Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

Dissertação de Mestrado

Uma Abordagem para Offloading em Múltiplas Plataformas Móveis

Philipp Bernardino Costa

Fortaleza – Ceará
2014

Philipp Bernardino Costa

Uma Abordagem para Offloading em Múltiplas Plataformas Móveis

Dissertação de Mestrado submetida à Coordenação do Programa de Pós-graduação em Ciência da Computação (MDCC) da Universidade Federal do Ceará (UFC) como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. José Neuman de Souza, PhD.

Co-Orientador: Prof. Fernando Antonio Mota Trinta, DSc.

Fortaleza – Ceará
2014

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Ciências e Tecnologia

C875a Costa, Philipp Bernardino.
Uma abordagem para *offloading* em múltiplas plataformas móveis / Philipp Bernardino Costa. – 2014.
104 f. : il. color., enc. ; 30 cm.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Departamento de Computação, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2014.
Área de Concentração: Sistema de Informação.
Orientação: Prof. Dr. José Neuman de Souza.
Coorientação: Prof. Dr. Fernando Antonio Mota Trinta.

1. Computação em nuvem. 2. Dispositivos móveis. 3. Computação móvel. I. Título.

CDD 005

Uma Abordagem para Offloading em Múltiplas Plataformas Móveis

Dissertação de Mestrado submetida à Coordenação do Programa de Pós-graduação em Ciência da Computação (MDCC) da Universidade Federal do Ceará (UFC) como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Aprovado em ___/___/_____

Banca Examinadora

_____

Prof. José Neuman de Souza, PhD. (Orientador)
Universidade Federal do Ceará – UFC

_____

Prof. Fernando Antonio Mota Trinta, DSc. (Coorientador)
Universidade Federal do Ceará – UFC

_____

Prof. Danielo Gonçalves Gomes, DSc.
Universidade Federal do Ceará – UFC

_____

Prof. Carlos André Guimarães Ferraz, DSc.
Universidade Federal do Pernambuco – UFPE

Eu dedico essa dissertação primeiramente aos meus pais, Ruth e Facundo, aos meus professores e amigos, que me apoiaram em tudo que foi necessário para trilhar esta jornada.

Agradecimentos

Agradeço a **Deus**, pelas oportunidades que surgiram e poderão surgir, pelas pessoas que me foram apresentadas e que ainda serão, por ter me dado perseverança e *insight* para solucionar todos os problemas que surgiram, ao longo desta caminhada, além de ter tido sorte de nascer em uma família maravilhosa.

Aos meus pais **Ruth** e **Facundo**, por sempre me oferecer um apoio incondicional em todos os sentidos, no que for necessário, para concluir esta e os próximos trabalhos que possam surgir.

Aos meus orientadores, Professor **Trinta** e Professor **Neuman**, que me orientaram ao longo desta árdua jornada. O Professor **Trinta** dedicou horas adicionais fora do seu horário de trabalho, para corrigir meus materiais, além de fornecer o capital intelectual necessário (artigos, *softwares*, etc.) para realizar esta dissertação de mestrado.

Aos professores **Danielo** e **Carlos**, que compõem a banca examinadora e certamente contribuirão para o sucesso da solução desenvolvida.

Também sou bastante agradecido ao Professor **Paulo**, pelas longas conversas sobre o tema que escolhi, pelas correções e dicas das coisas que escrevia, além do artigo que publicamos juntos na trilha principal da SBRC 2014.

Agradeço ao **GREat** pelo ambiente de pesquisa e trabalho, além de poder interagir com pessoas maravilhosas, que me ensinaram bastante no âmbito profissional. Também sou grato aos gerentes do projeto **Bruno** e **Perote**, que sempre flexibilizavam os meus horários, para realizar as minhas diversas atividades de mestrado.

Por fim, a todos que de alguma forma me apoiaram para a conclusão desse trabalho.

"Eu nunca fiz nada por acaso, nem nenhuma das minhas invenções surgiu por acaso; elas vieram pelo fruto do trabalho."

Thomas A. Edison

Resumo

Os dispositivos móveis, especificamente os *smartphones* e os *tablets*, evoluíram bastante em termos computacionais nos últimos anos, e estão cada vez mais presentes no cotidiano das pessoas. Apesar dos avanços tecnológicos, a principal limitação desses dispositivos está relacionada com a questão energética e com seu baixo desempenho computacional, quando comparado com um *notebook* ou computador de mesa. Com base nesse contexto, surgiu o paradigma do *Mobile Cloud Computing* (MCC), o qual estuda formas de estender os recursos computacionais e energéticos dos dispositivos móveis através da utilização das técnicas de *offloading*. A partir do levantamento bibliográfico dos *frameworks* em MCC verificou-se, para o problema da heterogeneidade em plataformas móveis, ausência de soluções de *offloading*. Diante deste problema, esta dissertação apresenta um *framework* denominado de MpOS (*Multi-platform Offloading System*), que suporta a técnica de *offloading*, em relação ao desenvolvimento de aplicações para diferentes plataformas móveis, sendo desenvolvido inicialmente para as plataformas Android e Windows Phone. Para validação foram desenvolvidas para cada plataforma móvel, duas aplicações móveis, denominadas de *BenchImage* e *Collision*, que demonstram o funcionamento da técnica de *offloading* em diversos cenários. No caso do experimento realizado com *BenchImage* foi analisado o desempenho da aplicação móvel, em relação à execução local, no *cloudlet server* e em uma nuvem pública na Internet, enquanto no experimento do *Collision* (um aplicativo de tempo real) foi analisado o desempenho do *offloading*, utilizando também diferentes sistemas de serialização de dados. Em ambos os experimentos houve situações que era mais vantajoso executar localmente no *smartphone*, do que realizar a operação de *offloading* e vice-versa, por causa de diversos fatores associados com a qualidade da rede e com volume de processamento exigido nesta operação.

Palavras-chave: Heterogeneidade de Plataformas Móveis, Mobile Cloud Computing e Offloading.

Abstract

The mobile devices, like smartphones and tablets, have evolved considerably in last years in computational terms. Despite advances in their hardware, these devices have energy constraints regarded to their poor computing performance. Therefore, on this context, a new paradigm called Mobile Cloud Computing (MCC) has emerged. MCC studies new ways to extend the computational and energy resources, on mobile devices using the offloading techniques. A literature survey about MCC, has shown that there is no support heterogeneity on reported studies. In response, we propose a framework called MpOS (Multi-platform Offloading System), which supports the offloading technique in mobile application development, for two mobile platforms (Android and Windows Phone). Two case studies were developed with MpOS solution in order to evaluate the framework for each mobile platform. These case studies show how the offloading technique works on several perspectives. In BenchImage experiment, the offloading performance was analyzed, concerning to its execution on a remote execution site (a cloudlet on local network and public cloud in the Internet). The Collision application promotes the analysis of the offloading technique performance on real-time application, also using different serialization systems. In both experiments, results show some situations where it was better to run locally on smarphone, than performing the offloading operation and vice versa.

Keywords: Mobile Platform Heterogeneity, Mobile Cloud Computing and Offloading.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

Sigla	Significado
API	Application Programming Interface
BSON	Binary JavaScript Object Notation
BTS	Base Transceiver Station
CPU	Central Processing Unit
CSV	Comma-seperated values
DHCP	Dynamic Host Configuration Protocol
DLL	Dynamic Link Libary
E/S	Entra e Saída
FPS	Frame Per Second
GPS	Global Positioning System
GPU	Graphics Processing Unit
IP	Internet Protocol
JIT	Just-in-time
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LTE	Long-Term Evolution
MCC	Mobile Cloud Computing
MP	Mega Pixels
MpOS	Multi-platform Offloading System
MV	Máquina Virtual
P2P	Peer-to-Peer
ORM	Object-Relational Mapping
RPC	Remote Procedure Call
RTT	Round-To-Trip
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
UML	Unified Modeling Language
VnC	Virtual Network Computing
WP	Windows Phone
XML	Extensible Markup Language

Capítulo 1

Introdução

Os dispositivos móveis como *smartphones* e *tablets* são utensílios cada vez mais importantes no cotidiano das pessoas. Com isso, a rápida adoção destes dispositivos levou a popularidade das aplicações e serviços móveis em diferentes aspectos da vida moderna, permitindo também que sejam acessados a qualquer momento, independente da hora e do lugar. Segundo os pesquisadores Kenney e Pon (2011), o desenvolvimento de aplicações móveis tem aumentado rapidamente por causa de dois fatores: (i) melhoria nas plataformas móveis (e.g. Android, Windows Phone e iOS) e no *hardware* dos dispositivos móveis e (ii) possibilidade de "monetizar" aplicações e serviços por meio de canais de vendas já consolidados, como Apple Store, Google Play Store e Windows Store.

Apesar dos avanços tecnológicos, os dispositivos móveis ainda são limitados em termos computacionais quando comparados com um computador de mesa ou um *notebook* devido a fatores relacionados com suas dimensões físicas e restrições térmicas. Nos últimos anos, a questão energética tem se destacado como a maior limitação dos dispositivos móveis, dada a evolução mais rápida das tecnologias de *hardware* (processadores, memórias e sensores em geral), em comparação com a evolução tecnológica empregada nas baterias [Satyanarayanan *et al.* 2009, Fernando *et al.* 2013].

Uma solução para superar estas limitações de recursos é através do paradigma *Mobile Cloud Computing* (MCC), que surgiu nesse contexto e envolve uma combinação de tecnologias em diversas áreas, sendo as principais: computação em nuvem, computação móvel e redes sem fio. O MCC aproveita os recursos e os serviços da computação em nuvem, além da sua crescente adoção, por parte dos usuários móveis tornando o MCC uma área de pesquisa bastante promissora [Shiraz *et al.* 2013].

1.1 Motivação

O paradigma do *Mobile Cloud Computing* foca nos benefícios que podem ser alcançados pelos dispositivos móveis, quando se delega uma operação de armazenamento ou processamento de dados, para um ambiente de execução remoto com maior capacidade computacional. Assim, este tipo de operação pode proporcionar tanto economia de energia, quanto um aumento no desempenho computacional do dispositivo móvel, sendo esta operação denominada inicialmente

na literatura por *cyber foraging* [Satyanarayanan *et al.* 2001], ou mais recentemente chamada de técnica de *offloading* [Cuervo *et al.* 2010]. Segundo Verbelen *et al.* (2012), o *offloading* pode ser executado em um ambiente de execução remoto baseado em máquinas virtuais na nuvem pública ou em qualquer máquina que esteja em uma mesma rede local no qual estão os dispositivos móveis. É importante destacar que *offloading* é diferente do modelo cliente-servidor tradicional, segundo no qual o cliente sempre solicita um conteúdo ou função do servidor. Esta técnica também é diferente do modelo de migração usado em grades computacionais, onde um processo pode ser migrado para fins de balanceamento de carga [Kumar *et al.* 2013]. Porém, utilizar o *offloading* não é sinônimo de aumento garantido de desempenho do dispositivo móvel.

O desenvolvimento de aplicações móveis que adotam o modelo do MCC enfrenta inúmeros desafios e problemas herdados das diferentes áreas que compõem este paradigma, como por exemplo, segurança, mobilidade, restrições energéticas e computacionais dos dispositivos móveis, além da questão da heterogeneidade em MCC. Esta questão é bastante abrangente, podendo ser limitada apenas ao *offloading* em diferentes plataformas móveis. Segundo alguns pesquisadores, como Sanaei *et al.* (2013), a heterogeneidade tem sido pouco explorada na literatura pelos principais trabalhos da área, porque os desenvolvedores de *frameworks* estavam focados no desenvolvimento de sistemas capazes de realizar a técnica de *offloading*, além de tomar a decisão de quando realizar esta operação. Por isto, muitos dos trabalhos existentes na área mostram protótipos e experimentos realizados apenas para uma determinada plataforma móvel e ambiente de execução remoto.

A motivação para desenvolver um *framework* em MCC com esta natureza de multiplaforma está na própria dinâmica da indústria de aplicativos móveis. Os grandes serviços de Internet como: Facebook, Twitter, Skype, Waze e dentre outros, possuem extensão de suas atividades no ambiente de aplicações móveis. Todos estes serviços suportam a questão da multiplataforma, visando maximizar a quantidade de usuários que podem interagir com seus sistemas e consequentemente aumentar a visibilidade dos seus anúncios, gerando assim, receita para suas empresas.

No mercado existem exemplos de *framework* que suportam múltiplas plataformas, como por exemplo, *framework* de mapeamento objeto-relacional (ORM) Hibernate[1]. Esta ferramenta possui duas versões, as quais podem ser utilizadas na linguagem Java e C#[2]. A vantagem desse tipo de *framework* é o reuso do conhecimento já adquirido dos desenvolvedores para trabalhar em outro projeto, baseado na linguagem C# e que utilize o mesmo *framework* de ORM. Com isso, o

[1] http://hibernate.org/
[2] http://nhforge.org/

tempo de treinamento da equipe para utilizar este *framework* em outra linguagem seria menor, trazendo assim, produtividade para equipe de desenvolvimento e quando necessário, flexibilizar a portabilidade de um projeto de uma tecnologia para outra.

Entretanto, para alguns pesquisadores como Sanaei *et al.* (2013) seria relevante à existência de um *framework* em MCC que suportasse a operação de *offloading* em diferentes plataformas móveis, principalmente quando se considera o seguinte cenário: Uma empresa precisa desenvolver um aplicativo móvel multiplataforma que precise também suportar a técnica de *offloading*; Atualmente, para resolver esta problemática, essa empresa terá de utilizar pelos menos duas soluções distintas de *framework* em MCC, para resolver seu problema. Portanto, é desejável a existência de um *framework* que trate dessa questão do *offloading* em múltiplas plataformas móveis, a fim de padronizar o desenvolvimento, a implantação e a interação dos componentes do aplicativo móvel com uma infraestrutura computacional.

1.2 Objetivos e Contribuições

Com base nas motivações apresentadas, esta dissertação de mestrado tem o objetivo de aprofundar a respeito do tema de *Mobile Cloud Computing* pretendendo contribuir para área, através do desenvolvimento de um *framework* que realiza a técnica de *offloading* em diferentes plataformas móveis. Este *framework* é denominado de MpOS (*Multi-platform Offloading System*), sendo voltado para os desenvolvedores de aplicativos móveis que desejam adotar a técnica do *offloading* computacional, na granularidade[3] de método no aplicativo móvel. Isso significa que um desenvolvedor durante o desenvolvimento de sua aplicação marcará um método candidato para realizar a técnica de *offloading* de forma oportuna para um determinado ambiente de execução remoto. O *framework* MpOS foi desenvolvido para as plataformas móveis Android e Windows Phone.

O *framework* MpOS também possui diversas funcionalidades que apoiam a técnica de *offloading*, tanto na parte cliente, quanto na parte servidora do processo. Dentre estas funcionalidades pode-se destacar o sistema de descoberta de serviço, que localiza dinamicamente uma máquina servidora e outros serviços de rede que são utilizados pelo *framework*. O MpOS possui uma funcionalidade que realiza automaticamente o processo de implantação do serviço de *offloading*, quando este não estiver em funcionamento no servidor remoto. Por fim, o *framework* possui um sistema de *Profiler* de Rede, que avalia as condições da rede e apoia a decisão do MpOS de realizar ou não uma operação de *offloading*.

[3] Existem diversos níveis de *offloading*, sendo explicado no próximo capítulo deste trabalho.

Este trabalho possui também diversas contribuições relacionadas com a questão de como os dados são enviadas para o servidor remoto, mostrando uma forma alternativa de serialização que pode acelerar o desempenho da operação de *offloading*. O sistema de *Remote Procedure Call* (RPC) neste trabalho foi desenvolvido especificamente para suportar a operação de *offloading*, em cima de métodos marcados pelos desenvolvedores, no tempo de desenvolvimento do aplicativo móvel. Finalmente, na plataforma Windows Phone, diversos componentes foram construídos, para que o MpOS tivesse funcionalidades equivalentes ao que foi desenvolvido na plataforma Android.

1.3 Metodologia

A metodologia científica utilizada neste trabalho é caracterizada resumidamente nos seguintes itens:

1. **Revisão de Literatura:** Inicialmente, foi realizada uma revisão bibliográfica, usando o indexador *Scorpus* com a seguinte *string* de busca *ALL (("mobile cloud computing" OR "mobile computing") AND "doudlet")*, filtrando também todos os artigos, a partir do ano de 2009 em diante. Nesta busca foram coletados 123 artigos, dentre estes quatros *surveys* [Dihn *et al.* 2011, Shiraz *et al.* 2013, Fernando *et al.* 2013 e Sanaei *et al.* 2013], que auxiliaram a filtrar e selecionar desta coleta, os principais artigos da área de *Mobile Cloud Computing*;
2. **Seleção dos Trabalhos Relacionados:** Após a revisão inicial do tema *Mobile Cloud Computing* foram escolhidos trabalhos que desenvolveram algum *framework* para realizar a operação de *offloading* em um aplicativo móvel, e que pudessem ser comparados com o *framework* MpOS em diversas perspectivas;
3. **Definição do *framework* MpOS:** Com base na revisão da literatura e na seleção dos trabalhos relacionados, foi projetado um *framework* para realizar a operação de *offloading* em múltiplas plataformas móveis. Durante a leitura dos trabalhos relacionados, foram coletados aqueles requisitos que são comuns a todos os *frameworks*. No decorrer desse processo também foram definidos requisitos inexistentes nestes trabalhos relacionados, mas que seriam desejáveis para compor a solução do MpOS;
4. **Avaliação dos Resultados:** Após a definição do *framework* MpOS, foram desenvolvidas duas aplicações como prova de conceito: uma que realiza um processamento pesado e outra de tempo-real. Nesta dissertação é demonstrada como foi definido os componentes que realizam a operação de *offloading*, em cada aplicativo

móvel e foram conduzidos diversos experimentos, avaliando separadamente os resultados obtidos, para cada plataforma móvel.

1.4 Organização da dissertação

Esta dissertação está organizada em seis capítulos. O capítulo atual descreveu uma breve introdução ao tema, contextualizando e motivando os assuntos abordados nesta dissertação. Além disso, foram definidos os objetos e as contribuições, como também, a metodologia utilizada para desenvolver este trabalho.

No Capítulo 2 é apresentada a fundamentação teórica acerca do tema *Mobile Cloud Computing*. Entre os conceitos apresentados estão a definição do tema, as arquiteturas que podem ser assumidas e os desafios da área. No final são apresentados os trabalhos relacionados, além de um resumo comparativo entre estes trabalhos e como eles influenciaram no desenvolvimento do *framework* MpOS.

O Capítulo 3 apresenta a proposta do MpOS através da metodologia da execução do projeto do MpOS, da elicitação dos requisitos funcionais e não funcionais, da definição da arquitetura, além da visão geral de como funciona o processo de configuração e execução do MpOS.

No Capítulo 4 é apresentada com detalhes como foram especificados os elementos propostos, a saber, os componentes arquiteturais de cliente denominado de MpOS API e componente de servidor chamado de MpOS Plataforma.

O Capítulo 5 apresenta os dois aplicativos móveis desenvolvidos, usados para avaliar e validar o uso do *framework* MpOS. Também neste capítulo é apresentada a definição das funcionalidades e a arquitetura desses aplicativos móveis, além da discussão sobre os resultados obtidos durante sua execução dos experimentos.

Finalmente, o Capítulo 6 descreve de forma resumida os resultados alcançados, bem como as conclusões deste trabalho, apresentando também as possíveis melhorias a serem consideradas nos trabalhos futuros.

Capítulo 2
Mobile Cloud Computing

O *Mobile Cloud Computing* surgiu para contornar as limitações dos dispositivos móveis em relação ao desempenho e consumo de energia. Esse capítulo será apresentado a fundamentação teórica do tema MCC, que engloba a conceituação do tema, as arquiteturas e os desafios da área, como também será apresentada os trabalhos relacionados, comparando suas características com o *framework* MpOS.

2.1 Introdução

Atualmente não existe um consenso sobre o conceito do paradigma *Mobile Cloud Computing*, o que existe é a opinião de diversos pesquisadores em relação ao assunto, sendo destacado três destas opiniões.

Segundo Dinh *et al.* (2011), o conceito do MCC surgiu com objetivo de fornecer uma variedade de serviços equivalentes aos da nuvem, adaptados à capacidade dos dispositivos com recursos limitados de computação, além da melhoria das infraestruturas de telecomunicações, a fim de aperfeiçoar o provisionamento de serviços.

De acordo com Sanaei *et al.* (2013), MCC estende a tecnologia de computação móvel tirando proveito dos recursos elásticos da computação em nuvem e das tecnologias de rede, podendo assim servir os dispositivos móveis em qualquer lugar e a qualquer hora, através da Internet.

Para Shiraz *et al.* (2013), o MCC é o mais recente paradigma computacional prático, que estende a visão da computação utilitária de computação em nuvem, para aumentar os limites tanto dos recursos computacionais, quanto dos recursos energéticos dos *smartphones*, através da utilização de alguma técnica de *offloading* que atua no nível do aplicativo móvel. Nesta dissertação de mestrado é considerada esta definição para conceituar os propósitos para os quais o *framework* MpOS se propõe.

2.2 Arquiteturas de Mobile Cloud Computing

As arquiteturas em *Mobile Cloud Computing* podem ter diversas perspectivas, que serão definidas a seguir. Para alguns autores, como Dinh *et al.* (2011), Qi e Gani (2012), e Bahl *et al.* (2012), o paradigma do MCC pode ser simplesmente dividido em componentes de computação em nuvem e computação móvel, conforme mostrado pela Figura 2.1.

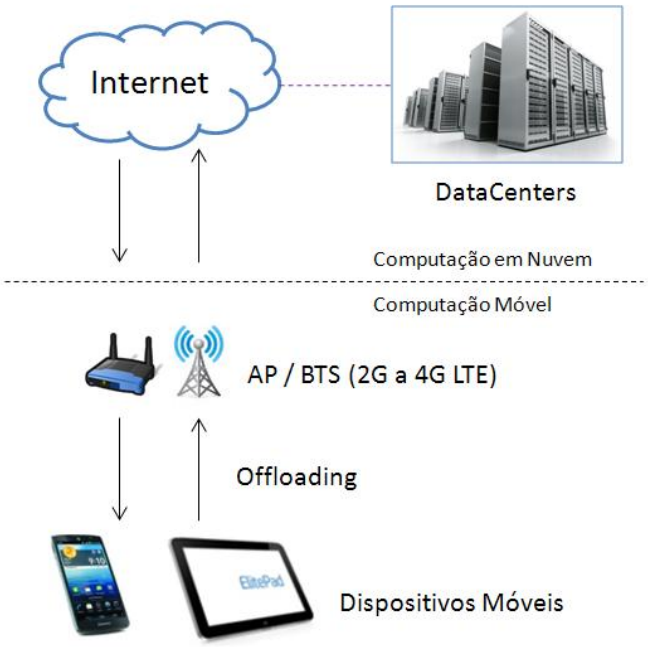

Figura 2.1. Arquitetura Geral do *Mobile Cloud Computing* [Qi e Gani 2012]

A Figura 2.1 apresenta a interação dos componentes que compõem arquitetura geral do MCC. A dinâmica desta arquitetura funciona da seguinte forma: os usuários dos dispositivos móveis interagem com os serviços de *offloading*, que estão disponíveis em uma nuvem pública, e são conectados por meio de uma conexão Wi-Fi (*Access Point*), ou via estação rádio base (BTS) que utiliza algumas das tecnologias de Internet móvel, como 2G até 4G LTE. Em MCC, existem também outros modelos arquiteturais que tratam de eventos e problemas específicos, e que não são abordados nesta arquitetura geral.

O modelo arquitetural *cloudlet* (ver Figura 2.2) foi introduzido por Satyanarayanan *et al.* (2009) e tem como principal objetivo aproximar os serviços de *offloading* para serem executados em máquinas locais, ao invés de utilizar uma nuvem pública para este fim. Estas máquinas locais podem variar deste de *notebooks* até servidores, que estão disponíveis em uma mesma rede local Wi-Fi. O tipo de máquina que compõe um *cloudlet* deve ser proporcional à sua utilidade. Por exemplo, para uso pessoal, uma máquina do porte de um *notebook* ou computador de mesa, talvez seja suficiente para servir de *cloudlet*. No entanto, em grandes ambientes, como em *shopping centers*

ou aeroportos, é necessário que o *cloudlet* seja composto por uma robusta infraestrutura computacional de servidores, que suportem proporcionalmente as demandas dos usuários.

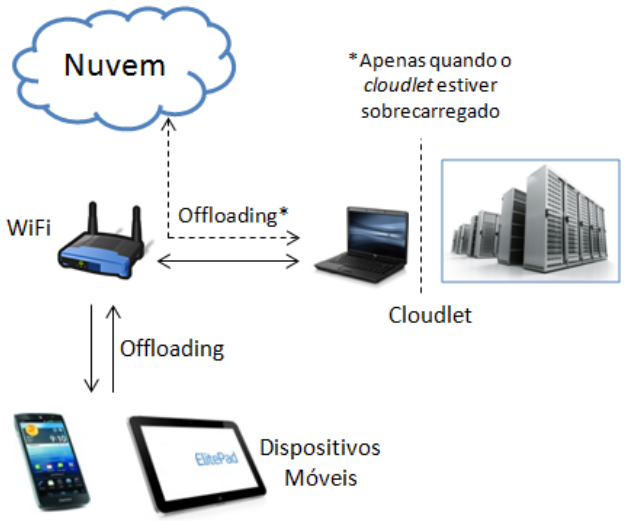

Figura 2.2. Modelo Arquitetural *Cloudlet* [Fernando *et al.* 2013]

A ideia é utilizar as redes Wi-Fi, uma vez que estas, em geral, possuem velocidades maiores e latência menores, por serem menos congestionadas do que as redes de celulares. Como consequência, um *cloudlet* consegue entregar um serviço de melhor qualidade do que o proposto pela arquitetura geral do MCC, que utiliza a infraestrutura de Internet para interagir com os serviços de *offloading*. De acordo com Satyanarayanan *et al.* (2009), este modelo arquitetural (ver Figura 2.2), em caso de sobrecarga também suportaria estender os recursos computacionais de um *cloudlet* para uma nuvem pública na Internet a fim de continuar atendendo as demandas dos usuários locais. Contudo, por questões de delimitação do escopo, será desconsiderada nesta dissertação a ideia de estender os recursos físicos do *cloudlet* para uma nuvem pública, sendo assim, denominado de *cloudlet server*, a modificação desse modelo arquitetural.

Outros autores também destacam as vantagens do modelo arquitetural *cloudlet*. Bahl *et al.* (2012) propuseram que a utilização do conceito de *cloudlet* é mais evidente em uma determinada categoria de aplicações que são sensíveis à percepção de latência, como aplicativos de processamento de imagem, visão computacional, reconhecimento de voz e jogos. Em um estudo realizado no Brasil pelos pesquisadores da UFC [Costa *et al.* 2014], foi analisado o impacto da qualidade da Internet móvel 4G na utilização de *cloudlets*. Este trabalho realizou diversos experimentos, dentre eles, uma experiência sobre o desempenho do aplicativo móvel, quando se executa uma operação de *offloading* em uma nuvem pública utilizando a Internet móvel 4G, em comparação com a mesma operação quando realizada a partir de um *doudlet*. Os resultados obtidos mostraram uma vantagem para a utilização do *doudlet* em relação à Internet móvel mesmo sendo de quarta geração (4G).

Por fim, existe outra proposta de arquitetura em MCC (ver Figura 2.3), que utiliza estruturas de redes ponto a ponto (P2P) para prover serviços de *offloading* baseada no conceito de computação colaborativa. Os usuários cedem parte dos recursos computacionais de seus dispositivos móveis, para resolver uma tarefa em comum a todos os outros usuários [Fernando *et al.* 2013].

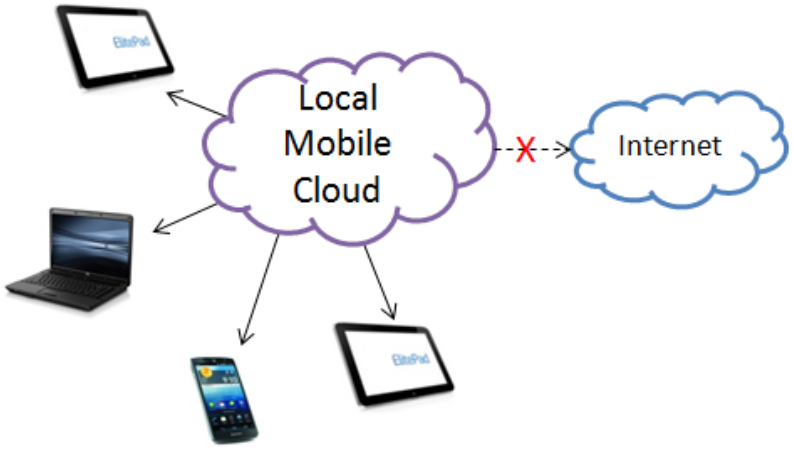

Figura 2.3. Modelo Arquitetural baseado em P2P [Fernando *et al.* 2013]

Neste modelo arquitetural apenas os dispositivos que fazem parte da rede P2P, poderão enviar ou receber tarefas para serem executadas. Logo, este tipo de arquitetura não precisa de acesso à Internet, podendo ser bastante útil em ocasiões de desastres naturais ou regiões sem nenhuma cobertura tecnológica, como a floresta Amazônica. Porém, em situação normal, as pessoas querem maximizar o desempenho de suas aplicações, além de aumentar a autonomia da sua bateria, gerando assim, conflitos de interesse para adoção desse modelo arquitetural [Fernando *et al.* 2013].

2.3 Desafios

A definição desse paradigma mostra que o MCC engloba diversas áreas, concebendo assim, diversos desafios técnicos que são herdados por cada uma dessas áreas, conforme listado a seguir:

- **Restrições energéticas:** Qi e Gani (2012), e Sanaei *et al.* (2013) afirmam que as tecnologias das baterias incrementam por volta de 5% ao ano do seu tempo de vida e que não haverá no curto prazo, um grande salto tecnológico para melhoria deste cenário. Sendo assim, faz-se necessário recorrer às técnicas de *offloading* para conservar os recursos energéticos dos dispositivos móveis;

- **Restrições computacionais:** Estas restrições acontecem por causa da mobilidade dos aparelhos não sendo relacionada por limitações computacionais da tecnologia móvel atual. Estes dispositivos são construídos levando em conta primeiro suas particularidades físicas como, peso leve, tamanho compacto, *design* confortável ergonomicamente, dentre outros aspectos. Assim, depois de estabelecidas suas restrições físicas, as características relacionadas com a parte computacional dos aparelhos móveis são secundárias, apesar da indústria reconhecer o "apetite" dos usuários por aplicações computacionalmente intensivas [Ha *et al.* 2013];

- **Qualidade das redes sem fio:** Tal qualidade é influenciada por uma série fatores, como largura de banda disponível, latência entre os pontos de conexão e taxa de perda de pacotes, que variam ao longo do dia, de acordo com a densidade dos usuários que utilizam os serviços de rede sem fio. De acordo com Sanaei *et al.* (2013) e Bahl *et al.* (2012), uma baixa qualidade no enlace sem fio pode diminuir o desempenho do *offloadin*g consequentemente da aplicação móvel, por causa do efeito da retransmissão dos dados, desencadeando também, um aumento no tempo de transferência das informações envolvidas na sessão. Assim, grande parte desses problemas de rede sem fio é associado com as redes de celulares congestionadas, porém da mesma forma as redes Wi-Fi padecem dos mesmos males, quando existe uma grande concentração de pessoas utilizando-as;

- **Consistência da Plataforma Distribuída:** As tarefas e os pedaços de código de uma aplicação que foram selecionados para realizar a operação de *offloading* devem manter a consistência, isto é, não devem ser corrompidos durante todo o processo de execução remota. Por consequência, desenvolver este procedimento "leve" e transparente é um desafio, que traz todas as questões relacionadas aos sistemas distribuídos, como transparência de localização, de falha, de concorrência e dentre outras questões [Shiraz *et al.* 2012];

- **Mobilidade:** Este é considerado, o mais importante atributo dos *smartphones*, além de ser classificado como uma questão séria de pesquisa em MCC. Segundo, Shiraz *et al.* (2012), a necessidade de prover de forma transparente, imperceptível e ininterrupto, o acesso aos ambientes remotos, que podem estar nas nuvens ou na mobilidade entre *cloudlets*, ainda é um desafio em aberto;

- **Segurança e Privacidade:** Este é um assunto transversal em diversos domínios da ciência da computação, não podendo ser diferente com o paradigma do MCC. Com a popularização dos *smartphones*, os *cybers* criminosos passaram ameaçar estes aparelhos utilizando as mesmas técnicas dos computadores de mesa, através da utilização de vírus e explorando diversas vulnerabilidades[4] das plataformas móveis. De acordo com Dinh *et al.* (2011), o MCC deve apoiar a criação de aplicativos de segurança, como Antivírus, que permitem utilizar o poder computacional de um ambiente externo, para verificar os arquivos suspeitos de forma mais rápida, do que executar essa verificação localmente no dispositivo móvel.

