o serviço;

Notificações ponto-a-ponto

- NotificationPTPSession
  - void sendMessage(asynchronousMessage): enviar mensagem para um serviço de notificações;
- NotificationPTPAdmin
  - void removeService(): desativa o serviço;

Observe que *ServiceSession* e *ServiceAdmin* são classes da API, e ao mesmo tempo, pontos de extensão. Já o *MessageDispatcher* é uma classe que tem a função de evitar que as classes base do middleware fiquem acoplados a alguma classe de extensão, isto é, alguma classe relacionada a um paradigma de comunicação específico. Assim, sempre que é necessário instanciar ou invocar uma classe de extensão, o *message dispatcher* é chamado, através da operação *dispatch(sessionId, operationReference),* a qual utiliza mecanismos de reflexão. A classe *Operation Reference*, bem como o processo para descobrir, através de reflexão, qual classe de extensão deve ser chamada em cada operação será descrito mais adiante.

3.8.2 Classes internas

Um outro tipo de classes do Multi-MOM são as classes internas, mostradas na Figura 24.

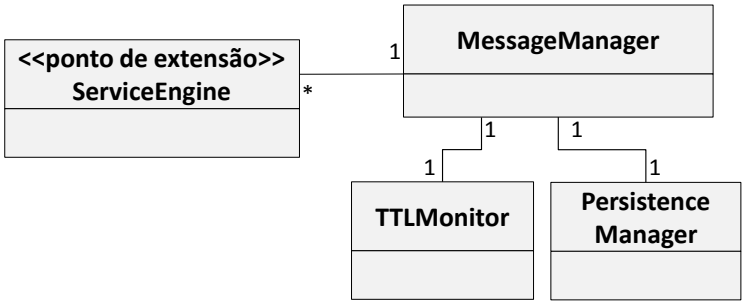

Figura 24 - Classes internas do middleware

A classe *MessageManager* é responsável por manter numa estrutura de dados as mensagens gerenciadas pelo middleware. Para auxiliar no gerenciamento dessas mensagens, são definidas também as classes *TTLMonitor* e *PersistenceManager*. A classe *TTLMonitor* implementa métodos para verificar se as mensagens ainda estão válidas ou já expiraram, e também uma *thread* que periodicamente varre todas as mensagens gerenciadas pelo middleware e faz essa verificação. Já a classe *PersistenceManager* é responsável por manter as mensagens marcadas como persistentes armazenadas na memória não-volátil do dispositivo.

O *ServiceEngine* é uma classe interna do middleware, e, ao mesmo tempo, um ponto de extensão no qual as semânticas de entrega de mensagens dos paradigmas de comunicação são de fato implementadas. Cada classe estendida a partir de S*erviceEngine* define as semânticas de um paradigma específico, similar aos pontos de extensão definidos em *ServiceSession* e *ServiceAdmin*. A diferença é que estes últimos definem um conjunto de classes que oferecem uma API para as aplicações que utilizam o middleware, baseado nas operações específicas de cada paradigma, enquanto que o *ServiceEngine* e suas extensões implementam a lógica interna de cada paradigma de comunicação.

Ou seja, as extensões de *ServiceEngine* provêem implementação para os métodos que são definidos por cada extensão de *ServiceAdmin* e *ServiceEngine*. Por exemplo, o *TupleSpaceEngine*, define as semânticas para recuperar mensagens por meio da especificação de um conjunto de propriedades desejadas, enquanto o *MessageQueueEngine* define semânticas para recuperar mensagens seguindo a ordem de inserção (FIFO). Já o *PublishSubscribeEngine* mantém as listas de assinantes dos

canais de eventos e, assim que recebe uma mensagem, cuida do encaminhamento dessa mensagem para cada um dos assinantes.

3.8.3 Classes de transporte de dados

As classes de transporte de dados são utilizadas para carregar dados entre a aplicação e o middleware, e também entre dispositivos, através da rede. Entre elas estão as classes *Message* e seus dois subtipos *SyncrhonousMessage* e *AsynchronousMessage*, descritos na Seção 3.6, bem como a classe *ServiceReference*, descrita na Seção 3.7. Há ainda outras duas classes, *ServiceParadigm* e *OperationReference*.

ServiceParadigm é também uma classe de ponto de extensão. Ela é utilizada para que a aplicação possa indicar qual paradigma de comunicação ela deseja ao criar um novo serviço, por exemplo. Assim, no método *createService(serviceName, serviceParadigm, properties)*, o parâmetro *serviceParadigm*, ao invés de ser uma string, é um subtipo de *ServiceParadigm*. Outro uso dessa classe é como um dos atributos de *ServiceReference*, para indicar qual o paradigma de comunicação é utilizado pelo serviço referenciado.

O problema de se utilizar string nesse caso é que, novos paradigmas são adicionados por qualquer desenvolvedor que deseje estender o middleware, e com isso, uma inconsistência nos nomes utilizados para referenciar os paradigmas seria suficiente para causar erro no funcionamento do middleware. Visto que strings não são verificadas na compilação, essas inconsistências causariam erros inesperados em tempo de execução. Assim, ao invés de uma string indicar o paradigma desejado, um subtipo de *ServiceParadigm* deve ser provido para cada paradigma adicionado ao Multi-MOM.

Já *OperationReference* é um objeto que encapsula as requisições de operações específicas dos paradigmas de comunicação. Ele é utilizado pelo *MessageDispatcher* para receber as operações das classes da API (subtipos de *ServiceAdmin* e *ServiceSession*), e repassar essas operações para as classes internas (subtipos de *ServiceEngine*). Como explicado anteriormente, o *MessageDispatcher*, não pode estar acoplado a nenhuma classe específica dos paradigmas, portanto, ele sempre recebe as requisições encapsuladas num objeto *OperationReference*, e a partir dos dados contidos nesse objeto, o *MessageDispatcher* utiliza mecanismos de reflexão para invocar a classe do paradigma adequado para tratar uma determinada requisição.

3.9 Extensibilidade do Projeto

As crescentes demandas exigidas para sistemas de software fazem com que eles freqüentemente tenham que desempenhar funções além do escopo para o qual foram inicialmente planejados. Essas funcionalidades nem sempre podem ser antecipadas durante a fase inicial do desenvolvimento, o que evidencia a necessidade de projetar sistemas que sejam facilmente extensíveis durante o seu ciclo de vida. Isso se torna ainda mais crítico no contexto de sistemas de middleware para dispositivos móveis, pois as possibilidades de aplicações para estes cenários são constantemente descobertas.

Por outro lado, se o middleware for projetado para atender a uma grande variedade de aplicações, possivelmente incluirá funcionalidades que serão utilizadas apenas por uma pequena parte das aplicações. Enquanto essas funcionalidades podem ser vitais para algumas aplicações, incluí-las como parte fundamental do middleware aumentará muito o espaço ocupado em memória pelo middleware e fará apenas aumentar o *overhead* para a maioria das outras aplicações.

Neste sentido, o Multi-MOM permite escolher quais paradigmas de comunicação farão parte de cada instância da plataforma. Ainda mais, além dos paradigmas que foram discutidos na Seção 3.4, novos paradigmas de comunicação podem ser criados e adicionados à plataforma, simplesmente pela adição de novos componentes em um conjunto de pontos de extensão previamente definidos. Da mesma maneira, a remoção de paradigmas de comunicação – para otimização da plataforma – é feita através da remoção dos componentes relativos ao paradigma de comunicação dos pontos de extensão.

É importante destacar que essa customização é feita sem que seja necessário alterar nenhum código existente, isto é, na adição ou remoção de um paradigma de comunicação, tanto os componentes relativos às funcionalidades comuns, como os componentes relativos aos paradigmas de comunicação já existentes permanecem sem modificação. Desta forma, o projeto do middleware é fechado para modificações e aberto para extensões, como é sugerido pelo princípio de projeto aberto-fechado [Gamma94].

Para dar suporte a essa flexibilidade, uma abordagem de customização foi definida baseada nos seguintes conceitos [Morais10d]:

- **Herança:** Os pontos de extensão são representados por classes genéricas, as quais devem ser estendidas (herdadas) para implementar comportamentos específicos dos paradigmas de comunicação (como mostrado na Seção 3.8);
- **Convenção de nomes:** Todo ponto de extensão do Multi-MOM tem um nome estabelecido. Desta forma, as classes de extensão devem obedecer a essa convenção para que sejam reconhecidas pelos componentes base do middleware;
- **Reflexão:** Para que não haja dependência ou acoplamento entre as classes que implementam funcionalidades comuns (fixas) e as classes que implementam funcionalidades variáveis (paradigmas de comunicação), o middleware utiliza mecanismos de reflexão.

3.9.1 Pontos de Extensão

Seguindo o princípio de projeto aberto-fechado e a abordagem de customização definida, é possível fazer com que o middleware incorpore novos comportamentos. Isso é feito simplesmente pela extensão de um conjunto de classes base, sem que seja necessário alterar código existente. Neste sentido, o middleware define um conjunto de pontos de extensão, na forma de classes que devem ser estendidas, como explicados na Seção 3.8). Assim, o objetivo desta seção é detalhar como esses pontos de extensão funcionam em conjunto.

No Multi-MOM são definidos 4 pontos de extensão:

- ServiceSession: Oferece às aplicações um conjunto de operações específicas para interagir com os serviços de comunicação dos diversos paradigmas.
- ServiceAdmin: É obtido quando alguma aplicação cria um novo serviço de mensagens. As classes derivadas de *ServiceAdmin* oferecem métodos para tarefas relacionadas à administração dos serviços.
- ServiceEngine: Oferece um ponto de extensão no qual as semânticas de entrega de mensagens dos paradigmas de comunicação são de fato implementadas. Diferentemente dos dois pontos de extensão anteriores, este não faz parte da API do middleware, é utilizado apenas internamente.
- ServiceParadigm: Utilizado para indicar o paradigma de comunicação adotado por um serviço. Evita com que sejam utilizadas strings para definir os paradigmas de

comunicação, pois inconsistências nas mesmas só seriam detectadas em tempo de execução, causando erros inesperados.

3.9.2 Convenção de Nomes

A convenção de nomes para pontos de extensão do Multi-MOM define duas regras básicas. A primeira é que todo ponto de extensão deve ter um nome, e as classes que o estendem tem que seguir esse nome. Os nomes dos pontos de extensão foram apresentados na Seção 3.9.1, e são: ServiceSession, ServiceAdmin, ServiceEngine e ServiceParadigm. As classes de extensão têm que retirar o nome "Service" e substituir pelo nome do paradigma de comunicação. Por exemplo, as classes do espaço de tuplas foram nomeadas: TupleSpaceSession, TupleSpaceAdmin, TupleSpaceEngine e TupleSpaceParadigm. A Figura 25 mostra esses pontos de extensão.

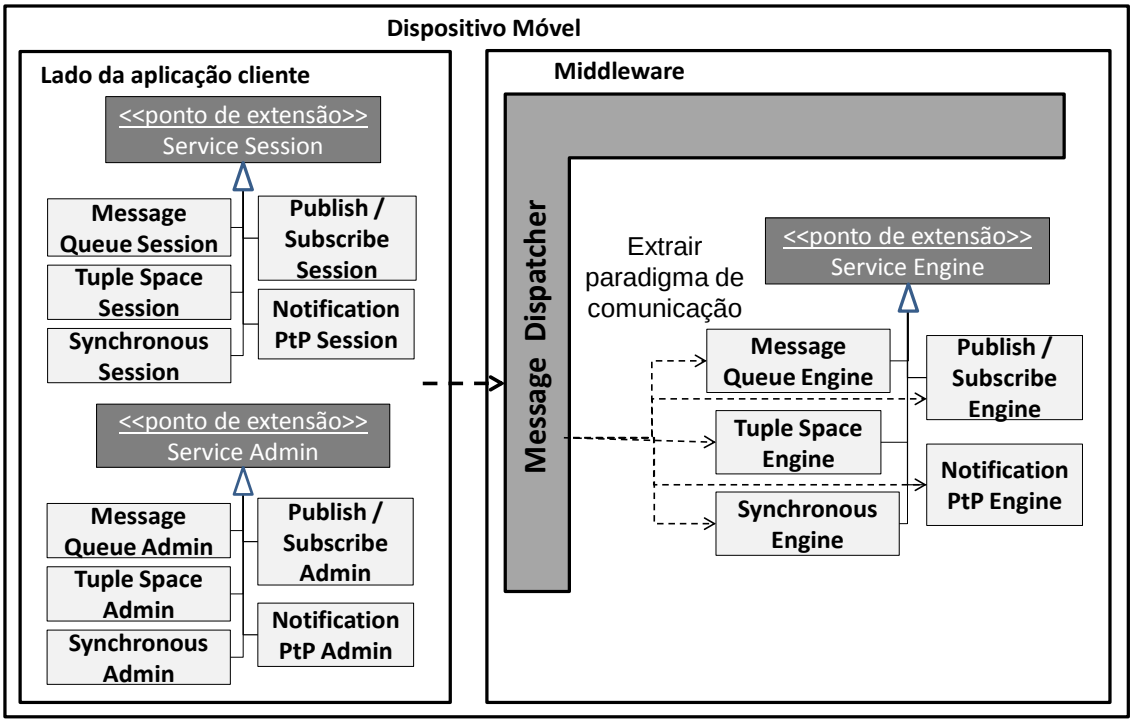

Figura 25 - Pontos de extensão do Multi-MOM

A segunda regra é que, para as classes relativas aos paradigmas de comunicação, os nomes utilizados pelas suas classes de extensão têm que ser consistentes em todos os pontos de extensão. Observe na Figura 25 que o nome escolhido para designar o paradigma espaço de tuplas foi "TupleSpace". Assim o nome "TupleSpace" foi substituído pelo nome "Service" em todos os pontos de extensão. Da mesma forma

ocorre com o paradigma publish/subscribe, que substitui o nome “Service” dos pontos de extensão pelo nome “PublishSubscribe”. Esse padrão deve ser seguido por todos os paradigmas de comunicação.

3.9.3 Funcionamento do Mecanismo de Reflexão

O mecanismo de reflexão é utilizado para evitar que componentes base do middleware, isto é, relativos às funcionalidades comuns, fiquem acoplados aos componentes opcionais/variáveis, isto é, aos componentes específicos de cada paradigma de comunicação.

Todo código que utiliza reflexão está concentrado no *message dispatcher*. Assim, é o *message dispatcher* que faz com que uma operação chamada pela aplicação, referente a um paradigma de comunicação específico, seja resolvida por um componente interno daquele mesmo paradigma de comunicação. Por exemplo, quando uma aplicação chama a operação *takeTuple()* num *TupleSpaceSession*, é o *message dispatcher* que vai encaminhar aquela operação para o *TupleSpaceEngine*.

Isso acontece da seguinte forma: a classe *TupleSpaceSession*, ao chamar o *message dispatcher*, passa para ele todas as informações sobre a operação desejada, *takeTuple()* nesse exemplo, encapsuladas no objeto *Operation Reference*. O *Message Dispatcher* extrai do objeto *operation reference* a informação de que a operação é referente ao paradigma espaço de tuplas, indicado pela classe *TupleSpaceParadigm*. Pela convenção de nomes, o nome do paradigma é “TupleSpace”, então o nome do componente que implementa as semânticas daquele paradigma é *TupleSpaceEngine*. Logo, o nome do componente recém-descoberto, *TupleSpaceEngine*, será utilizado para realizar a invocação do mesmo, através de reflexão.

A Figura 25 ilustra o papel desempenhado pelo *message dispatcher* em extrair as informações sobre o paradigma utilizado para então fazer uso de mecanismos de reflexão. É importante destacar que, apesar do mecanismo de reflexão adicionar um certo *overhead*, o uso de sessões tende a amenizar esse problema. Isto é, a reflexão é realizada somente quando uma aplicação obtém a referência ao serviço pela primeira vez. As chamadas subseqüentes às operações daquele serviço são identificadas por um identificador (ID) de sessão, as quais não precisam fazer o uso de reflexão.

3.10 Interação dos Componentes

Para definir como os diversos componentes do Multi-MOM trabalham em conjunto, esta seção apresenta a modelagem da interação entre os componentes, a qual é representada, basicamente, através de diagramas de colaboração. Primeiro é apresentado na Figura 26 como acontece a criação de um serviço de mensagem. Nesse exemplo o serviço a ser criado é um espaço de tuplas.

Primeiro, a aplicação chama o método *createService()* oferecido pelo componente *service manager*. Os parâmetros especificados indicam o nome escolhido para o serviço e o paradigma de comunicação desejado. O *service manager* por sua vez, chama um método com o mesmo nome e mesmos parâmetros em *service locator*, o qual fica responsável por anunciar este serviço na rede. O próximo passo é chamar o *message dispatcher* para que este, por meio de reflexão, obtenha uma referência para o componente que implementa o serviço, e invoque nele a operação de criação de serviços. Neste exemplo, o serviço é um espaço de tuplas, então o componente *TupleSpaceEngine* é chamado, retornando um objeto *TupleSpaceAdmin*, que será utilizado pela aplicação para gerenciar o serviço.