- **Heterogeneidade:** Sanaei *et al.* (2013) em seus estudos afirmou que existem três tipos de heterogeneidade em MCC: (i) relacionada com a plataforma móvel, (ii) com ambiente de execução remoto, e (iii) com as redes sem fio. Estes tipos também podem ser agrupados em duas categorias: (i) denominada de heterogeneidade vertical, que ocorre sobre um único tipo de um mesmo aspecto, por exemplo, uma plataforma móvel com suas diferentes versões; e (ii) heterogeneidade horizontal, quando a diferenciação ocorre sobre diferentes tipos de um mesmo aspecto, por exemplo, comparação entre plataformas móveis diferentes.

Durante o processo de revisão de literatura foram elencados estes principais desafios do paradigma MCC. Também não foi encontrada nenhuma solução que contemplasse todos estes aspectos e que também esteja disponível para o público em geral testar e comprovar, como foram solucionados estes desafios.

O *framework* MpOS tratou principalmente dos desafios relacionados com as restrições computacionais do dispositivo móvel, sendo também o objetivo da operação de *offloading* trazer ganho de desempenho na execução de uma aplicação móvel. O *framework* também prover a consistência da plataforma distribuída, através da transparência de localização [Kurose and Ross, 2012], pois para o aplicativo, não tem diferença entre executar localmente no dispositivo móvel e remotamente em um servidor remoto. Em caso de qualquer erro no ambiente de execução remoto, o MpOS é executado localmente no dispositivo móvel.

Por fim, a dissertação também abordou o desafio da heterogeneidade horizontal, em relação a todos os aspetos citados. Pois, o *framework* MpOS foi desenvolvido para duas plataformas móveis, além disso, executou alguns experimentos em dois ambientes de execução remotos (*cloudlet* e nuvem pública), para dois tipos de rede sem fio (Wi-Fi e 4G).

[4] http://blog.trendmicro.com/trendlabs-security-intelligence/exploiting-vulnerabilities-the-other-side-of-mobile-threats/

2.4 Trabalhos Relacionados

A área de *Mobile Cloud Computing* é composta pela fusão de diversas outras áreas, tendo como principal objeto de estudo as técnicas de *offloading*. Nesta seção serão apresentados alguns conceitos específicos da técnica de *offloading*, em conjunto com os trabalhos realizados pela comunidade científica e que estão relacionados com a proposta do *framework* MpOS.

2.4.1 *Offloading* Baseado na Migração de Máquina Virtual

São métodos de *offloading* que consistem na transferência de estados de uma máquina virtual de um dispositivo móvel para outro, sem precisar alterar o código fonte das aplicações móveis.

2.4.1.1 *Framework* Kimberley

Satyanarayanan *et al.* (2009) através do *framework* Kimberley, propuseram um *offloading* baseado na migração de máquina virtual (MV). Durante, o experimento do *framework* foi utilizado o dispositivo móvel Nokia N810, que usava o sistema operacional Maemo (uma variante do Linux) e possibilitava instalar aplicativos desse dispositivo, dentro de uma máquina virtual Linux.

Portanto, caso fosse necessário migrar algum aplicativo móvel, uma MV precisava ser preparada antes no *cloudlet* e a versão alterada dessa MV seria copiada para o dispositivo móvel. Caso fosse necessário realizar o procedimento de *offloading* baseado na migração de MV, seria migrada apenas a diferença da *MV base* com a *MV alterada* para o *cloudlet*, em um processo denominado de "síntese dinâmica de máquina virtual". Depois de realizada a migração, o dispositivo móvel interage com o aplicativo remoto, através de um cliente de VnC.

Esta proposta tem a vantagem de não precisar alterar o código fonte das aplicações, além de apresentar pela primeira vez o conceito da arquitetura *doudlet*, que aproxima o ambiente de execução do *offloading* dos usuários móveis. Entretanto, esta abordagem é limitada para tratar da questão da heterogeneidade em plataformas móveis, pois empresas como Microsoft e Apple, no primeiro momento, não tem interesse de fornecer imagens de suas respectivas plataformas móveis (Windows Phone e iOS), para executar aplicações remotas dentro de um servidor. Existem também outros complicadores que são relatados pelos próprios autores, como o tempo de implantação das MVs, que demoram no mínimo 30 segundos em uma rede de 100Mbps para estarem prontas para utilização.

2.4.1.2 *Framework* CloneCloud

Os pesquisadores Chun *et al.* (2011) apresentaram um trabalho que modifica a máquina virtual da plataforma Android (Darvik VM) para suportar a técnica de *offloading* na granularidade de *threads*. Assim, qualquer *smartphone* Android que adotar essa máquina virtual modificada, a aplicação automaticamente passará a suportar a técnica de *offloading*, sem a necessidade de alterar o código fonte dos aplicativos desenvolvidos.

Entretanto, o CloneCloud realiza um processo de particionamento *offline*, ou seja, particiona inicialmente uma aplicação (no momento da implantação da MV modificada), através de um sistema de *solver*, que decide automaticamente, quais *threads* serão realizadas as operações de *offloading*. Este sistema escolhe as partições, a partir do componente de *profiling*[5], que avalia o tempo de execução do aplicativo móvel e a energia gasta pelo dispositivo. O *solver* guarda sua decisão de *offloading* em um arquivo de configuração, para ser usada pela MV modificada, durante a execução do aplicativo móvel. O *solver* também verifica se as *threads* possui alguma restrição, por exemplo, estado compartilhado entre duas ou mais *threads*, invocação de procedimento de E/S ou de *hardware* (GPS, GPU, acelerômetro, dentre outros), não poderão ser candidatas para realizar a operação de *offloading*.

Portanto, durante a execução do aplicativo móvel, uma *thread* que é selecionada para *offloading*, o CloneCloud migra antes o estado "atual" da MV do cliente, para depois migrar a *thread* e continuar a sua execução, no servidor remoto. Na hora de retornar a *thread* "processada", o mesmo procedimento é feito, no sentido servidor para o cliente móvel.

Apesar das vantagens, o *framework* CloneCloud é limitado a uma única plataforma móvel, não tratando do desafio da heterogeneidade entre plataformas móveis. Para cada versão nova do Android, os desenvolvedores do CloneCloud precisam também lançar uma nova modificação da máquina virtual, podendo ser oneroso a adaptação dessa solução para cada nova versão da plataforma Android. Por fim, o particionamento da aplicação é *offline* exigindo um passo adicional para o CloneCloud analisar um aplicativo móvel e gerar o "plano de *offloading*" das partes que serão migradas para o servidor remoto.

[5] É um processo no qual um sistema analisa uma demanda sobre diversas perspectivas, por exemplo, *profiling* de aplicação é avaliado o uso da CPU, o consumo de memória e etc.

2.4.2 *Offloading* Baseado no Particionamento do Aplicativo

O particionamento de uma aplicação móvel é abordagem mais comum de *offloading*, podendo ser tanto da forma estática, como da forma dinâmica.

2.4.2.1 Particionamento Estático

O particionamento estático de uma aplicação é definido geralmente no tempo de desenvolvimento de um aplicativo móvel. Os desenvolvedores de forma manual ou automática selecionam os segmentos de código, para realizar a operação de *offloading*, independente de qualquer condição, deste que tenha conectividade entre o dispositivo móvel e o ambiente de execução remoto [Shiraz *et al.* 2012]. O particionamento *offline* citado no trabalho do CloneCloud [Chun *et al.* 2011] pode ser também considerado particionamento estático do aplicativo móvel, por selecionar antes da execução as partes que serão realizadas as operações de *offloading*.

2.4.2.1.1 Proposta do projeto Spectra

O projeto Spectra [Flinn, Narayanan e Satyanarayanan 2001] é um componente desenvolvido para habilitar a execução remota do *framework* Aura. Este projeto pode ser considerado o primeiro trabalho que realiza atividade de particionamento (no caso estático), baseado no monitoramento dos recursos (CPU, rede, bateria e estado da *cache*) em aplicações móveis.

Neste trabalho, os programadores precisam criar diversos planos de execução, para cada aplicação móvel. Essa atividade envolve cadastrar todos os métodos do aplicativo, dando um peso de referência para cada método, além de determinar a localidade da execução, como local ou remota. No Spectra, o motor de decisão escolhe o plano de execução, que maximiza a função utilidade, além de apoiar a decisão, baseado em um determinado momento do monitor de recursos.

Mesmo este *framework* sendo considerado o pioneiro da área de particionamento, o mesmo tem como principal desvantagem moldar a construção da aplicação empregando um modelo específico de execução remota que é empregado pelo Spectra, sendo similar ao sistema de RPC. O *framework* também exige que o programador desenvolva diversos planos de execução, sendo equivalente ao processo de fazer um *profiling* manualmente do aplicativo móvel, para melhor apoiar a escolha do motor de decisão do *framework* Spectra.

2.4.2.1.2 *Framework* Hyrax

O *framework* Hyrax [Marinelli 2009] implementou uma versão do *framework* Hadoop[6] fundamentado no modelo de programação MapReduce [Dean e Ghemawat 2004] para dispositivos móveis. O Hyrax trabalha com o conceito de particionamento estático, através das funções *map* e *reduce*.

Segundo os autores [Dean e Ghemawat 2004], a função *map* é aplicada para um conjunto de dados de entrada e produz uma lista com diversos pares (chave e valor), para uma saída intermediária. Estes pares são agrupados em partições e enviados para a função *reduce*, que produz a partir destas partições o resultado final do processamento. Assim, os desenvolvedores precisam apenas se preocupar em implementar essas funções *map* e *reduce*, enquanto todo sistema abstrai a complexidade envolvida na execução dessas funções.

O *framework* Hyrax possui como desvantagem, exigir que os aplicativos móveis sejam desenvolvidos baseado no paradigma MapReduce. Assim, esta restrição limita os tipos de aplicativos móveis que podem usufruir desse *framework*, além das diversas dificuldades reportadas pelo autor em relação à utilização desse paradigma, em um ambiente de dispositivos móveis.

2.4.2.2 Particionamento Dinâmico

Neste tipo de particionamento o *framework* decide dinamicamente em tempo de execução, sobre a localidade (local ou remota) da execução de um segmento de código marcado para realizar a operação de *offloading* em um aplicativo móvel. Essa decisão pode ser baseada em diversos fatores como, custo energético para execução local ou remoto do processamento, qualidade da rede sem fio e complexidade computacional da partição selecionada para realizar o procedimento de *offloading*.

2.4.2.2.1 *Framework* AlfredO

Os pesquisadores Giurgiu *et al.* (2009) desenvolveram um *framework* denominado de AlfredO, que foi construído em cima do *framework* OSGi[7], no qual o aplicativo móvel passa a ser desenvolvido orientado à módulos (*bundle* do OSGi) e estes módulos são distribuídos entre os dispositivos móveis e os servidores remotos. Portanto, AlfredO é responsável por decidir sobre quais *bundles* serão executados localmente ou remotamente, nesse ambiente distribuído.

[6] http://hadoop.apache.org/
[7] http://www.osgi.org/

O *framework* produz um "grafo do consumo dos recursos", através de um *profiling* sobre todos os *bundles* que compõe o aplicativo móvel. Esse *profiling* produz diversas informações relativas com a quantidade de memória consumida, tráfego de rede gerado e tamanho do código do *bundle*. O próximo passo do AlfredO é pré-processar o grafo, separando os *bundles* locais e remotos, diminuindo assim, o espaço de busca dos algoritmos de otimização.

Este *framework* aplica as duas formas de particionamento do aplicativo: (i) particionamento estático através da aplicação do algoritmo ALL (corte global ótimo), sendo apenas utilizado para otimização *offline*, ou seja, o sistema prepara as partições do aplicativo, para os diversos tipos de dispositivos móveis; (ii) particionamento dinâmico utiliza o algoritmo K-Step (busca local ótima), onde as partições devem ser calculadas em tempo de execução, uma vez que os dispositivos móveis também interagem com outros recursos.

Os próprios autores reconhecem que utilizar o *framework* OSGi para desenvolvimento de aplicativos móveis é bastante "pesado", além de modificar a forma de como estes aplicativos são construídos, por causa do desenvolvimento orientado à *bundles* exigido pelo OSGi. A estratégia de particionamento utilizada é também bastante complexa, por causa principalmente das atividades de *profiling*, decisão, migração e implantação de componentes. Segundo Shiraz *et al.* (2012), o *framework* AlfredO precisa estar continuamente sincronizado com a nuvem, mantendo sempre o *smartphone* no estado ativo, consumindo evidentemente mais energia do dispositivo móvel.

2.4.2.2.2 *Framework* MAUI

O *framework* MAUI [Cuervo *et al.* 2010] foi desenvolvido para plataforma do Windows Mobile 6.5 e suporta o particionamento dinâmico com o mínimo de alteração no código fonte, através da "anotação" dos métodos que são passiveis de *offloading*. O MAUI tem como principal meta, utilizar a técnica de *offloading* para reduzir o consumo de energia durante a execução dos aplicativos móveis.

Segundo os autores Cuervo *et al.* (2010), o sistema operacional do Windows Mobile suporta a portabilidade do código, por causa da plataforma de execução .NET *Framework*, que compila o código desenvolvido para uma linguagem intermediária, possibilitando assim, executar esse código, em diferentes arquiteturas de processadores. O suporte a programação reflexiva é fundamental para o funcionamento do *framework* MAUI, na identificação dos métodos remotos e extração dos estados necessários para realizar a operação de *offloading*.

O desenvolvedor que estiver utilizando o MAUI deve ter alguns cuidados para anotar os métodos para execução remota, pois se houver neste métodos algum componente de UI,

interação com E/S do dispositivo móvel, como abertura de arquivos, leitura de dados do GPS ou relacionado com componentes externos de rede, estes métodos não deverão ser anotados. Se por algum motivo, o programador anote algum método restrito, vai ser disparado um erro e a execução do aplicativo móvel retorna a ser local de forma transparente.

O procedimento de *profiling* (custo de CPU) feito pelo componente do MAUI *Profile* é executado para cada método anotado. Este componente também utiliza informação do processo de serialização para determinar os custos da rede, que são envolvidos na transferência de um estado do aplicativo móvel para os servidores remotos. O MAUI *Profile* ainda mensura a qualidade da rede sem fio, através dos valores de latência e largura de banda. O MAUI *Solver* tem como valores de entrada o MAUI *Profile* e resolve um problema de programação linear, para decidir sobre a viabilidade ou não, de migrar um método anotado para um servidor remoto.

O *framework* MAUI possui como limitação suportar uma única plataforma móvel, o Windows Mobile, não tratando, portanto do problema da heterogeneidade do processo de *offloading*. O trabalho MAUI também usa um sistema de serialização de dados baseado em XML, sendo bastante pesado, principalmente se for necessário mover uma grande quantidade de dados durante a sessão de *offloading*. No entanto, o processo de serialização do MAUI utiliza um sistema de serialização, que transfere apenas a diferença entre uma operação de *offloading* e outra, podendo ainda ser ineficiente caso tenha muita mudança entre os XML transferidos.

2.4.2.2.3 *Framework* Scavenger

O Scavenger [Kristensen e Bouvin 2010] é um *framework* que utiliza também chamadas RPC como método de *offloading*, sendo um método similar ao empregado pelo *framework* MAUI. O foco desse trabalho é realizar a operação de *offloading* visando ganho no desempenho do aplicativo móvel.

Este trabalho tentou também tratar da questão da heterogeneidade de plataformas móveis, utilizando a linguagem Python, que executa teoricamente em qualquer plataforma móvel do mercado. Como o *framework* foi desenvolvido baseado nessa linguagem, os aplicativos móveis também precisam ser desenvolvidos utilizando Python. O ambiente de execução remoto utilizado foi o Stackless Python[8] que tem a capacidade de oferecer uma instalação dinâmica e executar de código móvel em Python.

O Scavenger utiliza o conceito de *decorator* da linguagem Python para o desenvolvedor marcar os métodos que são passiveis de *offloading*, sendo que a principal restrição dessa marcação

[8] http://www.stackless.com/

é a necessidade do método ser autocontido, ou seja, não pode chamar outros métodos. O *framework* utiliza dois *profiling*, onde o primeiro procura estimar o tamanho da tarefa, condições da rede e a velocidade do dispositivo atual, e outro *profiling* busca estimar apenas a capacidade de processamento das máquinas que podem receber esse processamento. No processo de decisão também são considerados dados históricos do *profiling*. Diferente das abordagens anteriores, a decisão é feito em cima de uma formula simples e uma tabela de referência da execução média em cada plataforma usada.

O problema principal dessa abordagem é que nenhuma das três maiores plataformas móveis do mercado (Android, iOS e Windows Phone), utiliza nativamente a linguagem Python para o desenvolvimento de aplicativos móveis. Outro empecilho da abordagem é que os próprios autores não conseguiram validar o *framework* utilizando duas plataformas móveis diferentes.

2.4.2.2.4 *Framework* ThinkAir

O *framework* ThinkAir [Kosta *et al.* 2012] promete trazer um mecanismo para suportar alocação de recurso sobre demanda no ambiente de execução remoto (*cloudlet* ou nuvem pública), baseado na carga de trabalho de uma operação de *offloading*, além de explorar o paralelismo no ambiente de execução remota. Os quatro principais objetivos desse trabalho são: (i) facilidade no uso, através da anotação de métodos que podem fazer a operação de *offloading* (sem restrição); (ii) trazer transparência da localidade da execução dos métodos que são passíveis de *offloading*, (iii) o objetivo do *offloading* é aumentar o desempenho e reduzir o consumo de energia e (iv) escalar dinamicamente o poder computacional do servidor, de acordo com a exigência da operação de *offloading*.

No ThinkAir os desenvolvedores podem também escolher uma das quatros políticas para decidir, quando realizar ou não, a operação de *offloading*. A primeira política baseada no histórico do tempo de execução. A segunda baseada no histórico do consumo de energia do aparelho. A terceira é uma combinação das outras duas políticas. Enquanto a quarta engloba a terceira política e o custo da utilização da nuvem pública. As informações dessas políticas são alimentadas a partir de três *profiles* relacionados com o dispositivo, o aplicativo e a rede. Segundo os autores o modelo para estimar o consumo da energia foi o mesmo utilizado pelo trabalho PowerTutor [Zhang *et al.* 2010], sendo também utilizado o mesmo dispositivo móvel (HTC Dream) nos testes de avaliação do artigo.

O artigo afirma não tratar da questão da segurança dos dados e de autenticação dos usuários que podem utilizar os serviços remotos na nuvem. O trabalho precisa modificar

levemente o código fonte, do aplicativo para poder suportar a operação do *offloading*, além de não suportar a questão da heterogeneidade da plataforma móvel. No caso da paralelização de um *offloading* em múltiplos clones (baseado em MV do Android), os autores afirmam ter componentes que permitam o desenvolvedor criar algoritmos em paralelos sem modificar o código do ThinkAir. No entanto, os autores não explicam e nem demonstram com exemplos, como seria essa implementação apenas afirmam, que o ThinkAir divide as tarefas, através do particionamento das entradas, entre as diversas máquinas virtuais secundárias.

2.4.3 Comparação entre os Trabalhos Relacionados

Todos os trabalhos relacionados influenciaram de forma direta ou indireta, no planejamento dos componentes e na construção da solução do *framework* MpOS. O *framework* Kimberley [Satyanarayanan *et al.* 2009] utilizou a migração de máquina virtual, em conjunto com conceito de *cloudlets* e motivou outros trabalhos, como MAUI [Cuervo *et al.* 2010], CloneCloud [Chun *et al.* 2011] e ThinkAir [Kosta *et al.* 2012], a realizar testes de desempenho do *offloading*, usando diferentes tipos de redes (3G e Wi-Fi dedicado) para acessar o servidor remoto. O *framework* MpOS utiliza este mesmo tipo de cenário de rede e ambiente de execução remoto (nuvem pública e *cloudlet*) nos seus experimentos, utilizando também uma Internet móvel 4G, que se difere destes trabalhos que utilizaram apenas o 3G.

O projeto Spectra [Flinn, Narayanan e Satyanarayanan 2001] pode ser considerado um dos pioneiros para realizar a execução remota usando a questão do particionamento estático e adotando um modelo (estilo RPC) para adequar o aplicativo móvel para realizar a execução remota. Outros trabalhos como MAUI, ThinkAir e Scavenger [Kristensen e Bouvin 2010], utilizam um conceito menos intrusivo para realizar a operação de *offloading*, baseado apenas na marcação[9] de método, além de utilizar o conceito de particionamento dinâmico, no qual é decidido em tempo de execução se vale a pena realizar uma operação de *offloading*. No entanto, o CloneCloud é a solução menos intrusiva, pois esta solução modifica a máquina virtual do dispositivo móvel, permitindo assim, que qualquer aplicativo móvel realize a operação de *offloading* sem a necessidade de alterar o código fonte da aplicação.

O *framework* MpOS utiliza um sistema próprio de *offloading* baseado nas ideias do RPC tradicional [Thurlow 2009], além de empregar a mesma ideia de marcação de método, para realizar as operações de *offloading*. Estes métodos marcados possuem diversas restrições não podendo ter internamente os seguintes elementos: (i) chamadas a itens de interface de usuário; (ii) chamadas a recursos de E/S de arquivo, banco de dados e rede; (iii) acesso a outros recursos e

[9] Conhecido como *annotations* na linguagem Java e *attributes* na linguagem C#.

sensores do dispositivo móvel. Este tipo de restrição também acontece durante a seleção da *thread* para *offloading* no CloneCloud e na escolha do método remoto pelo desenvolvedor no *framework* MAUI. No caso dos tipos de particionamento do aplicativo, o MpOS suporta as duas formas dinâmico e estático, deixando a critério do programador qual utilizar.

O *framework* Hyrax [Marinelli 2009] mostra outra forma de particionamento estático, como também demonstra, a dificuldade de adaptar certos paradigmas de programação como o MapReduce, para ser utilizado em um ambiente de dispositivos móveis. No caso do *framework* do AlfredO [Giurgiu *et al.* 2009] é mostrado o uso do particionamento dinâmico, porém utiliza um sistema de decisão muito "pesado", além de utilizar um componente, que mexe na "forma natural" de desenvolver aplicações em dispositivos móveis.

O sistema de decisão do *framework* MpOS é mais simples, quando comparado com sistemas de outros trabalhos como MAUI, CloneCloud e ThinkAir. No caso do MpOS a decisão é tomada em cima das informações produzidas pelo *profiler* de rede, para decidir rapidamente onde fazer a operação de *offloading*.

Nesta seção de trabalhos relacionados, oito trabalhos foram listados, porém apenas o Scavenger tentou tratar da questão da heterogeneidade do *offloading* em plataformas móveis de forma parcial, pois não conseguiu validar a sua solução utilizando duas plataformas móveis diferentes. A solução do MpOS definiu e desenvolveu um *framework*, para duas plataformas móveis, a fim de tratar dessa questão, mostrando também experimentos e resultados, em relação a estas duas plataformas.

Os *frameworks* do MAUI, CloneCloud e ThinkAir desenvolveram um sistema de implantação de serviço dinamicamente, enquanto o *framework* do Kimberley possui suporte ao um sistema de descoberta de serviço, para localizar a presença de um *cloudlet* dinamicamente na rede sem fio. O *framework* MpOS suporta também estas duas funcionalidades.

Finalmente, a Tabela 2.1 apresenta de forma simplificada uma comparação entre os trabalhos relacionados, com a proposta do *framework* MpOS, usando os seguintes parâmetros de comparação: Objetivo do *Offloading* (OO), Tipo de Particionamento (TP), Granularidade (GR), Tipos de *Profiling* (PR), Descoberta de Serviço (DS), Implantação de Serviço (IS), Multiplataforma (MP).

Tabela 2.1. Comparação entre os trabalhos relacionados

Framework	OO	TP	GR	PR	DS	IS	MP
Kimberley (2009)	Desempenho	-	Migração de MV	Rede	Sim	Sim	Não
CloneCloud (2011)	Desempenho e Economia de Energia	Estático	Thread	CPU e Rede	Não	Sim	Não
Spectra (2001)	Desempenho	Estático	Método	CPU, Rede e Energia	Sim	Não	Não
Hyrax (2009)	Desempenho	Estático	Map and Reduce	Não possui	Não	Não	Não
AlfredO (2009)	Desempenho	Estático e Dinâmico	OSGi Bundle	CPU e Rede	Sim	Sim	Não
MAUI (2010)	Economia de Energia	Dinâmico	Método	CPU e Rede	Não	Sim	Não
Scavenger (2010)	Economia de Energia e Desempenho	Dinâmico	Método	CPU e Rede	Sim	Sim	Parcial
ThinkAir (2012)	Desempenho	Dinâmico	Método	CPU, Rede	Não	Sim	Não
MpOS	Desempenho	Estático e Dinâmico	Método	Rede	Sim	Sim	Sim

2.5 Considerações Finais do Capítulo

Este capítulo apresentou a discussão teórica sobre o tema *Mobile Cloud Computing*, mostrando também os principais componentes arquiteturais e os principais desafios dessa área. Na parte dos trabalhos relacionados, foram elencados oito destes trabalhos que influenciaram no desenvolvimento do *framework* proposto.

Nesta seção de trabalhos relacionados, também foram apresentados alguns conceitos específicos da técnica de *offloading*, como a migração de máquina virtual e as duas formas de particionamento da aplicação, sendo apresentado no final da seção, um resumo comparativo entre os *frameworks* e o que cada um influenciou em relação às funcionalidades do MpOS. É bom ressaltar que todos os *frameworks* citados foram desenvolvidos para uma determinada plataforma móvel. No caso do *framework* MpOS existem duas implementações para cada plataforma móvel, utilizando funcionalidades em comum e uma mesma arquitetura, que exigiu um maior esforço para construir o *framework* MpOS.

O próximo capítulo apresenta o *framework* MpOS através de uma metodologia de projeto, mostrando os principais requisitos funcionais e não funcionais, bem como uma visão geral em relação ao processo de configurar e executar o *framework*.

Capítulo 3

Framework MpOS

Os conceitos e os desafios apresentados nesta dissertação acerca do tema *Mobile Cloud Computing* motivam a criação do MpOS (*Multi-platform Offloading System*), um *framework* que realize a operação de *offloading* em múltiplas plataformas móveis. Este capítulo apresenta uma visão geral da proposta, bem como metodologia e arquitetura utilizadas para o desenvolvimento da solução.

3.1 Introdução

Os componentes de arquitetura e as especificações dos requisitos que compõem o *framework* MpOS foram inspirados nos diversos trabalhos relacionados, que foram apresentados na Seção 2.4. De acordo com Satyanarayanan *et al.* (2009), Dinh *et al.* (2011), Fernando *et al.* (2013), os *frameworks* em MCC possuem uma estrutura similar de funcionamento. Por exemplo, possuem componentes de (i) descoberta de serviço; (ii) implantação dinâmica de serviço; (iii) realização de um *profiling* de rede ou de processo; e (iv) execução da operação de *offloading*, que pode acontecer em diversos níveis de granularidade, como *thread*, método, componente, dentre outros níveis.

O principal objetivo desta proposta é tratar da questão do *offloading* em diferentes plataformas móveis utilizando no caso o Android e o Windows Phone, por serem plataformas que são similares tecnicamente. O MpOS enquadra-se na categoria de trabalhos que realizam a técnica do *offloading* em nível de método usando chamada remota de procedimento (RPC) e com suporte ao particionamento estático ou dinâmico do aplicativo móvel.

Neste trabalho, por limitação de escopo, nenhum mecanismo de segurança e autenticação foi desenvolvido para ser usado durante a execução da operação de *offloading*, como também não foram consideradas as questões relacionadas com a escalabilidade do servidor, onde centenas de clientes poderiam acessar um mesmo *cloudlet* ou serviço de *offloading* na Internet.

3.2 Metodologia do Projeto

A metodologia do *framework* MpOS realizou primeiramente a elicitação dos requisitos, a partir da análise dos trabalhos relacionados na Seção 2.4. Por meio, desta análise foram escolhidos os requisitos funcionais e não funcionais, para compor um *framework* que realizasse a

operação de *offloading* em múltiplas plataformas móveis. No entanto, todo *framework* em *Mobile Cloud Computing* deve possuir no mínimo esses três requisitos funcionais e dois requisitos não funcionais:

I. Requisitos Funcionais:
   1. O *framework* deve realizar a operação de *offloading* computacional, baseado em algum nível de granularidade (como *thread*, método, componente, dentre outros), em relação à migração do processamento;
   2. O sistema deve ter um componente para realizar *profiling* de rede e/ou *profiling* de execução, guardando também o último resultado ou mantendo um histórico destes;
   3. O *framework* deve decidir sempre realizar uma operação de *offloading* (particionamento estático) e/ou tomar a decisão autonomamente (particionamento dinâmico), com base nos resultados gerados pelo sistema de *profiling*. (trocar esse requisito pela a capacidade de implantação dinâmica)

II. Requisitos não Funcionais:
   1. O *framework* deve ser capaz de recuperar-se das falhas de conectividade;
   2. O sistema não pode interferir na corretude do aplicativo móvel;

As análises dos trabalhos relacionados indicam também que os *frameworks* em MCC, utilizam uma arquitetura distribuída, baseada no modelo tradicional Cliente/Servidor. Neste modelo, a parte cliente é representada pelos dispositivos móveis, enquanto a parte servidora é demonstrada pelos servidores remotos ou *endpoint*. Um servidor remoto pode ser constituído por máquinas próximas, que estão dentro de uma mesma rede Wi-Fi, através do conceito de *cloudlet server*, ou mesmo, por máquinas hospedadas dentro de uma nuvem pública na Internet. No tocante ao *framework* MpOS, o mesmo conceito arquitetural é empregado.

Este trabalho também adotou uma metodologia iterativa e incremental (ver Figura 3.1), para desenvolver o *framework* MpOS. A primeira versão deste *framework* foi desenvolvida para a plataforma Android, para depois ser construído uma versão equivalente na plataforma do Windows Phone. A escolha destas plataformas ocorre por causa da semelhança entre as linguagens de programação Java e C#, como também de suas APIs.

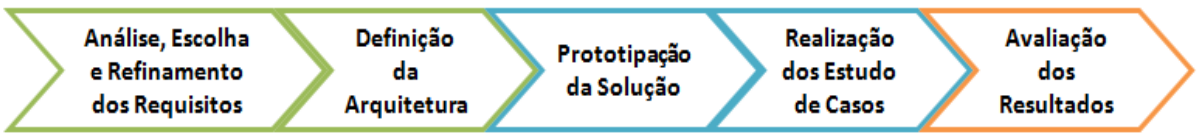

Figura 3.1. Processo de desenvolvimento iterativo e incremental

Assim, na primeira etapa desse processo (ver Figura 3.1) foi realizado uma análise, escolha e refinamentos dos requisitos, a partir dos trabalhos relacionados dessa dissertação. A próxima etapa foi definida uma arquitetura de suporte para a solução do *offloading* em múltiplas plataformas móveis. A terceira etapa foi definida um protótipo da solução do *framework* MpOS com base nos requisitos elencados. Na penúltima etapa foram desenvolvidos dois estudos de casos para demonstrar a viabilidade do *framework* MpOS, em diferentes plataformas móveis. E finalmente, na última etapa foi realizada uma avaliação dos estudos de casos, através da criação de diversos experimentos que avaliam o desempenho e o funcionamento do *framework* MpOS.

As seguintes subseções apresentam os requisitos funcionais e não funcionais do MpOS, como também será mostrado uma visão geral da configuração e do funcionamento da execução do *framework*, finalizando com a composição da arquitetura.

3.2.1 Requisitos Funcionais

Nesta seção são definidos os requisitos funcionais (ver Tabela 3.1), que são necessários para construir o *framework* MpOS, sendo também priorizados em três níveis (Desejável, Importante ou Essencial).

Tabela 3.1. Requisitos funcionais do *framework* MpOS

Identificação	Descrição	Prioridade
RF 01	Permitir que a inicialização do *framework* fosse efetuada através de um único ponto de forma simplificada.	Desejável
RF 02	O *framework* deve ser configurado através da marcação da classe de inicialização do aplicativo móvel.	Essencial
RF 03	O sistema deve permitir marcar as variáveis de instância da classe de inicialização, para serem dinamicamente instanciadas.	Essencial
RF 04	O MpOS deve interceptar todos os métodos de uma variável de instância que foi inicializada dinamicamente.	Essencial
RF 05	O *framework* deve permitir a marcação dos métodos das interfaces que podem realizar a operação de *offloading*.	Essencial
RF 06	O MpOS deve prover um serviço de descoberta de *cloudlet server* em uma rede local sem fio.	Desejável
RF 07	O sistema deve prover a descoberta de serviços, que estão localizados em um determinado servidor remoto.	Importante

RF 08	O *framework* deve prover um sistema de implantação dinâmica de um serviço de *offloading* que é inexistente em um servidor remoto.	Essencial
RF 09	O MpOS deve possuir um sistema de *profiling* de rede para avaliar a qualidade de uma rede entre o cliente móvel e um servidor remoto.	Essencial
RF 10	O *framework* deve possuir um sistema para decidir sobre um método interceptado, quando realizar ou não, uma operação de *offloading* a partir dos dados que foram produzidos pelo *profiling* da rede.	Essencial
RF 11	O sistema deve adotar um mecanismo próprio de *offloading* baseado em um sistema de RPC.	Essencial
RF 12	Permitir que o desenvolvedor definisse manualmente um protocolo para serializar informações da operação de *offloading*.	Importante
RF 13	O sistema deve permitir consulta aos resultados de *profiling* de rede	Desejável
RF 14	O MpOS deve permitir o fornecimento de estatísticas de rede, em relação à última operação de *offloading* realizada.	Desejável
RF 15	O *framework* deve possuir um serviço de ambiente de execução remoto para tratar da operação de *offloading* para cada uma das plataformas móveis.	Essencial
RF 16	Permitir a configuração da parte servidora do *framework* por meio de um arquivo de configuração.	Desejável
RF 17	O sistema (servidor remoto) após detectar a presença de um cliente móvel na rede local sem fio deve lhe informar o seu endereço IP.	Importante
RF 18	O MpOS depois de implantar um serviço de *offloading*, deve informar a porta deste serviço para o sistema de descoberta de serviço.	Essencial
RF 19	O *framework* deve armazenar todos os resultados de *profiling* de rede, que foram gerados pelo dispositivo móvel.	Desejável

3.2.2 Requisitos não Funcionais

Nesta seção são definidos os requisitos não funcionais (ver Tabela 3.2) do *framework* MpOS, como também sua prioridade (Desejável e Essencial) em relação ao planejamento.

Tabela 3.2. Requisitos não funcionais do *framework* MpOS

Identificação	Descrição	Prioridade
RnF 01	Utilizar na parte cliente as plataformas móveis Android 4.x ou superior e Windows Phone 8.x ou superior.	Essencial
RnF 02	Utilizar na parte servidora o sistema operacional do Windows 7 com Java SE 7 e .NET Framework 4.0, ou versões superiores.	Essencial
RnF 03	Utilizar o Banco de Dados SQLite para guardar as informações do cliente no servidor remoto.	Essencial
RnF 04	Empregar um sistema de log circular para registrar as atividades do servidor remoto.	Desejável
RnF 05	A rede sem fio deve permitir o uso do protocolo UDP Multicast na porta 31000 usado para o cliente descobrir a presença de um *cloudlet* nesta rede.	Desejável
RnF 06	O *framework* no lado cliente em caso de erro interno ou externo (no servidor remoto) deve ser capaz de recupera-se sozinho não interferindo no funcionamento normal e na corretude da aplicação móvel.	Essencial
RnF 07	O sistema do MpOS no servidor remoto em situação de erro não deve deixar de processar as requisições dos outros clientes.	Essencial
RnF 08	O *framework* em caso de erro no servidor remoto deve enviar uma mensagem de exceção para o cliente móvel para ser mostrado no log da aplicação móvel.	Desejável
RnF 09	O MpOS deve ser capaz de processar altas taxas de requisição de RPC para realizar a operação de *offloading* em aplicativos de tempo-real.	Desejável
RnF 10	Todos os serviços de rede em execução no servidor remoto e que utiliza o protocolo TCP devem ser *multithreads*, para maximizar o desempenho do sistema.	Essencial

3.2.3 Visão Geral do MpOS

Com a definição dos requisitos funcionais e não funcionais foi estabelecido um cenário de como seria o processo de configuração e funcionamento do *framework* MpOS, para depois estabelecer a arquitetura da solução. O aplicativo móvel interage com uma biblioteca cliente fornecido pelo *framework* MpOS e possibilita que sejam realizadas as operações de *offloading*, para um determinado servidor remoto. Enquanto, no servidor remoto o MpOS é responsável por diversas atividades de redes como a descoberta e a implantação de serviço, *profiling* de rede, além do ambiente para executar os serviços de *offloading*.

Nas próximas subseções são descritos os processos de configuração do *framework* MpOS, logo em seguida é apresentado funcionamento desse *framework* considerando a primeira execução do aplicativo móvel.

3.2.3.1 Processo de Configuração

O desenvolvedor precisa definir uma série de configurações para integrar o *framework* MpOS em conjunto com sua aplicação móvel, tanto na parte cliente, quanto na parte servidora deste *framework*. A Figura 3.2 apresenta os passos que o desenvolvedor deve tomar para configurar a sua aplicação, em conjunto com MpOS.

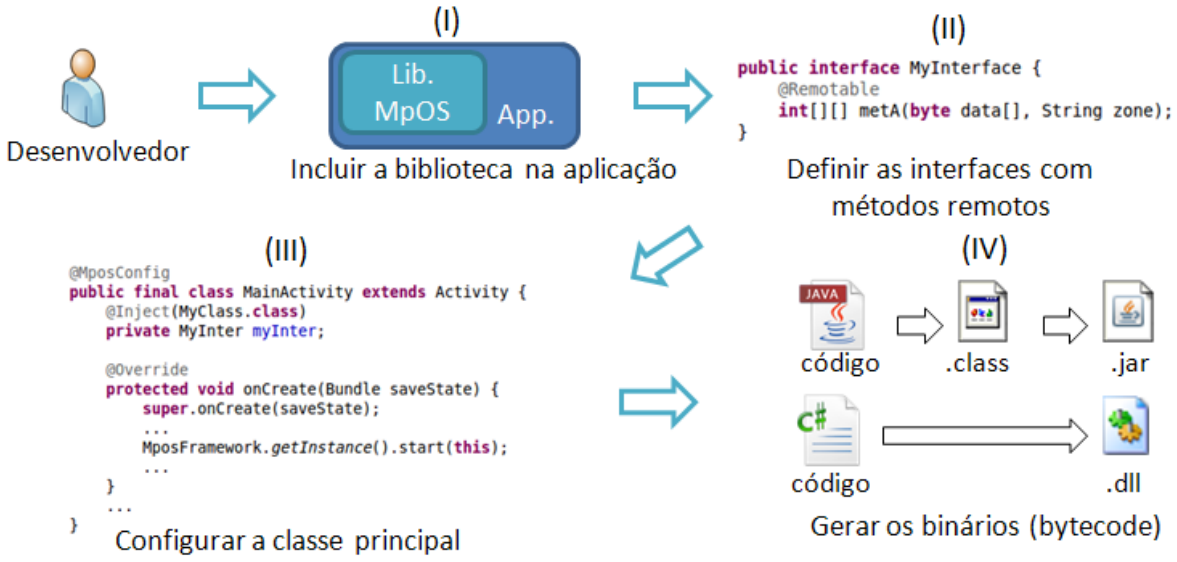

Figura 3.2. Passos da configuração diente do MpOS

No MpOS o processo de configuração inicializa com a integração da biblioteca cliente em conjunto com o projeto da aplicação móvel. Os próximos passos serão realizados diversas configurações diretamente no código fonte da aplicação, por meio de marcações de código conforme apresentado na Figura 3.2.

No segundo passo o desenvolvedor precisa criar ou definir, as interfaces que possuem métodos candidatos para realizar a operação de *offloading* por meio da marcação no código fonte. Esta marcação possui também três itens de configuração: (i) define a forma do particionamento, se é dinâmico ou estático, sendo por padrão dinâmico; (ii) habilita os status da rede e do processamento, em relação à operação de *offloading*, sendo desabilitado por padrão e (iii) define o *endpoint* prioritário para executar a técnica de *offloading*, sendo por padrão executado no *cloudlet server*. Os métodos que são marcados pelo desenvolvedor devem possuir obrigatoriamente um ou mais parâmetros, além de ter um retorno, pois a técnica de *offloading* no *framework* MpOS utiliza de um sistema de RPC que demanda destas características para executar remotamente um método.

Após definir as interfaces que possuem métodos passíveis para realizar a operação de *offloading*, o próximo passo seria integrar na inicialização do aplicativo móvel, o processo de inicialização do *framework* MpOS. Em ambas as plataformas móveis, a inicialização do aplicativo acontece por meio de uma classe principal [Mednieks *et al.* 2012, Whitechapel e McKenna 2013].

Nesta classe é necessário utilizar uma marcação que permita o desenvolvedor configurar a execução do *framework* MpOS, em conjunto com a aplicação móvel de acordo com a sua necessidade. Os itens que podem ser configurados na marcação são apresentados pela Tabela 3.3.

Tabela 3.3. Itens de configuração da marcação na classe principal

Item	Configuração Padrão
Definir o tipo de *profiling* de rede (*light* ou *full*).	*Profiling light*
Definir um *endpoint* secundário na Internet ou Intranet, para realizar a operação de *offloading*.	Nenhum *endpoint* definido
Permitir a captura dos detalhes sobre o dispositivo móvel (modelo, operadora e conectividade usada).	Habilitado
Permitir a detecção da localidade do dispositivo móvel, por meio do GPS ou sistema de localização similar.	Desabilitado
Habilitar o sistema de descoberta de *cloudlet server* no Wi-Fi.	Habilitado
Ativar o sistema de decisão dinâmica para operação de *offloading*.	Ativado
Habilitar o sistema de implantação de serviço de *offloading*.	Habilitado

O desenvolvedor precisa definir as variáveis de instância que são do tipo das interfaces que possuem métodos marcados para realizar a operação de *offloading*. Se o desenvolvedor instanciar diretamente estas variáveis, não há como interceptar as chamadas dos métodos a partir

do *framework* MpOS. Por causa disso, o programador deve marcar as variáveis de instância, definindo na propriedade dessa marcação, o tipo concreto que será instanciado dinamicamente por estas variáveis. Assim, este processo de marcação permitirá que o MpOS intercepte todos os métodos invocados, a partir dessas variáveis que foram dinamicamente instanciadas.

No último passo da configuração do MpOS, o desenvolvedor precisa gerar um binário do aplicativo desenvolvido, em uma linguagem intermediária para cada plataforma móvel. Após a criação destes binários, estes arquivos junto com as bibliotecas que compõe o aplicativo móvel, precisam ser salvos nos *asserts* do projeto em desenvolvimento. Com isso o desenvolvedor prepara sua aplicação para realizar o processo de implantação de serviço em um determinado servidor remoto. É necessário que seja mantida o versionamento da aplicação e da atualização dos binários do aplicativo móvel, para evitar a ocorrência de erros proveniente da utilização de dependências que estão desatualizadas no lado do servidor.

O processo de configuração do MpOS no lado servidor é apresentado na pela Figura 3.3. Neste caso, o desenvolvedor precisa implantar todos os arquivos e executáveis necessários para o funcionamento do *framework* MpOS em um *cloudlet server* ou servidor remoto em uma nuvem pública. No final desse processo, o desenvolvedor precisa definir em um arquivo de configuração o IP da máquina servidora e caso seja necessário também pode mudar os valores das portas em relação aos diversos serviços de rede que são suportados pelo *framework*.

Figura 3.3. Passos da configuração servidor do MpOS

3.2.3.2 Processo de Execução

Depois de definir no aplicativo móvel as configurações do MpOS API conforme visto na subseção anterior, será descrito de forma simplificada nesta subseção, o funcionamento da aplicação móvel em conjunto com este *framework*, sendo consideradas neste cenário as seguintes restrições: (i) o desenvolvedor utilizou a configuração padrão para as propriedades das marcações de código, (ii) o *framework* MpOS executa somente em um *cloudlet server*, não possuindo nenhum serviço de *offloading* ou aplicativo previamente implantado e (iii) neste cenário foi utilizado o procedimento de serialização automática. É importante ressaltar que todas as marcações com

suas propriedades que são definidas em tempo de desenvolvimento de *software*, serão obtidas pelo *framework* MpOS em tempo de execução, através de processos de inspeção ou reflexão do código [Forman *et al.* 2004].

No processo de inicialização do aplicativo móvel, o *framework* MpOS é invocado simultaneamente com esse procedimento. Durante a inicialização deste componente, os serviços de rede serão disparados de acordo com as configurações que foram definidas na marcação da classe principal. Desse modo, os seguintes serviços de rede como: (i) descoberta de um *cloudlet* na rede local Wi-Fi; (ii) descoberta de serviços de rede e (iii) implantação de serviço, estarão em funcionamento como ilustrado na Figura 3.4.

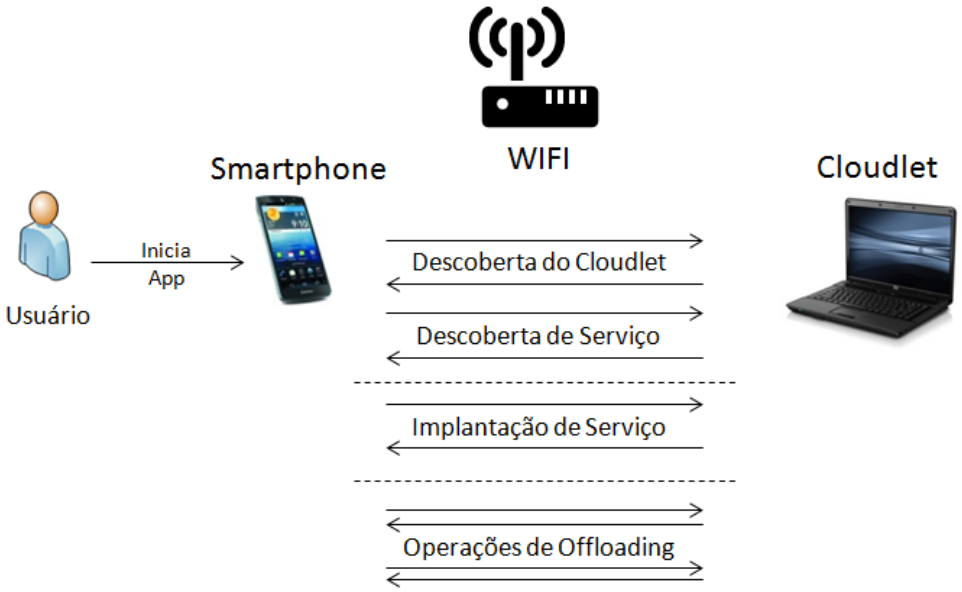

Figura 3.4. Processo de execução do MpOS

Ainda com relação ao processo de inicialização do *framework*, o procedimento de injeção de dependência [Fowler 2004] é realizado apenas sobre as variáveis de instância que são marcadas na classe principal do aplicativo móvel. Este procedimento verifica se uma variável marcada deriva de uma interface válida, ou seja, aquela que possui pelo menos um método com a marcação para realizar a operação de *offloading*. Caso seja positiva essa verificação a variável é instanciada dinamicamente para um *proxy* de objeto [Gamma *et al.* 1994], baseado no tipo concreto que foi definida na propriedade da marcação dessa variável. Depois disso, o sistema de injeção de dependência irá associar dinamicamente o *proxy* de objeto com a variável que foi marcada, permitindo assim, que os métodos destas variáveis sejam interceptados pelo MpOS ao serem invocados pelo aplicativo móvel.

Após concluir o processo de inicialização do MpOS API, o primeiro componente de rede que interage com um servidor remoto é a descoberta de serviço (ver Figura 3.4). Este sistema

possui uma funcionalidade que descobre em uma rede local sem fio, a presença de um *cloudlet*. O cliente neste caso anuncia sua presença por meio de diversas mensagens de *multicast*. O MpOS no lado servidor possui um serviço de rede para escutar este tipo de mensagem e responde para o destinatário com o endereço IP do *endpoint*, conforme foi definido no arquivo de configuração do servidor remoto.

Com isso, o cliente poderá interagir diretamente com o servidor remoto para descobrir as portas dos outros serviços de rede, como a implantação de serviço, *profiling* de rede e o próprio serviço de *offloading*. Nesta operação o *framework* passa para o servidor remoto o nome do aplicativo com sua versão, além do tipo de plataforma móvel que a aplicação executa. Na parte servidora, o sistema de descoberta de serviço verifica com base nestas informações, se existe um determinado serviço de *offloading*. No entanto, conforme dito nesta subseção o servidor remoto não possui nenhum serviço de *offloading* previamente implantado, e dessa maneira, a pesquisa não retornará nenhum resultado. Por causa disso, o servidor remoto responderá ao cliente, passando como mensagem, todas as portas dos serviços fixos de rede, como implantação de serviço e *profiling* de rede, além de atribuir um valor negativo, para a porta que representa o serviço de *offloading*. O cliente ao receber esta mensagem, vai perceber que o serviço de *offloading* não está disponível no servidor remoto, por causa do valor negativo da porta, inicializando assim, o procedimento de implantação de serviço.

Este procedimento de implantação de serviço consiste na transferência dos arquivos que estão disponíveis nas dependências da aplicação móvel, além do envio do nome da aplicação com sua versão, para um selecionado *endpoint*. O servidor remoto recebe estes arquivos e estas informações que representam uma aplicação móvel criando posteriormente um serviço de *offloading*. No final da operação de implantação de serviço, o servidor envia para o cliente o número da porta onde está sendo executado este serviço de *offloading*, e disponibiliza esta porta para o sistema de descoberta de serviço, para que outros clientes possam ter acesso ao mesmo serviço de *offloading*.

No final da execução da descoberta de serviço, o sistema de decisão dinâmica (SDD) é invocado e funciona em paralelo com restante da aplicação móvel. Este sistema funciona em conjunto com o serviço de *profiling* de rede e avalia de tempos em tempos, se as condições de rede são apropriadas para realizar uma operação de *offloading*. Dessa forma a avaliação consiste na medição do RTT (*Round-to-Trip*) médio entre o cliente e o servidor remoto, sendo realizados a cada 45 segundos pelo SDD. Se o resultado dessa coleta exceder o limiar de 80ms, significa que a rede está inadequada para o MpOS realizar a operação de *offloading*. Caso contrário o MpOS poderá realizar a operação de *offloading*. A tomada de decisão fica disponível globalmente pelo

framework MpOS, sendo atualizados a cada 45 segundos. O valor desse limiar também pode ser definido durante o processo de configuração do aplicativo móvel, porém o valor padrão desse limiar está relacionado com *offloading* em aplicativos de tempo real, no qual um alto valor do RTT pode atrapalhar o desempenho da aplicação como o todo. Por fim, o SDD funciona apenas para aqueles métodos que são marcados para realizar a operação de *offloading* empregando também a propriedade do particionamento dinâmico, pois se o desenvolvedor definir o particionamento estático entende-se que o desenvolvedor deseja sempre realizar a operação de *offloading* para um servidor remoto independente das condições de rede.

O sistema de *offloading*, por sua vez funciona em conjunto com a execução do aplicativo móvel através das variáveis de instância que foram injetadas dinamicamente pelo *framework* MpOS. A Figura 3.5 apresenta um fluxograma do funcionamento da operação de *offloading*.

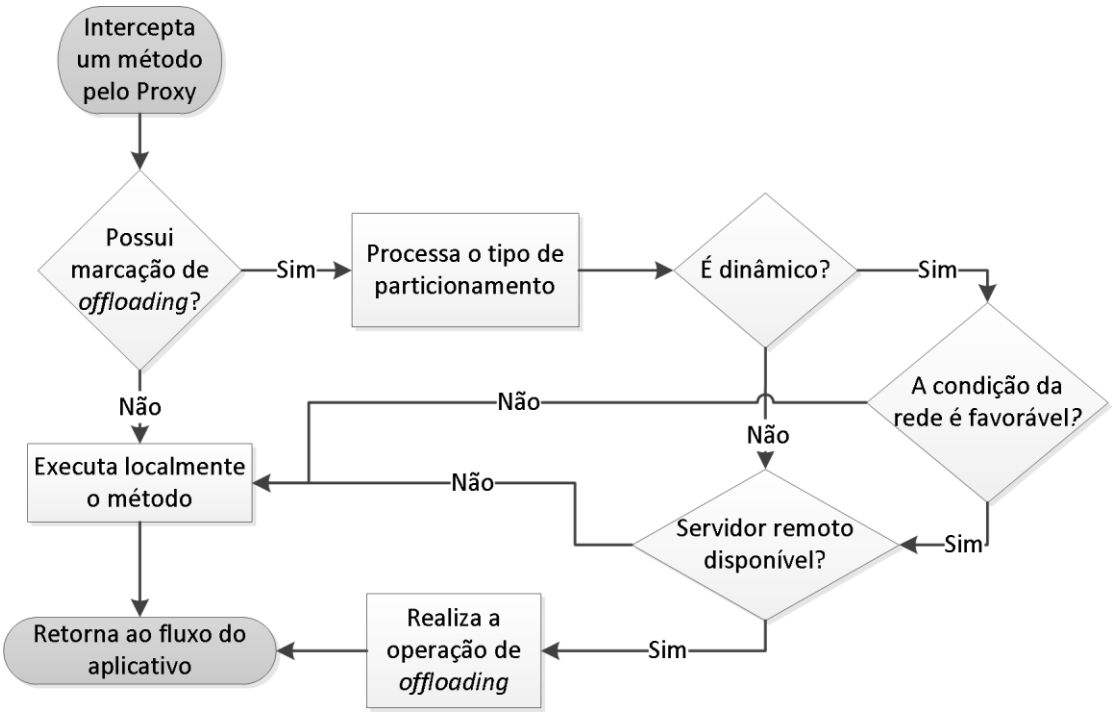

Figura 3.5. Funcionamento do processo de *offloading*

Durante a execução da aplicação, o *framework* MpOS intercepta uma chamada de método e o sistema se esse método está vinculado com a marcação da operação de *offloading*. Caso seja verdade, esta marcação será processada para obter o tipo de particionamento utilizados. Se o particionamento for dinâmico, o sistema verificará se as condições da rede são favoráveis para realizar a operação de *offloading*, além da disponibilidade do servidor remoto. Se for verdade o MpOS irá realizar a operação de *offloading*. No entanto, na hipótese do particionamento estático, o sistema verifica apenas se o servidor remoto está disponível, realizando também a operação de *offloading* caso seja verdadeiro independente da decisão produzida pelo SDD. Portanto, qualquer

caminho alternativo a esses que foram descritos o método interceptado executará localmente no dispositivo móvel.

É bom destacar que a operação de *offloading* acontece da seguinte forma, um cliente passará para o servidor remoto as seguintes informações: (i) o nome do método interceptado; (ii) o nome do objeto que utiliza este método; e (iii) os parâmetros deste método interceptado; sendo estas informações serializadas de acordo com o processo que foi definido, durante o tempo de desenvolvimento do aplicativo móvel. No lado servidor, o serviço de *offloading* recebe do cliente estas informações e instancia o objeto que invoca o método remoto que foi requisitado, usando neste caso as dependências do aplicativo móvel que foi previamente implantado no servidor remoto. Depois de instanciado o objeto será invocado dinamicamente o método que o cliente requisitou, passando na sequencia os parâmetros dos métodos que foram recebidos durante esta sessão de *offloading*. O resultado da execução do método (no servidor) será enviado para o cliente remoto, usando o mesmo procedimento de serialização que foi utilizado no inicio desta sessão. Na parte cliente, o valor retornado pelo servidor é substituído no retorno do método interceptado, seguindo posteriormente com o fluxo do aplicativo.

Este sistema de *offloading* (na parte do cliente do *framework* MpOS) também garante a transparência de localização [Kurose e Ross, 2012] em relação à execução da chamada de um método que foi marcado para realizar a operação de *offloading*. Pois, o aplicativo móvel não sabe diferenciar se essa execução da chamada do método ocorreu localmente no aparelho ou remotamente em um servidor remoto.

3.2.4 Arquitetura da Solução

Para dá suporte ao conjunto de requisitos funcionais e ao cenário definido nesta visão geral, foi definido que o *framework* MpOS seguirá o modelo tradicional Cliente/Servidor de aplicações distribuídas. Pois é quase natural do framework existir uma parte cliente denominada de MpOS API e uma parte servidora chamada de MpOS Plataforma. A Figura 3.6, apresenta uma visão geral dessa arquitetura, mostrando as interações entre as camadas.

Nessa arquitetura existem dois tipos de interações de redes. O primeiro tipo é composto por interações unidirecionais[10] ou momentâneas, pois acontecem apenas durante a inicialização do aplicativo móvel, como por exemplo, as interações de descoberta de serviço e as interações de implantação de serviço. O segundo tipo de interação é classificada por interações bidirecionais[11] e

[10] Indica o sentido de quem fornece dados.
[11] Ambas as partes interagem uma com a outra durante a execução dos serviços.

longas, porque acontecem durante todo o período de execução do aplicativo móvel, como por exemplo, as interações de *offloading* e a medição da qualidade da rede.

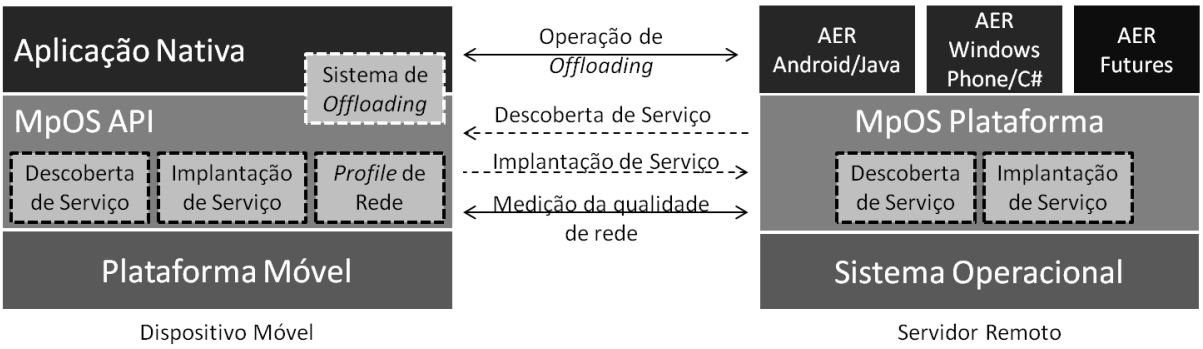

Figura 3.6. Arquitetura do MpOS

A arquitetura do *framework* MpOS é divida em três camadas. A primeira camada é representada na porção cliente pelo aplicativo móvel e na porção servidora pelos Ambientes de Execução Remotos (AER). Estes AER são compostos por serviços de rede que processam as diversas operações de *offloading* em diferentes plataformas móveis, conforme mostrado na Figura 3.6. Um servidor remoto pode executar diversos AER, permitindo assim, processar diversas requisições de heterogêneas plataformas móveis.

A segunda camada é composta pelos componentes, MpOS API e MpOS Plataforma, com seus diversos serviços de rede, que são apresentados nesta seção apenas resumidamente. Com relação ao componente do MpOS API existem quatro serviços de redes:

a) **Descoberta de Serviço:** o objetivo desse serviço é descobrir um MpOS Plataforma que esteja sendo executando em uma mesma rede sem fio, como acontece com a arquitetura *cloudlet*. Além disso, esse serviço pode também requisitar informações sobre outros serviços que estejam sendo executados no servidor remoto, como Serviço de *Offloading*, *Profile* de Rede e dentre outros. Caso seja configurado na inicialização do aplicativo móvel um *endpoint* de Internet, este serviço de descoberta realizará o mesmo procedimento de requisição, em relação a um *endpoint* remoto independentemente se o MpOS API conseguiu ou não, localizar a presença de um *cloudlet* na rede Wi-Fi;

b) **Implantação de Serviço:** quando a Descoberta de Serviço não detecta nenhum Serviço de *Offloading* no servidor remoto, a Implantação de Serviço é usada para enviar todas as dependências (arquivos binários e de bibliotecas), do aplicativo móvel para este servidor instanciar um novo Serviço de *Offloading*. É importante destacar que uma vez que o Serviço de *Offloading* está em execução para uma especifica

aplicação móvel, este serviço também estará disponível para todos os outros usuários de *cloudlet* ou na Internet, caso esteja hospedado em um serviço de nuvem pública.

c) ***Profile* de Rede:** o objetivo desse serviço é mensurar a qualidade de uma conexão de rede entre o dispositivo móvel e o servidor remoto. As métricas empregadas neste serviço para mensurar essa qualidade de rede são: RTT, *jitter* e largura de banda. Desta forma estas métricas permitem o Serviço de *Offloading* decidir quando realizar ou não, uma operação de *offloading*.

d) **Serviço de *Offloading*:** esse serviço é responsável por interceptar os métodos daquelas variáveis que são instanciadas dinamicamente pelo *framework*, decidindo também, a localidade (local ou remoto) da sua execução. Essa decisão é baseada nos resultados obtidos pelo *Profile* de Rede e no tipo do particionamento (estático ou dinâmico) empregado na marcação desse método.

Os serviços mencionados acima também executam no lado do servidor, através do MpOS Plataforma. É importante destacar que alguns desses serviços, como Descoberta de Serviço e *Profile* de Rede, são portáveis, ou seja, o mesmo componente é usado por todas as plataformas móveis. Enquanto outros serviços como a Implantação de Serviço e o Serviço de *Offloading* são específicos para cada tipo de plataforma móvel.

Por último, a terceira camada da arquitetura MpOS é representada pela plataforma móvel (Android ou Windows Phone) no lado cliente e pelo sistema operacional (Windows) na parte servidora.

3.3 Considerações Finais do Capítulo

A proposta do *framework* MpOS (*Multi-platform Offloading System*) apresenta uma solução de *framework* em *Mobile Cloud Computing* que realiza a operação de *offloading* inicialmente para duas plataformas móveis distintas (Android e Windows Phone).

Outras plataformas móveis como Firefox OS e iOS não foram contemplados no desenvolvimento deste trabalho, por causa de diversos fatores como: (i) limitação do tempo para codificar o *framework* proposto para as duas plataformas selecionadas; (ii) a linguagem do Objective-C e API Cocoa apresentam uma alta curva de aprendizagem; (iii) o ambiente de execução remoto para plataforma do iOS exige que seja utilizado o sistema operacional Mac OSX; e (iv) a baixa adesão ao Firefox OS (2% de *market share*) não compensaria o desenvolvimento devido ao seu pouco tempo de lançamento.