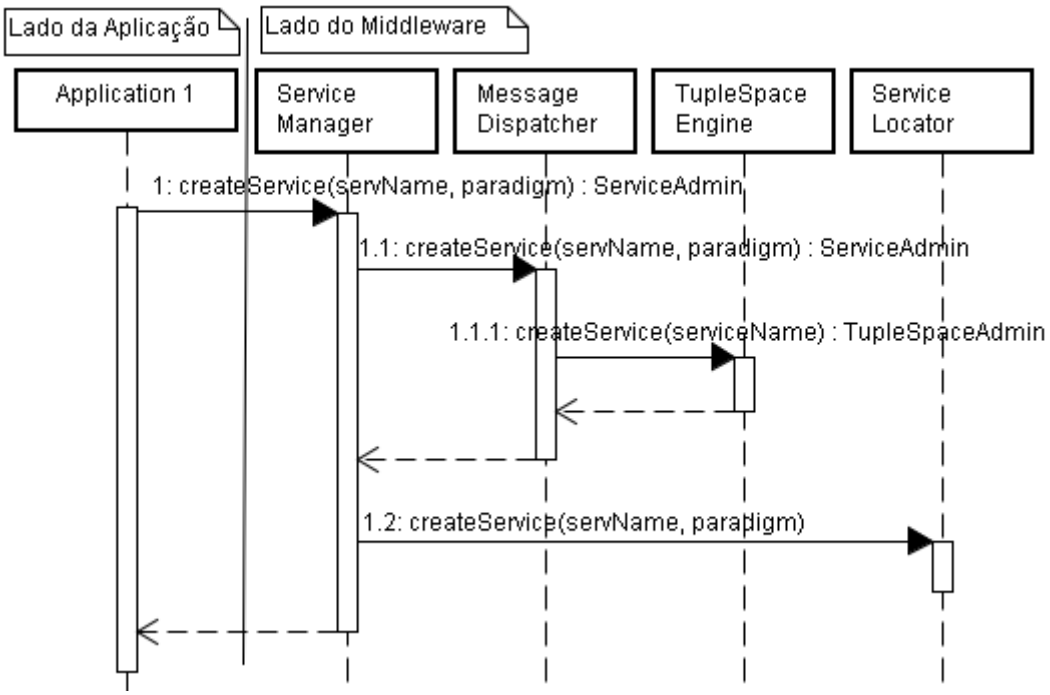

Figura 26 - Criação de serviço de mensagem

Uma vez que o processo de criação do serviço é finalizado, este serviço passa a estar disponível na rede para que aplicações remotas possam acessá-lo. A Figura 27

mostra como as aplicações obtêm referências aos serviços de mensagens disponíveis na rede.

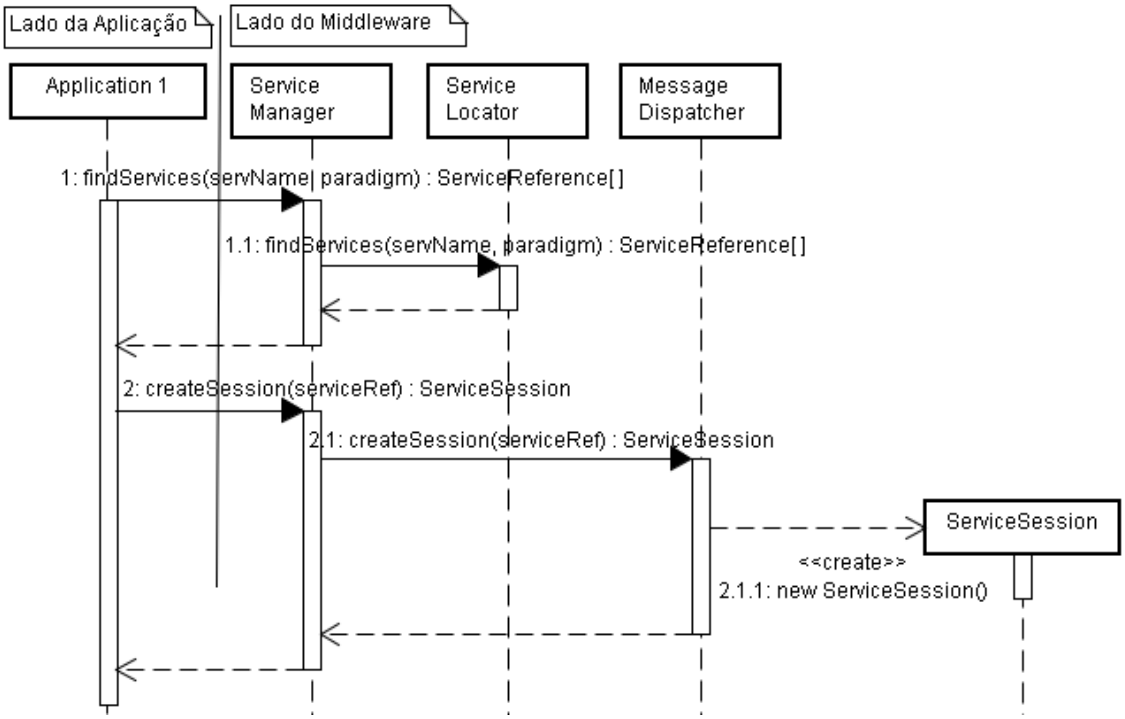

Figura 27 - Obtendo referências aos serviços de mensagens

Primeiro, a aplicação cliente descobre quais são os serviços disponíveis na rede através do método *findServices()* oferecido pelo *service manager*. Note que, essa operação permite a especificação do nome do serviço desejado e/ou o paradigma do serviço. Esses parâmetros podem também ser omitidos, o que resultará no retorno de todos os serviços disponíveis na rede. Em seguida, o *service manager* chama um método com o mesmo nome em *service locator*, o qual utiliza o protocolo de localização de serviços mencionado na Seção 3.3.4 para obter informações sobre os serviços da rede, retornando um arranjo com as referências aos serviços encontrados (*ServiceReference [])*. Objetos *service reference* são aqueles que encapsulam toda a informação de acesso a um serviço, tal como endereço IP, porta, paradigma utilizado pelo serviço e o nome do serviço.

Em seguida, dentre as referências a serviços retornadas, a aplicação escolhe uma, a qual indica o serviço que deseja-se interagir. Assim, a aplicação chama *createSession()* no *service manager*, passando o objeto *service reference* que corresponde ao serviço desejado. O *service manager* então chama o *message*

dispatcher, que fará a criação da sessão, que será representada por um objeto subtipo de *service session*, de acordo com o paradigma utilizado pelo serviço.

Apesar do serviço estar executando num dispositivo remoto, a criação da sessão também é registrada localmente, pelo *message dispatcher*. Esse registro consiste de uma associação entre o objeto *service reference* e um ID de sessão. O ID de sessão atua de forma similar a um *cookie* HTTP, que será utilizado nas próximas invocações àquele serviço, evitando assim que o middleware tenha que utilizar reflexão novamente para descobrir qual subclasse de *service engine* deverá ser usada para tratar tais invocações. Desta forma, o *overhead* introduzido pelo uso de mecanismos de reflexão é amenizado pelo uso de sessões, já que este tipo de mecanismo é usado apenas na criação da sessão.

Além de evitar o uso de reflexão em cada chamada, o ID de sessão diminui também a quantidade de dados transferidos entre as aplicações e o middleware, pois o ID de sessão está também associado ao objeto *service reference*, que encapsula todas as informações sobre um determinado serviço. Assim, essas informações não têm que ser repassadas a cada invocação a um mesmo serviço. É importante considerar esses dados passados entre as aplicações e o middleware, pois estes executam em processos distintos do sistema operacional. Conseqüentemente, uma cópia dos dados é feita para que os dados sejam transferidos entre processos.

A Figura 28 ilustra a relação entre os objetos do lado da aplicação e o middleware. O *message dispatcher* registra todas as sessões criadas por aplicações e encaminha as mensagens para o subtipo de *service engine* apropriado, baseado no ID de sessão previamente associado.

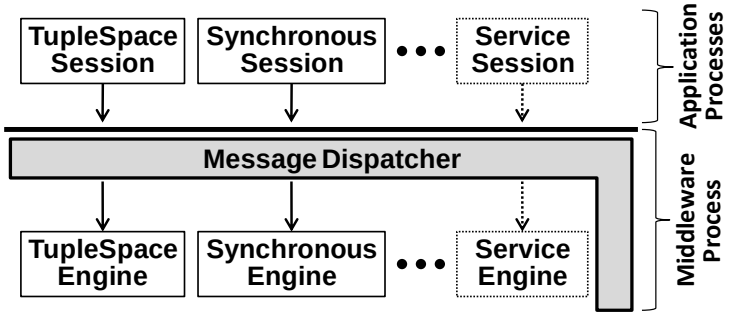

Figura 28 - O componente Message Dispatcher e a separação de processos

A Figura 29 ilustra os passos para o envio de uma mensagem a um serviço de mensagens, continuando a seqüência mostrada na Figura 27, na qual a aplicação obteve

uma referência a um espaço de tuplas, e em seguida recebeu um objeto *TupleSpaceSession* em resposta a criação de uma sessão. Como explicado anteriormente, é através dos objetos derivados de *service session* que as aplicações interagem com os serviços de mensagens, enviando e recebendo mensagens de acordo com as semânticas de cada paradigma.

Inicialmente, a aplicação chama *sendMessage()* no *TupleSpaceSession*, que, por sua vez, chama o *message dispatcher*, passando o ID de sessão e referência da operação chamada. A referência da operação é encapsulada num objeto o*peration reference*, descrito na Seção 3.8.3. Esse objeto encapsula as informações de uma operação específica de um paradigma, incluindo seu nome e os parâmetros. É importante destacar que é esse objeto que possibilita o *message dispatcher* de receber requisições de todos os paradigmas sem estar acoplado a nenhum deles. Isto é, o *message dispatcher* recebe um objeto *operation reference*, verifica qual componente deve tratar aquela operação e repassa a chamada ao componente adequado.

Através do ID de sessão, o *message disptacher* identifica que a operação deve ser tratada pelo *TupleSpaceEngine*. Este último, por sua vez, transfere a mensagem para o *message manager*, que verifica o destino da mensagem. Se a mensagem é para um espaço de tuplas executando localmente, o *message manager* mantém a mensagem localmente. Caso contrário, se a mensagem é para um espaço de tuplas remoto, o *message manager* chama o *message exchanger* para encaminhar a mensagem pela rede. Se o *message exchanger* não puder estabelecer uma conexão com o dispositivo remoto, o *message manager* mantém a mensagem e tenta retransmitir novamente mais tarde.

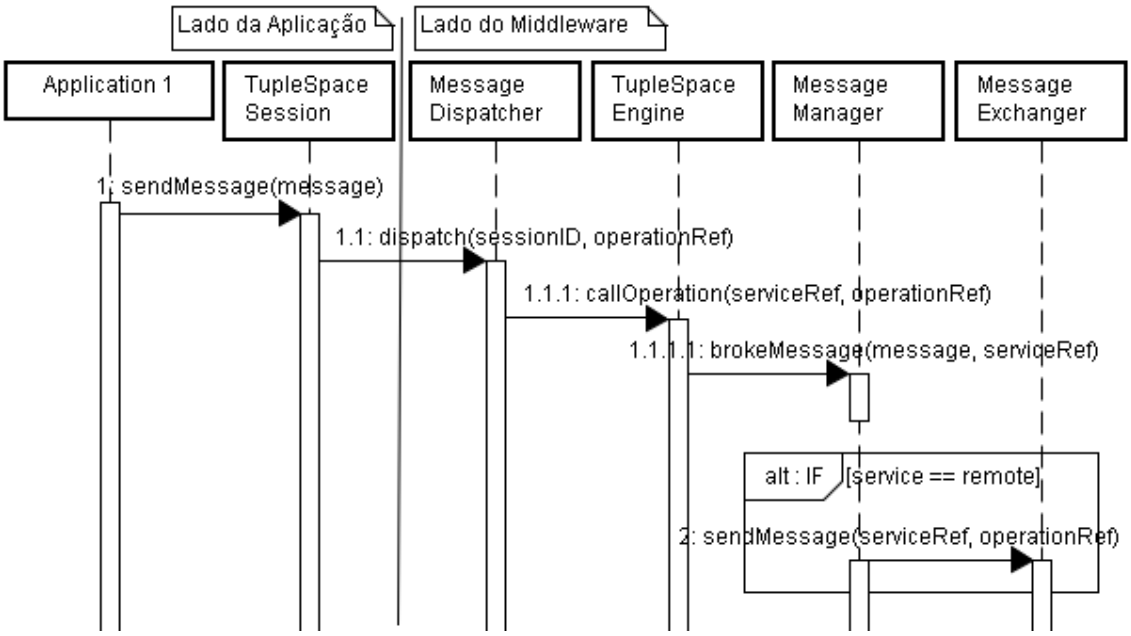

Figura 29 - Interação no envio de uma mensagem

Continuando este cenário, a Figura 30 ilustra a interação dos componentes no outro dispositivo, isto é, no dispositivo que recebeu a mensagem enviada na figura anterior (Figura 29). Essa mensagem trafega através da rede e é recebida pelo componente *message exchanger*, que implementa um s*erver socket* que aguarda conexões, e ao receber uma mensagem, repassa-a diretamente para o *message dispatcher*. Este último, como explicado anteriormente, é quem vai saber qual subtipo de *service engine* chamar para tratar aquela mensagem, de acordo com o paradigma associado.

No caso do paradigma espaço de tuplas, o *TupleSpaceEngine* é o componente responsável pelo tratamento semântico adequado para aquela mensagem. Mensagens enviadas para espaço de tuplas são mantidas por estes serviços até que outras aplicações interessadas busquem-na. Assim, o *TupleSpaceEngine* passa a mensagem para o *message manager,* que é reponsável por guardar todas as referências às mensagens mantidas pelo middleware. Se o paradigma fosse *publish/subscribe,* a mensagem ao invés de ficar armazenada, esperando ser recuperada, seria encaminhada para todas as aplicações que registraram interesse previamente. Como outro exemplo, se o paradigma fosse notificações ponto-a-ponto, a mensagem seria entregue diretamente à aplicação criadora daquele serviço ao qual a mensagem foi direcionada.

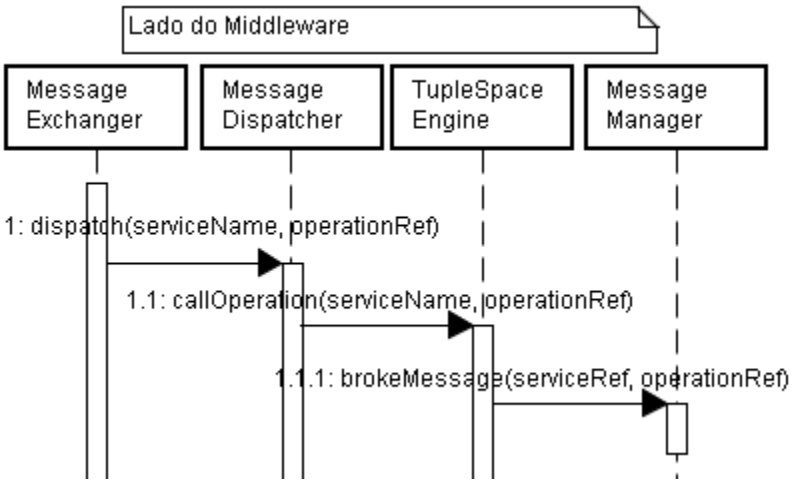

Figura 30 – Recebimento de mensagem através da rede

3.11 Considerações finais

Neste capítulo, foram apresentadas as funcionalidades e características do middleware proposto, o qual define uma plataforma extensível que oferece suporte a aplicações móveis colaborativas, para que as mesmas não tenham que lidar com as dificuldades da programação em redes móveis.

Nesse contexto, o middleware provê suporte a múltiplos paradigmas de comunicação, para que a utilização do middleware não fique restrita a um cenário de aplicação específico. Os paradigmas providos são: Publish/Subscribe, Espaço de Tuplas, Filas de Mensagens, Mensagens Síncronas e Notificações Ponto-a-Ponto. Além de prover esses paradigmas, são consideradas as restrições dos dispositivos móveis e da comunicação no meio sem fio, e a partir destas limitações são definidas funcionalidades de suporte aos paradigmas providos. Vale ressaltar que essas funcionalidades são definidas de forma que sejam comuns aos paradigmas de comunicação, para que o middleware não sobrecarregue os recursos já limitados dos dispositivos móveis.

Quanto ao projeto extensível, a abordagem de customização definida permite que o middleware seja instanciado com qualquer combinação dos paradigmas providos, bem como possibilita a criação de novos paradigmas somente com a adição de novas classes em pontos de extensão previamente estabelecidos. É importante destacar que essa customização não requer alteração de código existente, isto é, seguindo a abordagem de customização definida os paradigmas podem ser incluídos/excluídos somente pela inclusão/exclusão das classes específicas de cada paradigma.

4 Implementação e Cenário de Aplicação

Este capítulo apresenta detalhes de implementação do Multi-MOM, incluindo a apresentação das principais tecnologias envolvidas, bem como as justificativas para uso das mesmas. Em seguida são apresentados como foram realizados os testes e a preparação para implantação no ambiente de execução. Por fim, é apresentado um cenário de aplicação e uso do Multi-MOM, o qual tem por objetivo ilustrar os benefícios da abstração multi-paradigma no ambiente móvel.

4.1 Implementação do Protótipo

Para validar a arquitetura e o projeto do middleware proposto, um protótipo foi implementado utilizando a plataforma Android [Android10a]. Android é uma plataforma para dispositivos móveis, do tipo celulares e *smartphones*, desenvolvida em Novembro de 2007 pela Open Handset Alliance, um grupo de 71 empresas de tecnologia lideradas pelo Google. Diferentemente de outras plataformas para dispositivos móveis (ex.: Symbian, iPhone OS, Windows Mobile), Android é uma plataforma de código aberto, baseada no sistema operacional Linux e utilizada em dispositivos de várias marcas (ex.: Samsung, HTC, Motorola, Sony Ericsson).

Apesar do pouco tempo no mercado, comparado às plataformas existentes, Android é atualmente a segunda plataforma mais popular para *smartphones* nos Estados Unidos, estando presente em 28% dos *smartphones* vendidos no primeiro trimestre de 2010 [Reuters2010]. Considerando que uma das principais motivações para o desenvolvimento de sistemas de middleware é que ele possa ser reusado em diversas aplicações, a utilização de uma plataforma amplamente adotada tende a aumentar esse nível de reuso.

O modelo de componentes definido pelo Android provê um tipo de componente no estilo de serviços de segundo plano (*background services*), nos quais podem ser

implementadas as tarefas de processamento não vistas diretamente pelas aplicações interativas. Desta forma, o middleware é provido como um serviço de segundo plano, acessível às aplicações por meio do mecanismo de IPC do Android. Além da segurança provida pela separação do middleware em seu próprio processo, este isolamento possibilita que o middleware seja compartilhado por duas ou mais aplicações que necessitem de funcionalidades de comunicação em rede, podendo as mesmas estar executando em paralelo ou uma de cada vez. Desta maneira, diversas aplicações podem contar com uma única plataforma para lidar com os desafios da comunicação em rede.

A Figura 31 apresenta uma visão geral das principais tecnologias envolvidas na implementação do protótipo.

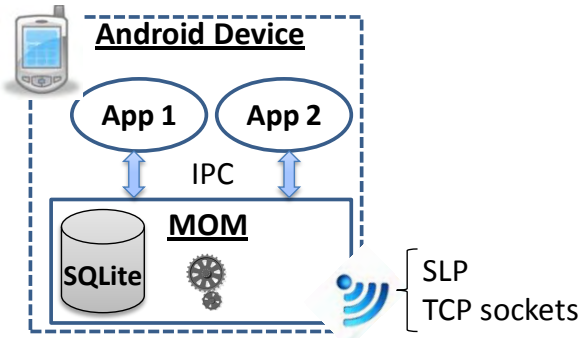

Figura 31 - Visão geral da implementação

Aplicações Android são desenvolvidas em Java Standard Edition, que possibilita o uso de mecanismos de reflexão para inspecionar o código e dinamicamente criar invocações de métodos. Outras plataformas para dispositivos móveis (ex.: Symbian, Windows Mobile) geralmente dão suporte apenas ao Java Micro Edition, uma versão limitada do Java para celulares que não tem suporte à reflexão, o que impossibilita o uso da abordagem de extensibilidade do middleware proposto.

Mensagens persistentes são armazenadas através de um banco de dados relacional embarcado. Neste tipo de banco de dados, não há um serviço separado, ao invés disso, o engenho do banco de dados é provido por uma biblioteca do SQLite [SQLite10] auto-contida e de baixa carga computacional. Desta forma, o banco de dados é criado apenas internamente ao middleware, sendo o armazenamento das mensagens transparente para as aplicações. O único controle que as aplicações têm nesse aspecto é a possibilidade de marcar as mensagens como persistentes ou não.

As mensagens são apagadas do banco de dados sempre que são também retiradas da memória, ou seja, quando o serviço de mensagem correspondente deixa de utilizar aquela mensagem, ou quando ela é enviada com sucesso ao dispositivo remoto ao qual ela deve ser entregue. Além disso, sempre que o middleware é iniciado, é verificado se existem mensagens previamente armazenadas no banco de dados, se existirem e elas ainda não houverem expirado, elas são então restauradas para a memória principal.

É importante destacar que todo o acesso ao banco de dados é feito na linguagem SQL padrão, o que possibilita que outro provedor de banco de dados seja utilizado ao invés do SQLite sem a necessidade de alterar código algum. Entretanto, o SQLite já é por padrão provido pela plataforma Android, assim, uma biblioteca adicional não precisa ser incluída.

Para anúncio e a descoberta de serviços, é utilizado o Service Location Protocol (SLP), um protocolo simples e de leve carga computacional. Uma importante característica do SLP é o seu comportamento adaptativo. No SLP, quando um Agente de Diretório (DA – *Directory Agent*) está presente, clientes utilizam essa entidade central para registrar e descobrir serviços. Se não há um DA presente, os clientes enviam mensagens de *multicast* na rede, e assim dispositivos que tiverem serviços disponíveis respondem a esta mensagem. Isso permite que os serviços do middleware proposto possam operar tanto em redes infra-estruturadas, com entidades centralizadoras, como também em redes *ad hoc*. A implementação desse protocolo é provida pelo jSLP [Rellermeyer06], uma biblioteca de código aberto, baseada em Java e otimizada para dispositivos móveis.

Por fim, o transporte de mensagens entre dispositivos é feito através de sockets TCP. O protocolo TCP foi escolhido por prover garantia de entrega dos dados, simplificando a programação do middleware. A utilização do protocolo UDP tornaria necessária a implementação de um mecanismo de controle de erros para a entrega de mensagens. A vantagem é que esse último requer menos tempo para transmitir dados, já que nenhum controle é feito durante a transmissão. Assim, o ideal é que o middleware possa ser customizado para prover um protocolo ou outro. Apesar disso, essa possibilidade de escolha do protocolo não é provida no protótipo atual, sendo sugerida como trabalho futuro.

4.2 Metodologia de Testes

Os testes foram realizados para verificar se o produto final atende aos requisitos especificados, bem como descobrir falhas ou defeitos no sistema. Para executar a tarefa de teste, diversas técnicas e metodologias são encontradas na literatura. De acordo com [Pressman05], a atividade de testes deve começar com foco em partes individuais, e a medida que o teste progride, este foco vai passando para o sistema como um todo. Desta forma, os testes do middleware proposto foram realizados baseados em duas abordagens complementares: *testes de unidade* e *testes de sistemas*.

Os testes de unidade devem ser focados na menor unidade do projeto de software. No middleware proposto, os testes de unidade foram criados para cada um dos componentes do projeto arquitetural mostrado na Seção 3.5. Nessa etapa, os testes foram feitos de forma caixa-branca, isto é, testando cada um dos fluxos de controle, de forma a descobrir erros na implementação interna de cada um desses componentes.

Uma vez que boa parte dos componentes estão implementados e funcionando corretamente de forma individual, é necessário verificar se os mesmos funcionam de maneira adequada quando colocados todos juntos. Nesta etapa, o objetivo era verificar se o conjunto de componentes que compõem o Multi-MOM funcionavam de maneira adequada para satisfazer os requisitos aos quais ele foi proposto. Para isso, a abordagem de teste de sistemas divide-se em duas: *testes de integração* e *teste de releases* (teste funcional).

Nos *testes de integração*, verificamos se os componentes são chamados corretamente e se transferiam os dados corretos no tempo correto por meio de suas interfaces. Para tal, usamos uma estratégia chamada de integração *bottom-up*, onde integramos primeiro os componentes de mais baixo nível, como os que cuidam das conexões de rede para transferir mensagens, bem como anunciar e descobrir serviços. Em seguida, foram adicionados os componentes que cuidam do armazenamento de mensagens e a lógica interna de cada paradigma de comunicação. Por fim, foram incluídos os componentes que se comunicam diretamente com as aplicações.

Para automatizar o processo de criação dos códigos de testes dessa fase, foi utilizado o *framework* de teste *JUnit* [Massol03], o que permitiu garantir que toda vez que um novo código era integrado ou alterado, todos os testes eram executados novamente para garantir que a alteração não afetou as demais interfaces ou

características do componente. Esse processo de re-execução dos testes a cada alteração é chamado de *teste de regressão* [Pressman05].

Já nos testes de *releases*, ou como são mais conhecidos, testes funcionais, utiliza-se uma abordagem caixa-preta, onde o objetivo é concentrar-se nas funcionalidades, e não na implementação do software. Para isso, utilizamos uma abordagem baseada em cenários [Sommerville07]. Foram escolhidos maneiras comuns que as aplicações podem fazer uso dos serviços de mensagens providos pelo middleware. Por exemplo: uma aplicação cria um serviço de espaço de tuplas e insere algumas tuplas, em seguida, outra aplicação, rodando em outro dispositivo, busca a referência para aquele serviço, retira uma tupla e insere outras. Os outros paradigmas de comunicação foram testados de forma similar, seguindo as dinâmicas de interação de cada um.

4.3 Montagem e Implantação

Por fim, nesse estágio acontecem operações que preparam uma versão executável do software e a montagem da aplicação para o ambiente de produção. No contexto dessa dissertação, o middleware foi preparado para as avaliações descritas no próximo capítulo.

O artefato de software do Multi-MOM é por si só um artefato finalizado, que provê os cinco paradigmas de comunicação abordados nesta dissertação. Entretanto, por ser modelado como uma Linha de Produto de Software, o mesmo permite que variações sejam derivadas a partir do conjunto de componentes que compõe o Multi-MOM. Essas variações são manipuladas a partir dos pontos de extensão definidos na Seção 3.6. Com isso, pode-se selecionar quais paradigmas de comunicação farão parte de cada instância da plataforma, retirando paradigmas, caso seja necessário otimizar a plataforma, ou acrescentando novos paradigmas, para atender a novos requisitos de distribuição.

A seleção dos paradigmas de comunicação é feita em tempo de compilação, apenas pela inclusão/exclusão das classes relacionadas a cada paradigma de comunicação. É importante destacar dois pontos: (i) essas classes são colocadas em pontos de extensão previamente definidos; (ii) nenhuma outra parte do código do middleware é alterada para seleção dos paradigmas de comunicação.

Abaixo segue a organização das classes de extensão em pacotes:

- **br.ufpb.compose.multimom**: Pacote em que ficam as classes de funcionalidades comuns de sistemas de mensagens. As classes desse pacote não são alteradas no processo de seleção de paradigmas de comunicação;
- **br.ufpb.compose.multimom.engine**: Responsável pelas classes que herdam de *Service Engine*, as quais implementam a lógica interna dos paradigmas de comunicação. Ex.: *TupleSpaceEngine, PublishSubscribeEngine*.
- **br.ufpb.compose.multimom.session:** Classes que implementam *Service Session*, provêem operações para as aplicações clientes utilizarem os serviços de mensagens. Ex.: *TupleSpaceSession, PublishSubscribeSession.*
- **br.ufpb.compose.multimom.admin:** Classes que implementam *Service Admin*, provêem operações de gerenciamento dos serviços de mensagens. Ex.: *TupleSpaceAdmin, PublishSubscribeAdmin*.
- **br.ufpb.compose.multimom.messagetype:** Classes que implementam os subtipos do objeto *Message*. Ex.: *SynchronousMessage* e *AsynchronousMessage*.

Após a seleção das classes contidas nos pacotes acima, deve ser gerado um artefato de software em forma binária, pronto para implantação em qualquer dispositivo Android. O artefato binário é gerado no formato .apk, utilizado por todas as aplicações compatíveis com a plataforma Android. Para as análises de desempenho e de tamanho ocupado em memória definidas no próximo capítulo, foram derivadas diferentes instâncias do Multi-MOM, sendo 5 instâncias contendo apenas um paradigma de comunicação por vez, e uma outra instância contendo todos os paradigmas de comunicação juntos.

Para um desenvolvedor de aplicações utilizar o Multi-MOM, ele tem duas opções. Ele pode obter o arquivo executável do middleware, através do qual ele terá disponível os 5 paradigmas de comunicação aqui discutidos. Esse arquivo executável pode ser instalado em qualquer dispositivo Android, e não precisa de nenhuma alteração. Como segunda opção, o desenvolvedor pode obter o código fonte do middleware, para que ele possa fazer suas próprias customizações, isto é, estender o middleware com novos paradigmas e/ou remover paradigmas existentes, de acordo com suas necessidades.

4.4 Cenário de Aplicação

Os benefícios de um middleware multi-paradigma para computação móvel incluem não somente o potencial para atender uma ampla variedade de cenários de aplicação, mas também a possibilidade de uma única aplicação beneficiar-se de tal abordagem para dar suporte às diferentes formas de interação que ela pode desempenhar. Com o objetivo de mostrar a necessidade de uma abordagem multi-paradigma para o desenvolvimento de aplicações móveis, esta seção apresenta como uma aplicação exemplo pode se beneficiar do Multi-MOM para um jogo colaborativo. Através de um exemplo simples, tal aplicação demonstra como as operações disponibilizadas pelo Multi-MOM são naturais e suficientes para dar suporte às variadas necessidades de interação das aplicações móveis colaborativas.

4.4.1 Aplicação Exemplo

A aplicação exemplo é um jogo de quebra-cabeça de múltiplos jogadores (*multi-player*). Neste jogo, um grupo de 3 jogadores (J1, J2 e J3) cooperam para solucionar o quebra-cabeças. O jogo começa com um dos jogadores (J1) criando um espaço compartilhado na rede e carregando as peças do quebra-cabeça para este espaço. Os outros jogadores que desejam participar do jogo devem primeiro encontrar o anfitrião do jogo na rede e então obter os dados necessários para entrar no jogo, tais como a figura a ser montada, o tempo limite para completar o jogo e os jogadores conectados.

Depois de conectados, os jogadores vão tentar montar as peças. A cada peça montada por um dos jogadores, deve ser disseminada uma mensagem aos outros jogadores informando tal ocorrência, para que eles não trabalhem numa peça que já foi montada. Desta forma, uma comunicação baseada em eventos é também necessária. Além disso, se um novo jogador entrar no jogo depois do seu início, ou se um jogador desconectar por um certo tempo e depois voltar a participar do jogo, eles devem ter a possibilidade de obter o estado atual do jogo.

Em adição, quando um jogador (J2, por exemplo) desconecta, ele deve continuar apto a montar seu quebra-cabeças, apesar de suas jogadas não poderem ser disseminadas aos outros jogadores. Assim que J2 voltar à rede, todas as suas mudanças no quebra-cabeça devem ser visualizadas por J1 e J3, assim como as mudanças feitas por J1 e J3 devem ser disponibilizadas para J2. O jogo deve também prover aos

jogadores um meio de bate-papo, isto é, enviar mensagens de texto para todos os outros jogadores ou para algum específico.

4.4.2 Projeto da Aplicação

A aplicação deve ser instalada em cada um dos dispositivos que irão participar do jogo. A aplicação provê toda lógica do jogo de quebra-cabeças, incluindo imagens, gerenciamento de jogadores e configurações de partidas. Já o Multi-MOM é usado como infraestrutura de comunicação, com o objetivo de facilitar a comunicação entre dispositivos, e livrar os desenvolvedores da aplicação de ter que se preocupar com detalhes de baixo nível da comunicação em rede.

Nessa aplicação, toda imagem do quebra-cabeça a ser montada é subdividida em partes menores. Essas partes menores são as peças do quebra-cabeça, e cada uma possui um número identificador. Assim, se J2 consegue colocar uma peça no seu lugar correto, ou seja, montar aquela peça, a aplicação precisa apenas que o número da peça montada por J2 seja disseminado na rede. Os dispositivos de J1 e J3 receberão uma notificação de que aquela peça com identificador especificado foi montada, e em seguida atualizarão a visão local destes jogadores. A Figura 32 ilustra uma imagem do quebra-cabeça dividida em partes menores, cada uma com seu identificador. Note que esses identificadores não aparecem para os jogadores, esses números são mostrados na figura apenas para ilustrar como a aplicação gerencia suas peças.