A motivação para o *framework* MpOS adotar um mecanismo próprio de RPC foi por causa das diversas questões técnicas em relação as soluções existentes de RPC, como XML-RPC[12] e JSON-RPC[13]. A principal limitação destas soluções está associada com, a necessidade de enviar um dado binário (foto ou vídeo) para um servidor remoto, sendo preciso codificar e decodificar essa informação para Base64[14]. Consequentemente, este processo aumenta o tamanho dos dados a serem transferidos durante a chamada remota, além do *overhead* computacional envolvido nesse processo de codificação, que pode potencialmente prejudicar o desempenho do aplicativo móvel. Além disso, uma solução própria de RPC permite em alguns casos, como de aplicações de tempo-real, que o desenvolvedor deva escrever manualmente o protocolo de serialização e deserialização usado pelo RPC. Esse protocolo se propõe em aumentar o desempenho da operação de RPC em termos computacionais, além de reduzir o tamanho dessas chamadas, não sendo permitido pelas outras soluções de RPC que foram citadas anteriormente.

Por fim, este capítulo especificou uma metodologia de trabalho, que definiu os requisitos funcionais e não funcionais, além de apresentar uma visão geral em relação à configuração e a execução do *framework* MpOS. O capítulo também mostrou o desenho da arquitetura do *framework* (ver Figura 3.6), explicando resumidamente os seus componentes. O próximo capítulo apresenta em detalhes o funcionamento dos componentes MpOS API e MpOS Plataforma.

[12] http://xmlrpc.scripting.com/spec
[13] http://json-rpc.org/wiki/specification
[14] http://tools.ietf.org/html/rfc4648

Capítulo 4

Protótipo do Framework MpOS

Após a fase de planejamento do *framework* MpOS, que envolveu as especificações dos requisitos e a definição dos componentes da arquitetura, o próximo passo seria demonstrar com detalhes, como foram especificados os elementos propostos para as duas plataformas móveis, com seus respectivos serviços de rede.

4.1 Introdução

O *framework* MpOS, conforme apresentado na Figura 3.6, possui dois tipos de componentes. O primeiro componente denominado de MpOS API interage diretamente com aplicação móvel, sendo também responsável pela operação de *offloading*. Dessa maneira, o processamento do aplicativo pode ser "exportado" ou "terceirizado", para executar em máquinas próximas (conceito *cloudlet*), ou mesmo executar em um *endpoint* remoto na Internet, pretendendo assim, aumentar o desempenho dessa aplicação móvel.

O segundo componente, MpOS Plataforma, está ligado com a parte servidora do *framework* MpOS e executa como um serviço de sistema no sistema operacional Windows. Com isso, o *framework* suporta a questão da heterogeneidade da operação de *offloading* nos dispositivos móveis que utilizam Android e Windows Phone. É bom ressaltar que o MpOS Plataforma pode ser também instalado no sistema operacional Linux ou Mac OSX, porém terá suporte apenas ao *offloading* em relação a plataforma móvel do Android. Pois, a plataforma do Windows Phone utiliza componentes, como Microsoft .NET Framework, que executam exclusivamente no sistema operacional Windows. O MpOS Plataforma também é responsável por executar diversos serviços de rede, relacionado com as atividades de Descoberta de Serviço, Implantação de Serviço, *Profile* de Rede e Serviço de *Offloading*, sendo todos desenvolvidos com tecnologias do Java SE e .NET Framework, de modo a fornecer suporte as necessidades específicas de cada plataforma móvel.

Nas próximas seções serão detalhados os componentes do MpOS API e do MpOS Plataforma, bem como os serviços relacionados com cada um desses componentes.

4.2 MpOS API

Nesta seção será discutido o processo de integração do MpOS API com um aplicativo móvel, além do processo de funcionamento com relação a inicialização do *framework* em conjunto com esta aplicação móvel, apresentando também alguns detalhes da implementação do *framework* MpOS.

O processo de inicialização de um aplicativo móvel é similar em ambas as plataformas móveis. No caso do Android é preciso ter uma classe principal do tipo *Activity*, para inicializar essa aplicação [Mednieks *et al.* 2012]. O MpOS API deve ser integrada com aplicação, através do método *onCreate* da *Activity* principal, conforme apresentado pela Figura 4.1 (b). Essa classe principal precisa também utilizar a marcação *MposConfig* para definir as configurações do MpOS API, de acordo com a necessidade do desenvolvedor, conforme explicado na subseção 3.2.3.1. A Figura 4.1 (b) mostra a marcação *MposConfig* utilizando a configuração padrão do *framework* MpOS.

```
[MposConfig]
public partial class App : Application
{
    [Inject(typeof(BallController))]
    private BallUpdater controllerSerializable;

    [Inject(typeof(BallControllerCustom))]
    private BallUpdaterCustom controllerCustom;

    public App()
    {
        MposFramework.Instance.Start(this, new ProxyFactory());

        (...)

        Debug.WriteLine("[App]: Collision Particle started");
    }
    (...)

}
```

(a)

```
@MposConfig
public final class MainActivity extends Activity {

    @Inject(BallControllerCustom.class)
    private BallCustomUpdater<Ball> controllerCustom;

    @Inject(BallController.class)
    private BallUpdater<Ball> controllerSerializable;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        (...)
        MposFramework.getInstance().start(this);
        (...)
        Log.i(clsName, "Collision Particle started");
    }
    (...)
}
```

(b)

Figura 4.1. Exemplo de inicialização e configuração do MpOS API na classe principal

A inicialização do Windows Phone se difere do Android apenas em relação à classe principal, que é estendida do tipo *Application* [Whitechapel and McKenna, 2013]. Nesta situação, o desenvolvedor integra o MpOS API com aplicação móvel, por meio do construtor dessa subclasse, sendo mostrado na Figura 4.1 (a). A mecânica de configuração do MpOS API na plataforma do Windows Phone acontece da mesma maneira, que na versão Android, por meio do conceito de *attributes* da linguagem de programação C# similar ao conceito de *annotations* em Java.

O desenvolvedor precisa definir e marcar com a marcação *Inject*, aquelas variáveis de instância (ver Figura 4.1) que são do tipo das interfaces que possuem métodos marcados para realizar a operação de *offloading*, permitindo assim, que o *framework* MpOS intercepte estes métodos e decida onde executá-los (localmente no aparelho ou remotamente). Por fim, o

processo de inicialização do MpOS API acontece apenas durante a invocação do procedimento *start* na classe *MposFramework*, devendo ocorrer no momento da inicialização do aplicativo móvel. A Figura 4.2 apresenta os componentes que compõem a inicialização do *framework* MpOS API, sendo demonstrado através de um diagrama de classe por meio da notação UML (*United Modeling Language*).

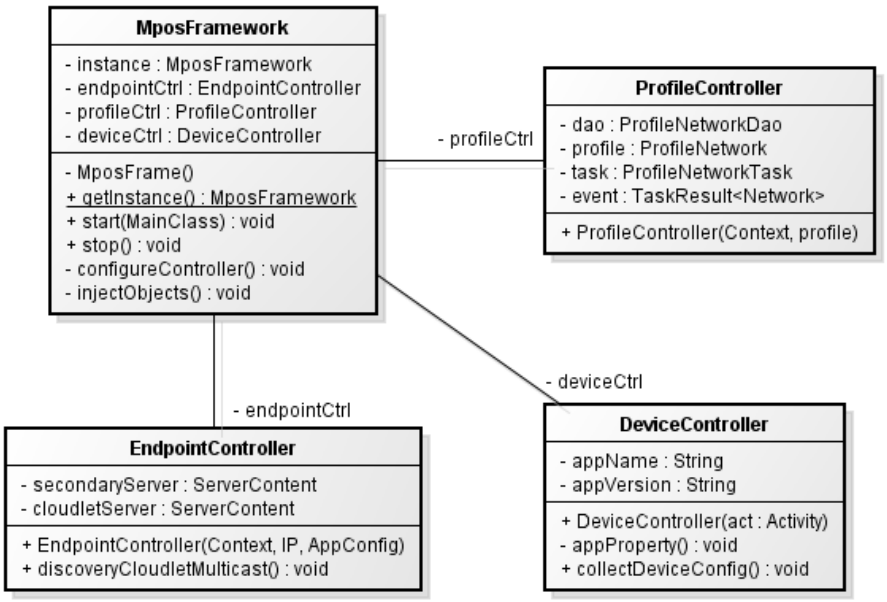

Figura 4.2. Diagrama de classes da Inicialização do MpOS API

A classe *MposFramework* (Figura 4.2) emprega dois padrões de projeto: (i) *Singleton*; e (ii) *Façade*, sendo este último, composto apenas por controladores do sistema. O primeiro padrão garante que seja instanciado e acessado globalmente no projeto, enquanto que o segundo padrão centraliza o acesso as instâncias de todos os controladores do sistema. O método *start* possui um mecanismo interno de controle para garantir que o mesmo seja chamado uma única vez pelo aplicativo móvel, evitando que sejam disparados vários processos de inicialização do MpOS API em um mesmo aplicativo. Por sua vez, o método *start* recebe como parâmetro, a própria classe de inicialização do aplicativo móvel, e internamente invoca os seguintes métodos: (i) *configureController*; e (ii) *injectObjects*.

O método *configureController* é responsável por iniciar e configurar todos os controladores do sistema, sendo nesse caso três alinhados de acordo com as definições da marcação *MposConfig*. A Figura 4.3 mostra o funcionamento desse método através de um diagrama de sequência UML, considerando neste caso as configurações de padrão da marcação *MposConfig*.

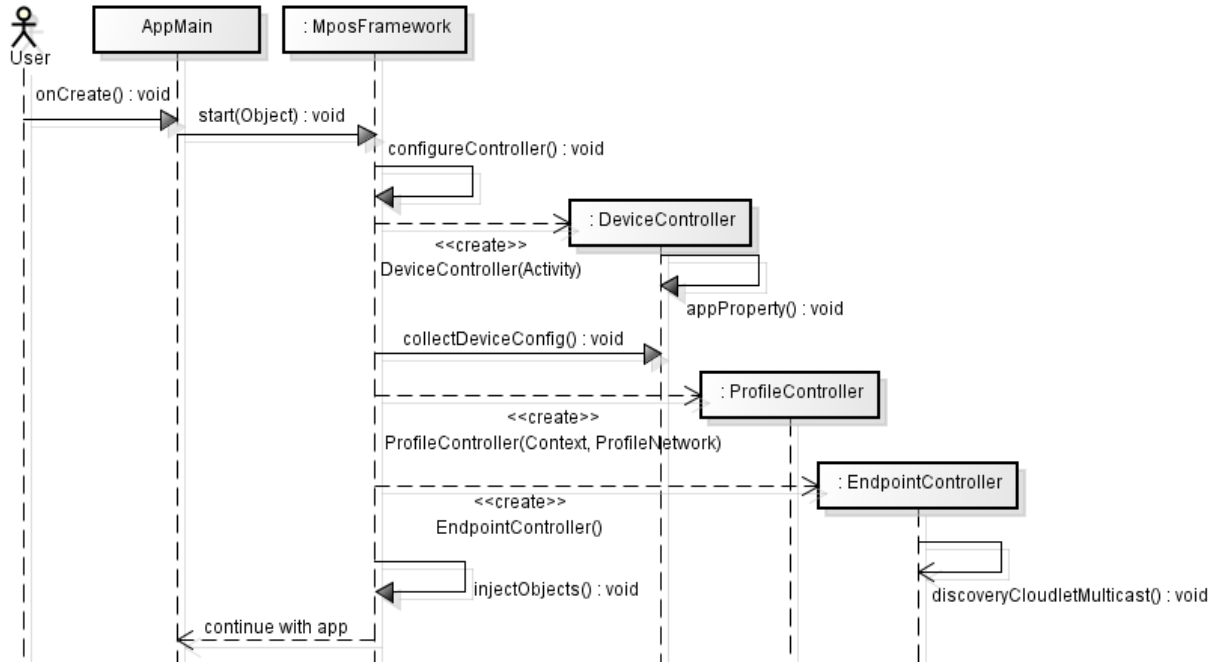

Figura 4.3. Configuração dos controladores no *MposFramework*

O primeiro controlador a ser iniciado no método *configureController* (Figura 4.3), é o controlador do dispositivo (*DeviceController*), responsável por obter o nome e a versão do aplicativo móvel em execução, além de coletar diversas informações sobre o aparelho em si. O próximo controlador é relacionado com controle do sistema de *profiling* e recebe na sua inicialização, o tipo de *profiling* de rede (*light* ou *full*) para ser utilizado, durante a execução do *framework* MpOS. Por último, o controlador *EndpointController* é responsável por controlar o acesso com servidor remoto, tratando da conectividade, dos serviços disponíveis, além do controle dos outros serviços de rede como Descoberta de Serviço (em rede local Wi-Fi e no *endpoint* secundário), Implantação de Serviço e Sistema de Decisão Dinâmica (SDD). Por causa dessa configuração padrão do *MposConfig*, apenas a Descoberta de Serviço no *cloudlet* foi invocado, através de uma chamada assíncrona, ou seja, uma chamada que dispara uma *thread*[15].

Por fim, na classe do *MposFramework*, o método *injectObjects* é responsável pelo processo de injeção de dependências, em relação às variáveis de instância que foram definidas na classe principal do aplicativo móvel. Como previamente explicado na subseção 3.2.3.2. as próximas seções detalham o funcionamento dos componentes de Descoberta de Serviço, Implantação de Serviço, *Profile* de Rede e Serviço de *Offloading*.

[15] A vantagem dessa abordagem é evitar que o fluxo principal do aplicativo seja bloqueado por operações de leitura, relacionada com algum processo de E/S do programa.

4.2.1 Descoberta de Serviço

O componente de Descoberta de Serviço é fundamental para o funcionamento do *framework*, pois detecta todos os serviços de rede, como Implantação de Serviço, *Profile* de Rede e Serviço de *Offloading*, que são necessários para as atividades do MpOS API. Este componente possui uma funcionalidade secundária que serve para anunciar a presença de um cliente móvel em uma rede local sem fio, obtendo como resposta o endereço IP de um *cloudlet*, que esteja presente nesta mesma rede sem fio. Sendo assim, o *framework* é capaz de realizar o processo de descoberta de serviço em até dois *endpoints* diferentes, de acordo com a configuração da aplicação móvel, conforme definido no *MposConfig*. A Figura 4.4 apresenta um diagrama de classe em UML que representa a estrutura do componente de Descoberta de Serviço.

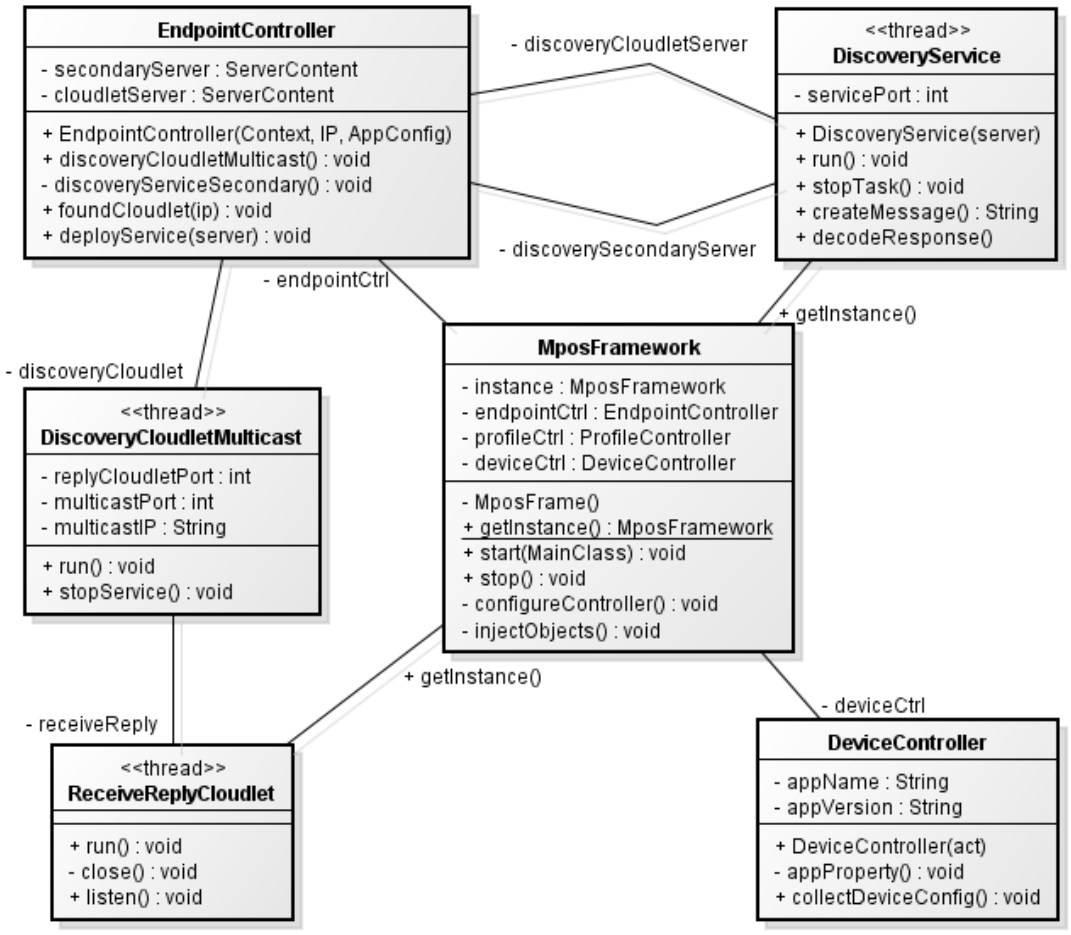

Figura 4.4. Diagrama de classe da Descoberta de Serviço

Para descrever esta arquitetura de Descoberta de Serviço (Figura 4.4) deve-se considerar que existe uma demanda desse serviço em relação a dois *endpoints* diferentes, sendo um localizado dinamicamente na rede local Wi-Fi e outro definido estaticamente pelo desenvolvedor da aplicação. Assim, durante a inicialização do controlador *EndpointController*, os métodos

discoveryCloudletMulticast e *discoveryServiceSecondary* serão invocados assincronamente e explicados nesta sequencia. A Figura 4.5 apresenta os passos seguidos pelo método *discoveryCloudletMulticast*, que dispara uma *thread* do tipo *DiscoveryCloudletMulticast*. Esta *thread* fica em funcionamento até localizar a presença de um *cloudlet server* na rede local Wi-Fi. O sistema também dispara outra *thread* do tipo *ReceiveReplyCloudlet*, responsável por realizar o procedimento de escuta em relação à resposta que é emitida pelo *cloudlet server*.

O sistema emite uma requisição de UDP Multicast a cada 0.5 segundo, sendo realizada no máximo de 60 dessas requisições. O motivo desses limites é para permitir que o servidor possa captar o anúncio do cliente na rede local sem fio. Nesse caso, o servidor pode demorar entre 1 até no máximo 30 segundos, para detectar anúncio desse cliente na rede local. O sistema também utiliza um IP (classe D) e uma porta fixa, comum ao cliente e ao servidor remoto.

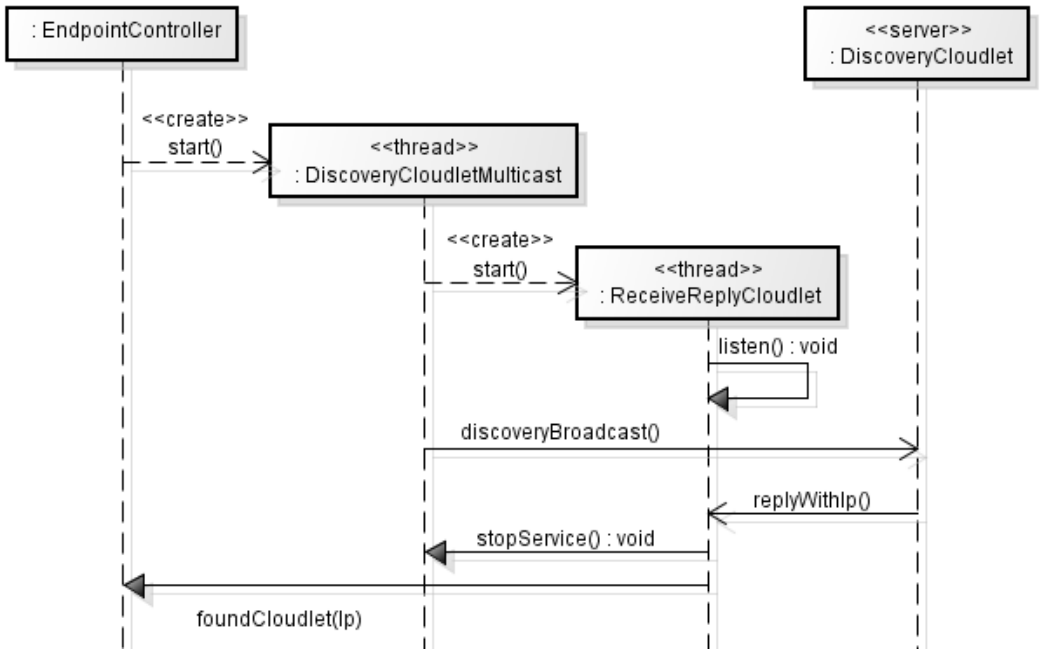

Figura 4.5. Processo de descoberta de *cloudlet* em rede local

Caso haja na rede local Wi-Fi a presença de um *cloudlet server* com o sistema do MpOS Plataforma instalado, o *doudlet* responderá ao cliente móvel usando outra porta fixa, por meio do protocolo UDP Unicast. O servidor envia como mensagem para o cliente o valor do IP da sua interface de rede. O cliente recebe esta mensagem e finaliza a execução da *thread DiscoveryCloudletMulticast*, por meio do *stopService*. Por fim, o sistema notifica o IP recebido com *EndpointController* usando o método *foundCloudlet*, finalizando assim, a execução do *ReceiveReplyCloudlet*.

Caso a classe *ReceiveReplyCloudlet* não consiga dentro do *timeout* de 35 segundos, receber alguma notificação de Unicast UDP, esta *thread* será desligada e a classe *DiscoveryCloudletMultiast*

vai esperar 10 segundos para reiniciar a sua execução, até localizar a presença de um *cloudlet* ou ser desligado na finalização do aplicativo móvel. A escolha destes e de outros valores de tempo, como *timeout* e reinicio dos sistemas, foram inspirados a partir de outros protocolos de configuração dinâmica, como DHCP [Droms 1997].

Após ter descoberto um *cloudlet* na rede e o desenvolvedor ter definido também um *endpoint* secundário, será executados (ver Figura 4.4) os seguintes métodos: (i) *foundCloudlet* e (ii) *discoveryServiceSecondary*. Ambos os métodos são disparados a partir de *threads* do tipo *DiscoveryService* (Figura 4.6), no qual cada *thread* fica em execução a procura dos serviços de rede em um determinado servidor remoto.

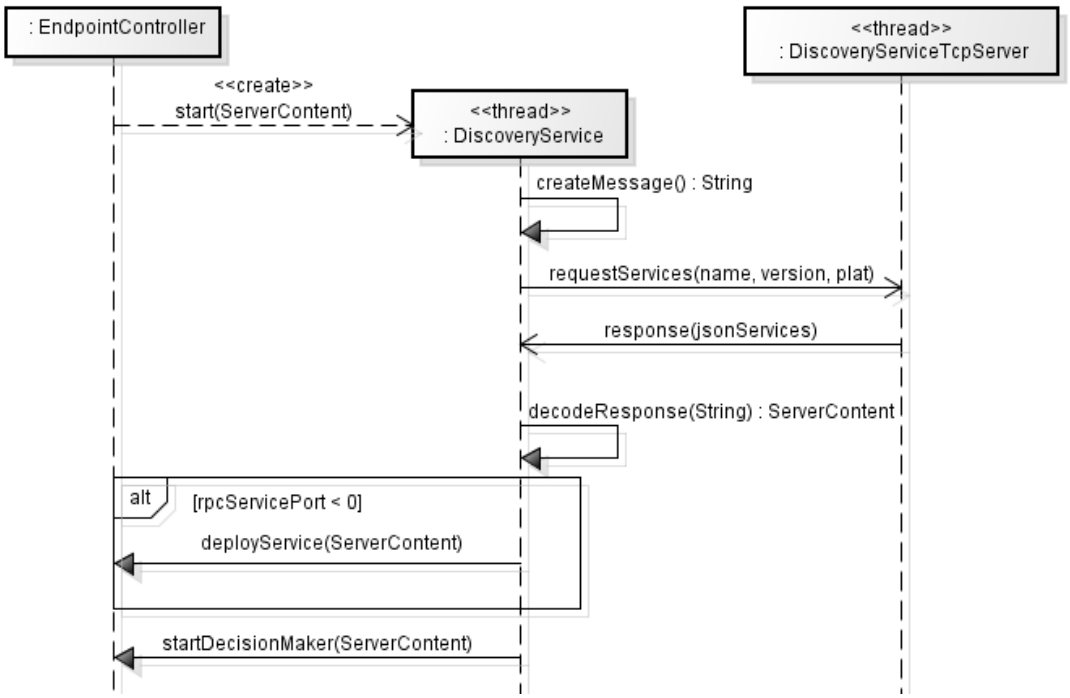

Figura 4.6. Processo de descobertas de serviços para um determinado servidor remoto

O sistema de descoberta de serviço durante sua inicialização, cria uma mensagem de requisição, baseada no nome, versão do aplicativo móvel e da plataforma móvel em execução, que foram obtidos no controlador *DeviceController*. Após isso, o cliente se conecta ao servidor remoto usando o protocolo TCP, por meio de uma porta fixa. Durante a primeira interação com servidor remoto, este serviço de rede envia uma mensagem "perguntando" sobre os serviços de rede que estão disponíveis neste servidor.

No lado servidor essa mensagem é processada e a requisição do cliente é respondida por meio de uma mensagem no formato *json*[16], que descreve todas as portas dos serviços de rede que estão em execução no servidor. Caso não exista um Serviço de *Offloading* em execução será

[16] http://json.org/

atribuído na mensagem de resposta do servidor um valor negativo para esta porta, conforme apresentado na Figura 4.7.

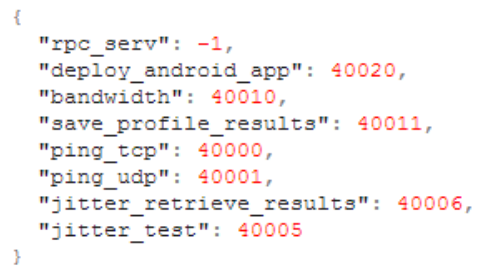

Figura 4.7. Exemplo de resposta do servidor em *json*

Quando o cliente receber a resposta do servidor, a mensagem será decodificada do formato *json*, para um objeto do tipo *ServerContent*. Caso seja detectado um valor negativo para a porta do Serviço de *Offloading*, o *EndpointController* será notificado através do método *deployService* e receberá por parâmetro o objeto *ServerContent* que foi criado anteriormente. Com isso, o procedimento *deployService* tem o objetivo de iniciar assincronamente o sistema de Implantação de Serviço.

Na situação em que a mensagem de resposta do servidor não chega antes do *timeout* de 10 segundos, neste caso a classe de *DiscoveryService* vai esperar outros 20 segundos para repetir de novo o procedimento que foi explicado anteriormente (Figura 4.6). Este processo de repetição deve acontecer até o sistema de Descoberta de Serviço, conseguir detectar os serviços de rede em relação ao *endpoint* destino ou ser desligado na finalização do aplicativo móvel.

No final da execução do *DiscoveryService*, o sistema de Sistema de Decisão Dinâmica (SDD) é invocado a partir do *EndpointController*, por meio do método *startDecisionMaker*, passando como parâmetro o mesmo *ServerContent* que foi definido durante a execução do componente de Descoberta de Serviço. O SDD funciona em conjunto com o *Profile* de Rede e será detalhado o seu funcionamento na seção referente ao Serviço de *Offloading*.

4.2.2 Implantação de Serviço

O funcionamento desse processo de Implantação de Serviço consiste no envio do binário do aplicativo móvel e das suas dependências que serão necessárias para realizar a operação de *offloading* em um determinado servidor remoto. Como mencionado na Seção 4.2.1, o componente de Implantação de Serviço ocorre posteriormente a Descoberta de Serviço, através do método assíncrono *deployService* no *EndpointController*. A Figura 4.8 apresenta o componente de Implantação de Serviço, sendo representada em um diagrama de classe UML.

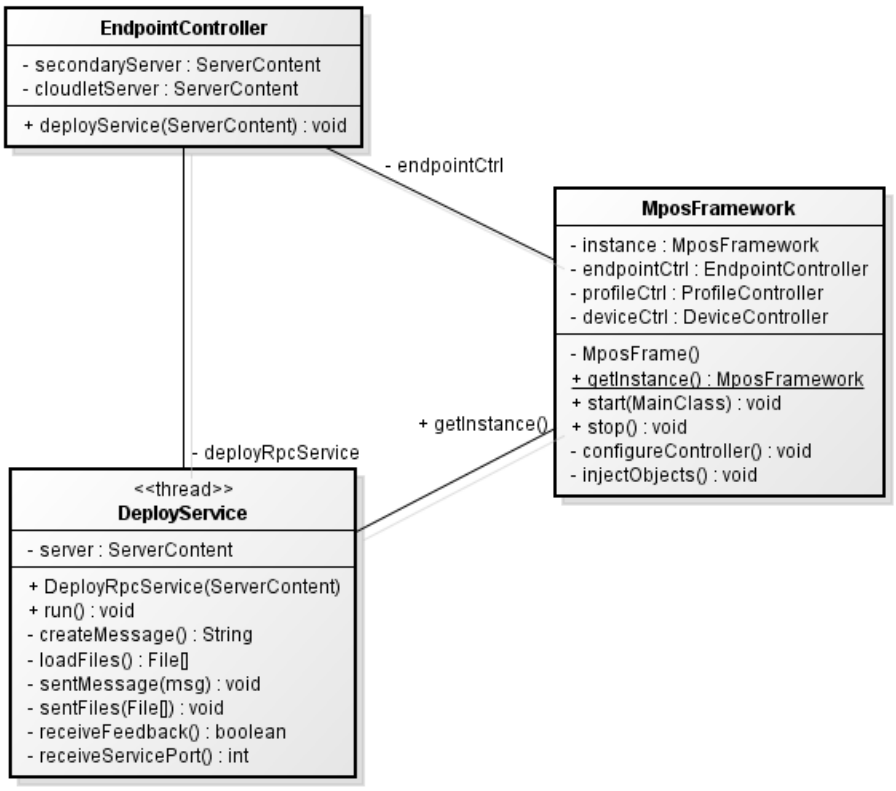

Figura 4.8. Diagrama de classe da Implantação de Serviço

O procedimento *deployService* dispara uma *thread* do tipo *DeployService* e durante sua inicialização recebe o objeto do tipo *ServerContent* (ver Figura 4.8), criando uma mensagem através do método *createMessage*, que descreve o aplicativo móvel que será implantado no servidor remoto. A mensagem criada é composta com nome, versão e plataforma móvel que executa o aplicativo móvel.

Neste serviço o cliente conecta-se com o servidor remoto, por meio do protocolo TCP enviando na primeira interação de rede, a mensagem que foi criada anteriormente. Para continuar a execução do serviço, o cliente espera por uma mensagem de confirmação do servidor através do método *receiveFeedback*. Depois de confirmado o recebimento, os arquivos relacionados com as dependência da aplicação são carregados por meio do procedimento *loadFiles*. Em seguida, outra mensagem informa a quantidade de arquivos que serão enviados do cliente para o servidor remoto. O cliente procede a execução da *thread*, após receber a mensagem de confirmação do servidor.

Na sequencia o cliente envia para o servidor remoto, cada arquivo que estiver na pasta de dependência, seguindo sempre o mesmo "protocolo" de enviar primeiro o nome do arquivo, seguido pelo o tamanho do arquivo e por último o seu conteúdo; através do *Socket* no método *sentFiles*.

O servidor quando receber estas três informações do "protocolo", enviará para o cliente uma mensagem de confirmação, para que este continue enviando o próximo "protocolo", até terminar de enviar todos os arquivos para o servidor remoto.