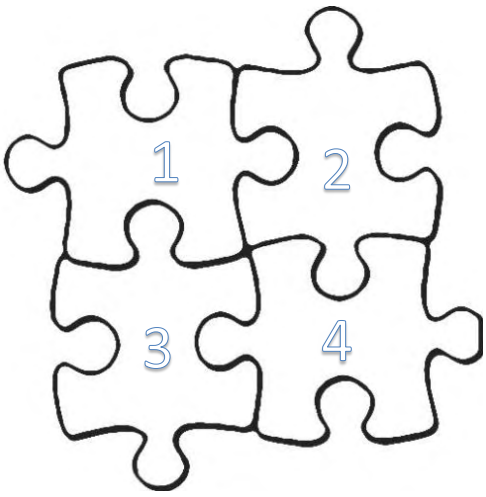

Figura 32 – Organização das peças do jogo quebra-cabeça

Para dar suporte a essa aplicação, os objetos que serão utilizados na comunicação entre os dispositivos são modelados na Figura 33. Note que *StateUpdate* é

o único que não possui relacionamento direto com os outros objetos. Isto acontece porque este objeto será utilizado para transportar as atualizações de estado do jogo. Assim, cada vez que um jogador montar uma peça, a aplicação indicará o número da peça no campo *assembledPieceID* e transmitirá apenas esse número para os outros jogadores. Observe que esse campo, *assembledPieceID*, é obtido a partir do campo *pieceID* do objeto *ImagePiece*. Já o restante dos objetos será utilizado para um jogador entrar no jogo, pois especificam todas as informações necessárias para iniciar uma partida.

A aplicação descrita acima faz uso de pelo menos três paradigmas de comunicação: espaço de tuplas, publish/subscribe e notificações ponto-a-ponto. Primeiro, J1, o anfitrião do jogo, cria e registra na rede um espaço compartilhado no qual os dados sobre o jogo são disponibilizados para os outros jogadores. Este espaço compartilhado corresponde a um serviço de espaço de tuplas. No espaço de tuplas, os objetos mostrados na Figura 33 serão encapsulados na forma de objetos do tipo *message*. É importante lembrar que todo transporte de dados através do middleware utiliza objetos *message*. Assim, os objetos da aplicação supracitados serão carregados no corpo das mensagens.

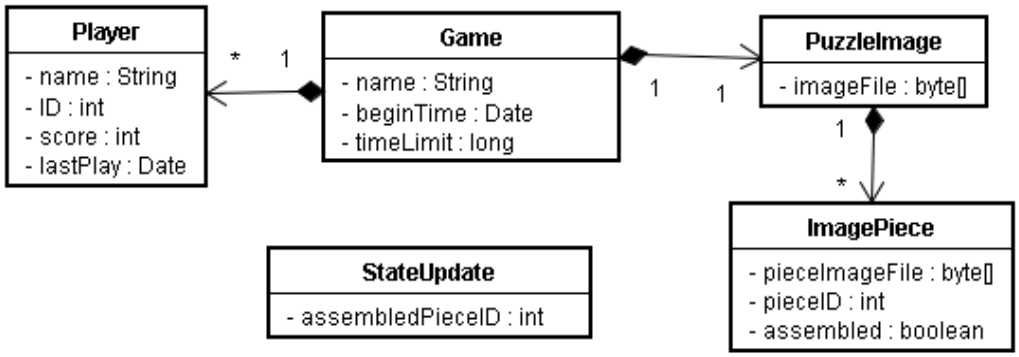

Figura 33 - Projeto da aplicação cliente

Quando a aplicação cria o espaço de tuplas, o middleware registra este serviço na rede através do componente *service locator*. Assim, quando os outros jogadores fizerem uma busca na rede por informações sobre o jogo, o middleware executando no dispositivo de J1, o anfitrião do jogo, responderá informando o endereço do espaço de tuplas para que possam ser obtidas as mensagens com informações sobre o jogo.

Os eventos gerados pelos jogadores correspondem a montagem de uma nova peça no quebra-cabeças. Tais eventos são representados pelo objeto *StateUpdate*, o qual é empacotado num objeto *message*, e então distribuído para todos os jogadores através do serviço publish/subscribe. É importante observar que todos os jogadores são publicadores e ao mesmo tempo assinantes desse serviço, já que eles devem produzir eventos quando uma nova peça é montada, como também eles devem receber atualizações de montagem das peças dos outros jogadores. Sempre que um jogador recebe uma atualização de estado (*StateUpdate*), sua visão local do jogo é também atualizada.

A Figura 34 mostra o pedaço de código relacionado à criação e registro dos serviços de espaço de tuplas e publish/subscribe. A referência *middlewareServiceLocator* representa o objeto que dá acesso ao middleware, ele é obtido através do mecanismo de IPC provido pelo Android. No método *createService()*, o primeiro parâmetro indica o nome do serviço sendo criado e registrado na rede, enquanto o segundo parâmetro indica o paradigma de comunicação deste serviço.

```
1. middlewareServiceLocator.createService (“PuzzleGame”, new TupleSpaceParadigm() );
2. middlewareServiceLocator.createService (“PuzzleGame”, new PublishSubscribeParadigm() );
```

Figura 34 - Criação de serviços de mensagens

A criação de tais serviços é feita somente pelo anfitrião do jogo (J1), o qual vai prover os serviços de espaço de tuplas e publish/subscribe. A partir daí, J2 e J3 devem obter as referências remotas para estes serviços (linhas 1 e 2 da Figura 35). Em seguida, para evitar o uso de reflexão em cada chamada ao serviço, as linhas 3 e 4 criam uma sessão com esses serviços. Internamente ao middleware, a criação de sessões é feita pela associação das referências aos serviços (objeto *service reference*) com o subtipo de *service engine* adequado. Por exemplo, sempre que uma operação for realizada utilizando a referência de serviço identificada como *tSpaceRef*, o middleware tem gravada uma associação da mesma com um *TupleSpaceEngine*, e assim encaminha a operação diretamente para ele, sem precisar fazer uso de reflexão.

```
1. ServiceReference tSpaceRef = middlewareServiceLocator.findService("PuzzleGame", new
   TupleSpaceParadigm() );
2. ServiceReference pubsubRef = middlewareServiceLocator.findService("PuzzleGame", new
   PublishSubsribeParadigm() );
3. TupleSpaceSession tSpaceSession = middlewareServiceLocator.createSession(tSpaceRef);
4. PublishSubscribeSession pubsubSession = middlewareServiceLocator.createSession(pubsubRef);
5. Properties prop = new Properties();
6. prop.setProperty("GameCurrentState");
7. Message gameCurrentStateTuple = tSpace.readTuple(prop);
8. pubsubSession.subscribe();
9. Message msg = pubsubSession.createMessage();
10. msg.setBody(stateUpdate);
11. pubsubSession.sendMessage(msg);
```

Figura 35 - Acessando os serviços de mensagens

Depois disso, a aplicação pode invocar operações que irão de fato enviar e receber mensagens. Neste contexto, para que J2 e J3 possam começar a jogar, eles precisam requisitar o estado atual do jogo, utilizando o espaço de tuplas compartilhado (Figura 35, linhas 5 a 7). Em adição, as linhas 8 a 11 mostram o código necessário para inscrever-se num serviço publish/subscribe, bem como enviar as atualizações de estado geradas pelo jogador.

É importante observar que, na distribuição de atualizações usando o paradigma publish/subscribe, os dados são disponibilizados de forma transiente, ou seja, somente no momento que são publicados. Assim, para possibilitar que novos jogadores entrem no jogo depois do mesmo ter início, o estado global do jogo deve ser mantido sempre atualizado no espaço de tuplas. Além disso, se um jogador perder conexão por algum tempo, ele pode ainda continuar montando o quebra-cabeças. Neste caso, a aplicação continua gerando mensagens de atualizações (S*tateUpdate*), enquanto que o middleware vai guardando estas mensagens localmente no *buffer* de envio até que a conexão seja re-estabelecida, e ele possa então transmitir estas mensagens.

Por fim, se um jogador deseja bater papo através de mensagens de texto, um serviço de notificações ponto-a-ponto pode dar suporte a esta funcionalidade. Para isso, cada jogador deve criar localmente seu serviço de notificações ponto-a-ponto, para que recebam mensagens enviadas diretamente por outros dispositivos, sem ter que passar por um serviço intermediário.

4.4.3 Considerações sobre o cenário de aplicação

Jogos estão entre os tipos de aplicações mais populares encontrados no Android Market [Android10b], a loja *online* de aplicativos da plataforma Android. Muitos desses jogos já dão suporte ao modo multi-jogador, no qual vários jogadores, cada um com seu dispositivo, fazem parte de um único jogo, interagindo através da rede. O jogo de quebra-cabeças apresentado é baseado numa aplicação descrita em [Murphy06], o qual apresenta as características de uma classe geral de aplicações em que o compartilhamento e a distribuição de dados são elementos chave. Esta estratégia de compartilhamento e distribuição de dados pode facilmente ser adaptada para outros domínios de aplicação, tal como compartilhamento de mídias e edição colaborativa de documentos.

Como pôde ser visto, o Multi-MOM apresenta uma API simples e intuitiva para a programação em rede utilizando múltiplos paradigmas de comunicação baseado em mensagens. Desta forma, desenvolvedores de aplicações ficam livres para projetar a lógica específica da sua aplicação, e não tem que reimplementar, a cada aplicação, funcionalidades que são comuns para toda aplicação em rede.

Sem a utilização de um sistema de middleware, como o Multi-MOM, o desenvolvimento desses tipos de aplicações exigiria programação em baixo nível e conhecimentos distintos do domínio da aplicação em desenvolvimento, a qual geralmente utiliza a comunicação em rede apenas para dar suporte às suas funcionalidades. Assim, o Multi-MOM atua como uma plataforma reusável para aplicações móveis distribuídas, evitando que o código relativo à programação em rede fique em segundo plano, o que a tornaria repetitiva e mais predisposta a erros.

4.5 Considerações Finais

A implementação de sistemas distribuídos para computação móvel é uma tarefa complexa que requer o tratamento de diversos requisitos que não são encontrados normalmente no desenvolvimento de aplicações tradicionais. Entre eles, está o tratamento para as freqüentes desconexões devido à mobilidade de dispositivos e a necessidade de prover sistemas de baixa carga computacional, adequados para executar em dispositivos de capacidades bem mais limitadas que os computadores tradicionais.

Nesse contexto, esse capítulo apresentou detalhes da implementação do Multi-MOM, mostrando como os componentes especificados no Capítulo 3 foram mapeados para efetiva utilização num dispositivo móvel. A escolha das tecnologias envolvidas levou em conta principalmente a viabilidade de utilização das mesmas na prática, como no caso da plataforma Android, por exemplo, que vem sendo utilizada cada vez mais por diversos fabricantes de *smartphones*. Igualmente, o banco de dados embarcado SQLite é adotado por padrão em diversos sistemas operacionais para dispositivos móveis, tal como iPhone OS, Symbian, BlackBerry, bem como por aplicativos bem conhecidos, como Mozilla Firefox e Skype.

Este capítulo mostrou também como é feita a seleção de funcionalidades no Multi-MOM. Esta seleção de funcionalidades permite escolher quais paradigmas de comunicação serão providos em cada instância do middleware. A disponibilidade de múltiplos paradigmas de comunicação permite que o middleware atenda a diversos cenários de aplicação distintos. Essa abordagem facilita o desenvolvimento de aplicações móveis distribuídas, já que os desenvolvedores de aplicações não têm que adquirir, ou desenvolver, um novo sistema de middleware para cada tipo de aplicação. Ao invés disso, podem contar com uma plataforma unificada e extensível que atende a diversos domínios de aplicação.

Não obstante, caso a aplicação necessite de uma forma de interação que não é oferecida por nenhum dos paradigmas disponíveis, pode-se facilmente estender o middleware com muito menos esforço do que seria gasto na implementação de um novo sistema de middleware. Por outro lado, o Multi-MOM pode ser customizado para melhor atender a dispositivos com maiores limitações. Isto é feito pela redução da quantidade de paradigmas de comunicação oferecidos, diminuindo assim a quantidade de memória exigida pelo middleware.

Por fim, o cenário de aplicação descrito demonstra a necessidade de utilizar-se uma abordagem multi-paradigma para facilitar o desenvolvimento de aplicações móveis, ilustrando como uma aplicação que necessita de variadas formas de interação em rede pode se beneficiar do Multi-MOM. Com a utilização do Multi-MOM, evita-se que desenvolvedores de aplicações tenham que lidar com os problemas de comunicação em redes móveis. Ao invés disso, os desenvolvedores utilizam apenas as operações providas pelo middleware, numa API simples e intuitiva, deixando todo trabalho repetitivo e de baixo nível para o middleware.

5 Avaliação

Constantemente, visando resolver ou amenizar os diversos problemas encontrados no desenvolvimento de software, novas soluções e tecnologias são apresentadas aos engenheiros de software. Entretanto, segundo [Travassos02], novos métodos, técnicas, linguagens e ferramentas não deveriam ser apenas sugeridos, publicados ou apresentados para a venda sem experimentação. De forma complementar, [Babar04] afirma que mesmo as tecnologias desenvolvidas baseadas em boas práticas ou padrões consolidados não podem assegurar que os resultados esperados sejam obtidos. Portanto, além de desenvolver a tecnologia, é também importante avaliar experimentalmente se a mesma atende aos objetivos e requisitos definidos inicialmente.

Neste contexto, este capítulo apresenta uma avaliação do middleware com relação ao espaço ocupado em memória e o nível de reuso das funcionalidades comuns entre os paradigmas de comunicação. Além disso, este capítulo também descreve uma avaliação de desempenho, na qual é medida a vazão de mensagens para cada um dos paradigmas de comunicação.

5.1 Espaço em Memória e Reusabilidade do Código

Esta parte da avaliação tem os seguintes objetivos:

- Verificar a efetividade do projeto arquitetural, considerando o reúso do código comum na adição/remoção de paradigmas de comunicação;
- Avaliar a adequação do middleware proposto para os tipos de dispositivos aos quais se pretende dar suporte.

Para isso, foram criadas cinco instâncias diferentes do Multi-MOM. Cada uma dessas instâncias inclui o código comum entre os paradigmas e o código específico de apenas um dos paradigmas de comunicação. Assim, existe uma instância que possui apenas o paradigma Publish/Subscribe, outra que possui apenas o paradigma Espaço de Tuplas, e assim por diante. A partir dessas diferentes instâncias, foi medida a

quantidade de código específico de cada paradigma de comunicação, bem como a quantidade de código que é comum entre tais paradigmas.

Para medir o nível de reuso atingido pelo projeto arquitetural, foi utilizada a métrica de reuso definida em [Frakes05]. Esta métrica é obtida pela divisão da quantidade de linhas de código reusado pela quantidade total de linhas de código. Assim, o cálculo de nível de reuso no Multi-MOM foi feito da seguinte forma:

- *(código comum entre os paradigmas) / (código total de cada instância)*

O **código comum** resultou em 2019 linhas. Já o **código total** de cada instância é mostrado na coluna "Código total" da Tabela 2, obtido pela soma da coluna "Código específico" com a quantidade de código comum (2019). Por exemplo, o nível de reuso da instância Publish/Subscribe foi calculado da seguinte forma: 2019 / 2347. A coluna "código específico" representa o código relacionado especificamente a cada um dos paradigmas.

Toda medição de tamanho de código fonte foi obtida com o suporte da ferramenta Eclipse Metrics plugin.

Tabela 2 - Linhas de código e reusabilidade

Instância	Código comum (linhas)	Código específico (linhas)	Código total (linhas)	% Reuso
Publish/Subscribe	2019	328	2347	86,02%
Fila de Mensagens		255	2274	88,8%
Espaço de Tuplas		264	2283	88,4%
Notificações PaP		248	2267	89,06%
Síncrono		255	2274	88,8%
Middleware Completo		1074	3093	65,2%

Estes números mostram que, na implementação de um paradigma de comunicação específico, o código adicionado, relativo a esse paradigma, representa por volta de 13% do código total, resultando num nível de reuso médio de 87%. Na literatura de engenharia de software, é estimado que de 40% a 60% de código é comum entre aplicações do mesmo domínio [Frakes05]. No Multi-MOM, mesmo na instância do middleware que provê 5 paradigmas de comunicação distintos (Middleware

Completo), 65% do código é comum e compartilhado entre esses paradigmas. Neste sentido, as funcionalidades comuns do middleware podem ser consideradas efetivamente reusáveis.

Utilizando novamente o conjunto de instâncias definidos na Tabela 2, a Tabela 3 mostra os valores de espaço ocupado em memória para cada instância do middleware, incluindo: o tamanho do arquivo em memória flash (estático), isto é, a memória não-volátil do dispositivo; e o tamanho do processo em execução na memória RAM (dinâmico). O tamanho do arquivo estático foi obtido do arquivo gerado como unidade de implantação, que no Android é um arquivo .apk, enquanto que o tamanho dinâmico foi obtido do processo em execução, como a diferença retornada entre os métodos *java.lang.Runtime totalMemory()* e *freeMemory()*. É importante destacar que os valores do processo em execução foram obtidos depois da inicialização e criação de um serviço de cada paradigma, como forma de garantir que, uma e somente uma instância de cada classe disponível naquela instância estaria em memória.

Tabela 3 - Espaço ocupado em memória

Instancia	Tamanho na memória flash	Tamanho em tempo de execução (RAM)
Aplicação vazia	13 KB	2372,9 KB
Publish/Subscribe	95,8 KB	2517,3 KB
Fila de Mensagens	95,1 KB	2512,8 KB
Espaço de Tuplas	95,1 KB	2513,1 KB
Notificações PaP	95,2 KB	2514,9 KB
Síncrono	95,2 KB	2513,9 KB
Middleware Completo	101 KB	2587,9 KB

Além das instâncias criadas anteriormente, contendo um dos paradigmas de comunicação por vez, foi criada também uma “Aplicação vazia”. O objetivo da criação dessa aplicação é dar uma idéia de qual é o tamanho mínimo que uma aplicação Android pode ter. Assim, a partir dela, podemos saber quanto do tamanho final do middleware é dado pelos arquivos criados por padrão pelo Android e sua máquina virtual, e quanto é relacionado ao código implementado pelo Multi-MOM. Essa aplicação não contém nenhum código do Multi-MOM, mas apenas o código mínimo

necessário para se ter uma aplicação Android, similar ao que seria uma aplicação Java comum, com uma única classe, que contém apenas o método *main()* em branco.

Na Tabela 3, podemos perceber que a maioria das instâncias do middleware ocupa em memória flash por volta de 95 KB, enquanto a aplicação vazia apresentou tamanho de 13 KB. Assim, para saber exatamente quanto o código de cada instancia do middleware representa no tamanho total do arquivo gerado, basta subtrair 13 de cada um dos números da coluna "Tamanho na memória flash". Da mesma forma pode ser feito para a coluna "Tamanho em tempo de execução". Neste caso, subtraindo-se a quantidade de memória total requerida pelo Multi-MOM em **tempo de execução** (por volta de 2500 KB) pelo valor requerido por uma aplicação vazia (2372 KB), percebe-se que o Multi-MOM adiciona apenas 200 KB, em média, em tempo de execução. Ou seja, dos 2500 KB do tamanho ocupado em memória pelo Multi-MOM, por volta de 2372 KB desse valor é relativo especificamente à máquina virtual do Android.

Assim, comparando esses valores, o tamanho do Multi-MOM é apenas 6% maior do que o tamanho de uma aplicação vazia em tempo de execução, o que leva à conclusão de que o espaço ocupado em memória pelo middleware proposto é compatível com outras aplicações Android.

A última linha da Tabela 2 e da Tabela 3 refere-se à instância do middleware contendo todos os paradigmas de comunicação juntos (Middleware Completo). Nesta instância, enquanto o total de linhas de código cresceu por volta de 25%, comparado às outras instancias provendo um paradigma de cada vez, o tamanho do arquivo estático (memória flash) cresceu apenas 5% e o tamanho do processo em execução em torno de 3%.

Esse crescimento discreto deve-se ao fato de que o Android converte os arquivos .class num formato mais compacto, chamado Dalvik Executable (.dex), que é mais adequado para sistemas de recursos escassos. Ao contrário do formato .class utilizado no Java comum, os arquivos .dex utilizados pelo Android incluem múltiplas classes num único arquivo .dex. Além de diminuir a quantidade de arquivos gerados, esse formato elimina redundâncias geradas pelo compilador Java padrão.

Por fim, os aparelhos Android encontrados no mercado atualmente apresentam configurações de memória não-volátil medida em gigabytes e uma média de memória RAM de 256 MB [GSMArena10]. Considerando estas configurações, os resultados

apresentados pelas medições de espaço ocupado em memória podem ser considerados bastante satisfatórios. Conseqüentemente, pode-se dizer que o tamanho do middleware é adequado para os dispositivos aos quais ele foi projetado.

5.2 Avaliação de Desempenho

O propósito da análise de desempenho é avaliar a vazão de mensagens do middleware proposto, a partir de uma perspectiva externa, isto é, a partir da visão das aplicações clientes fim-a-fim. Para isso, foi medida a vazão na troca de mensagens nos cinco paradigmas de comunicação considerados neste trabalho, primeiro usando apenas serviços locais, e em seguida, incluindo serviços acessíveis através da rede.

Duas aplicações foram criadas. A primeira aplicação (aplicação servidora) cria junto ao middleware os serviços de mensagens, tal como serviço de espaço de tuplas ou serviço publish/subscribe. A segunda aplicação (aplicação cliente) acessa estes serviços para enviar/receber mensagens. É importante destacar que as medições de cada paradigma seguem a sua própria dinâmica de interação, por exemplo, no espaço de tuplas uma aplicação precisa colocar a mensagem no espaço compartilhado, e depois essa mensagem deve ser retirada, enquanto no de notificações ponto-a-ponto, a mensagem sai de uma aplicação diretamente para outra. Desta forma, foi seguido o princípio de que cada troca de mensagem representa a interação mínima necessária para: produzir a mensagem, passar pelo middleware e em seguida consumir a mensagem.

Os experimentos foram feitos através da troca de 100 mensagens e do cálculo do número médio de mensagens trocadas por segundo. Cada um desses testes foi repetido entre 10 e 15 vezes, até que se atingisse um intervalo de confiança de 95% e um erro padrão de 5%. Antes do início das medições, as primeiras 30 trocas de mensagens foram desconsideradas até que o sistema atingisse um estado estável. Os valores obtidos desses experimentos são detalhados a seguir, nas Subseções 5.2.1 e 5.2.2.

5.2.1 Serviços Locais

A Figura 36 mostra os resultados das trocas de mensagens num cenário onde não há interação com a rede, simulando uma situação em que uma aplicação está enviando uma mensagem para um serviço que está sendo executado no próprio dispositivo. A intenção neste cenário é medir quão rápido as mensagens podem passar pela pilha de componentes do middleware, excluindo o atraso da rede.

Estes experimentos foram executados usando o emulador Android do Google [Android10a], executando num *laptop* com Windows Vista, processador Intel Pentium Dual Core, 1.6GHz e 1 GB de memória RAM. A vazão foi medida para diferentes tamanhos de carga útil (*payload*) das mensagens: 0, 1024, 4096, 8192 e 16384 bytes.

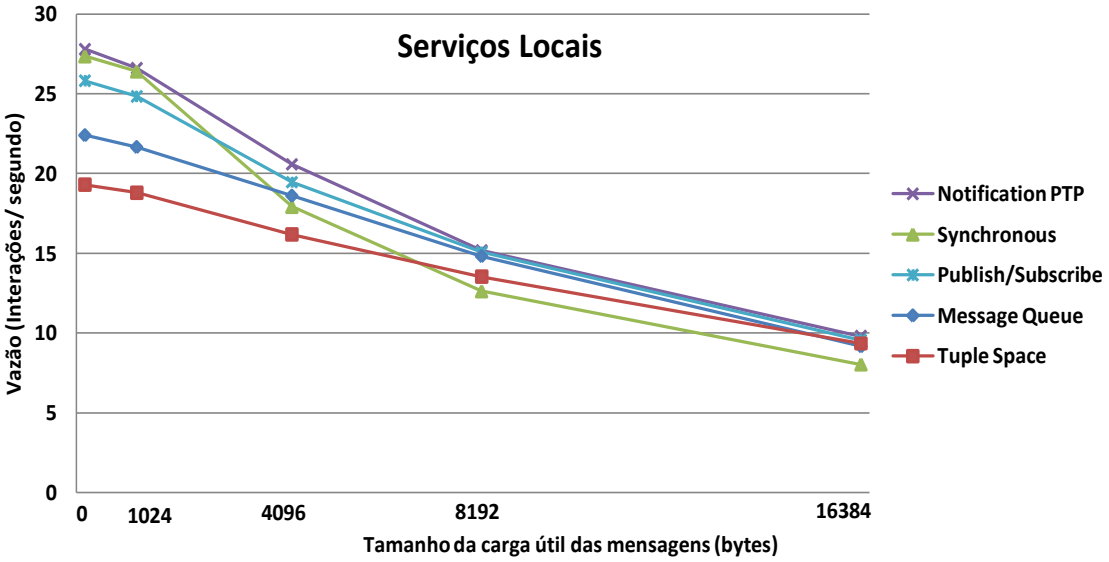

Figura 36 - Vazão de troca de mensagens utilizando serviços locais

No gráfico da Figura 36 é possível perceber que a vazão do sistema cai à medida que o tamanho das mensagens aumenta. Neste cenário, apesar de não ter que transmitir as mensagens pela rede, mais processamento tem que ser feito para mensagens maiores. Esse processamento consiste em passar a mensagem da aplicação origem para o middleware, fazer todo o processamento interno de cada paradigma de comunicação e então tornar essa mensagem disponível para aplicação destino.

É importante destacar que as mensagens são passadas das aplicações para o middleware (e vice-versa) através de chamadas entre processos (IPC). Para isto, os dados da mensagem são empacotados, transferidos de um processo para outro, e então desempacotados, similar ao processo de serialização / deserialização do Java. Além disso, o tamanho do *heap* de memória aumenta junto com o tamanho das mensagens, e assim a máquina virtual em que o middleware está sendo executado precisa fazer coleta de lixo (*garbage collection*) com mais freqüência do que com mensagens pequenas.

5.2.2 Serviços Remotos

Neste cenário, as aplicações comunicantes estão executando em dois dispositivos diferentes, conectados por meio de um rede sem fio 802.11g (Wi-Fi). O primeiro

dispositivo é emulado no mesmo computador utilizado no cenário anterior, enquanto o segundo dispositivo é emulado num outro *laptop* com Windows Vista, Intel Core 2 Duo, 1.6GHz e 2 GB de RAM. Como as aplicações estão executando em dois dispositivos diferentes, cada um deles deve ter uma instância do middleware executando localmente.

Enquanto no cenário anterior, toda troca de mensagens era feita localmente, entre aplicações dentro de um mesmo dispositivo, neste cenário, toda troca de mensagens é feita através da rede. Como dito anteriormente, o objetivo é simular a interação mínima necessária para produzir uma mensagem e consumi-la. No cenário de serviços remotos, as mensagens são produzidas e enviadas para um serviço remoto (remoto na perspectiva da aplicação produtora), e também consumidas, ou seja, recuperada ou notificada, a partir de um serviço remoto (remoto na perspectiva da aplicação consumidora).

É importante destacar que as formas de interação entre aplicações – produção e consumo de mensagens – seguem as características de cada paradigma. A Figura 37 ilustra essas formas de interação. Nesta figura, cada retângulo representa um dispositivo onde está sendo executada uma parte do sistema, podendo esta parte ser a aplicação produtora, aplicação consumidora, ou um serviço intermediário, como o espaço de tuplas, por exemplo. Já as setas que chegam e saem desses retângulos representam a passagem de mensagens de um dispositivo para outro, através da rede.

Nestes cenários, além de incluir todo o processamento que já era feito no cenário de serviços locais, é acrescentado, no mínimo, mais um processo de serialização / deserialização para transportar as mensagens de um dispositivo para outro através da rede, bem como o estabelecimento de conexões e o transporte propriamente dito. Conseqüentemente, como mostrado na Figura 38, a vazão nestes cenários tende a ser menor que os valores obtidos nos serviços locais. Entretanto, comparado aos resultados dos serviços locais, a queda da vazão para mensagens maiores é bem menor no cenário de serviços remotos. Este fato leva a conclusão de que o *overhead* maior deste cenário está associado ao estabelecimento de conexões dos sockets TCP.

É importante destacar também que a execução dos testes dentro de um emulador impõe naturalmente mais um *overhead*, principalmente na comunicação em rede, já que o emulador Android não tem acesso direto à rede, sendo necessária a utilização de um

proxy – Simple TCP datapipe [Auriemma10] – para fazer as mensagens saírem do emulador para outros computadores da rede local.

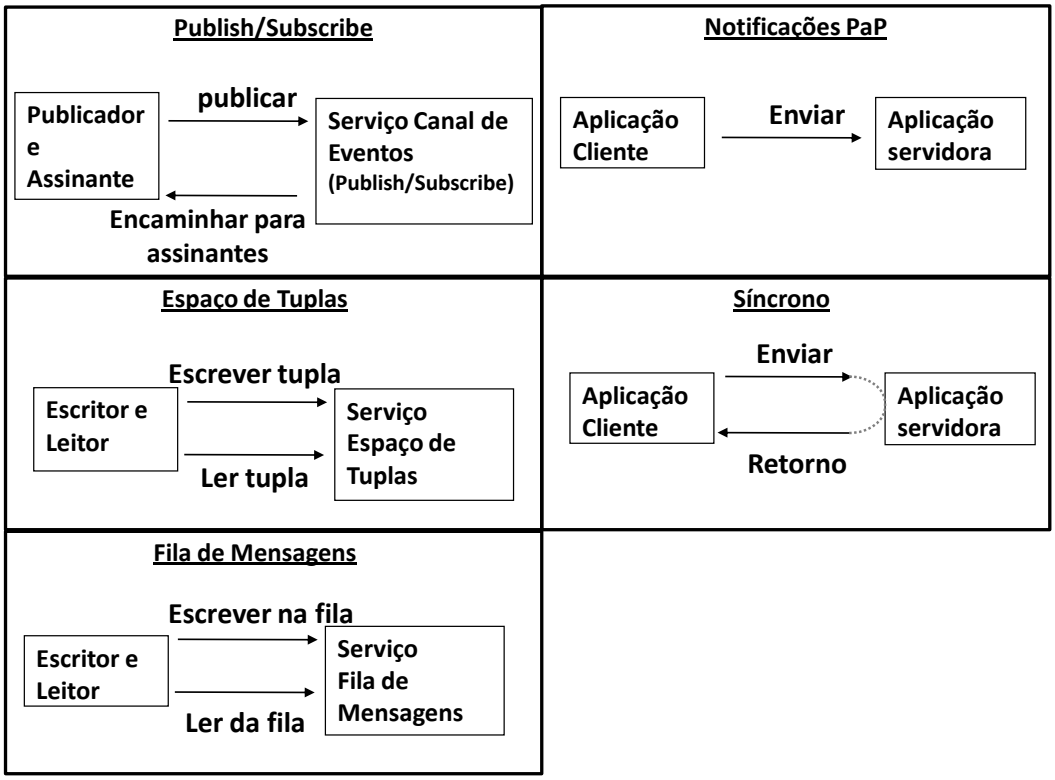

Figura 37 - Interações entre serviços remotos

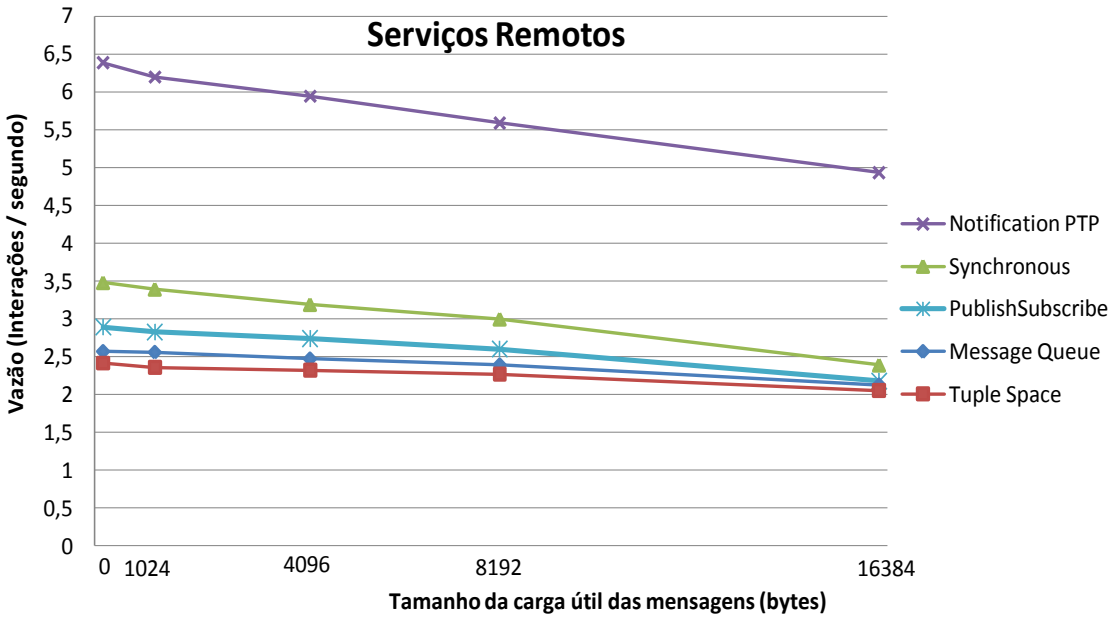

Figura 38 - Vazão de troca de mensagens utilizando serviços remotos

No gráfico acima, os valores da vazão obtidos pelo paradigma Notificações ponto-a-ponto chamam atenção por serem bem acima da média dos outros paradigmas. Estes resultados são alcançados devido ao seu modelo de interação unidirecional. Isto é, uma aplicação envia a mensagem e a outra recebe a mensagem do outro lado, havendo apenas uma passagem pela rede (Figura 37). Por outro lado, nos outros paradigmas, as mensagens passam duas vezes pela rede. No paradigma síncrono, há uma mensagem enviada e uma outra retornada. Já nas filas de mensagens e espaço de tuplas, a aplicação envia a mensagem pela rede, e depois busca a mensagem num serviço remoto. Por fim, no paradigma Publish/Subscribe as mensagens são enviadas para o canal de eventos e depois encaminhadas para outros dispositivos com aplicações assinantes. Como resultado de um modelo de interação mais simples, no qual as mensagens passam pela rede somente uma vez, a vazão do paradigma Notificações ponto-a-ponto é consideravelmente maior.