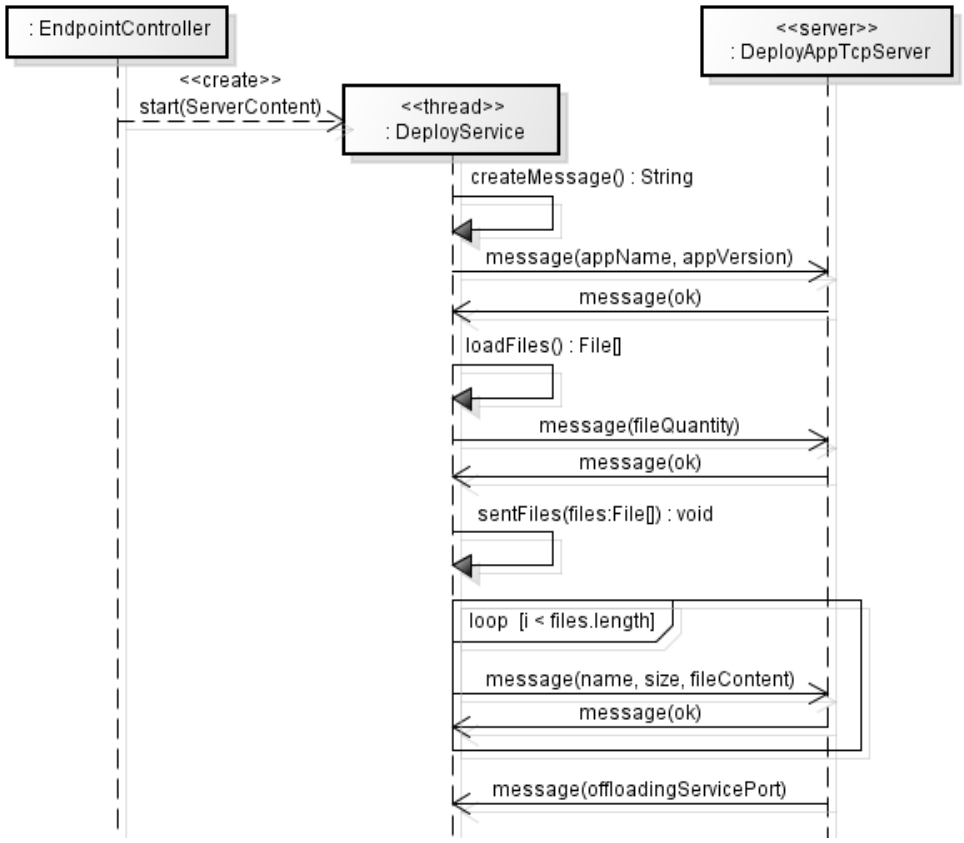

Figura 4.9. Processo de Implantação do Serviço

No final da execução da *thread DeployService* (Figura 4.9), o cliente aguarda receber do servidor remoto, o número da porta onde o serviço de *offloading* está sendo executado, atualizando assim, o objeto *ServerContent*. Enquanto isso, no servidor remoto o serviço também passa a ser disponibilizado na Descoberta de Serviço, para que outros clientes que utilizem o mesmo aplicativo móvel faça uso desse serviço recém-implantado.

4.2.3 *Profile* de Rede

Esse componente tem objetivo realizar o *profiling* da rede, isto é, avaliar a qualidade da rede entre o cliente móvel e um servidor remoto destino, por meio de diversas métricas de rede. O *Profile* de Rede foi desenvolvido com base nas ideias e metodologia de [Huang *et al.* 2012] e nos cálculos de [Demichelis and Chimento 2002]. As portas dos serviços de rede que compõe esse sistema de *profiling* podem ser definidas manualmente ou automaticamente por meio do sistema de Descoberta de Serviço. A Figura 4.10 apresenta o componente de *Profile* de Rede, através de um diagrama de classe UML.

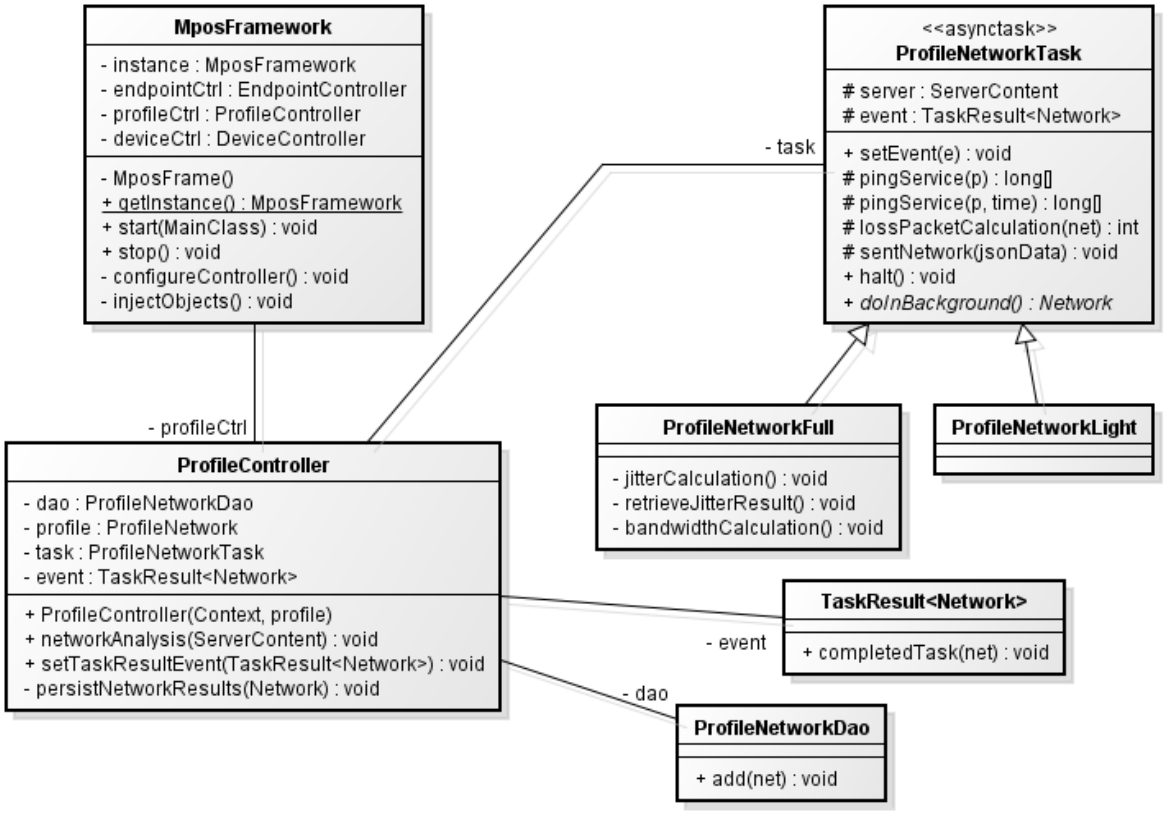

Figura 4.10. Diagrama de classe do *Profile* de Rede

Como explicado anteriormente na Seção 4.2, o controlador do *ProfileController* é instanciado durante o processo de inicialização do *MposFramework* e recebe por parâmetro o tipo do *profiling* de rede que foi definido no *MposConfig*. Assim de acordo com a Figura 4.10, para executar a operação de *profile* de rede, os seguintes procedimentos serão necessários.

Primeiro deve-se registrar um evento (*TaskResult*) para receber o resultado da avaliação de rede, pois a execução do *Profile* de Rede acontece de forma assíncrona. Após isso, o procedimento (*networkAnalysis*) deve ser chamado para realizar a análise da rede, de acordo com tipo de *profiling* definido. No MpOS API existem dois tipos de *profiling*. (i) *light* e (ii) *full*.

O processo de *profiling light* é considerado um teste simples e rápido. Porque, demorar em média oito[17] segundos para avaliar a qualidade de uma determinada rede, sendo representada pela classe *ProfileNetworkLight* na Figura 4.10. Nesse tipo de *profiling* é realizado sete medições do RTT (*pingService*), para cada um dos protocolos TCP e UDP, além de contabilizar quantos pacotes foram perdidos durante esta medição dos *pings*, seguindo o seguinte critério. Para cada valor de RTT que tiver um valor igual a 0ms ou acima de 1500ms, será classificado como pacote perdido, durante a execução dessa avaliação.

[17] Depende da carga do meio de transmissão, essa estimativa foi baseada na observação da execução do profiling de rede, em uma rede WiFi.

Enquanto, no *profiling full* é definido pela classe *ProfileNetworkFull* (Figura 4.10), que demora por volta de 40 segundos para terminar todo o processo de avaliação da rede, em relação ao servidor remoto. Este *profile* possui as mesmas funcionalidades do *profiling light*, além de fazer a operação de *jitter* no protocolo UDP usando o método *jitterCalculation*. Este nível de *profile*, realiza a medição da largura de banda disponível, através do método *bandwidthCalculation* usando o protocolo TCP para realizar esta tarefa.

Para os dois tipos de *profile*, os resultados produzidos no final de cada execução são encapsulados no objeto do tipo *Network*. No final da execução, o componente envia para o servidor remoto estas informações geradas pelo *profiling* de rede, além de outras informações relativas ao aparelho do cliente. Por fim, o controlador (*ProfileController*) também persiste este objeto *Network* localmente no banco de dados do aparelho móvel.

4.2.4 Sistema de *Offloading*

Este sistema de *offloading* atua no nível do aplicativo móvel, por meio da marcação *Remotable* empregada para definir os métodos das interfaces que devem ou não realizar a operação de *offloading*. A Figura 4.11 apresenta o componente do Sistema de *Offloading* através de um diagrama de classe em UML. A classe *ProxyHandler* (do tipo *InvocationHandler*) tem a responsabilidade de interceptar os métodos que são invocados a partir de um *proxy* do objeto, por meio do procedimento *invoke* da classe *ProxyHandler*. Essa classe possui um método estático, denominado de *newInstance*, sendo chamado pela classe *MposFramework* durante o processo de injeção de dependências. Este método serve para criar dinamicamente os *proxy* de objeto, a partir de uma interface e tipo concreto que é passado por parâmetro nesse método *newInstance*.

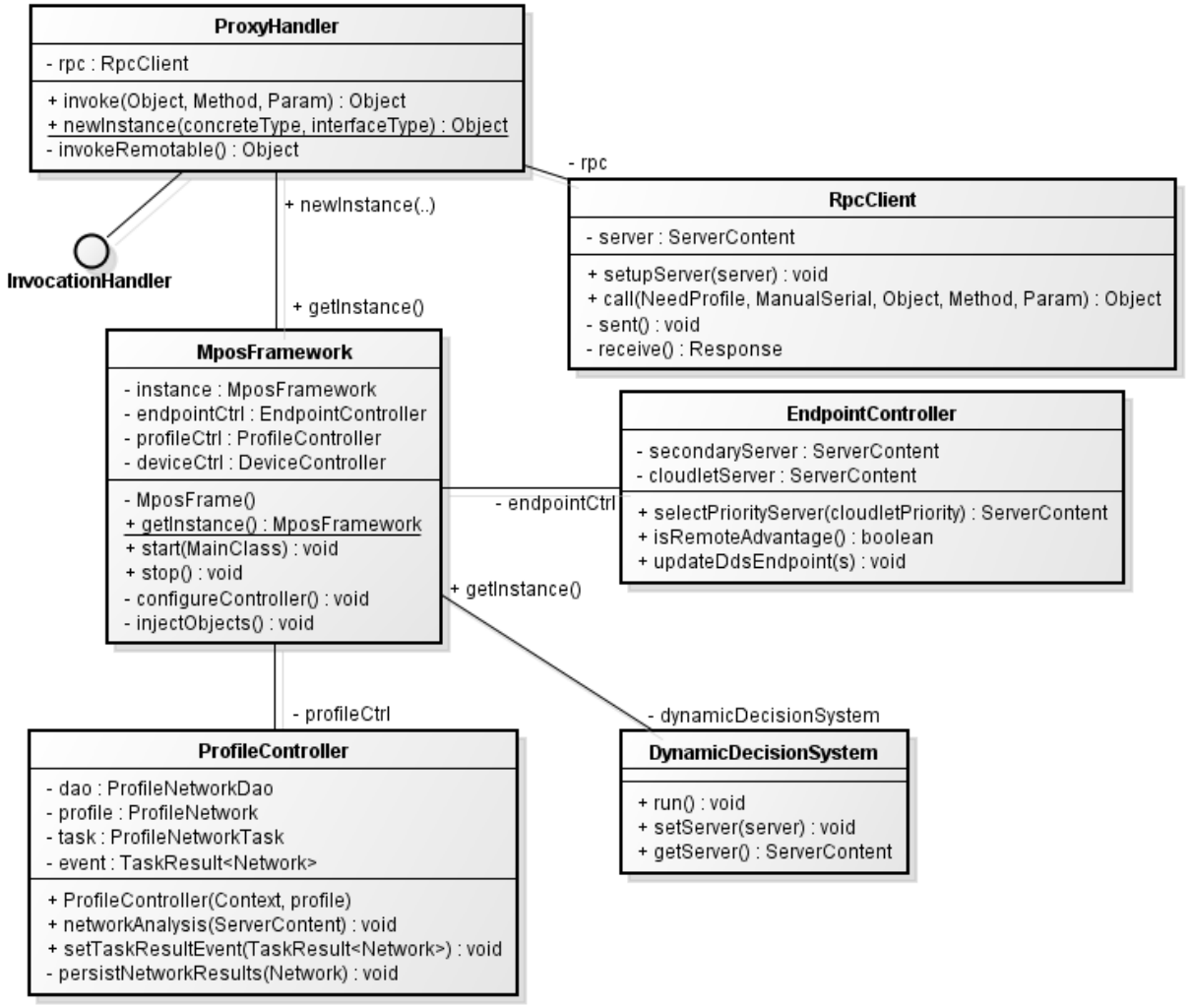

Figura 4.11. Diagrama de classes do Sistema de *Offloading*

Após instanciar um *proxy* do objeto, o procedimento *invoke* da classe *ProxyHandler* (Figura 4.12), intercepta todas as chamadas de métodos que são invocados do *proxy* de um objeto. O procedimento *invoke* possui os seguintes parâmetros: (i) uma instância original do objeto (sem *proxy*); (ii) método interceptado e (iii) seus parâmetros em uma lista de objetos. A partir destas informações, o *ProxyHandler* inspeciona o método interceptado à procura da marcação *Remotable*. Caso não encontre, este método interceptado é invocado para ser executado localmente no dispositivo móvel, utilizando reflexão computacional [Forman *et al.* 2004].

Porém, caso seja encontrada a marcação *Remotable*, as propriedades da marcação vão especificar o comportamento do procedimento *invoke* em relação ao processo de *offloading*. A primeira propriedade inspecionada determina a prioridade onde será executado o processo de *offloading*. O valor dessa prioridade é transmitido para o método *selectPriorityServer* do *EndpointController*, que retorna como resultado um *ServerContent* representando o *endpoint* disponível.

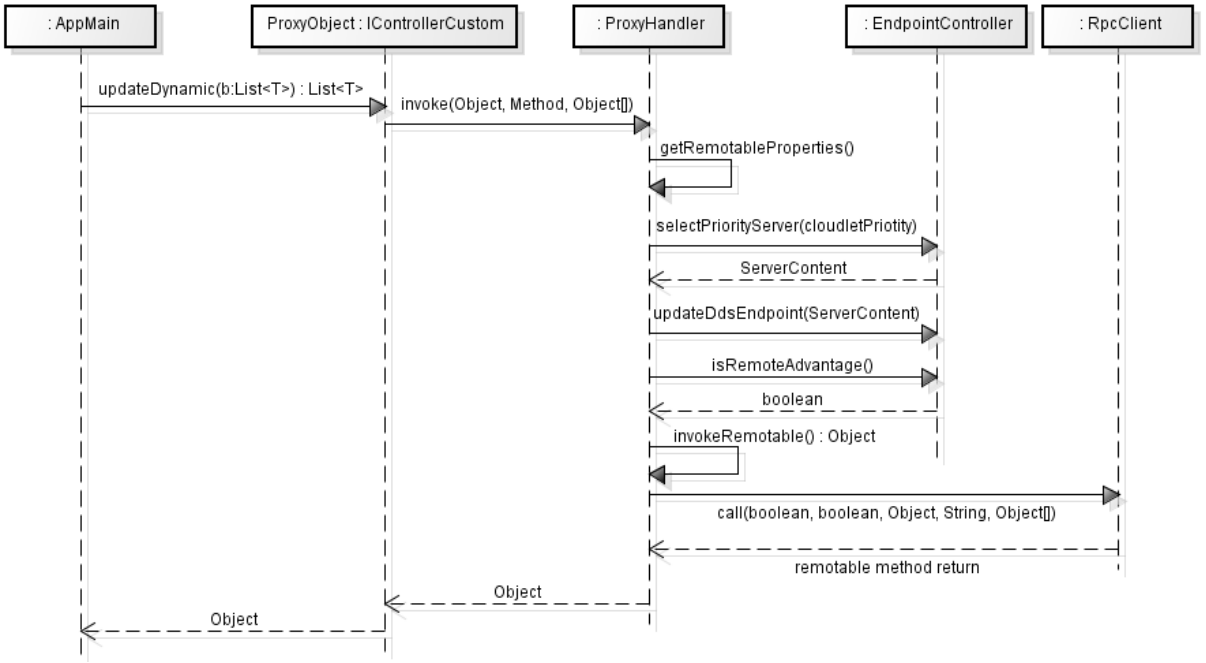

Figura 4.12. Processo de decisão para realizar uma operação de *offloading*

A segunda propriedade da marcação define a forma do particionamento da técnica do *offloading*, sendo sempre realizado (regra do particionamento estático) ou dependendo das condições da rede (regra do particionamento dinâmico). Caso essa propriedade selecione o particionamento do tipo dinâmico, o procedimento *invoke* (ver Figura 4.12) vai sempre atualizar o *endpoint* do Sistema de Decisão Dinâmica (SDD), por meio do método *updateDdsEndpoint* no *EndpointController*.

Em seguida, o procedimento *invoke* vai verificar se o momento atual é vantajoso, para realizar ou não a operação de *offloading* usando o método *isRemoteAdvantage*. Se não for vantajoso, o método que foi passado por parâmetro no procedimento *invoke* é chamado para ser executado localmente no dispositivo móvel. Caso contrário, o método interceptado é chamado para ser executado remotamente no *endpoint* selecionado, por meio do método *invokeRemotable* da classe *ProxyHandler*.

Este procedimento (Figura 4.13) interage com uma classe de RPC cliente (*RpcClient*), desenvolvida especialmente para tratar da técnica de *offloading*, em métodos que foram interceptados e selecionados pela classe *ProxyHandler*. O procedimento configura inicialmente o servidor remoto no *RpcClient*, por meio do método *setupServer* e realiza a operação de *offloading* usando uma chamada remota de procedimento (RPC) síncrona, através do procedimento *call* do *RpcClient*.

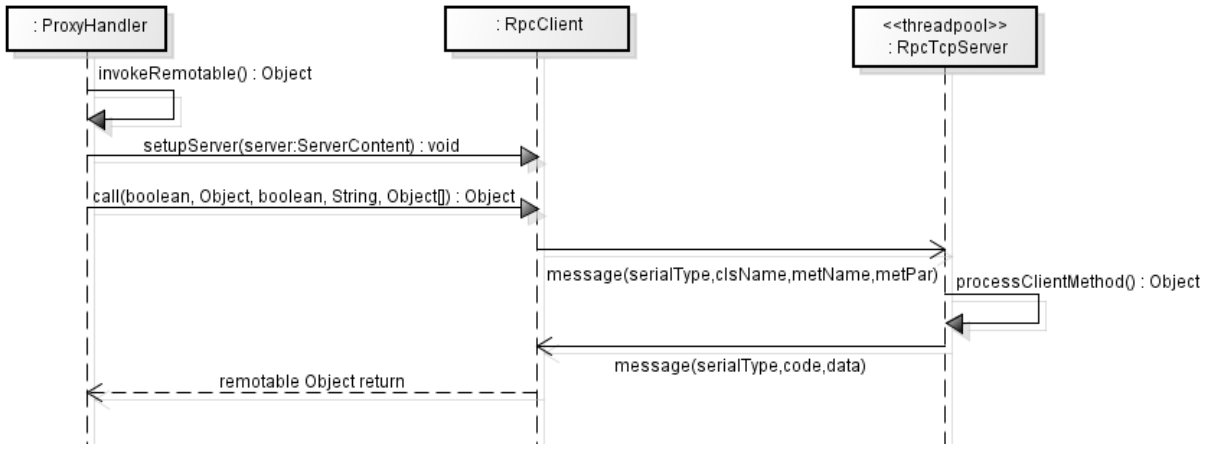

Figura 4.13. Processo de realização da operação de *offloading*

O procedimento *call* (Figura 4.13) conecta ao servidor remoto, por meio do protocolo TCP enviando inicialmente o seguinte fluxo de *bytes* pelo *Socket*. O tipo de serialização usada na transmissão e se o servidor deve fazer ou não, o *profiling* da chamada de RPC. Em seguida o "nome completo" da classe, que inclui também o nome do pacote (Java) ou do *namespace* (C#), além do nome do método interceptado. Por fim, o sistema envia os valores dos parâmetros do método, para ser usado durante a invocação remota do método no servidor remoto. Em seguida, após realizar a chamada remota, o cliente espera no método *receive* pelo resultado do método remoto.

Essa resposta segue também o seguinte fluxo de *bytes*. O primeiro *byte* indica o tipo de serialização usada no retorno da chamada remota, junto com o código que corresponde ao tipo da mensagem se é de retorno ou erro. O restante do *Socket* é destinado para montar o valor de retorno do método remoto ou construir uma mensagem de erro.

Caso o servidor remoto envie uma mensagem de erro, no lado cliente é disparado uma exceção dentro do MpOS API para avisar o desenvolvedor sobre o erro ocorrido no servidor remoto. O motivo desse erro pode ser gerado por qualquer evento, por exemplo, o desenvolvedor esqueceu-se de atualizar as dependências no lado cliente, invocando assim, um método que não existe no servidor remoto. Caso do servidor remoto retorne o valor da chamada remota, esse resultado é montado e retornado pelo procedimento *call* no *RpcClient* até o procedimento *invoke*, que retorna para o aplicativo móvel o resultado do método que foi interceptado pelo *ProxyHandler* anteriormente.

É bom destacar que o processo de serialização que foi descrito anteriormente é diferente nas duas plataformas móveis. No caso da plataforma Android, o desenvolvedor pode serializar automaticamente os parâmetros de um método interceptado, caso seus tipos suportem a interface *Serializable*. Caso este desenvolvedor deseje ter mais desempenho no processo de serialização, a

interface que possui os métodos de *offloading* pode estender da interface *RpcSerializable*, conforme apresentado na Figura 4.14.

```
public interface RpcSerializable {

    public void writeMethodParams(DataOutput out, String methodName, Object params[]);

    public Object[] readMethodParams(DataInput in, String methodName);

    public void writeMethodReturn(DataOutput out, String methodName, Object returnParam);

    public Object readMethodReturn(DataInput in, String methodName);
}
```

Figura 4.14. Representa a interface *RpcSerializable*

Com essa interface (Figura 4.14), o desenvolvedor pode descrever manualmente o processo de serialização e deserialização em relação aos parâmetros e ao retorno dos métodos que estão envolvidos no procedimento de *offloading*. Assim, quando um objeto original implementar essa interface, o *RpcClient* invoca o método *writeMethodParams* para serializar os parâmetros do método interceptado. No lado servidor, o método *readMethodParams* é usado para deserializar tais parâmetros. Quando o servidor envia o resultado do retorno do método remoto, o procedimento *writeMethodReturn* é invocado no servidor para serializar esse retorno, enquanto no lado cliente, o método *readMethodReturn* é usado para deserializar o retorno do método remoto.

Na plataforma do Windows Phone também existe a mesma interface de *RpcSerializable*, com a mesma funcionalidade descrita anteriormente. O que difere da plataforma do Android é o processo de serialização automática, que no Windows Phone utiliza uma biblioteca baseada na ideia do BSON[18] (Binary-JSON). As versões utilizadas neste trabalho do Windows Phone e do .NET Framework não possuem uma funcionalidade em comum de serialização nestes dois ambientes, equivalente com que ocorre na plataforma do Android e do Java SE, através da interface *Serializable*. No entanto, essa implementação baseada no BSON possui algumas limitações como não ter suporte ao tipo *Dictionary*, dentre outras limitações relacionadas a outros tipos da API do Windows Phone.

A vantagem do procedimento de serialização manual (ou de baixo nível) é a redução do tamanho dos dados enviados para o servidor remoto, além de um menor *overhead* em relação ao processamento envolvido neste processo de serialização. A principal desvantagem desta técnica é a dificuldade de depurar erros em caso de deslize do programador durante o desenvolvimento do seu protocolo, podendo travar o procedimento de *offloading* ou disparar alguma exceção na execução do procedimento.

[18] http://bsonspec.org/

Por fim, o Sistema de Decisão Dinâmica (SDD) é um subsistema do Sistema de *Offloading*, que inicia depois da Descoberta de Serviço, conforme foi explicado na Seção 4.2.1. O SDD executa em paralelo com todo o resto da aplicação móvel e utiliza do *Profile* de Rede para avaliar a cada 45 segundos, se as condições da rede é favorável para realizar uma operação de *offloading*, atualizando essa decisão no *EndpointController* através de uma variável *boolean*. A lógica por trás desse sistema de decisão é realizada a partir da média dos valores que são coletados do *ping* após o processo do *profiling* de rede. Se essa média estiver abaixo do limiar de 80 ms, então é favorável realizar a operação de *offloding*. Caso contrário, não se deve fazer o *offloading*. Os trabalhos relacionados não revelam também os limites utilizados para tomada de decisão, então com base nas observações dos diversos resultados de *ping* que foram produzidos em uma rede local Wi-Fi, o limiar de 80 ms foi definido para o *framework* MpOS, podendo ser customizada pelo desenvolvedor de acordo com suas necessidades.

4.3 MpOS Plataforma

O MpOS Plataforma é um serviço de sistema, que executa diversos serviços fixos de rede, como Descoberta de Serviço, Implantação de Serviço, *Profile* de Rede e Serviço de *Offloading*, e tem como objetivo servir todas as requisições, que são feitas a partir do cliente MpOS API. Os serviços de *Profile* de Rede e Descoberta de Serviço são classificados como serviços portáveis, pois estes serviços possuem uma única implementação no servidor usando a tecnologia Java, que servem qualquer tipo de plataforma móvel. Os outros serviços de rede como a Implantação de Serviço e Serviço de *Offloading*, não são portáveis, pois cada serviço possuem implementações especificas para cada plataforma móvel, sendo assim, desenvolvidos usando Java e C# respectivamente.

O serviço do MpOS Plataforma é composto por dois subsistemas. O primeiro subsistema considerado principal, pois foi desenvolvido em cima da tecnologia do Java SE, e possui todas as implementações dos serviços de rede que foram citados anteriormente. O segundo subsistema desenvolvido com .NET Framework (C#) possui apenas as implementações do serviço de Serviço de *Offloading* e Implantação de Serviço.

Esses subsistemas compartilham dois recursos externos, sendo o primeiro um arquivo de configuração do MpOS Plataforma, denominado de *config.properties*, enquanto o outro recurso *mpos.data* é relacionado ao banco de dados em SQLite. Antes de inicializar o sistema do MpOS Plataforma é necessário que o desenvolvedor forneça no arquivo de configuração, o IP do servidor remoto para que o sistema possa associar as portas dos serviços de rede, com uma das interfaces de rede na máquina local. O desenvolvedor de acordo com sua necessidade, também

pode customizar no arquivo de configuração, o nome e a porta dos diversos serviços fixos de rede.

A Figura 4.15 ilustra a arquitetura do MpOS Plataforma como um diagrama de classe usando a notação UML. O processo de inicialização do MpOS Plataforma acontece na classe *MposMain* a partir do subsistema principal, sendo também invocado neste momento em paralelo o executável do subsistema secundário. Em ambos os subsistemas existe uma classe denominada de *ServiceController*, que utiliza do padrão de projeto *Singleton* e começa as suas atividades no procedimento *start*.

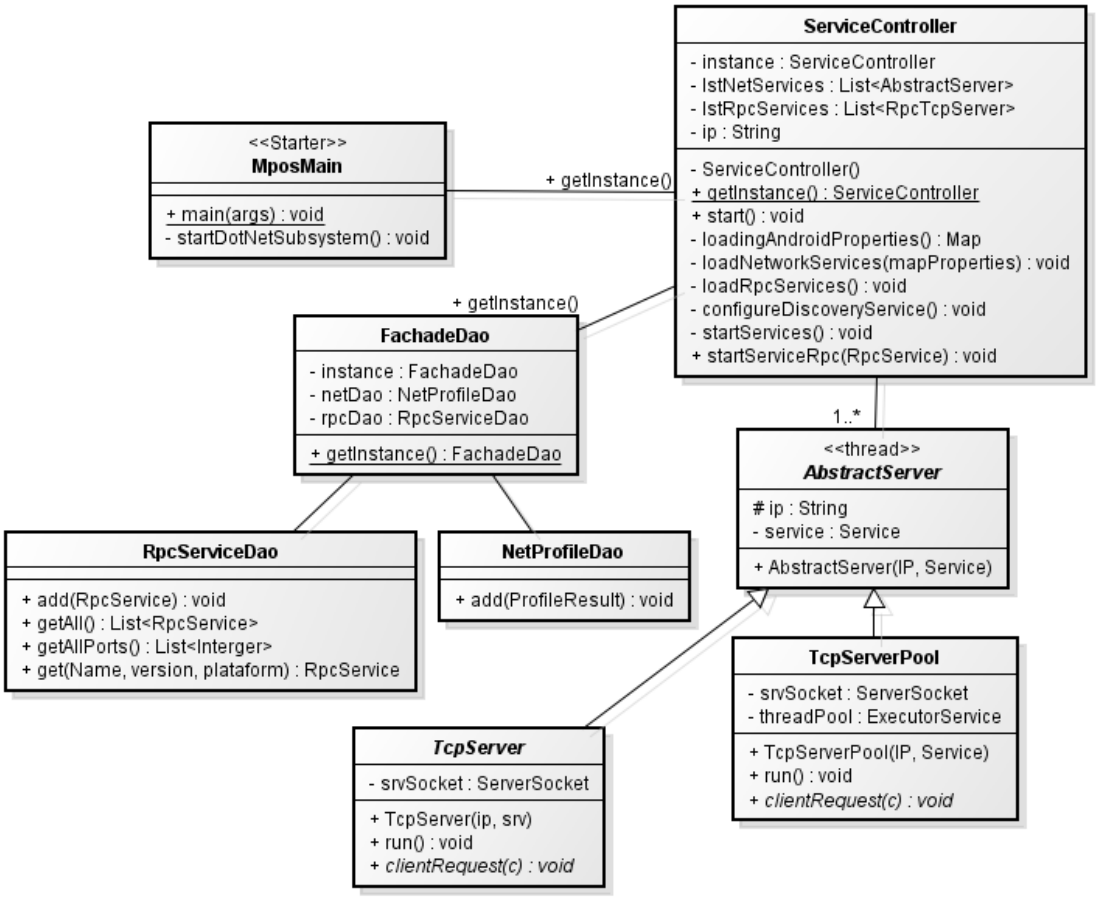

Figura 4.15. Diagrama de Classe da inicialização do MpOS Plataforma

O método *start* (Figura 4.16) carrega inicialmente todas as configurações definidas no arquivo de configuração, para chamar os seguintes métodos na sequência: (i) *loadNetworkServices*; (ii) *loadRpcServices*; (iii) *configureDiscoveryService* e (iv) *startServices*. No entanto, no subsistema secundário, esse método *start* vai lançar apenas a *thread* referente à Implantação de Serviço e os métodos de *loadRpcServices* e *startServices* que serão chamados.

O primeiro método *loadingNetworkServices* (Figura 4.16) cria uma lista denominada de *lstNetServices*, e instancia todos os serviços "fixos" de rede que foram definidos no arquivo de configuração. O próximo procedimento referente ao *loadRpcServices* consulta no banco de dados local, sobre a existência de algum Serviço de *Offloading* que tenha sido implantado no servidor previamente. Caso existam esses serviços, os objetos do tipo *RpcTcpServer* serão criados passando os seguintes parâmetros de IP e *RpcService*, e armazenados em uma lista denominada de *lstRpcServices*.

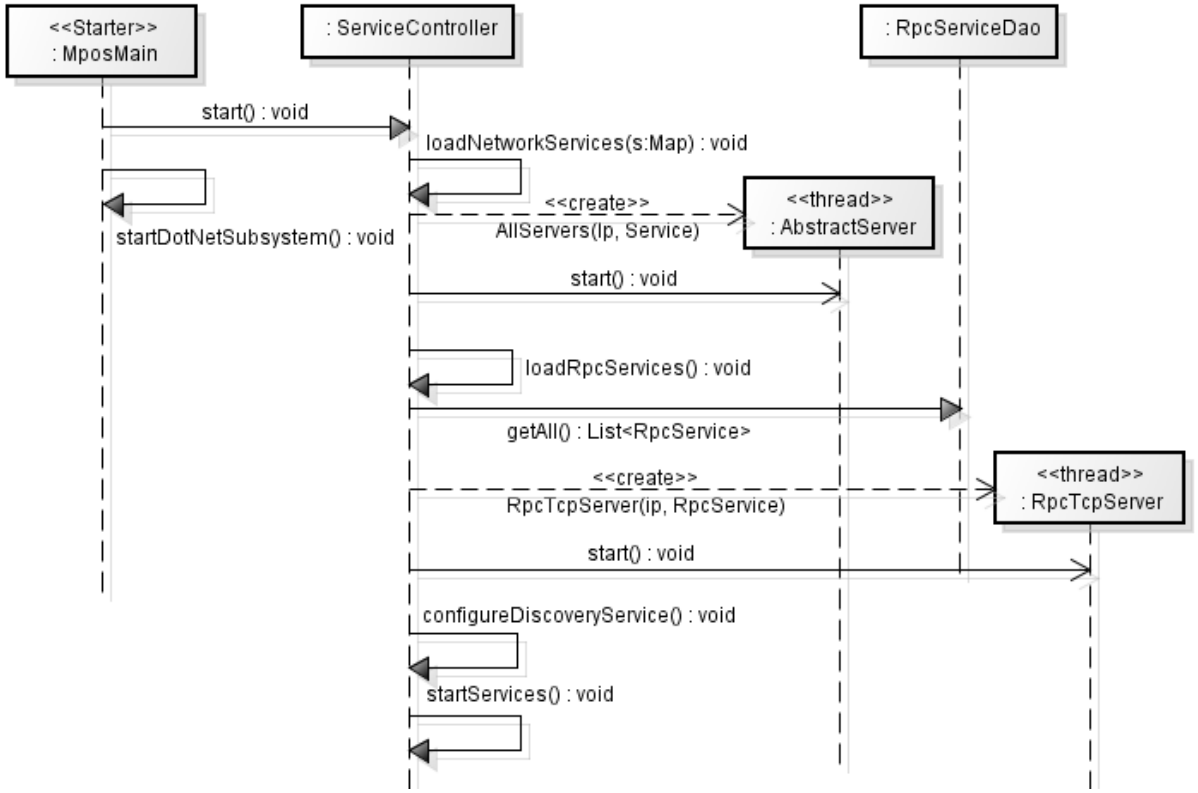

Figura 4.16. Processo de inicialização do MpOS Plataforma

O método *configureDiscoveryService*, instância o servidor de Descoberta de Serviço e constroem com base nos serviços fixos de rede, as mensagens de respostas no formato *json*. Por último, o procedimento *startServices* percorre as duas listas com serviços de redes instanciados, inicializando cada serviço, por meio de uma *thread*.

O MpOS Plataforma possui três tipos de implementações de servidores. A primeira implementação é dos servidores que utilizam o protocolo UDP, como *multicast*, RTT e *jitter*. Esse tipo de servidor executa em uma mesma *thread* não disparando assim, novas *threads* para tratar das requisições dos clientes, ou seja, estes servidores são implementados no formato *single-thread*.

O outro tipo de implementação do servidor é relacionado com protocolo TCP, representando também a maioria dos serviços de redes do componente MpOS Plataforma, como

RTT, Descoberta de Serviço, Implantação de Serviço, dentre outros. Essa implementação de servidor, diferente no que ocorre no servidor UDP, trabalha no formato *multithread*, isto é, atende paralelamente os múltiplos clientes, disparando sempre uma nova *thread* para cada requisição solicitada por eles. No *framework* existe uma classe abstrata denominada de *TcpServer* (Figura 4.15), que auxilia no desenvolvimento de novos serviços de rede, nos quais utilizam o protocolo TCP. Isso auxilia o desenvolvedor a se preocupar apenas com a lógica de negócio do serviço de rede, não precisando gerenciar o disparo de novas threads e dentre outras atividades.

A última implementação de servidor utiliza também o protocolo TCP, porém se diferencia da anterior, por atender as requisições dos clientes, por meio do padrão *thread pool*[19]. Esse padrão tenta reusar as *threads* do sistema, para minimizar o *overhead* envolvido na criação e destruição destas *thread*s. O único serviço de rede que usa dessa implementação é o Serviço de O*ffloading* visando principalmente melhorar o tempo de resposta das chamadas remotas, quando ocorrer múltiplas requisições dos clientes.

As próximas seções serão detalhadas o funcionamento dos componentes de Descoberta de Serviço, Implantação de Serviço, *Profile* de Rede e Serviço de *Offloading*, no MpOS Plataforma.

4.3.1 Descoberta de Serviço

Esse serviço de rede possui dois tipos de implementações de servidores, uma pertencente ao protocolo UDP Multicast, enquanto a outra é relacionada com protocolo TCP. O servidor *DiscoveryMultiastService* (Figura 4.17) tem o objetivo de detectar em uma rede local sem fio, a presença de um cliente que anuncia sua presença na rede, usando o protocolo UDP Multicast. O servidor, após detectar esse anúncio, deve responder ao cliente enviando o IP definido no arquivo de configuração, através do protocolo UDP Unicast.

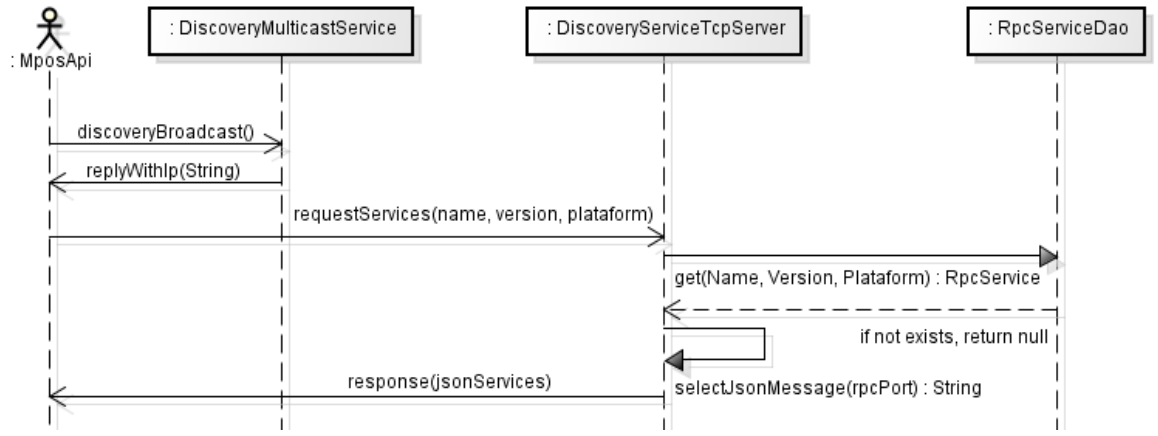

Figura 4.17. Processo de Descoberta de Serviço no servidor

[19] http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html

O servidor *DiscoveryServiceTcpServer* (Figura 4.17) tem o propósito de processar as requisições dos clientes que podem estar, tanto em uma rede local sem fio, quanto na Internet. O servidor recebe desses clientes uma mensagem contendo o nome do aplicativo, versão e tipo da plataforma móvel, que por sua vez, utiliza dessas informações, para buscar no banco de dados local, se existe algum Serviço de *Offloading* em funcionamento.

Durante essa operação, uma mensagem de resposta[20] no formato *json* é enviada, de acordo com a solicitação do cliente móvel. Caso exista um Serviço de *Offloading*, o valor da porta em relação a este serviço é adicionado na mensagem de resposta, senão um valor negativo é atribuído no lugar, para indicar a inexistência do Serviço de *Offloading* no servidor remoto. Por fim, o servidor envia de volta para o cliente esta mensagem de resposta, com todos os serviços de rede definidos e a *thread* de requisição do cliente é finalizada no término da chamada do *clientRequest*.

4.3.2 Implantação de Serviço

Esse serviço no MpOS Plataforma foi desenvolvido em cima da lógica do servidor TCP (Figura 4.15). Durante o funcionamento do processo de Implantação de Serviço (Figura 4.18), o cliente envia na primeira interação de rede, o nome e a versão do aplicativo móvel e logo em seguida, o servidor confirma o recebimento dessa mensagem.

A Implantação de Serviço no servidor remoto cria um diretório e utiliza como nome esses dois atributos que foram passados, através da chamada do método *checkDiretory*. Esse método cria um diretório dentro da pasta *app_dep/android*, se a Implantação de Serviço estiver interagindo com a plataforma Android ou dentro da pasta *app_dep/windowsphone*, no caso Windows Phone. Na sequencia da execução um objeto do tipo *RpcService* é criado utilizando também estes dois atributos (nome e versão do aplicativo) que foram passados.

As próximas interações de rede (Figura 4.18), o servidor recebe primeiro a quantidade de arquivos que serão processados no servidor, em seguida os arquivos são transmitidos um de cada vez para o servidor remoto, sendo salvo cada arquivo através do procedimento *saveFile*. O *path* que representa o arquivo salvo é adicionado também em uma lista no objeto *RpcService*. Após salvar todos os arquivos no servidor remoto, um número de porta é aleatoriamente gerada no método *generateServicePort* e representa um ponto de acesso para um Serviço de *Offloading* que foi recém-implantado neste servidor remoto. O valor dessa porta deve estar entre 36000 e 37000, sendo também única em relação a todos os outros Serviços de *Offloading*, sendo também guardado no objeto *RpcService* criado no inicio desse processo.

[20] A existência de múltiplas mensagens de resposta, por causa da Implantação de Serviço não é um serviço portável.

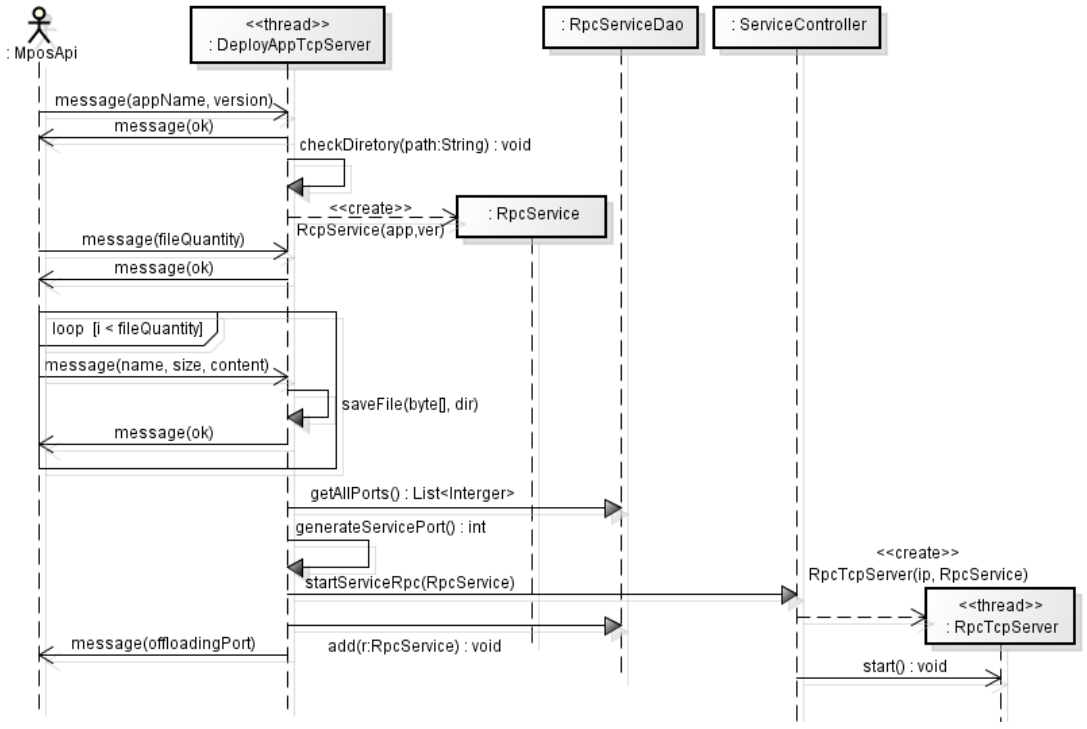

Figura 4.18. Processo de Implantação de Serviço em um servidor remoto

Em seguida, o Serviço de *Offloading* é instanciado, por meio do método *startServiceRpc* (Figura 4.18) do *ServiceController* passando por parâmetro o objeto *RpcService*, que possui todos os dados necessários para inicializar o Serviço de *Offloading*, com base nas dependências do aplicativo móvel. Por causa do funcionamento desse método *startServiceRpc*, que a Implantação de Serviço não é um sistema portável, sendo necessário desenvolver uma versão para cada tipo de plataforma móvel, para permitir assim, invocar o Serviço de *Offloading*.

As informações contidas no objeto *RpcService* também é persistido no banco de dados local, para ser disponibilizado no sistema de Descoberta de Serviço e permitir que outros clientes tenha acesso a esse mesmo serviço que foi recém-implantado no sistema. No final do processo de Implantação de Serviço, o servidor devolve para o cliente o número da porta do Serviço de *Offloading* que foi implantado no servidor remoto.

4.3.3 *Profile* de Rede

O componente de *Profile* de Rede possui seis implementações de servidores disponíveis, em diferentes protocolos de rede, para permitir que o cliente avalie as condições do meio físico da rede, entre ele e o servidor remoto. Cada servidor é responsável por uma atividade de *profiling* de rede. A atividade de RTT ou *ping* é implementada por dois tipos de servidores (ver Figura 4.15), um relacionado com a implementação do protocolo TCP e outro desenvolvido com protocolo UDP.

A atividade de *jitter* também utiliza de dois tipos de servidores, sendo um servidor destinado para realizar o cálculo do *jitter*, sendo desenvolvido em cima do protocolo UDP. Ao passo que o outro servidor é responsável apenas pela entrega dos resultados da operação de *jitter* usando o protocolo TCP.

A atividade para calcular a largura de banda disponível, entre o cliente e o servidor remoto é implementada através do protocolo TCP. Um dado aleatório é gerado em tempo de execução pelo cliente sendo enviado para o servidor remoto. Assim, o servidor remoto contabiliza a quantidade de dados que foram transferidos na sessão e quanto tempo levou para transferir toda essa informação, gerando no final o resultado da largura de banda em relação ao *upload* do cliente. No caso do *download* do cliente, o mesmo procedimento é realizado, porém começando a partir do servidor remoto.

Por fim, o *Profile* de Rede também possui um servidor de rede que é responsável pela persistência dos resultados gerados pelo *Profile* de Rede cliente (MpOS API), além de algumas informações relacionadas com aparelho móvel. Este servidor de persistência de informação também foi desenvolvido em cima do protocolo TCP.

4.3.4 Serviço de *Offloading*

O Serviço de *Offloading* no MpOS Plataforma é conhecido também por ambiente de execução remoto (AER). Esse serviço de rede utiliza a implementação do servidor TCP, com suporte ao padrão de *thread pool*. A motivação para utilizar esse padrão é melhorar o desempenho das múltiplas chamadas remotas que podem ser realizadas por um cliente através do processo de reciclagem das *threads*, ao invés de sempre instanciar uma nova *thread* para cada chamada de RPC no servidor remoto.

Durante a criação do Serviço de *Offloading*, o objeto do tipo *RpcService* é passado por parâmetro e serve para descrever o funcionamento desse serviço. Consequentemente, esse objeto fornece uma lista de dependências em relação a um determinado aplicativo móvel, no qual foi implantado anteriormente no servidor remoto. Ao longo do processo de inicialização do Serviço de *Offloading*, as dependências do aplicativo móvel é carregada também dinamicamente no contexto dessa *thread*, permitindo que através dessas dependências os objetos sejam acessados, durante a execução do RPC feita a partir dos clientes no MpOS API.

Esse procedimento de carregar as dependências dinamicamente é possível, pois as plataformas móveis Android e Windows Phone compilam o código fonte para uma linguagem intermediária (*bytecode*) da plataforma JVM (Android/Java) e .NET Runtime (Windows

Phone/C#), possibilitando através de processos reflexivos realizar o procedimento de RPC. Contudo, é preciso ressaltar que no Android o *bytecode* da JVM ainda é compilado para as instruções da máquina virtual Dalvik, durante o processo de empacotamento do aplicativo móvel [Mednieks *et al.* 2012]. Enquanto no Windows Phone, as DLL que contém o *bytecode* do .NET Runtime que são empacotadas no aplicativo móvel e o compilador JIT compila estas DLL para instruções nativas do processador ARM em tempo de execução [Whitechapel and McKenna, 2013]. Por causa disso, o Serviço de *Offloading* não é portável, sendo também necessário que seja desenvolvido, usando as mesmas tecnologias e linguagens de programação das plataformas móveis, no caso JVM (Android/Java) e .NET Runtime (Windows Phone/C#), para permitir que as dependências do aplicativo móvel possa ser executadas no servidor remoto.

O Serviço de *Offloading* possui duas formas de serialização, uma automática usando o processo *Serializable* (Java) ou BSON (C#), e outra denominada manual, que ocorre por meio da interface *RpcSerializable* (ver Figura 4.14), sendo esta comum nas duas implementações. Durante o procedimento de RPC (Figura 4.19), o cliente envia para o AER as seguintes informações: (i) o tipo de serialização e deserialização usado na transmissão, podendo também ativar ou não, o modo de *profiling* (ou estatísticas do RPC) das chamadas remotas; (ii) "nome completo" da classe; (iii) nome do método remoto e (iv) o valor dos parâmetros desse método remoto.

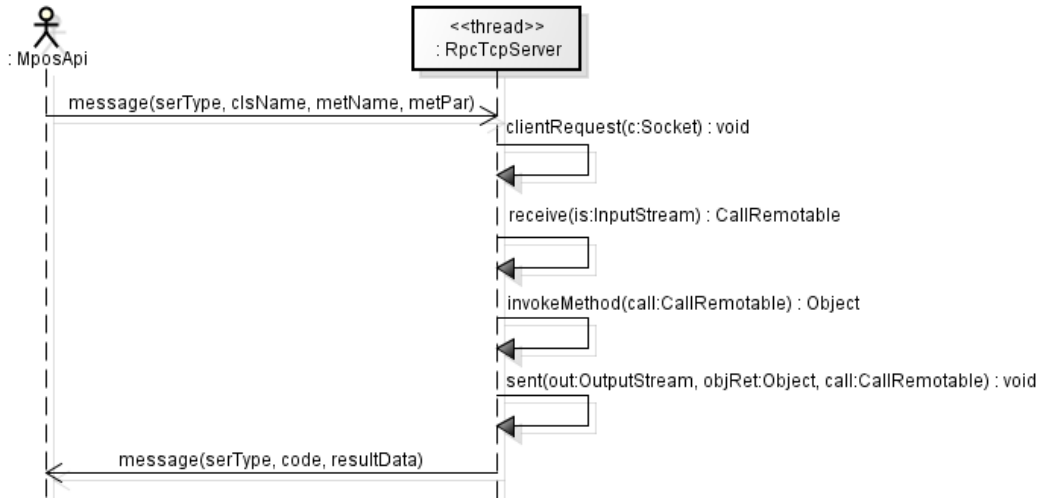

Figura 4.19. Execução do Serviço de *Offloading* no servidor

No servidor remoto (Figura 4.19), o Serviço de *Offloading* define um objeto do tipo *CallRemotable*, para receber do cliente essas quatro informações, que foram recebidas. O primeiro atributo recebido pelo *Socket* define o tipo de deserialização usada nessa sessão e verifica se precisa fazer a coleta das estatísticas do RPC. Os próximos dois atributos são referentes ao nome da classe e nome do método, que é chamado pelo cliente no AER. O nome da classe serve nesta apenas para instanciar dinamicamente um objeto necessário para invocar o método passado para

ser executado remotamente usando diversos processos de inspeção de código. O último atributo passado é relacionado com os valores dos parâmetros desse método remoto. Se durante essa sessão foi adotada a serialização manual, o objeto que foi instanciado previamente irá invocar o método *readMethodParams*, passando por parâmetro, o nome do método remoto e o restante do *stream* do *Socket*. Senão, o *stream* realiza o procedimento de deserialização automática, extraindo assim, os valores dos parâmetros necessários para executar o método remoto no servidor.

Após de definir os atributos do objeto *CallRemotable*, o próximo passo é invocar o método remoto (ver Figura 4.19) usando um processo de inspeção ou reflexão de código, sobre o objeto que foi instanciado anteriormente. O sistema realiza a inspeção de cada método desse objeto verificando sempre, o nome e quantidade de parâmetros, em relação ao método remoto que foi enviado pelo cliente. Ao localizar um método candidato, o sistema de reflexão o invoca passando também todos os seus parâmetros necessários para sua execução. Caso também exista uma sobrecarga desse método candidato, com a mesma quantidade de parâmetros, mas com tipos diferentes, em relação aos tipos utilizados no método remoto; o sistema irá tentar invocar esse método candidato disparando na sequência um erro na execução. Nesse momento, o método *invokeRemote* irá "ignorar" o erro, chamando o próximo método candidato, até conseguir executar o método corretamente, se ele supostamente existir. Pois, pode haver situações no qual o programador atualizou seu aplicativo, porém esqueceu-se de atualizar as dependências e de mudar a versão do aplicativo móvel, levando assim, a requisitar no AER, métodos ou objetos que fazem parte das novas dependências, mas não está disponível no servidor remoto por ter ainda a versão antiga das dependências da aplicação.

Em caso de sucesso da invocação dinâmica do método, o valor retornado pela sua execução, será enviado para o cliente remoto, caso contrário uma mensagem de erro é que será enviada. Depois de definido o valor de retorno (Figura 4.19), o servidor escreve o seguinte protocolo de resposta. O primeiro *byte* do *stream* do *Socket* é definido o tipo de serialização utilizada no envio desses dados, sendo no caso a mesma utilizada no começo dessa sessão de RPC. Apenas o tipo serialização pode ser mudado para o automático, caso ocorra alguma condição de erro durante a tentativa de executar um método remoto, no AER.

Se no decorrer dessa sessão for exigido informações sobre as estatísticas do RPC, os seguintes dados, além da resposta do método remoto precisa ser passada para o cliente remoto, como tamanho e tempo do *upload* e o tempo da execução remota do método. No caso da informação sobre o tamanho e o tempo de *download*, elas são obtidas apenas no lado do cliente. Por fim, se a serialização utilizada nessa sessão de RPC for do tipo manual, ou do tipo *RpcSerializable*, o objeto que foi instanciado previamente, irá invocar o procedimento

writeMethodReturn, passando por parâmetro, o nome do método remoto, o valor do retorno do método remoto e o *stream* do *Socket* que escreve essas informações pela rede.

No final da execução do Serviço de *Offloading*, a *thread* que foi utilizada nesta sessão de RPC, entra no processo de reciclagem da *thread*, por meio de um sistema que implementa o padrão *thread pool*. Essa sessão de RPC também não guarda nenhuma informação interna do cliente para ser usada em um momento posterior, sendo também caracterizada essa sessão de RPC como *stateless*.

4.4 Considerações Finais do Capítulo

O *framework* MpOS (*Multi-platform Offloading System*) foi apresentado neste capítulo com todos os detalhes inclusive de funcionamento dos componentes, tanto na parte cliente (MpOS API), quanto na parte servidora (MpOS Plataforma). Desse modo, o capítulo expôs uma solução de *offloading* para duas plataformas móveis Android e Windows Phone, e realiza esta operação de *offloading* no nível de método em relação a uma aplicação móvel, durante o seu tempo de execução.

No entanto, por se tratar de um projeto multiplataforma, alguns componentes não estão disponibilizados na plataforma móvel do Windows Phone e .NET Framework, sendo assim, necessário desenvolver tais componentes para se ter soluções que tivesse implementações equivalentes nas duas plataformas móveis. Dos componentes que foram desenvolvidos, podemos destacar o desenvolvimento de um *script* T4[21], que gera o código em tempo de compilação, tendo como resultado a criação de uma classe, denominada de *ProxyFactory*, que simula o funcionamento da classe *Proxy* da API do Android.

O outro item que precisou ser desenvolvido para a plataforma do Windows Phone e .NET Runtime foi um sistema de serialização automática e portável, entre a plataforma móvel e o servidor remoto, usando uma biblioteca de serialização, baseada no BSON. Finalmente, outros componentes também precisaram ser desenvolvidos, como o sistema de *logging*, leitura de um arquivo *properties* e criação de alguns objetos adaptadores, para ter certos tipos que existem apenas no Android, como *Runnable* e *AsyncTask*.

[21] http://msdn.microsoft.com/en-us/library/bb126445.aspx

Capítulo 5

Experimentos

Com objetivo de avaliar e validar o *framework* MpOS foram desenvolvidos duas aplicações móveis para a plataforma Android e Windows Phone. Este capítulo tem o objetivo de apresentar a definição das funcionalidades e da arquitetura dessas aplicações, discutindo também os resultados obtidos durante a execução dos experimentos.

5.1 Introdução

Para avaliar os componentes propostos no capítulo anterior foram desenvolvidas como prova de conceito, duas aplicações móveis do tipo *CPU Bound*. Este tipo de aplicação é caracterizado por depender apenas da capacidade de processamento da CPU para executar uma determinada tarefa, pois a variação do desempenho dos outros componentes, como quantidade de memória RAM e velocidade da E/S, não acelera ou atrasa a execução dessa tarefa. Contudo, quando se aplica a técnica de *offloading*, estes aplicativos propostos deixam de ser *CPU Bound* para ser *I/O Bound*, por depender mais do meio de transmissão (no caso as redes sem fio), do que a capacidade da CPU, onde será executado este processamento.

Nesta dissertação a primeira aplicação desenvolvida é denominada de *BenchImage* [Costa *et al.* 2014] e executa uma tarefa relacionada com o processamento de uma foto. O segundo aplicativo denominado de *Collision* é voltado para a computação gráfica e simula a colisão de diversas esferas dentro da tela do *smartphone* em um espaço 2D. Estes aplicativos demonstram como é possível utilizar o *framework* MpOS, servindo também de exemplos para o desenvolvimento de outras aplicações que precisem adotar a técnica de *offloading*. Os exemplos mostram como a técnica de *offloading* permite acelerar a execução de uma aplicação, utilizando recursos de uma infraestrutura computacional que está disponível localmente em uma rede Wi-Fi, ou mesmo, em uma nuvem pública na Internet.

A metodologia dos experimentos utilizou sempre de dois dispositivos móveis, sendo um Android e outro Windows Phone. Os aplicativos também suportavam selecionar uma localidade em relação à execução do processamento, podendo ser executado localmente no aparelho ou remotamente através da operação de *offloading* em um determinado servidor remoto. No caso da aplicação *BenchImage* o *offloading* será realizado em diferentes localidades, podendo o AER está

localizado em uma mesma rede local sem fio, ou mesmo, na Internet hospedado em uma nuvem pública. No caso do aplicativo *Collision*, o AER está disponível apenas localmente em uma mesma rede Wi-Fi, no qual estão os dispositivos móveis. Porém, diferente do aplicativo *BenchImage* que usa apenas o sistema de serialização automático, o aplicativo *Collision* adota também o sistema de serialização manual, mostrando os resultados para ambos os sistemas de serialização.

Contudo, os experimentos não avaliam o ganho energético que poderia ser proporcionado com a operação de *offloading*, sendo neste caso apenas avaliado o ganho do desempenho, quando comparado com a execução local do processamento em um dispositivo móvel. Por fim, os resultados de todos os experimentos serão mostrados separadamente para cada plataforma móvel, pois não é objetivo desse trabalho comparar o desempenho da execução dos aplicativos entre as diferentes plataformas móveis.

Nas próximas seções, serão detalhados os aplicativos *BenchImage* e *Collision*, como também os ambientes utilizados nos experimentos e serão discutidos os resultados que foram obtidos na execução desses testes.

5.2 Aplicação BenchImage

O *BenchImage* é um aplicativo de processamento de imagem (tipo *CPU Bound*), que realiza a operação de aplicar um filtro sobre uma determinada foto selecionada pelo usuário. Este aplicativo possui quatro implementações de filtros e dispõe internamente de três tipos de fotos, que possuem diferentes resoluções que variam deste de 8MP até 0.3MP. A Figura 5.1 mostra a tela inicial desse aplicativo para a plataforma Android (à esquerda) e Windows Phone (à direita).

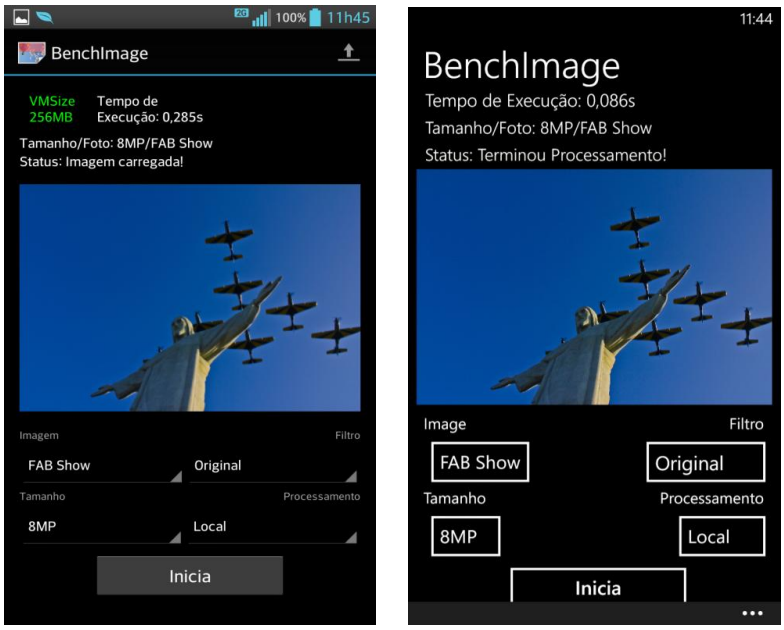

Figura 5.1. Tela inicial do *BenchImage* em diferentes plataformas móveis

Todas as fotos disponíveis no *BenchImage* estão no formato JPEG, sendo necessário a aplicação criar uma matriz de cores em um formato "cru" (*raw*) para permitir que o aplicativo manipule as imagens, de acordo com filtro que foi selecionado pelo usuário. O aplicativo disponibiliza os seguintes filtros de imagem: (i) Original (ii) *Sharpen*; (iii) *Cartoonizer* e (iv) *Red Tone*.

O primeiro filtro denominado de Original carrega e exibe a foto no aplicativo, sem realizar nenhuma operação de processamento de imagem. O segundo filtro disponível chamado de *Sharpen*, tenta aumentar ou diminuir a nitidez da imagem dependendo da máscara que está sendo usada na imagem. O próximo filtro, o *Cartoonizer*, tem o objetivo de transformar uma imagem em outra desenhada a lápis, conforme mostra a Figura 5.2. O último filtro do aplicativo, chamado de *Red Tone*, aplica um mapa de cores sobre uma foto, deixando-a no tom de avermelhado.