5.2.3 Desempenho de Trabalhos Relacionados

Nesta seção são mostrados resultados de outros trabalhos existentes na literatura para cada um dos paradigmas de comunicação considerados nesse trabalho. Cada um dos trabalhos a seguir aborda apenas um paradigma de comunicação isoladamente. Com isso, o objetivo não é fazer uma comparação direta, já que o conjunto de funcionalidades implementadas não é o mesmo. Ao invés disso, pretende-se colocar os resultados do Multi-MOM em perspectiva, mostrando que a abordagem multi-paradigma proposta atende a diversos cenários de aplicação simultaneamente, sem prejudicar os resultados de desempenho.

A Tabela 4 mostra resumidamente esses trabalhos, classificando-os pelo paradigma de comunicação provido, e apresentando uma breve descrição do trabalho, da configuração utilizada nos experimentos, do tamanho das mensagens utilizadas e os resultados obtidos por cada um deles. É importante destacar que estes resultados foram obtidos a partir de gráficos encontrados em artigos publicados, desta forma, os valores mostrados abaixo podem não estar exatos.

A escolha dos trabalhos para essa comparação levou em conta a adequação do trabalho a algum dos paradigmas de comunicação do Multi-MOM, a disponibilidade de resultados de experimentos em artigos da literatura, e se os mesmos são projetados para computação móvel. Apesar disso, não foi possível encontrar sistemas de espaço de

tuplas e de paradigma síncrono para computação móvel com resultados de experimentos especificados em artigos. Assim, esses dois tipos de sistemas são para computação tradicional, utilizando computadores de mesa (*desktop*). Inclusive, esses dois trabalhos, [Wells06] e [Kuhns99], obtiveram resultados melhores que o Multi-MOM. É importante destacar que em [Kuhns99], as mensagens não passam pela rede, ao invés disso, é utilizado o modo *loopback* para transferir a mensagem de uma aplicação para outra dentro de um mesmo computador.

Tabela 4 - Comparação de desempenho com plataformas existentes

Paradigma	Descrição	Configuração	Tamanho da Mensagem	Resultado *(msg/seg)
Publish / Subscribe	Sistema publish/subscribe para redes wi-fi ad hoc [Yoneki03]	Pentium III, 800MHz, 392 MB RAM	250 KB	1,1
Fila de Mensagens	Sistema de fila de mensagens para dispositivos móveis [Jung99]	Pentium PC	Não informado	0,9
Espaço de Tuplas	Espaço de Tuplas para redes fixas, baseadas em servidor [Wells06]	Pentium 4, 3.2 GHz, 1GB RAM	24 bytes	32
Síncrono	Middleware configurável baseado em CORBA [Kuhns99]	Pentium II Xeon 400 MHz em modo loopback	2048 bytes	4

* Valores aproximados

Além disso, nenhum dos trabalhos analisados utiliza emuladores, ao invés disso, são testados diretamente nas máquinas descritas na Tabela 5. Já o Multi-MOM foi testado dentro de um emulador, simulando um dispositivo real, o que naturalmente adiciona um *overhead*, principalmente na comunicação em rede, que requer um *proxy*, como explicado na Seção 5.2.1. Apesar disso, o Multi-MOM apresenta resultados compatíveis com os resultados dos trabalhos descritos acima.

É importante destacar que o objetivo da demonstração dos resultados de trabalhos relacionados não é fazer uma comparação direta, já que, tanto as funcionalidades implementadas, como também a configuração utilizada nos testes é diferente. Assim, o objetivo é mostrar que os resultados obtidos pelo Multi-MOM estão na mesma ordem

de magnitude de outros trabalhos similares. Entretanto, enquanto esses trabalhos provêem apenas um paradigma de comunicação isoladamente, o Multi-MOM provê um conjunto extensível de paradigmas de comunicação, e ainda assim, apresenta resultados similares.

5.3 Considerações finais

Este capítulo trouxe uma avaliação do Multi-MOM em questões relacionadas ao reuso de funcionalidades comuns entre os paradigmas de comunicação, ao tamanho ocupado em memória e ao desempenho.

A análise relativa à reusabilidade quantificou o grau de reuso das funcionalidades comuns do middleware, demonstrando a viabilidade da definição de uma plataforma com um conjunto de funcionalidades padrão e ao mesmo tempo com possibilidade de seleção e extensão de certas funcionalidades. Os resultados obtidos nesta análise mostram que a quantidade de código necessário para oferecer um novo paradigma de comunicação é bem menor do que se tivesse que ser implementada uma nova plataforma de middleware, sem reuso. Assim, novos sistemas de middleware podem ser desenvolvidos com menor esforço e mais rapidez, economizando recursos e gerando sistemas de melhor qualidade, já que desenvolvedores não tem que se preocupar em resolver os mesmos problemas repetidamente. Ao invés disso, ao utilizar o Multi-MOM, a tendência é que desenvolvedores acrescentem novos recursos e dêem foco às particularidades de suas aplicações.

Quanto à análise de tamanho ocupado em memória, foi mostrado que mesmo a instância provendo cinco paradigmas de comunicação apresentou um tamanho pouco superior aos tamanhos das instâncias que ofereciam apenas um dos paradigmas de forma isolada. Além disso, analisando os dispositivos baseados em Android atualmente disponíveis no mercado, ficou constatado que o middleware proposto adéqua-se perfeitamente as restrições de recursos desses dispositivos.

Por fim, na análise de desempenho foram identificados os principais gargalos do middleware proposto, que concentram-se: (i) nas transferências de mensagens entre a aplicação e o middleware, através de IPC; (ii) e principalmente no estabelecimento de conexões sockets entre dois dispositivos. Além disso, resultados obtidos de trabalhos relacionados foram listados com o intuito de mostrar que o middleware proposto,

mesmo oferecendo múltiplos paradigmas, tem desempenho comparável a outras plataformas especializadas num paradigma único.

6 Trabalhos Relacionados

O presente capítulo tem o objetivo de apresentar os principais trabalhos relacionados ao tema desta dissertação e compará-los com a solução proposta, analisando principalmente as funcionalidades definidas para tratar problemas comumente enfrentados na computação móvel, bem como o suporte a customização para atender a grande variedade de requisitos desses tipos de aplicações. Ao final do capítulo, é apresentada uma tabela que sintetiza as principais características de cada um desses trabalhos.

6.1 Java Message Service (JMS)

O Java Message Service (JMS) [Sun02] é uma especificação desenvolvida pela Sun Microsystems que provê uma API comum para aplicações Java se comunicarem através de mensagens. Assim como o Multi-MOM, a especificação do JMS provê uma interseção das principais funcionalidades comuns em sistemas de mensagens, tal como persistência e tempo de vida das mensagens, além de uma maneira comum para criação, envio e recebimento de mensagens utilizando dois paradigmas de comunicação: Fila de Mensagens e Publish/Subscribe.

No JMS há uma clara distinção entre os papéis de cliente e provedor dos serviços de mensagens. Os clientes podem enviar e receber mensagens acessando o provedor JMS, que é encarregado do gerenciamento e entrega dessas mensagens. Desta forma, os clientes são providos por componentes de baixa carga computacional, enquanto que as funcionalidades de servidor são geralmente implementadas em computadores com maior capacidade de recursos.

Apesar dessa abordagem ser adequada para sistemas de informação de grande porte, em redes móveis, nem sempre é possível contar com a presença de servidores de grande porte, pois é comum que dispositivos móveis se comuniquem diretamente uns com os outros. Diferentemente do JMS, o Multi-MOM, é projetado para operar em ambientes *peer-to-peer*, onde os dispositivos da rede têm papeis similares, e, assim,

podem se comunicar diretamente, oferecendo funcionalidades de servidor embutidas em cada dispositivo da rede, como persistência e descoberta de serviços.

No JMS, os clientes obtêm informações de serviços de mensagens existentes na rede através do Java Naming and Directory Interface (JNDI), um serviço de diretório que centraliza o registro de serviços. Esse serviço oferece transparência sobre a localização dos serviços de mensagens, mas por outro lado, exige que as aplicações saibam previamente o endereço deste serviço de diretório. O Multi-MOM, ao invés disso, utiliza um protocolo para descoberta de serviços que permite a descoberta de serviços sem nenhum conhecimento prévio sobre as redes, e sem a necessidade de um diretório central.

Por fim, a criação de serviços de mensagens no JMS (filas e tópicos) é feita de forma administrativa, ou seja, um administrador do sistema deve previamente criar tais serviços e registrá-los no serviço de diretório. No Multi-MOM, os serviços de mensagens são criados dinamicamente e sob demanda pelas aplicações cliente. Em resumo, apesar do JMS ser um padrão ‘de fato’ para aplicações de grande porte baseadas em mensagens, ele não é adequado para ambientes de alta dinamicidade, como os cenários das redes móveis.

6.2 JMS on Mobile Ad-Hoc Networks

Definido pelos autores como o primeiro middleware Java construído para redes móveis *ad hoc* (*Mobile Ad Hoc Networks* - MANETs), o “JMS on Mobile Ad-Hoc Network” [Vollset03] provê uma adaptação do JMS (*Java Message System*) para redes *ad hoc*. Este middleware tem como base o modelo de mensagens do JMS, tratando as características relativas à configuração e ao transporte de mensagens.

O transporte de mensagens é realizado utilizando a semântica publish/subscribe, na qual tópicos de interesse são mapeados em endereços *multicast*. Para isso, é implementado um protocolo de roteamento *multi-hop* no nível de aplicação, chamado Jomp (*Java on-demand multicast routing protocol*), que é baseado num protocolo já existente, chamado ODMRP (*On Demand Multicast Routing Protocol*). Apesar de ser baseado em JMS, esse sistema não implementa o paradigma de filas de mensagens, mas apenas o publish/subscribe.

Quanto às características de configuração, essa adaptação do JMS provê uma semântica de tópicos do tipo não-persistente, em que todos os dispositivos que participam de uma troca de mensagem devem ter uma cópia idêntica do arquivo de configuração. Esse arquivo de configuração contém informações sobre os tópicos publish/subscribe disponíveis para as aplicações. Apesar dessa abordagem tirar a necessidade da intervenção de um administrador de rede para configuração dos serviços, ela não traz flexibilidade. Ao invés de um administrador ter que criar os serviços, todo dispositivo passa a ter que ser configurado manualmente. Na direção oposta, no Multi-MOM, toda descoberta e criação de serviços – seja para tópicos publish/subscribe ou serviços de qualquer outro paradigma – é feita dinamicamente através de um protocolo de descoberta de serviços.

Outra restrição desse trabalho é que ele é baseado numa implementação do JMS chamada Arjuna-MS [Muro05], a qual exige dispositivos de alta capacidade computacional, como servidores. Além disso, o mesmo é implementado utilizando o Java padrão, sem utilizar nenhuma plataforma específica para dispositivos móveis, o que dificulta sua implantação em dispositivos reais.

6.3 Pronto

Pronto [Yoneki03] é um sistema de middleware para aplicações móveis que é também baseado no JMS. O Pronto propõe adaptações para o JMS, de forma que ele possa ser utilizado por dispositivos móveis de forma centralizada ou descentralizada. Na forma centralizada, chamada de "MobileJMS Client", é projetado um cliente de serviços JMS capaz de operar em dispositivos limitados. Assim, são retirados desse cliente algumas funções consideradas pelo autor como não essenciais, como por exemplo, alguns subtipos de mensagens e a propriedade que define a prioridade das mesmas.

Já na forma descentralizada, chamada "Serverless JMS", o objetivo é poder utilizar serviços de mensagens sem a presença de um servidor na rede. Para isso, o registro de serviços é feito de forma descentralizada, similar ao Multi-MOM, onde a busca por serviços é feita em *multicast*, e os dispositivos provendo serviços respondem em *unicast* informando o endereço dos seus serviços. Além disso, o transporte de mensagens é realizado também através de *multicast*. Entretanto, apesar dessa forma de transmissão ser eficiente para comunicações de "um para muitos", seria necessário a definição de funcionalidades extras para garantir a entrega de mensagens e controlar

quais clientes receberam cada mensagem, o que não é definido no Pronto. Como resultado dessa falta de controle de erros, o Pronto não tem como prover funcionalidades de persistência de mensagens e controle de TTL.

Além disso, o Pronto, assim como "JMS on MANETs", não implementa o paradigma de fila de mensagens do JMS, sendo esses dois baseados unicamente no paradigma publish/subscribe. Essa restrição limita os cenários de aplicações aos quais eles podem dar suporte. Já o Multi-MOM, provê cinco paradigmas de comunicação, além ser facilmente adaptável para suportar novos paradigmas, ou mesmos para selecionar um subconjunto entre os paradigmas existentes.

6.4 LIME

O LIME (*Linda in a Mobile Environment*) [Murphy06] é um middleware de compartilhamento de informação que estende o modelo de espaço de tuplas original do Linda [Gelertner85] e o torna mais adequado para ambientes móveis de alta dinamicidade. Ele define um espaço de tuplas permanente para cada dispositivo. Quando um host se conecta a outro, as regras para o compartilhamento de espaço de tupla são acionadas. Para isso, cada dispositivo mantém uma interface de espaço de tupla (*Interface Tuple-Space* - ITS) que contêm as tuplas que o dispositivo deseja compartilhar com os outros, representando o seu contexto acessível.

Quando um novo dispositivo entra em contato, o conteúdo de cada host da rede é dinamicamente atualizado com o conteúdo desse novo dispositivo, caracterizando a operação chamada de *engagement* (Figura 39). Nessa operação, as tuplas públicas de cada dispositivo são difundidas na rede, e cada dispositivo combina as tuplas dos outros dispositivos com as suas tuplas. É importante observar que esse modelo tende a sobrecarregar a rede, como pode ser imaginado numa rede com muitos dispositivos, ou com tuplas de tamanho grande. Além disso, numa rede em que dispositivos entram e saem da rede com muita freqüência, haverá um grande *overhead* nas tentativas de convergência de informações para combinar as tuplas desses dispositivos em movimento.

Além das informações trocadas entre aplicações que utilizam o middleware, existe um espaço de tupla transiente chamado *Lime System*, que contém detalhes sobre os dispositivos presentes no ambiente e os relacionamentos entre os mesmos,

fornecendo assim informações sobre a configuração do sistema. É possível também reagir a eventos para permitir que ações sejam tomadas em resposta à mudança na configuração do sistema, da mesma forma que, eventos podem ser usados para notificar as aplicações que novas tuplas estão disponíveis.

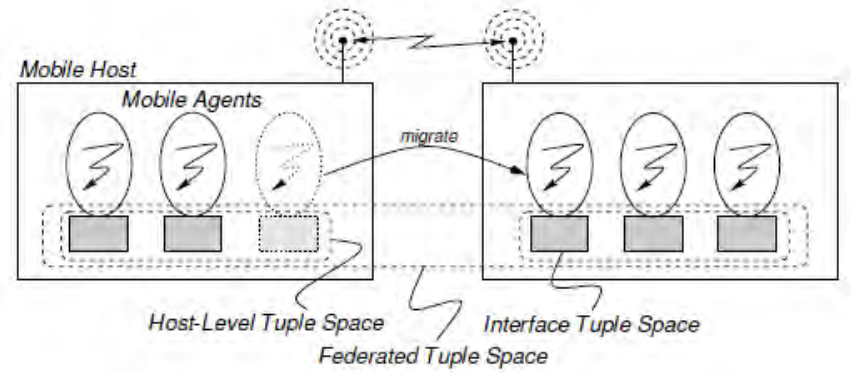

Figura 39 - Espaços de Tuplas compartilhados do LIME

Apesar de incluir esse suporte a eventos, caracterizando assim dois paradigmas de comunicação (espaço de tuplas e publish/subscribe), o LIME não dá suporte a extensão das funcionalidades, o que dificulta sua utilização em cenários de aplicação diferentes dos quais ele foi inicialmente projetado. O Multi-MOM, por sua vez, além de oferecer 5 paradigmas de comunicação, ainda permite facilmente acomodar novos comportamentos, reutilizando boa parte de seus componentes.

6.5 RUNES

RUNES [Costa05] é uma plataforma de middleware que oferece um modelo de componentes e uma API mínima para o desenvolvimento de aplicações móveis distribuídas. O RUNES divide áreas de funcionalidades típicas de sistemas de middleware em *frameworks* de componentes (CF), como por exemplo: Serviços de Localização, para lidar com funcionalidades que levam em conta a posição lógica e física do dispositivo; Descoberta e Anúncio de serviços, que permitem aplicações obterem acesso a componentes remotos, baseados em diversos protocolos, como UPnP, SLP e Bluetooth SDP; e Serviços de Interação, que permite modelar dinâmicas de interação entre aplicações.

É importante destacar que esses CF são independentes uns dos outros e podem ser implantados nos dispositivos individualmente. Entre eles, o CF de Serviços de Interação [Parlavantzas03] é o que mais se assemelha ao Multi-MOM. Este CF permite

projetar diversos modelos de interação entre aplicações, similar a idéia de paradigmas de comunicação, que no RUNES são chamados de *binding types*.

Para isso, é oferecida uma estrutura de alto nível, junto com uma API genérica, através dos quais podem ser derivados qualquer tipo de *binding type*, incluindo interações baseadas em mensagens, *streaming*, protocolos de leilão, invocação de métodos remotos, entre outros (Figura 40). Apesar do RUNES prover extensibilidade, permitindo a inclusão de praticamente qualquer tipo de interação (*binding types*), essa flexibilidade impede a definição de componentes comuns entre esses tipos de interação, o que faz aumentar o esforço necessário para implementar novos *binding types*.

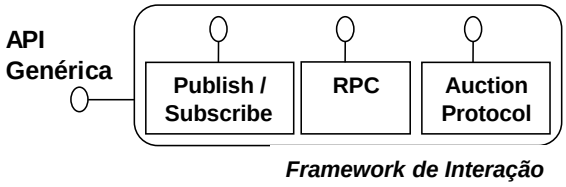

Figura 40 - Framework de Interação do RUNES

A única estrutura comum definida pelo RUNES para estes *binding types* é uma API genérica, que deve então ser estendida de acordo com cada tipo de interação. Diferentemente, o Multi-MOM baseia-se numa arquitetura integrada, a qual provê um conjunto de funcionalidades comuns e reusáveis entre os diversos paradigmas de comunicação, minimizando a quantidade de código necessária para prover soluções customizadas.

6.6 Quarterware

Singhai [Singhai98] propôs um dos trabalhos pioneiros quanto ao uso de soluções para o desenvolvimento de middleware customizável. O Quarterware define uma arquitetura de middleware formada por uma coleção de componentes genéricos visando a implementação de invocação de métodos remotos, serviço de diretório e grupos de objetos distribuídos. Estes componentes devem ser estendidos para derivar múltiplos padrões de middleware, como CORBA, Java RMI e *Message Passing Interface* (MPI).

A arquitetura do Quarterware possibilita a derivação de implementações de middleware por meio da customização de componentes individuais ou adicionando novos componentes, dependendo da funcionalidade requerida. Os componentes que devem ser customizados estão relacionados à: (i) empacotamento e desempacotamento

de dados (*marshaling / unmarshaling*); (ii) tipos de referências a serviços (ex.: endereço IP, endereço de memória local); (iii) transporte de dados (ex.: *sockets*, *pipes*, memória compartilhada); (iv) *dispatching* (pré-processar requisição, localizar e ativar servidor); (v) política de *threads* (ex.: uma *thread* por mensagem, *pool* de *threads*, *thread* única) e; (vi) protocolo de troca de dados (formato dos dados e seqüência de interações).

Assim como no Multi-MOM, as customizações são realizadas através de mecanismos de reflexão que permitem aos desenvolvedores a associação de versões customizadas desses componentes. A customização dos componentes supracitados permitem obter também otimizações baseadas em desempenho fim-afim. Essas otimizações estão relacionadas ao gerenciamento de memória (reduções de cópias, alocações dinâmicas e *caching*) e à comunicação (redução de latência, aumento da vazão e conservação de recursos).

O Quarterware foi projetado como uma arquitetura genérica e modular, a qual define uma estrutura geral unificada para desenvolvimento de vários tipos de middleware. Entretanto, essa estrutura deve ser complementada por desenvolvedores que desejem reusá-la. Esses desenvolvedores devem implementar componentes que provêem funcionalidades específicas, tais como componentes de gerenciamento de threads ou protocolo de transporte de dados, com intuito de oferecer soluções mais eficientes do que as soluções nativas existentes (ex: RMI, CORBA).

Por outro lado, o Multi-MOM oferece uma estrutura pronta para uso, onde paradigmas de comunicação podem ser selecionados e estendidos de acordo com os requisitos das aplicações e capacidades dos dispositivos. Além disso, o Multi-MOM permite que novos paradigmas de comunicação sejam acrescentados com um esforço relativamente pequeno, já que as principais dificuldades relativas à comunicação em redes móveis são tratadas pelos componentes base do Multi-MOM.

6.7 Proposta do Apel

Desenvolvido na Otto-von-Guericke-University Magdeburg, a proposta do Apel [Apel05] consiste em uma linha de produto de middleware ubíquo para suporte a comunicação entre dispositivos móveis. O objetivo da arquitetura baseada em linhas de produto é reduzir o espaço ocupado pelo middleware em memória, especializando o middleware em algum cenário de aplicação ou plataforma alvo.

Para dar suporte à variação de funcionalidades, esse trabalho utiliza uma abordagem baseada em componentes e *mixin layers*. *Mixin layers* permitem que um novo componente seja adicionado para prover uma funcionalidade que será reusada por vários componentes existentes. Por exemplo, supondo que existe um componente *Impressora* e um componente *Buffer*. Para adicionar controle de acesso a esses dois componentes, sem ter que alterar cada um deles separadamente, um novo componente do tipo *mixin* seria adicionado com a funcionalidade *Lockable*. Este novo componente então seria parametrizado, hora com o componente *Impressora*, hora com o componente *Buffer*. Assim para se ter uma impressora com controle de acesso, bastaria invocar o componente *Lockable* passando como parâmetro o componente *Impressora*. Da mesma forma aconteceria quando se desejasse um buffer com controle de acesso.

Nesse trabalho, funcionalidades de cliente e servidor são separadas com o intuito de colocar em cada dispositivo o mínimo de código necessário. Assim, gerenciar e registrar objetos remotos é função do servidor, enquanto enviar requisições é feito somente pelos clientes. Além dessas, há um módulo que provê funcionalidades básicas para comunicação entre clientes e servidores, e podem ser configuradas em termos de tipo de conexão (UDP ou TCP), direção da comunicação (unidirecional ou bidirecional) e estratégia de sincronização (síncrono ou assíncrono).

Como exemplo, são instanciadas três configurações diferentes de middleware, as quais são customizadas para requisitos de aplicações específicas: middleware para sensores-atuadores, middleware para invocação de objetos remotos e middleware para *Web Services*. Para esses três exemplos, além das componentes definidos na linha de produto, foram adicionadas bibliotecas específicas para prover funcionalidades específicas, como por exemplo, empacotamento e desempacotamento de requisições *web services*.

A contribuição desse trabalho consiste na definição de uma abordagem de configuração através de componentes e *mixin layers*, a qual pretende especializar o middleware para diferentes plataformas, bem como para diminuir a carga computacional. Por outro lado, nenhum mecanismo de tolerância a falhas é definido, os quais são extremamente necessários para comunicação entre dispositivos móveis utilizando redes sem fio. Em contrapartida, o Multi-MOM utiliza um modelo de comunicação baseado em mensagens, que é mais adequado para conexões intermitentes

do que invocações diretas, provendo funcionalidades como *buffer* de envio e persistência de mensagens.

6.8 Considerações finais

Os trabalhos relacionados foram apresentados e analisados neste capítulo. Primeiramente, foi descrito o JMS [Sun02], que é considerado um padrão 'de fato' para MOM e serve de base para vários outros sistemas desse tipo, entretanto ele não trata requisitos específicos para mobilidade. Em seguida foram abordados trabalhos que propõem adaptações de sistemas de mensagens para computação móvel. Por fim, foram apresentados trabalhos sobre sistemas de middleware customizáveis. A Tabela 5 ilustra uma breve comparação entre esses trabalhos.

Tabela 5 - Comparação entre trabalhos relacionados

<table>
<tr><th>Trabalho</th><th>Customizável / Extensível</th><th>Paradigmas de Comunicação</th><th>Funcionalidades relativas à mobilidade</th></tr>
<tr><td>JMS</td><td rowspan="3">Não</td><td>Publish/Subscribe e Fila de Mensagens</td><td>Nenhuma específica</td></tr>
<tr><td>JMS on MANET</td><td>Publish/Subscribe</td><td>Mapeamento de tópicos em endereços multicast</td></tr>
<tr><td>Pronto</td><td>Publish/Subscribe</td><td>Registro de serviços descentralizado e mapeamento de tópicos em endereços multicast</td></tr>
<tr><td>LIME</td><td>Espaço de Tuplas com suporte a eventos</td><td>Compartilhamento de tuplas de forma transparente</td></tr>
<tr><td>RUNES</td><td rowspan="3">Sim</td><td rowspan="3">Permite derivar qualquer paradigma</td><td>Define apenas arquitetura de alto nível e API genérica</td></tr>
<tr><td>Quarterware</td><td>Define arquitetura modular que deve ser estendida. O foco da customização está na melhora de desempenho.</td></tr>
<tr><td>Apel</td><td>O foco da customização é otimizar tamanho do middleware e adequar-se à diferentes plataformas.</td></tr>
<tr><td>Multi-MOM</td><td>Sim</td><td>5 paradigmas prontos para uso + suporte à extensão em alteração de código existente</td><td>Registro de serviços descentralizado, buffer de envio de mensagens, arquitetura integrada com reuso de componentes comuns.</td></tr>
</table>

A primeira coluna se refere ao nome do trabalho. A segunda coluna indica se o middleware em questão pode ser customizado / estendido, isto é, se é uma caixa preta que só pode ser modificado abrindo o código e modificando-o manualmente, ou se provê pontos de extensão que permite que outros desenvolvedores adicionem novos comportamentos sem ter que alterar o código existente. A terceira coluna indica quais

paradigmas de comunicação aquele middleware dá suporte. Alguns definem paradigmas de comunicação específicos, enquanto outros oferecem apenas a estrutura para que desenvolvedores implementem os paradigmas de comunicação de acordo com suas necessidades. Por fim, a quarta coluna destaca quais são as funcionalidades providas por esses sistemas relacionadas ao tratamento dos requisitos da computação móvel.

O JMS on MANETs [Vollset03], Pronto [Yoneki03] e LIME [Murphy06] apresentam plataformas que tratam requisitos específicos da comunicação na computação móvel. Esses trabalhos propõem abordagens para adaptar paradigmas de comunicação para lidar com as restrições impostas pela mobilidade. Todavia, cada um desses trabalhos trata de apenas um ou, no máximo dois paradigmas de comunicação. Segundo Costa [Costa07], middleware que oferece apenas um, ou um conjunto fixo de paradigmas de comunicação não é capaz de lidar com a diversidade de requisitos necessários para um amplo espectro de aplicações móveis.

Em relação a esses tipos de sistema, o Multi-MOM diferencia-se por oferecer cinco paradigmas de comunicação simultaneamente, bem como oferecer suporte a adição de novos paradigmas, que podem ser acrescentados para satisfazer necessidades específicas de novas aplicações. Vale ressaltar que essa extensão é feita sem alterar nenhum código existente, mas apenas acrescentando componentes em pontos de extensão bem definidos.

Já os sistemas de middleware customizáveis – RUNES [Costa05], Quarterware [Singhai98] e a proposta do Apel [Apel05] – permitem que diversos paradigmas de comunicação sejam derivados a partir de uma estrutura genérica. Entretanto, nenhum deles provê um conjunto comum e reusável de funcionalidades para tratar os requisitos da computação móvel. Assim, cada derivação de uma instância customizada dessas plataformas exige que tais tipos de funcionalidades sejam reimplementadas.

Além de aumentar o esforço para implementação de soluções customizadas, essa falta de integração tende a aumentar também o tamanho da plataforma, pois se uma instância da plataforma oferecer diversos paradigmas de comunicação distintos, possivelmente funcionalidades que tratam os requisitos de mobilidade serão repetidas em cada um dos paradigmas.

Segundo [Parlavantzas03], a definição de uma estrutura comum para uma plataforma customizável deve considerar os seguintes aspectos: (i) deve definir uma

estrutura padrão com funcionalidades comuns entre os possíveis sistemas deriváveis, de forma que essas funcionalidades não tenham que ser reimplementadas a cada customização; (ii) deve ser aberta o suficiente para possibilitar uma grande variedade de customizações. O Multi-MOM identifica funcionalidades comuns de sistemas de middleware para computação móvel e captura estas como componentes reusáveis. Assim, soluções customizadas podem ser derivadas sem ter que reimplementar tais funcionalidades, e além disso, como demonstrado pelo conjunto de paradigmas de comunicação oferecidos, a arquitetura definida é flexível o suficiente para derivar diversas variações de paradigmas de comunicação baseados em mensagens.

7 Considerações Finais

Neste trabalho, foi apresentado um middleware para computação móvel, denominado Multi-MOM, que tem por objetivo prover uma infra-estrutura para dar suporte a comunicação entre aplicações móveis distribuídas. Para isso, o middleware proposto define um modelo de comunicação baseado em mensagens, bem como um conjunto de mecanismos de tolerância a falhas e descentralização da comunicação, para que o modelo de comunicação baseado em mensagens possa adequar-se melhor ao contexto da mobilidade.

Vários sistemas de middleware têm sido propostos para tratar questões de comunicação na computação móvel, entretanto, uma das principais limitações das propostas atuais é que elas são projetadas para dar suporte a um paradigma de comunicação único e pré-definido (ex.: publish/subscribe). Essa restrição limita o escopo destas plataformas, no sentido de que elas não podem facilmente acomodar, ou serem estendidas para acomodar, a grande variedade de requisitos dos cenários móveis.

O Multi-MOM, por sua vez, oferece suporte a um conjunto abrangente e extensível de paradigmas de comunicação, tornando-o apto a atender uma ampla variedade de cenários de aplicação, sendo assim mais propenso a ser reusado do que uma solução baseada num paradigma específico. Os paradigmas de comunicação nativos oferecidos são: publish/subscribe, espaço de tuplas, fila de mensagens, notificações ponto-a-ponto e o modelo síncrono. É importante ressaltar que, apenas pela modificação de algumas propriedades, relacionadas às semânticas de entrega das mensagens, diversos outros paradigmas podem ser derivados. Entretanto, os paradigmas abordados neste trabalho são os mais debatidos na literatura e mais utilizados na computação móvel, como pode ser verificado em [Mascolo02] [Eugster03] [Aldred05].

O projeto arquitetural do Multi-MOM é baseado na abordagem de Linha de Produto de Software (LPS), a qual define um conjunto de sistemas com um núcleo comum e funcionalidades variáveis/opcionais que podem ser selecionadas para fazer

parte ou não de cada instância do middleware. Essa abordagem além permitir a customização do middleware, explora o reuso dos componentes necessários para prover os diversos paradigmas de comunicação, de forma a minimizar a utilização de recursos de memória necessários para execução do middleware.

Esses componentes reusáveis implementam funcionalidades comuns para lidar com as particularidades e restrições da comunicação no ambiente móvel, como funcionalidades para tratar problemas de transmissão (e retransmissão) de mensagens, descoberta de serviços e gerenciamento de mensagens. Assim, os componentes relativos aos paradigmas de comunicação reusam estas funcionalidades e incluem apenas o código relativo à semântica de entrega de mensagens de cada paradigma.

Além disso, apesar de definir um conjunto amplo de paradigmas de comunicação, é possível que algumas aplicações necessitem de algum paradigma de comunicação diferente dos oferecidos pelo Multi-MOM, ou simplesmente uma pequena variação de algum dos paradigmas definidos. Assim, por meio de pontos de extensão, é possível adicionar novos comportamentos à plataforma, isto é, novos paradigmas de comunicação. É importante destacar que essa extensão é feita pela adição de um conjunto de classes seguindo alguns padrões definidos, porém sem ter que alterar nenhum código existente.