Figura 5.2. Foto original na esquerda e foto após o filtro na direita

O filtro *Cartoonizer* é considerado o mais pesado do aplicativo *BenchImage*, porque realiza quatro outras aplicações de filtros, sendo executado nessa sequência: (i) transforma para escala de cinza a imagem original; (ii) clona a imagem do resultado anterior, invertendo as suas cores; (iii) aplica o filtro de *Sharpen* para embaçar a imagem (i) que foi gerada neste processo, e (iv) por meio da operação de *colorDodgeBlend* são misturados os resultados das imagens (ii) e (iii), formando uma nova imagem simulando o desenho a lápis, igual como mostrado na Figura 5.2.

O aplicativo *BenchImage* possui um modo de *benchmark*, que serve para avaliar o desempenho do dispositivo móvel. Este modo executa três vezes o filtro *Cartoonizer*, para cada resolução de foto, sendo neste caso cinco resoluções diferentes: 8MP, 4MP, 2MP, 1MP e 0.3MP; resultando dessa maneira de 15 execuções desse filtro.

É bom ressaltar que na execução de qualquer (tirando o original) filtro de imagem, o resultado produzido é também uma foto no formato JPEG, sendo salva no *storage* do dispositivo móvel. Outras informações relacionadas com a execução do filtro são também salvas no banco

de dados local, como: configuração escolhida na execução, tempo da execução e as estatísticas da rede, caso tenha sido realizado uma operação de *offloading*.

O aplicativo *BenchImage* possui uma funcionalidade para exportar as informações do banco de dados local, para um arquivo no formato CSV, que é salvo no *storage* do *smartphone*. Enfim, por limitação de escopo não será utilizado nesta aplicação o sistema de serialização manual para realizar o procedimento de *offloading*, sendo nesse caso utilizado apenas o sistema de serialização automático.

Nas próximas seções será explicada a arquitetura do aplicativo, como também o ambiente do experimento e a discussão dos resultados obtidos para cada plataforma móvel.

5.2.1 Arquitetura

Depois de definir as funcionalidades do aplicativo *BenchImage*, o próximo passo seria definir os componentes dessa aplicação através de um diagrama de classes, sendo apresentada pela Figura 5.3.

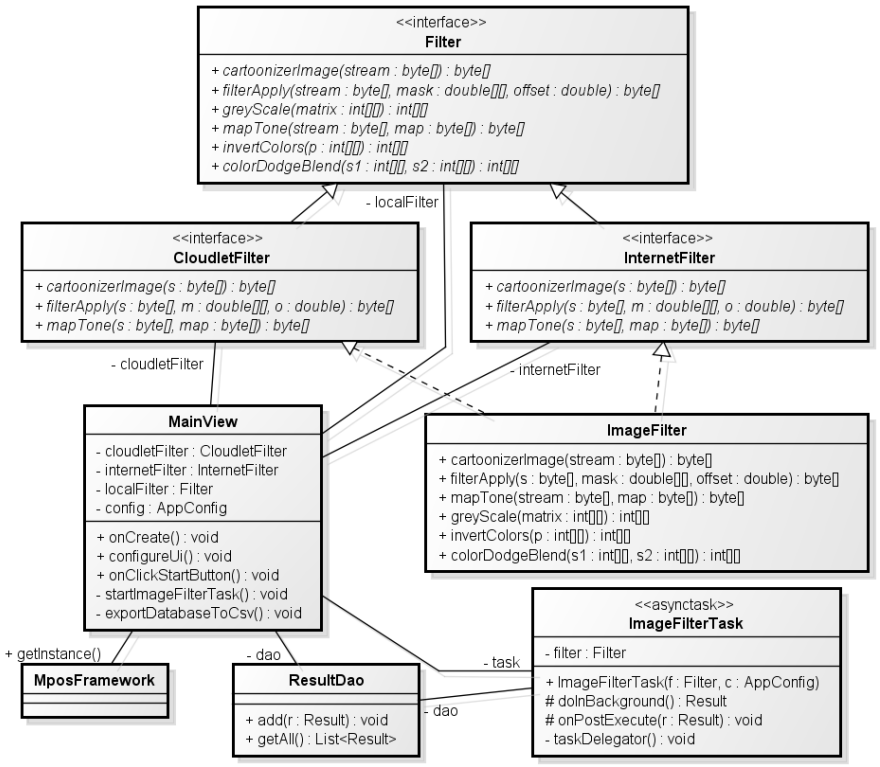

Figura 5.3. Diagrama de Classe do aplicativo *BenchImage*

O aplicativo *BenchImage* possui uma única tela que inicializa a partir da classe *MainView* (Figura 5.3) no método *onCreate*, pois esta classe serve para simplificar do processo de inicialização do aplicativo nas diferentes plataformas móveis. Esta classe também realiza o processo de configuração e inicialização do MpOS API, por meio da marcação *MposConfig* e da

classe *MposFramework*. A classe *MainView* possui diversas variáveis de instância, porém as principais variáveis estão destacadas na Figura 5.4 (a) para a plataforma Android e Figura 5.4 (b) no Windows Phone.

```java
@MposConfig(endpointSecondary = "54.94.172.61")
public final class MainActivity extends Activity {
    private Filter localFilter = new ImageFilter();

    @Inject(ImageFilter.class)
    private CloudletFilter cloudletFilter;

    @Inject(ImageFilter.class)
    private InternetFilter internetFilter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        MposFramework.getInstance().start(this);
        (...)

        Log.i(clsName, "Started BenchImage");
    }
    (...)
}
```

(a)

```java
[MposConfig("54.94.172.61")]
public partial class App : Application
{
    private Filter localFilter = new ImageFilter();

    [Inject(typeof(ImageFilter))]
    private CloudletFilter cloudletFilter = null;

    [Inject(typeof(ImageFilter))]
    private InternetFilter internetFilter = null;

    public App()
    {
        MposFramework.Instance.Start(this, new ProxyFactory());

        (...)

        Debug.WriteLine("[App]: Started BenchImage");
    }
    (...)
}
```

(b)

Figura 5.4. Configuração do *BenchImage* em conjunto com MpOS API

As variáveis *cloudletFilter* e *internetFilter* são instanciadas dinamicamente pelo MpOS API através da marcação *Inject* e usando o tipo concreto do *ImageFilter*, enquanto a variável *localFilter* é instanciada normalmente pelo desenvolvedor do aplicativo. A existência destas três variáveis está relacionada com a localidade da execução onde é chamado o método. A variável *localFilter* chamam métodos para ser executados localmente. Enquanto as variáveis *cloudletFilter* e *internetFilter* chamam métodos que podem ser executados através de uma operação de *offloading*, para um determinado servidor remoto, dependendo também da disponibilidade da rede. Por isso, na arquitetura da aplicação (Figura 5.3) existem três interfaces, denominadas de: (i) *Filter*; (ii) *CloudletFilter* e (iii) *InternetFilter*.

A primeira interface possui todos os métodos necessários para realizar as operações de processamento de imagem do aplicativo *BenchImage*, não possuindo nenhuma marcação para execução remota. Enquanto as outras duas interfaces (ii) e (iii) herdam da interface *Filter*, sobre-escrevendo os métodos que devem realizar a operação de *offloading*, através da marcação *Remotable*. A Figura 5.5 mostra apenas no caso do Windows Phone a marcação sobre as interfaces (ii) e (iii), sendo também similar na plataforma Android.

```
public interface CloudletFilter : Filter
{
    [Remotable(Status = true)]
    new byte[] MapTone(byte[] source, byte[] map);

    [Remotable(Status = true)]
    new byte[] FilterApply(byte[] source, double[][] filter, double factor, double offset);

    [Remotable(Offload.STATIC, Status = true)]
    new byte[] CartoonizerImage(byte[] source);
}

public interface InternetFilter : Filter
{
    [Remotable(Status = true, CloudletPriority = false)]
    new byte[] MapTone(byte[] source, byte[] map);

    [Remotable(Status = true, CloudletPriority = false)]
    new byte[] FilterApply(byte[] source, double[][] filter, double factor, double offset);

    [Remotable(Offload.STATIC, Status = true, CloudletPriority = false)]
    new byte[] CartoonizerImage(byte[] source);
}
```

Figura 5.5. Marcação dos métodos nas interfaces no WP

Deste modo a principal diferença entre as interfaces do *CloudletFilter* e *InternetFilter*, está relacionado com a prioridade em relação à localidade da operação de *offloading* (*cloudlet* ou Internet), sendo definido na própria marcação *Remotable*. É bom ressaltar que o método *CartoonizerImage* sempre irá fazer a operação de *offloading* salvo aconteça alguma indisponibilidade na rede, usando neste caso o particionamento do tipo estático (*Offloading.STATIC*).

Nesta arquitetura a tarefa de processamento de imagem é realizada, por meio da classe *ImageFilterTask* (ver Figura 5.3), que recebe na construção desta classe, o que foi definido na UI e no *MposConfig*, além de um objeto do tipo *Filter*, que implementa os diversos algoritmos de processamento de imagem que são usados no programa.

Durante a execução desta tarefa, a imagem que foi selecionada pelo usuário é carregada do *storage* e enviada para ser processada pelo método de acordo com a seleção do filtro por parte do usuário. Estes métodos podem ou não realizar a operação de *offloading* dependendo da instância passada. No final desse processamento uma foto no formato JPEG é retornada do método chamado, sendo salva no *storage* e os resultados da execução são guardados no banco de dados do dispositivo móvel. O aplicativo também exibe na UI, um *preview* da imagem processada, exibindo também o tempo total da execução do processo.

5.2.2 Ambiente do experimento

O *BenchImage* foi executado cinco vezes no modo de execução *benchmark*, resultando no total de 75 repetições do aplicativo para cada ambiente de execução. Cada plataforma móvel realizou este experimento de *benchmark* em 7 ambientes de execução diferentes, resultando no total de 525 repetições do aplicativo *BenchImage* para cada plataforma móvel. Os 7 ambientes de execução é composto pela execução local do aparelho e através da realização da operação de

offloading em cada um dos *cloudlet servers* (Ci7Mob e Ci5Desk) e máquinas hospedadas na nuvem pública da Amazon (EC2 Medium e EC2 Large). A forma de acesso dos *cloudlet servers* foi através de um ponto de acesso TP-Link WR740N (802.11n) dedicado exclusivamente para esta tarefa. Enquanto, no caso da nuvem pública foram utilizadas duas formas de conexões: (i) por meio da rede ADSL2+ (contrato de 10 Mbps para *download* e 0.5 Mbps para *upload*) conectado ao Wi-Fi e (ii) através da Internet móvel 4G LTE suportado por ambos os *smartphones*. A Tabela 5.1 apresenta mais detalhes sobre o *hardware* dos ambientes de execução.

Tabela 5.1. Configuração dos Ambientes de Execução do *BenchImage*

Ambiente de Execução	Configuração
WP Local	Nokia 925 / Windows Phone 8.0 Processador Qualcomm Krait, 2 *cores*, 1.5 GHz e 1 GB de RAM
Android Local	LG Optimus G E977 / Android 4.1.2 Processador Qualcomm Krait, 4 *cores*, 1.5 GHz e 2 GB de RAM
Cloudlet "Ci7Mob"	Windows 8.1 64-bits Processador Intel Core i7-4500U, 2 *cores*, 1.8 GHz e 8 GB de RAM
Cloudlet "Ci5Desk"	Windows 8.1 Pro 64-bits Processador Intel Core i5-4570, 4 *cores*, 3.2 GHz e 16 GB de RAM
EC2 Medium	Windows Server 2012 RTS / m3.medium 1 VCPU, 3 ECU e 3.75GB de RAM
EC2 Large	Windows Server 2012 RTS / m3.large 2 VCPU, 6.5 ECU e 7.5GB de RAM

5.2.3 Resultados e Discussão

O aplicativo *BenchImage* avalia durante os experimentos o desempenho da técnica de *offloading* na realização de processamentos longos como a aplicação do filtro *Cartoonizer* em uma determinada foto selecionada pelo usuário. O experimento fez uso do particionamento estático para forçar a utilização da operação de *offloading* independente da qualidade da rede. Todos os resultados apresentados como tempo total, tempo de computação e tempo de transferência, nas próximas seções foram obtidos através da média dos valores coletados, sendo também aplicado um intervalo de confiança de 95% sobre os valores coletados. Nas próximas seções serão discutidas a execução do *BenchImage* para cada plataforma móvel de acordo com suas peculiaridades, não realizando nenhuma comparação entre elas.

5.2.3.1 Android

O aplicativo *BenchImage* executou os experimentos a partir de um *smartphone* Android com suporte ao 4G, coletando durante sua execução diversas informações sobre o desempenho do aplicativo e a utilização da rede quando feito uma operação de *offloading*. Dentre estas informações

a primeira que será avaliada é o tempo total (em segundos) da execução do experimento, sendo mostrada na Tabela 5.2, para cada tamanho de foto e ambiente de execução.

Tabela 5.2. Tempo total de execução em segundos no Android

	8MP	4MP	2MP	1MP	0.3MP
Android Local	58,47 ± 2,15	28,72 ± 1,36	14,03 ± 0,30	7,20 ± 0,08	2,24 ± 0,05
Ci5Desk	4,08 ± 0,22	1,88 ± 0,07	1,11 ± 0,03	0,66 ± 0,02	0,28 ± 0,01
Ci7Mob	5,52 ± 0,34	2,23 ± 0,11	1,35 ± 0,09	0,76 ± 0,06	0,34 ± 0,03
EC2 Medium (4G)	12,99 ± 1,02	6,16 ± 0,35	4,28 ± 0,28	2,92 ± 0,21	1,78 ± 0,05
EC2 Medium (Wi-Fi)	93,18 ± 1,41	26,99 ± 1,16	15,31 ± 0,62	8,59 ± 0,45	5,28 ± 0,16
EC2 Large (4G)	9,75 ± 0,43	5,14 ± 0,28	3,47 ± 0,15	2,53 ± 0,18	1,68 ± 0,08
EC2 Large (Wi-Fi)	88,82 ± 1,40	25,67 ± 1,06	15,06 ± 0,92	8,17 ± 0,46	5,23 ± 0,20

A execução local do experimento teve um melhor desempenho, quando comparado a pior situação do ambiente de execução remoto (AER), utilizando uma nuvem pública (EC2 Medium e Large) e sendo acessada a partir da Internet Wi-Fi, que possui uma vazão de *upload* limitada a 0.5Mbps. Neste caso, a execução local teve um ganho de 37% e 57% sobre estes AER na aplicação do filtro em uma foto de 8MP e 0.3MP. Para outros tamanhos de fotos houve uma pequena perda no desempenho, como no caso da foto de 4MP e um ganho de menor proporção para os tamanhos de 2MP e 1MP. No entanto, a execução local perdeu para todos os outros cenários de AER e independente do tamanho da foto.

Para melhor visualizar a composição do tempo total em cada AER será apresentado na Figura 5.6 apenas o tempo de processamento (em segundos) para realizar a operação de *offloading* com relação à foto de tamanho de 8MP, 2MP e 0.3MP. Neste caso foram desconsiderados os tempos de transferências dos dados entre o dispositivo móvel e o AER.

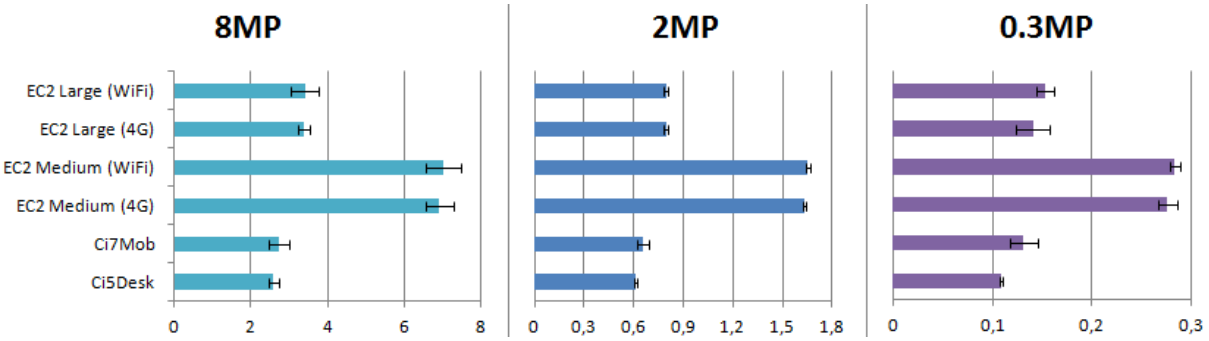

Figura 5.6. Tempo de processamento em segundos para diferentes tamanhos de fotos

Esta informação sobre o tempo de processamento mostra como a capacidade da CPU do *smartphone* Android é limitada, em comparação com a capacidade de uma máquina de maior porte, como um *notebook*, computador de mesa, ou mesmo, uma máquina virtualizada em um ambiente de nuvem pública. O Android Local no tempo total era 37% mais rápido na aplicação do filtro sobre uma imagem de 8MP, do que uma instância EC2 Medium e Large, usando uma conexão

lenta de *upload*. Porém, em termos computacionais estas instâncias são respectivamente 8.35 e 17.2 vezes mais rápidas, do que o processador Android. Esta diferença cresce 22 vezes, quando comparada com os processadores dos *cloudlet servers* sem virtualização. No caso dos tamanhos de 2MP e 0.3MP, a vantagem em termos computacionais, ainda continua na mesma proporção que descrito anteriormente para os mesmos ambientes de execução remotos.

Por último a Figura 5.7 compara o tempo médio (em segundos) para fazer o *download* e *upload*, entre o dispositivo móvel e o ambiente de execução remoto.

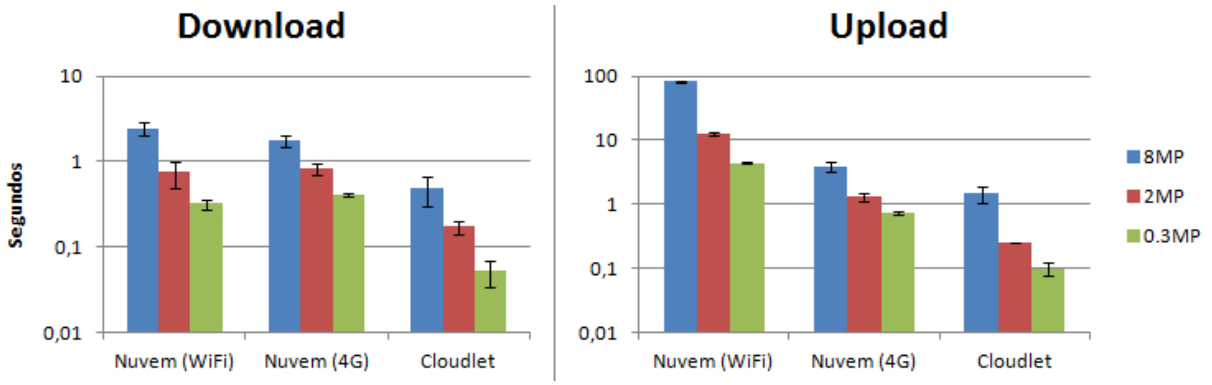

Figura 5.7. Tempo de transferência em segundos dos dados no Android

O que se observa que o tempo de transferência é menor na rede local sem fio, pois as taxas de transferências não estão limitadas por um provedor, uma vez que não passa pela Internet. Por isto, que os *cloudlet servers* em relação à plataforma Android possuem um menor tempo total de execução quando comparado com todos os outros AER. Porém a utilização da Internet móvel 4G não deixa muito a desejar, podendo ser uma alternativa para realizar a operação de *offloading* em uma nuvem pública, quando o usuário estiver em ambientes abertos (parques, shoppings ou aeroportos), ou mesmo, em mobilidade (deste que exista cobertura para isto) de um lugar a outro.

5.2.3.2 Windows Phone

Esta plataforma móvel realizou os mesmos experimentos que foram realizados na plataforma Android, sendo mostrado na Tabela 5.3 o tempo total da execução do experimento sobre os mesmos ambientes de execução remotos.

Tabela 5.3. Tempo total de execução em segundos no Windows Phone

	8MP	**4MP**	**2MP**	**1MP**	**0.3MP**
WP Local	21,87 ± 0,19	10,26 ± 0,10	5,27 ± 0,05	2,61 ± 0,05	0,84 ± 0,01
Ci5Desk	13,92 ± 0,10	7,10 ± 0,31	3,71 ± 0,16	2,10 ± 0,14	0,79 ± 0,05
Ci7Mob	16,55 ± 0,57	7,70 ± 0,21	4,21 ± 0,10	2,31 ± 0,07	0,94 ± 0,11
EC2 Medium (4G)	40,55 ± 2,68	18,80 ± 0,63	10,46 ± 0,23	6,23 ± 0,17	3,04 ± 0,22
EC2 Medium (Wi-Fi)	121,54 ± 2,53	42,07 ± 1,95	24,51 ± 1,03	13,41 ± 0,73	7,93 ± 0,59
EC2 Large (4G)	29,63 ± 2,99	13,07 ± 1,28	7,15 ± 0,36	4,57 ± 0,36	2,42 ± 0,27
EC2 Large (Wi-Fi)	108,10 ± 2,07	35,85 ± 1,46	21,66 ± 0,93	11,49 ± 0,60	6,93 ± 0,64

No Windows Phone a execução local do experimento teve um melhor desempenho, quando comparado com todos os resultados que foram executados na nuvem pública, independente do tamanho da instância e do meio de comunicação utilizado para acessar este recurso na Internet. Assim, a execução local foi em média cinco vezes mais rápida, do que o AER do EC2 Medium sendo acessado a partir do Wi-Fi. Ainda usando a nuvem pública com uma instância de maior porte, como o EC2 Large e usando a Internet móvel 4G, a execução local no aparelho ainda foi melhor por volta de 35.6% para as fotos de tamanho de 8MP e 2MP, ampliando esta vantagem para 2.8 vezes para a foto de 0.3MP. No entanto, a execução local leva desvantagem, quando comparado com o melhor *cloudlet server* (Ci5Desk) utilizado no experimento. Para fotos de 8MP este AER leva vantagem de 57%. Porém, continua caindo progressivamente até chegar a uma vantagem de apenas 6.3% para fotos de 0.3MP. No caso do Ci7Mob ele perde com a diferença de 10% em relação à foto de 0.3MP quando comparado com a execução local.

Para melhor entender a dinâmica por trás do tempo total será apresentada na Figura 5.8, apenas para o tempo de processamento (em segundos) para realizar a operação de *offloading* sobre os experimentos com fotos de tamanho de 8MP, 2MP e 0.3MP.

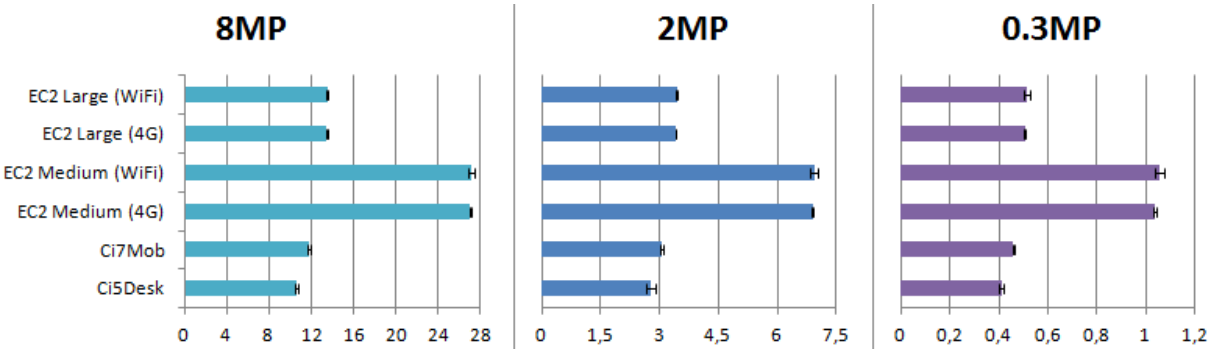

Figura 5.8. Tempo de processamento em segundos para diferentes tamanhos de fotos

O ambiente de execução remoto EC2 Medium continua mais lento por volta de 25%, quando comparado com a execução local no *smartphone*, em relação a esses três tamanhos de fotos. Porém, o ambiente de execução remoto EC2 Large é 60% mais rápido, do que a execução local no aparelho móvel. O *cloudlet server* Ci5Desk, consequentemente é mais rápido duas vezes em termo de processamento, do que a execução local no *smartphone*.

Por fim, a Figura 5.9 mostra o tempo de transferência (em segundos) para fazer o *download* e *upload*, entre o dispositivo móvel e os ambientes de execução remotos.

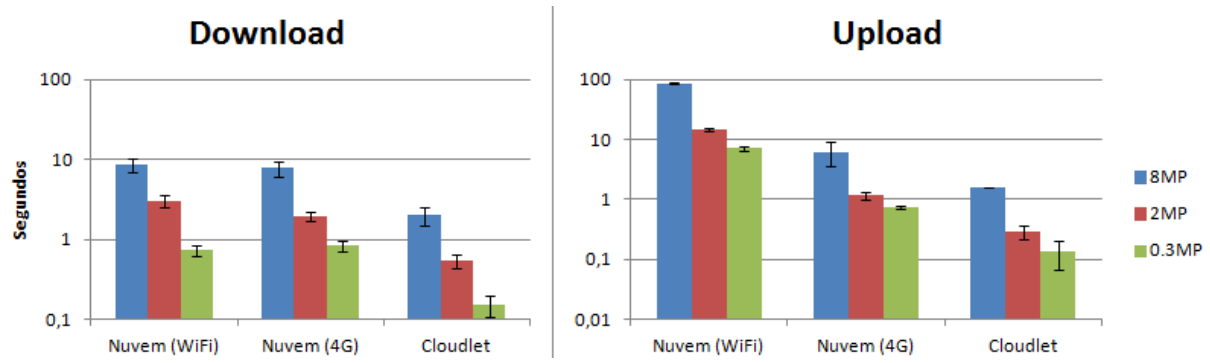

Figura 5.9. Tempo de transferência em segundos dos dados no Windows Phone

O tempo de transferência neste experimento da plataforma do Windows Phone ainda é menor na rede local sem fio (802.11n), justificando o desempenho dos AER baseado em *cloudlet servers*. A Internet móvel 4G teve uma taxa de *download* similar à internet ADSL2+, no entanto com uma maior vazão de *upload*, que ajuda a transferir os dados mais rapidamente para a nuvem pública. Assim, uma conexão mais rápida nos dois sentidos (*download* e *upload*) pode acelerar o tempo total da execução de um aplicativo em até 3.6 vezes, como acontece quando comparamos o EC2 Large (4G) com EC2 Large (Wi-Fi), para uma foto de 8MP. Isso reforça a ideia da dependência da técnica de *offloading*, em relação ao meio de transmissão utilizado, impactando diretamente no desempenho final do aplicativo.

5.3 Aplicação Collision

O *Collision* é um aplicativo de computação gráfica (tipo *CPU Bound*), que realiza diversos cálculos para detectar a colisão de objetos um com os outros ou com as fronteiras do ambiente virtual, limitado à tela do *smartphone* para cada *frame* renderizado no programa. A Figura 5.10 apresenta a tela inicial desta aplicação no seu estado inicial antes de iniciar a animação, para a plataforma Android (à esquerda) e Windows Phone (à direita).

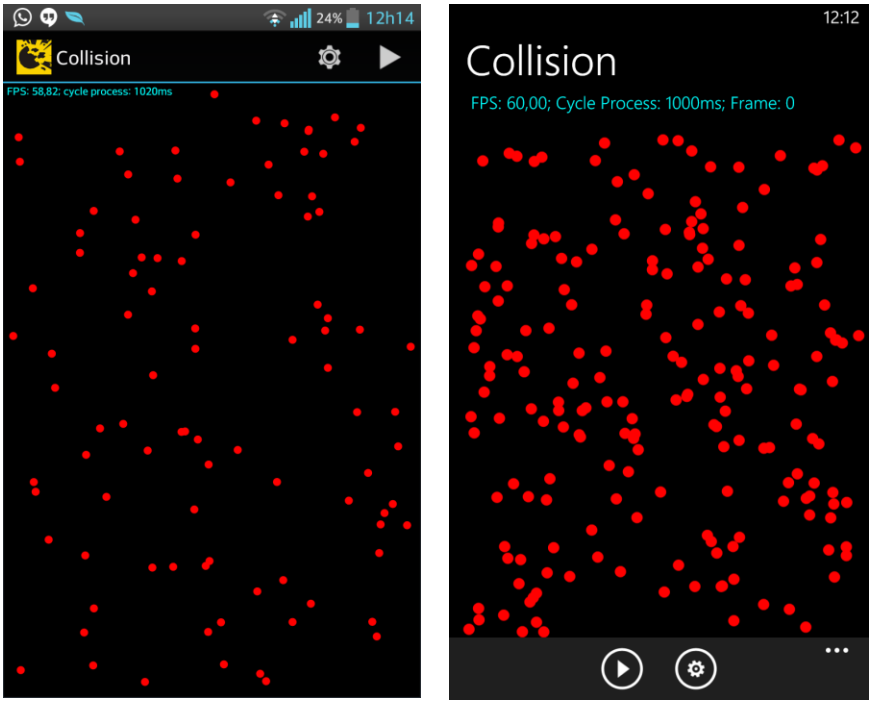

Figura 5.10. Aplicação *Collision* em diferentes plataformas móveis

Este aplicativo possui diversos tipos configurações que podem ser definidos pelo usuário, como iniciar a execução da aplicação escolhendo entre 250 até 1500 esferas, podendo também inicializar em um lugar fixo ou aleatoriamente dentro da tela do *smartphone*, como mostrado na Figura 5.10. O *Collision* permite duas formas de execução: (i) localmente no aparelho móvel, e (ii) remotamente em um *cloudlet server*, através da técnica de *offloading*. No entanto, por questões de escopo do projeto, este aplicativo não executará a técnica de *offloading* na Internet, apenas em um *cloudlet server* disponível localmente na rede sem fio. Por fim, a última configuração que pode ser definida no aplicativo é o tipo de serialização empregada no momento do processo do *offloading*, podendo ser automática ou manual.

As próximas seções serão explicadas a arquitetura do aplicativo, como também o ambiente do experimento e a discussão dos resultados obtidos.

5.3.1 Arquitetura

Depois de descrever as funcionalidades do aplicativo *Collision*, o próximo passo seria definir a arquitetura desse programa através de um diagrama de classes, conforme mostrado na Figura 5.11.