Com a finalidade de ilustrar os benefícios da abstração multi-paradigma no ambiente móvel provida pelo Multi-MOM, um cenário de aplicação foi descrito, ressaltando como as funcionalidades do Multi-MOM são simples de serem usadas pelas aplicações clientes, e ao mesmo tempo suficientes para tratar dos principais requisitos impostos pela comunicação em ambientes de mobilidade.

Por fim, foi feita uma avaliação do middleware proposto. Através de métricas de reuso de código foi mostrado que as funcionalidades comuns do middleware são efetivamente reusáveis. A avaliação mostrou também que o middleware proposto adéqua-se aos dispositivos aos quais ele foi projetado. Além disso, o desempenho do middleware foi mensurado e, mesmo oferecendo múltiplos paradigmas, seu desempenho ficou compatível com outros trabalhos especializados em um único paradigma de comunicação.

7.1 Contribuições

A principal contribuição do middleware proposto é a integração simultânea de diversos paradigmas de comunicação para computação móvel que, em geral, são identificados e encontrados nas propostas existentes de forma isolada. Essa abordagem permite atender aos requisitos de vários tipos de aplicações, e não somente de um conjunto restrito de aplicações, como é comum nas soluções baseadas num único paradigma.

Essa abordagem é importante para que diferentes aplicações possam utilizar uma mesma plataforma, facilitando a padronização das mesmas, e evitando que cada aplicação utilize uma plataforma de middleware distinta, o que causaria um maior consumo de recursos computacionais. Além disso, a abordagem multi-paradigma permite que uma mesma aplicação utilize simultaneamente diversas formas de interação, o que facilita ainda mais o desenvolvimento deste tipo de aplicação.

Além disso, outra importante contribuição dessa proposta é definição de um conjunto conciso de funcionalidades comuns entre sistemas de mensagens e o provimento destas como componentes reusáveis entre os paradigmas. Embora algumas funcionalidades já sejam utilizadas em sistemas de mensagens tradicionais, o Multi-MOM propõe adaptações das mesmas para que o modelo de mensagens adéqüe-se melhor aos cenários da mobilidade, pois, como afirma Capra [Capra02], implementações tradicionais de MOMs não são adequadas para tratar as freqüentes desconexões e as restrições de recursos observadas nos sistemas móveis.

Adicionalmente, esse conjunto de funcionalidades comuns, combinado com os pontos de extensão definidos, provêem uma infra-estrutura base para o desenvolvimento de novos sistemas de middleware, tornando a complexidade e o esforço de construção menor, através do reuso e da padronização. Com isso é possível construir middleware para comunicação entre aplicações móveis voltando-se basicamente para a definição das diferentes semânticas de entrega de mensagens específicas de cada paradigma.

7.2 Limitações eTrabalhos Futuros

A implementação de sistemas de middleware para computação móvel é uma tarefa complexa que envolve desde interesses comerciais, até conceitos de otimização, tolerância a falhas, e customização. Portanto, prover uma plataforma completa,

integrada e extensível é um grande desafio. Apesar de considerar todos esses requisitos, o Multi-MOM apresenta algumas limitações.

A seleção de funcionalidades do Multi-MOM é feita em tempo de compilação. Embora não seja necessário alterar código, esta seleção é feita através da adição/remoção de classes em pontos bem definidos. Assim, ao invés de realizar essa tarefa manualmente, uma sugestão de trabalho futuro seria a implementação de um sistema com interface gráfica para selecionar e gerar automaticamente instâncias do middleware, onde o desenvolvedor de aplicações apenas indicaria qual dos paradigmas ele deseja.

Além disso, considerando a dinamicidade dos cenários da mobilidade, seria interessante o provimento de recursos de adaptação dinâmica, isto é, permitir que as funcionalidades sejam adicionadas/removidas em tempo de execução, como pode ser visto em [Gilani04]. Assim, a adição de um paradigma para suprir a necessidade de uma nova aplicação não forçaria o middleware a ter que ser reinstalado manualmente.

Outra limitação do Multi-MOM é que os pontos de extensão permitem apenas a customização dos paradigmas de comunicação, isto é, permite que sejam customizadas apenas as semânticas de entrega das mensagens. Além desse tipo de funcionalidade, é interessante que se acrescente posteriormente a possibilidade de customizar outras funcionalidades, como por exemplo a tecnologia de rede que o middleware vai operar, ou o protocolo de transporte utilizado.

Atualmente o Multi-MOM só dá suporte às redes *Wi-Fi.* A definição de pontos de extensão para essa característica permitiria acrescentar suporte a redes Bluetooth, por exemplo. Já a customização do protocolo de transporte permitira substituir o protocolo TCP pelo UDP para troca de mensagens, desde que fosse acrescentado também uma forma de controle de erros para esse último.

De forma a evidenciar a viabilidade da abordagem multi-paradigma, o foco deste trabalho concentrou-se na definição de uma arquitetura reusável e num projeto extensível. Assim, devido à limitação de escopo, funcionalidades mais avançadas e muito específicas de cada paradigma de comunicação foram deixadas para serem implementadas, como extensões, por futuros desenvolvedores que desejem reutilizar o Multi-MOM.

Como exemplo de funcionalidades mais avançadas, o paradigma publish/subscribe ao invés de entregar as mensagens em *unicast* para cada assinante, poderia mapear cada tópico criado em endereços *multicast*, como pode ser visto em alguns trabalhos relacionados [Vollset03] [Yoneki03]. Já no paradigma de fila de mensagens poderia ser adicionado o ordenamento baseado em prioridades. Enquanto que no espaço de tuplas poderia ser implementado um mecanismo de bloqueio para certas tuplas, de forma que elas pudessem ser definidas como somente leitura, como em [Murphy06].

Estes são alguns exemplos de funcionalidades que podem ser encontradas em sistemas específicos de cada paradigma. Apesar de não oferecê-las no protótipo implementado, o Multi-MOM foi projetado para acomodar facilmente estas funcionalidades e também outras que possam vir a ser necessárias para os paradigmas de comunicação orientados a mensagens.

Aspectos de segurança também não foram considerados no projeto do Multi-MOM. Esse aspecto deve ser considerado tanto para descoberta e registro de serviços, como para troca de mensagens entre aplicações. Para isso, podem ser utilizados o campo “propriedades” das classes que encapsulam referências a serviços e das classes que encapsulam mensagens para carregar pares de chave pública e privada, garantindo autenticidade e confidencialidade na comunicação.

Por fim, no cenário de aplicação, não foi realizada a implementação da aplicação descrita. Esta foi apenas modelada e projetada, com o intuito de mostrar como a mesma se beneficiaria do uso do middleware proposto. Para validar o middleware num cenário real, além de implementar essa mesma aplicação, poderia ser feita uma pesquisa por aplicações existentes com requisitos similares, e reimplementá-las utilizando o Multi-MOM, comparando o esforço necessário para implementar uma mesma aplicação com e sem o middleware.

8 Referências

[Aldred05] Aldred, L., Aalst, W., Dumas, M. and Hofstede, A. *On the Notion of Coupling in Communication Middleware*. Lecture Notes in Computer Science. Vol. 3761. pp 1015-133. Springer, 2005.

[Android10a] Android Developer's Guide, http://developer.android.com. Acesso em: 29/06/2010.

[Android10b] Android Market, http://www.android.com/market/. Acesso em: 29/06/2010.

[Apple10] Apple iPhone App Store, http://www.apple.com/iphone/features/app-store.html. Acesso em: 29/06/2010.

[Apel05] Apel, S. e Bohm, K. *Towards the Development of Ubiquitous Middleware Product Lines*. Lecture Notes in Computer Science. Vol. 3427. pp 137-153. Springer, 2005.

[Auriemma10] Auriemma, L. Simple TCP proxy/datatype, http://aluigi.altervista.org/mytoolz.htm. Acesso em: 29/06/2010.

[Babar04] Babar, M. A., Zhu, L. and Jeffery, R. *A framework for classifying and comparing software architecture evaluation methods*. In: Proceedings of the Australian Software Engineering Conference, pp 309-318, Melbourne, Australia, Abril, 2004.

[Batista09] Batista, V. e Rosa, N. *Spontaneousware: A Middleware Framework for Mobile Ad Hoc Networks*. In: International Workshop on Middleware for Pervasive Mobile and Embedded Computing, 2009.

[Bellavista07] Bellavista, P. e Corradi, A. *The Handbook of Mobile Middleware.* Auerbach Publications, 2007.

[Capra03] Capra, L. Emmerich, W. Mascolo, C. *CARISMA: context-aware reflective middleware system for mobile applications*. In: IEEE Transactions on Software Engineering, Vol. 29, Issue: 10 pp. 929- 945, 2003.

[Clements02] Clements, P. and Northrop, L., *Software Product Lines: Practices and Patterns.* SEI Series, Addison-Wesley, 2002.

[Costa05] Costa, P., Coulson, G., Mascolo, C., Picco, GP., and Zachariadis, S. *The RUNES Middleware: A Reconfigurable Component-based Approach to Networked*

Embedded Systems. Proceedings of IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications, 2005.

[Costa07] Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G., Sivaharan, T., Weerasinghe, N. and Zachariadis, S. *The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario***.** In: 5th Annual IEEE International Conference on Pervasive Computing and Communications, 2007.

[Cugola02] Cugola, G., Jacobsen, H. *Using Publish/Subscribe Middleware for Mobile Systems*. In: ACM SIGMOBILE Mobile Computing and Communications Review. Vol. 6, No. 4, pp. 25-33, 2002.

[Curry04] Curry, E., Chambers, D., Lyons, G. *Extending Message-Oriented Middleware using Interception*. In: 3rd International Workshop on Distributed Event-based Systems. Edinburgh, Scotland, UK, 2004.

[Eugster03] Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A. (2003) *The Many Faces Of Publish/Subscribe***.** ACM Computing Surveys, Vol. 35, No. 2, June 2003, pp. 114–131.

[Emmerich00] Emmerich, W. *Software engineering and middleware: A roadmap*. Communications of the ACM, pp. 117-129, 2000.

[Frakes05] Frakes, W., Terry, C.: *Software Reuse and Reusability Metrics and Models*. Technical Report. Virginia Polytechnic Institute & State University. 2005.

[Geihs01] Geihs, K. *Middleware Challenges Ahead*. IEEE Computer, Vol. 34, Issue 6, pp. 24-31, 2001.

[Gelernter85] Gelernter, D. *Generative Communication in Linda*. In: ACM Transactions on Programming Languages and Systems. 1985.

[Gilani04] Gilani, W., Naqvi, N. and Spinczyk, O. *On Adaptable Middleware Product Lines.* Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware, pp. 207-213, 2004.

[Gomaa04] Gomaa, H. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, 2004.

[GSMArena10] GSM Arena, http://www.gsmarena.com. Acesso em: 29/06/2010.

[Hadim06] Hadim, S., Al-Jaroodi, J., Mohamed, N. *Middleware Issues and Approaches for Mobile Ad hoc Networks.* In: Proceedings of the 3rd IEEE Consumer Communications and Networking Conference Proceedings, Vol 1, pp. 431-436, Rio de Janeiro, 2006.

[Jung99] Jung, D., Paek, K., Kim, T. *Design of Mobile MOM: Message Oriented Middleware Service for Mobile Computing*. In: Proceedings of the International Workshop on Parallel Processing. 1999.

[Kaul06] Kaul, D., Gokhale, A. *Middleware specialization using aspect oriented programming.* Proceedings of the 44th ACM Southeast regional conference. pp. 319-324, Melbourne, USA 2006.

[Kuhns99] Kuhns, F., O'Ryan, C., Schmidt, D., Othman, O., Parsons, J. *The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware*. In: Proceedings of the IFIP 6th International Workshop on Protocols For High-Speed Networks. pp. 1-8, 1999.

[Linthicum01] Linthicum, D. *B2B Application Integration: e-Business-Enable Your Enterprise*, Addison-Wesley, 2001.

[Mascolo02] Mascolo, C., Capra, L., Emmerich, W. *Middleware for Mobile Computing***.** Tutorial. Proceedings of the International Conference on Networking. 2002.

[Massol03] Massol, V. and Husted, T. *JUnit in Action*. Greenwich, Manning Publications, 2003.

[Morais10a] Morais, Y., Burity, T., Elias, G. *Towards a Mobile Middleware Product Line: A Requirement Analysis.* Proceedings of the 7th International Conference on Information Technology: New Generations (ITNG), Las Vegas, EUA, 2010.

[Morais10b] Morais, Y.; Elias, G. *Integrating Communication Paradigms in a Mobile Middleware Product Line*. Proceedings of the 9th International Conference on Networks (ICN), Menuires, França, 2010.

[Morais10c] Morais, Y., Elias, G. *An Extensible, Multi-Paradigm Message-oriented Mobile Middleware*. In: Proceedings of the 5th International Conference on Software and Data Technologies. Atenas, Grécia, 2010.

[Morais10d] Morais, Y., Elias, G. *Customizing Message-oriented Mobile Middleware*. In: Proceedings of the 6th International Conference on Wireless and Mobile Communications (ICWMC). Valencia, Espanha, 2010.

[Morais09] Morais, Y.; Burity, T.; Elias, G. *A Systematic Review of Software Product Lines Applied to Mobile Middleware*. Proceedings of the 6th International Conference on Information Technology - New Generations (ITNG), Las Vegas, EUA, 2009.

[Muro05] Muro, E. Ingham, D., Simon, A. *JMS Clustering in Arjuna Message Service: Dependability, Performance and Manageability*. IADIS International Conference on Applied Computing. Algarve, Portugal, 2005.

[Murphy01] Murphy, A., Picco, G., Roman, G. *LIME: A Middleware for Physical and Logical Mobility*. In: 21st IEEE International Conference on Distributed Computing Systems, 2001.

[Murphy06] Murphy, A., Picco, G., Roman, G. *LIME: A coordination model and middleware supporting mobility of hosts and agents.* ACM Transactions on Software Engineering and Methodology (TOSEM). Vol. 15, 3. pp. 279—328. Julho, 2006

[Paller06] Paller, G. *Building Reflective Mobile Middleware Framework on Top of the OSGi Platform.* Reuse of Off-the-Shelf Components, pp. 354--367. Springer Berlin / Heidelberg, 2006.

[Parlavantzas03] Parlavantzas, N., Coulson, G., Blair, G. *An Extensible Binding Framework for Component-Based Middleware*. In: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing. Brisbane, Australia, 2003.

[Pressman05] Pressman, R. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 5th edition, 2005.

[Rellermeyer07] Rellermeyer, J., Alonso, G., Roscoe T. *R-OSGi: Distributed Applications through Software Modularization*. Lecture Notes in Computer Science. Vol. 4834. pp 1-20. Springer, 2007.

[Rellermeyer10] Rellermeyer, J.S.: jSLP project, Java Service Location Protocol. Disponível em: http://jslp.sourceforge.net. Acesso em: 29/06/2010.

[Reuters10] Thomson Reuters, http://www.reuters.com/. Acesso em: 29/06/2010.

[Research2Guidance10] Research2Guidance. *Global Smartphone Market Application Report 2010*, http://www.research2guidance.com/. Acesso em: 29/06/2010.

[Schantz01] Schantz, R., Schmidt, D. *Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications*. Encyclopedia of Software Engineering, Citeseer, 2001.

[Sommerville07] Sommerville, I. *Engenharia de Software*. 8 edição. Addison Wesley. 2007.

[SQLite10] SQLite. http://sqlite.org/. Acesso em: 29/06/2010.

[Sun10] Sun Microsystems. *Java Message Service (JMS) Specification*. Version 1.1, 2002.

[Tanenbaum02] Tanenbaum, A. S. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[Travassos02] Travassos, G. H., Gurov, D. e Amaral, E. A. G. (2002). "Introdução a Engenharia de Software Experimental". Relatório Técnico – ES – 590/02. Programa de Engenharia de Sistemas e Computação – COPPE/UFRJ. Rio de Janeiro – RJ.

[Wells06] Wells, C. *A Tuple Space Web Service for Distributed Programming*. In: International Conference on Parallel and Distributed Processing Techniques and Applications. 2006.

[Wuest05] Wuest, B. *Framework for middleware executed on mobile devices.* Dissertação de Mestrado. Informatik der Universität Kassel, 2005.

[Yoneki03] Yoneki, E. *Mobile Applications with a Middleware System in Publish-Subscribe Paradigm*. In: 3rd Workshop on Applications and Services in Wireless Networks. 2003.

[Zhang03] Zhang, C.; Jacobsen, H. *Quantifying aspects in middleware platforms*. In: Proceedings of the 2nd International Conference on Aspect-oriented Software Development. pp 130-139, 2003.