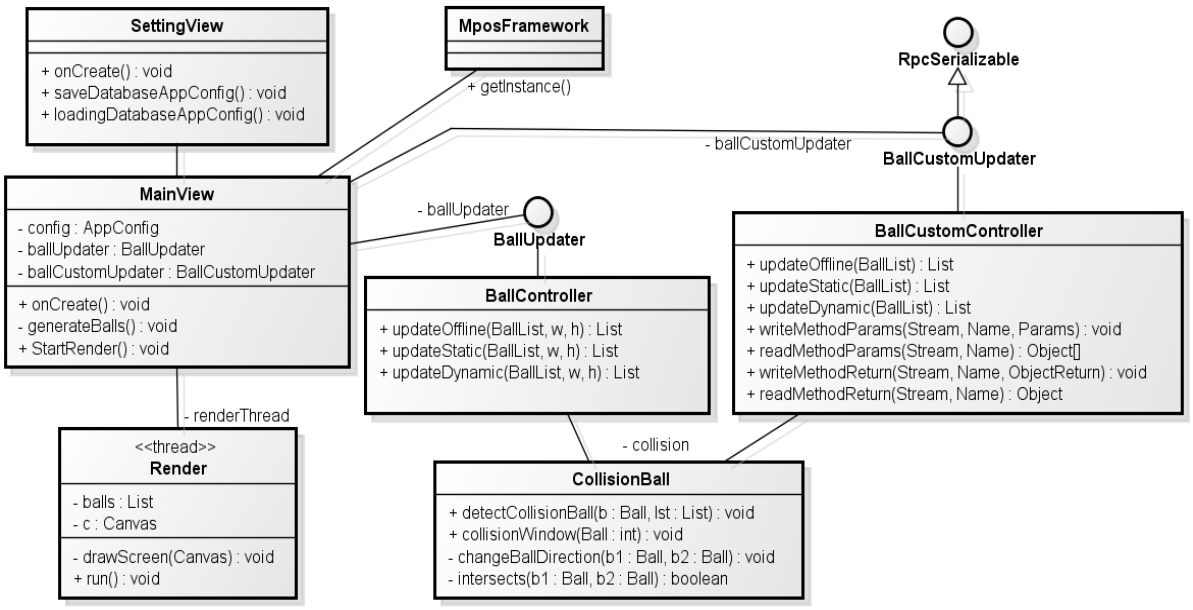

Figura 5.11. Diagrama de Classe do aplicativo *Collision*

Nesta arquitetura, as classes relacionadas com a *MainView* e com a *SettingView* são responsáveis por tratar, respectivamente da exibição das esferas e da configuração do aplicativo móvel. Também foi convencionado que o aplicativo móvel inicializa a partir da classe *MainView* apenas para simplificar o projeto (Figura 5.11) em relação às diferentes plataformas móveis. Na classe *MainView* foi realizado o processo de configuração e inicialização do MpOS API, por meio de diversas marcações e da classe *MposFramework*. As variáveis *ballUpdater* e *ballCustomUpdater* dessa classe são também instanciadas dinamicamente, através da marcação *Inject*.

Por se tratar de uma aplicação de computação gráfica, este aplicativo móvel possui uma classe de renderização, chamada de *Render*, que executa por meio de uma *thread* própria. Esta classe processa para cada quadro (ou *frame*) produzido uma lista de esferas que serão desenhadas na UI do aplicativo móvel. O *Render* interage com o controlador *BallController* ou *BallCustomController*, que são responsáveis por atualizar o próximo estado destas esferas (localidade e direção), detectando também algum processo de colisão, de acordo com as regras de negócio, que foram definidas na classe *CollisionBall*.

Como o processamento do aplicativo está concentrado nestes controladores, junto com a classe de negócio, então foi determinado nesta arquitetura, que os métodos destes controladores

podem realizar a operação de *offloading*. Por causa disso os controladores *BallCustomController* e *BallController*, implementam respectivamente as interfaces *BallUpdater* e *BallCustomUpdater*, que possuem métodos marcados para realizar a operação de *offloading*, como os métodos *updateStatic* e *updateDynamic*. As marcações destes métodos também definem o tipo do particionamento usado, sendo o particionamento estático para o método *updateStatic* e dinâmico para o método *updateDynamic*.

O motivo arquitetural para a existência de dois controladores está associado com o tipo de serialização utilizada para realizar a operação de *offloading*. No caso do controlador *BallController* é empregada a serialização automática tendo em consideração os parâmetros dos métodos e do seu retorno. Enquanto, a serialização manual é adotada pelo controlador *BallCustomController*, sendo também necessário que esse controlador implemente todos os métodos da interface *RpcSerializable* (ver Figura 4.14), para descrever o processo de serialização e deserialização, em relação aos métodos marcados para a operação de *offloading*.

5.3.2 Ambiente dos experimentos

A aplicação *Collision* foi executada dez vezes, com sessão de 90 segundos cada execução, sendo capturados a cada dez segundos os valores do FPS do momento. No final de cada execução foi produzida a média dos FPS obtidos durante essa sessão. Essas dez execuções foram repetidas para cada quantidade de esferas 250 (E250), 750 (E750) e 1500 (E1500), totalizando 30 repetições. Cada plataforma móvel realizou as 30 repetições em cada um dos cincos ambientes de execução, sendo o primeiro executado localmente no dispositivo móvel. Ao mesmo tempo em que os outros ambientes foram executados nos *cloudlet servers* (Ci7Mob e Ci5Desk), conforme definido na Tabela 5.1, usando também diferentes sistemas de serialização (manual e automática), com isso cada plataforma móvel executou 150 experimentos. Também foi utilizado o mesmo ponto de acesso TP-Link WR740N (802.11n), para conectar os *smartphones* com os *cloudlet servers*.

5.3.3 Resultados e Discussão

Este experimento avalia como seria o comportamento da técnica de *offloading*, quando empregado em um aplicativo de tempo-real, como o aplicativo *Collision*. Segundo alguns autores como Hughes *et al.* (2013), uma aplicação de computação gráfica deveria executar acima de 30 FPS, ou seja, produzir cada quadro (*frame*) demorando no máximo 33.34 ms, para que animação seja fluida na perspectiva do usuário da aplicação. O experimento realizado não mostra a execução do particionamento dinâmico do *offloading*, sendo limitado apenas ao particionamento estático, para facilitar a coleta de informações, quando estiver realizando a operação de *offloading*

independente da situação da rede local sem fio.

Os resultados mostram a execução do aplicativo *Collision* em FPS, sendo obtidos os valores através da média dos resultados coletados para cada quantidade de esferas e AER, além de aplicado um intervalo de confiança de 95% sobre estes valores. As próximas seções serão discutidas a execução do aplicativo *Collision* para cada plataforma móvel de acordo com suas peculiaridades não realizando nenhuma comparação entre elas.

5.3.3.1 Android

O aplicativo executou a partir de um *smartphone* Android e os resultados coletados são apresentados na Figura 5.12, para cada quantidade de esferas e AER. No caso dos ambientes de execução remotos (Ci7Mob e Ci5Desk), os resultados também estão associados com o tipo de serialização que foi empregada no momento da operação de *offloading* podendo ser automática (SA) ou manual (SM).

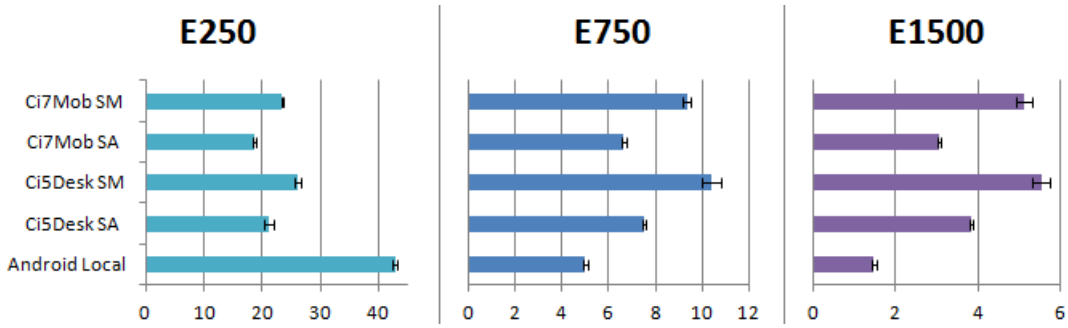

Figura 5.12. Resultado da Execução do *Collision* no Android em FPS

De acordo com a Figura 5.12, a execução do Android Local para o experimento de 250 esferas, teve um desempenho superior a 64% em comparação com a operação de *offloading* no *cloudlet* Ci5Desk, usando o sistema de serialização manual (SM). Por sua vez, quando se compara os sistemas de serialização em um mesmo *cloudlet server*, o SM teve um ganho de 24% em relação ao SA. Quando a quantidade de esferas foi triplicada para 750 unidades o desempenho da execução local caiu drasticamente. Isso permitiu que operação de *offloading* no *cloudlet* Ci5Desk usando a SM, obtivesse um ganho de desempenho de 100%, quando comparado com a execução local no dispositivo móvel. O sistema automático obteve também um ganho de desempenho, na ordem de 50% em relação à execução local. Porém, a diferença entre os sistemas de serialização utilizando um mesmo *cloudlet* é maior 38% para o sistema de SM, tendo como base de comparação o sistema de SA. Por fim, quando duplicamos a quantidade de esferas para 1500 unidades, a execução local continua perdendo para todas as operações de *offloading*. A execução no Ci5Desk SM e SA ficaram em torno de 3.72 e 2.58 vezes mais rápidas, do que a execução local no *smartphone* Android.

A explicação do crescimento do desempenho da serialização manual sobre a serialização automático acontece, pois a SA é baseado em um sistema nativo[22] e portável para serialização de objetos na plataforma Android e Java SE. Este sistema utiliza de um protocolo próprio, além de inspecionar dinamicamente as propriedades do objeto usando programação reflexiva, consumindo assim, mais recursos do aparelho. No caso do sistema de serialização manual, o próprio programador deve escrever o protocolo de serialização, usando um *DataStream*[23]. Caso seja necessário, o desenvolvedor também pode adicionar ou omitir, propriedades e estados internos do objeto, durante a operação de *offloading*. Contudo, nos experimentos realizados nenhuma informação dos objetos foi omitida durante a construção do protocolo de serialização.

A Tabela 5.4 mostra o tamanho das mensagens de *offloading* (em *bytes*), que são trocadas no aplicativo *Collision*, para cada sistema de serialização automática (SA) e manual (SM), em relação às quantidades de esferas. Estes dados foram obtidas através do MpOS API, sendo também fixos, por causa da natureza dos objetos envolvidos, que possui apenas propriedades numéricas.

Tabela 5.4. Tamanho das mensagens de *offloading* no sistema Android

	E250		**E750**		**E1500**	
	Download	**Upload**	**Download**	**Upload**	**Download**	**Upload**
SA	12738	12904	37738	37904	75328	75404
SM	11028	11081	33028	33081	66028	66081

O sistema de serialização manual, de acordo com a Tabela 5.4, transmite por volta de 13% menos *bytes*, do que o sistema automático, durante uma operação de *offloading*. Isso permite que o SM possua uma vazão maior de rede, do que o SA, além de processar mensagens menores e não utilizar de processos da programação reflexiva. A Figura 5.13 apresenta a média da vazão do *offloading*, para os dois sistemas de serialização, baseados nas médias dos FPS, que são obtidos durante cada intervalo da execução do experimento E250 e E1500, em relação ao ambiente de execução do Ci5Desk.

[22] http://docs.oracle.com/javase/tutorial/jndi/objects/serial.html
[23] Esse *DataStream* é relacionado com as *DataOutputStream* e *DataInputStream* da API do Java e Android.

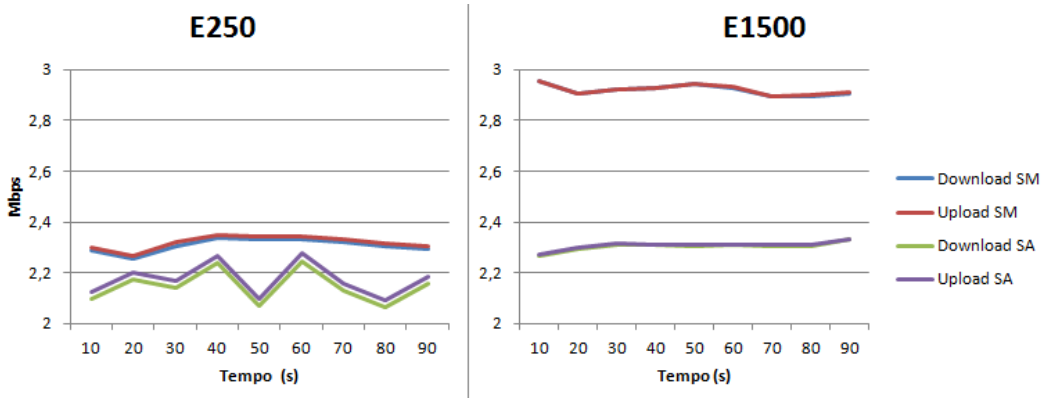

Figura 5.13. Vazão do *offloading* durante a execução do aplicativo no Android

Como se pode observar, a vazão disponível na rede sem fio é bem maior, do que a vazão utilizada pela técnica de *offloading*, sendo neste caso limitado pela capacidade de processamento para serializar e deserializar as informações, que são transmitidas entre as partes. Como o sistema SM utiliza menos recursos computacionais, ele se mantém sempre acima do sistema SA (em termos de vazão) na ordem de 7% para teste do E250 e acima de 25% para o teste do E1500.

Para finalizar a análise com relação à plataforma Android, a máquina representada pelo *cloudlet server* Ci7Mob, manteve sempre nos três testes executados, um desempenho inferior a 10%, quando comparado com o mesmo sistema de serialização empregado no Ci5Desk. Assim, em caso de ausência do *cloudlet* Ci5Desk, o Ci7Mob poderá ser utilizado para realizar a operação de *offloading*, com pouca penalidade no desempenho, sobre a execução do aplicativo *Collision* em 750 e 1500 esferas.

5.3.3.2 Windows Phone

Este experimento realizou na plataforma do Windows Phone, as mesmas atividades que foram definidas na plataforma Android. Desse modo, a Figura 5.14 apresenta os resultados coletados, para cada quantidade de esferas e ambiente de execução. No caso os ambientes (Ci7Mob e Ci5Desk) estão também associados com o tipo de serialização que foi empregada no momento da operação de *offloading* podendo ser automática (SA) ou manual (SM).

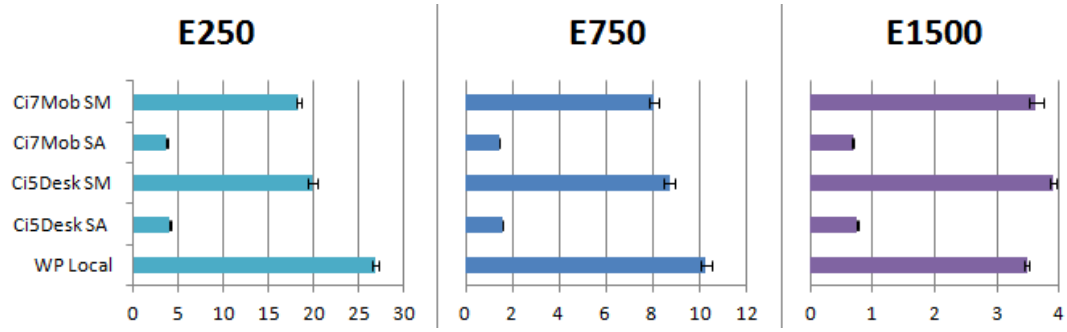

Figura 5.14. Resultado da Execução do *Collision* no Windows Phone em FPS

Os resultados da Figura 5.14 mostram que a execução local da aplicação ganhou em dois cenários E250 e E750, com as respectivas vantagens de 34.5% e 17.8%, quando comparado com o melhor ambiente de execução remoto Ci5Desk usando o sistema de SM. Porém, no resultado de 1500 esferas, a operação de *offloading*, teve uma vantagem de 12.5%, sobre a execução local no dispositivo móvel, usando as mesmas condições de AER anteriores.

O sistema de serialização automático, por sua vez teve o pior resultado, em relação a todos os cenários realizados, independente do ambiente de execução remoto utilizado. Este fato se dá pela utilização do formato BSON usado para serializar as informações que estão envolvidas na operação de *offloading* de forma portável no Windows Phone e C#. Este formato, como visualizado na Tabela 5.5, produz grandes mensagens por chamada de *offloading*, inviabilizando assim, seu uso em aplicativos de tempo-real.

Tabela 5.5. Tamanho das mensagens de *offloading* no sistema Windows Phone

	E250		**E750**		**E1500**	
	Download	**Upload**	**Download**	**Upload**	**Download**	**Upload**
SA	49074	49186	147074	147186	294574	294686
SM	10255	10319	30755	30819	61505	61569

A Tabela 5.5 mostra que a SA produz mensagens 4.8 vezes maior, do que o SM, impactando diretamente (em termos de processamento) no desempenho do aplicativo móvel conforme apresentado na Figura 5.14. O sistema de serialização manual na plataforma do Windows Phone funciona igualmente como foi descrito na plataforma Android, mudando apenas o tipo de *stream* utilizado nesta plataforma, denominado de *BinaryStream*[24]. Este sistema de serialização possui uma maior vazão de rede (por volta de 7%), quando comparado com o sistema da serialização automática, de acordo com a Figura 5.15. Esta figura também apresenta a média da vazão do *offloading* para os dois sistemas de serialização baseados na média dos FPS que são capturados, durante cada intervalo da execução dos experimentos E250 e E1500, em relação ao ambiente de execução do Ci5Desk.

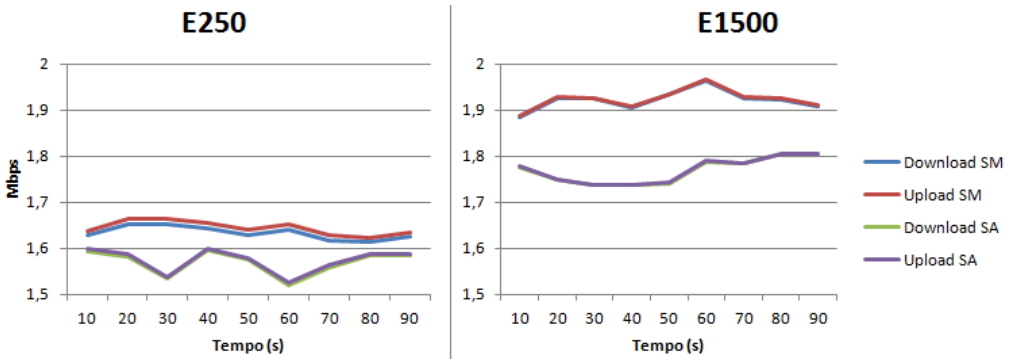

Figura 5.15. Vazão do *offloading* durante a execução do aplicativo no Windows Phone

[24] Composta pelas classes BinaryReader e BinaryWriter.

É importante ressaltar que a SM possui um menor tamanho de mensagem, do que o sistema SA, porém não possui uma vazão expressivamente maior, por causa da limitação da capacidade de processamento para realizar este processo de serialização. Caso deseje aumentar o desempenho do aplicativo móvel, outras estratégias devem ser adotadas, por exemplo, mudar os tipos dos atributos do objeto; para reduzir o tamanho das mensagens de *offloading* trocadas entre o cliente e o servidor.

Por fim, o *cloudlet* Ci7Mob ainda continua sendo mais lento do que Ci5Desk, na ordem de 7.5%, quando comparado com o mesmo sistema de serialização usado nos dois ambientes de execução remotos.

5.4 Considerações Finais do Capítulo

Este capítulo apresentou duas aplicações, denominadas de *BenchImage* e *Collision*, que fizeram uso do *framework* MpOS para suportar a técnica de *offloading*. Os experimentos que foram realizados no aplicativo *BenchImage* foram executados novamente utilizando agora uma nova versão desse aplicativo integrado com o *framework* MpOS, ao invés de fazer a operação de *offloading* manualmente (sem *framework* nenhum), como foi apresentado no artigo da SBRC 2014 [Costa *et al.* 2014].

Os resultados apresentados neste capítulo mostram que a execução da plataforma Android foi acelerada 14.3 vezes (usando o AER Ci5Desk e foto 8MP), quando comparado com a execução local do mesmo processo no *smartphone*. Usando uma Internet móvel 4G e uma instância da Amazon EC2 Large na nuvem pública, o desempenho do aplicativo no mesmo cenário (usando foto de 8MP) é acelerado 6 vezes. No caso da plataforma do Windows Phone sob as mesmas situações o ganho da técnica de *offloading* fica por volta de 57% quando executada no mesmo *cloudlet server* e perde em desempenho na situação da nuvem pública acessada pela rede 4G. Então os experimentos realizados no *BenchImage*, mostram que a rede é um fator determinante (além do poder de processamento do AER) para o desempenho da solução de *offloading*, indo de encontro com outros estudos que realizaram experimentos similares, usando a Internet móvel 3G em comparação com *cloudlet*, conforme encontrados nos trabalhos do MAUI [Cuervo *et al.* 2010], CloneCloud [Chun *et al.* 2011] e ThinkAir [Kosta *et al.* 2012].

Apenas para fins de documentação, quando é comparada a execução local entre as plataformas móveis nesta mesma aplicação do *BenchImage*, a plataforma do Windows Phone é em média 2.7 vezes mais rápida, do que a execução local no Android. A documentação da Microsoft

explica[25] que a execução de um aplicativo na plataforma do Windows Phone 8 é nativa, através da execução de imagem nativa. Enquanto, na plataforma do Android, segundo os autores [Mednieks *et al.* 2012], a execução do aplicativo ocorre em cima de uma máquina virtual, denominada de Dalvik, podendo ocorrer perda de desempenho para grandes exigências de processamento. É importante destacar que o aplicativo *BenchImage* foi desenvolvido *single-thread*, ou seja, utiliza apenas um *core* do processador dos dispositivos móveis, para realizar suas operações.

O aplicativo *Collision* mostrou em seus experimentos, como seria o desempenho da técnica de *offloading*, associado com uso de aplicações de tempo-real. Neste caso, o *offloading* não foi realizado em uma nuvem pública, pois esta classe de aplicação é altamente dependente da latência da rede, sendo executado apenas localmente nos *cloudlet servers* disponíveis em uma rede local sem fio. Os experimentos mostram que a técnica de *offloading* é ineficiente para pequenas quantidades de esferas (E250), sendo neste caso mais eficiente executar localmente. Contudo, na plataforma Android para grandes quantidades de esferas (E1500) a técnica de *offloading* pode acelerar mais de 2.5 vezes, dependendo do sistema de serialização que foi utilizado. Também é mais vantajoso, se o desenvolvedor adotar o sistema manual de serialização, por reduzir o custo computacional para serializar as informações no momento da operação de *offloading*. Contudo, apenas na plataforma do Windows Phone o desempenho do aparelho influenciou negativamente na técnica de *offloading*, porém esta técnica ganhou com uma pequena vantagem para a execução de 1500 esferas usando a serialização manual.

[25] http://msdn.microsoft.com/en-us/library/windows/apps/jj585401(v=vs.105).aspx

Capítulo 6
Conclusões

O tema *Mobile Cloud Computing* surgiu após a computação em nuvem e na "explosão" da utilização dos *smartphones*, sendo seguido com a chegada dos *tablets*. Apesar das diversas evoluções em termo de *hardware*, em alguns casos como apresentado nesta dissertação, os dispositivos móveis podem ser bem mais lentos, do que, um *notebook* dependendo da carga de processamento que foi exigida pela aplicação móvel. Neste contexto surgiu o paradigma do MCC, no qual tenta contornar as limitações dos dispositivos móveis através da exportação de processamento (*offloading*) para outros ambientes de execução.

As próximas seções deste capítulo descrevem na sequência as contribuições deste trabalho, discute as limitações do mesmo e apresenta sugestões de trabalhos futuros.

6.1 Contribuições

Esta dissertação apresentou um *framework* para a solução de *offloading* em múltiplas plataformas móveis (Android e Windows Phone), denominado de MpOS, que realiza a operação de *offloading*, através da marcação dos métodos, os quais são passíveis para realizar esta operação.

O trabalho contribuiu para as questões arquiteturais e de implementação de um *framework* desta natureza, além de explicar com detalhes, como cada componente funciona, conforme descrito no Capítulo 4, auxiliando também, aqueles pesquisadores que desejam implementar esta solução para outras plataformas, ou mesmo, estender as funcionalidades existentes do MpOS.

O MpOS apresenta um sistema próprio para descoberta de serviço e descoberta de *cloudlet* em uma mesma rede sem fio, no qual está os dispositivos móveis. O *framework* também suporta realizar a implantação de um serviço de *offloading* de forma automática em servidores remotos e *cloudlets*. Este trabalho desenvolveu um sistema de RPC próprio que utiliza de dois sistemas de serialização dos dados, sendo um automático e outro manual. Os experimentos demonstraram que a serialização manual traz ganho de desempenho para a técnica de *offloading*, quando se compara com a serialização automática. O sistema de *offloading* também suporta duas formas de particionamento, sendo uma estática e outra dinâmica de acordo com as condições da rede.

Por fim, durante o processo de desenvolvimento do *framework* MpOS para a plataforma do Windows Phone foi necessário desenvolver diversos componentes de apoio, como as classes Proxy, AsyncTask, dentre outras classes, que não existiam nesta plataforma móvel. O desenvolvimento dessas classes viabilizou a construção do *framework* no Windows Phone, permitindo que essa plataforma tenha funcionalidades equivalentes à versão do Android. É importante ressaltar que essa classe Proxy é produzida dinamicamente através de um gerador de código. Esse gerador é um componente do Visual Studio, que permite escanear todas as interfaces do projeto da aplicação do Windows Phone, e produz como resultado essa classe Proxy que permite gerar *proxy* de objetos, a partir de qualquer interface criada pelo desenvolvedor neste projeto da aplicação móvel.

6.2 Limitações

A primeira limitação desse trabalho é a exigência que os métodos marcados para a operação de offloading tenha algum tipo de retorno, exigindo algumas vezes que seja feita mudança na lógica do funcionamento da aplicação.

A serialização automática na plataforma móvel do Windows Phone suporta apenas serialização de tipos primitivos, vetores e listas, não suportando no caso o tipo dicionário e outras possíveis estruturas de dados. Como o processo de serialização utiliza o BSON para serializar as informações que são trocadas com o servidor, pode ter como consequência produzir mensagens grandes, além de reduzir o desempenho da operação de *offloading*.

O trabalho também não trata da questão do provisionamento sob demanda de recursos que estão disponíveis em uma nuvem pública ou em um *cloudlet server* implantado em uma infraestrutura virtualizada. Também não foram feitos experimentos com mais dispositivos móveis para saber o impacto de diversos usuários realizando a operação de *offloading* sobre um determinado servidor remoto.

6.3 Publicações

Durante o mestrado foi possível realizar duas publicações em âmbito nacional, todas relacionadas com a proposta da dissertação, conforme mostrada na Tabela 6.1. A primeira, com Qualis B3, apresenta uma proposta do *framework* MpOS em um Workshop de Teses e Dissertações (WTD). Enquanto a segunda, de Qualis B2, mostra uma análise do impacto do 4G Brasileiro na utilização de *cloudlet*, sendo apresentado na SBRC 2014.

Tabela 6.1. Lista de artigos publicados

Título	Autores	Conferência	Qualis
ltiplas Plataformas.	Philipp B. Costa, Fernando A. M. Trinta, José N. de Souza	19th Brazilian Symposium on Multimedia and the Web (WebMedia), 2013, Salvador.	B3
Uma Análise do Impacto da Qualidade da Internet Móvel na Utilização de Cloudlets	Philipp B. Costa, Paulo A. L. Rego, Emanuel F. Coutinho, Fernando A. M. Trinta, José N. de Souza	XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), 2014, Florianópolis.	B2

6.4 Trabalhos Futuros

Este trabalho poderá servir de base para novos estudos, por exemplos:

- Realizar estudos sobre a mobilidade da técnica de *offloading* entre *doudlet* e nuvem pública. Por exemplo, quando uma operação de *offloading* é iniciada a partir de um *cloudlet* e o usuário por algum motivo precisou sair da área de cobertura da rede Wi-Fi, mas deseja receber o resultado do *offloading* via Internet móvel de forma transparente em um ambiente aberto, sem retornar para Wi-Fi onde iniciou o processo;
- Desenvolver soluções que tratem da questão da escalabilidade do ambiente de execução remoto, conforme apresenta o trabalho do ThinkAir, Kosta *et al.* (2012), visando otimizar também o uso dos recursos externos, em um ambiente de nuvem pública ou em um *cloudlet server* de grande porte que seja virtualizado;
- Realizar estudos de como adicionar mecanismos de segurança relacionados com autenticação e conexão segura, em relação à operação de *offloading*, impactando minimamente em termos computacionais e no tamanho das mensagens que precisam ser trocadas entre os clientes e os servidores;
- Considerar no processo de decisão de uma operação de *offloading*, os custos envolvidos no consumo do plano de dados da Internet móvel, quando o *offloading* for direcionado para uma nuvem pública;
- Criar um *plugin* no Eclipse e Visual Studio que facilite configurar o *framework* MpOS, além de facilitar a geração das dependências dos aplicativos móveis, sendo necessário para realizar o procedimento de Implantação de Serviço de *offloading* no servidor remoto;
- Disponibilizar uma versão do *framework* MpOS para a plataforma móvel do iOS;

- Criar um portal na *web* bilíngue (português e inglês), para expor o *framework* proposto com sua documentação, visando aumentar a visibilidade do trabalho e permitir que a comunidade científica possa comparar ou contribuir com MpOS.

Referências Bibliográficas

Bahl, P., Han R. Y., Li E. L., Satyanarayanan, M. (2012). Advancing the state of mobile cloud computing. In Proceedings of the third ACM workshop on Mobile cloud computing and services (MCS '12), pp. 21-28.

Chun, Bg., Ihm, S., Maniatis, P., Naik, M., Patti, A. (2011). CloneCloud: elastic execution between mobile device and cloud. Proceedings of the 6th conference on Computer systems (EuroSys '11), New York, USA.

Costa, P. B., Rego, P. A. L., Coutinho, E. F., Trinta, F. A. M., Souza, J. N. (2014). Uma Análise do Impacto da Qualidade da Internet Móvel na Utilização de Cloudlets. In XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distrivuidos (SBRC 2014), Florianópolis, Santa Catarina, Brasil.

Cuervo, E., Balasubramanian, A., Cho, Dk., Wolman, A., Saroiu, S., Chandra, R., Bahl, P. (2010). Maui: Making *Smartphones* Last Longer with Code Offload. In Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys 2010), San Francisco, California, USA.

Dean, J., Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, USA.

Demichelis, C. and Chimento, P. (2002). RFC 3393: IP Packet Delay Variation Metric for IP Performance Metrics (IPPM).

Dinh, H. T., Lee, C., Niyato, D., Wang, P. (2011). A survey of mobile cloud computing: architecture, applications and approaches. Wireless Communication and Mobile Computing.

Droms, R. (1997). RFC 2131: Dynamic Host Configuration Protocol (DHCP).

Fernando, N., Loke, W. S., Rahayu, W. (2013). Mobile cloud computing: A survey. Future Generation Computer Systems, Elsever, v. 29, pp. 84-106.

Flinn, J., Narayanan, D., Satyanarayanan, M. (2001). Self-tuned remote execution for pervasive computing. Hot Topics in Operating Systems, 2001. In Proceedings of the Eighth Workshop, pp. 61-66.

Forman, I. R., Forman, N. (2004). Java reflection in action. Manning Publications.

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design patterns: elements of reusable object-oriented software. Pearson Education.

Giurgiu, I., Riva, O., Juric, D., Krivulev, I., Alonso, G. (2009). Calling the cloud: enabling mobile phones as interfaces to cloud applications. Proceedings of the ACM/IFIP/USENIX 10th international conference on Middleware (Middleware'09), Urbana, USA.

Ha, K., Pillai, P., Richter, W., Abe, Y., Satyanarayanan, M. (2013). Just-in-time provisioning for cyber foraging. Proceeding of the 11th annual international conference on Mobile systems, applications, and services (MobiSys'13), Taipei, Taiwan.

Huang, J., Qian, F., Gerber, A., Mao, Z. M., Sem, S., and Spatscheck, O. (2012). A close examitaion of performance and Power characteristics of 4g lte networks. In MobiSys 2012, proceedings ACM, pp. 225-238.

Hugles, J. F., Van Dam, A., McGuire, M., Sklar, D. F., Foley, J. D., Feiner, S. K., Akeley, K. (2013). Computer Graphics: Principles and Practice. Addison-Wesley Professional, vol 3.

Kenney, M., Pon, B. (2011). Structuring the Smartphone Industry: Is the Mobile Internet OS Platform the Key?. Journal of Industry, Competition and Trade, Springer, vol. 11, no. 3, pp. 239-261.

Kosta, S., Aucinas, A. Hui, P., Mortier, R., and Zhang, X. (2012). ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In INFOCOM, 2012 Proceedings IEEE, pp. 945-953.

Kristensen, M. D., Bouvin, N. O. (2010). Scheduling and development support in the Scavenger cyber foraging system. Pervasive and Mobile Computing, vol. 6, no. 6, pp. 677-692.

Kumar, K., Liu, J., Lu, Y.-H., and Bhargava, B. (2013). A survey of computation offloading for mobile systems. Mobile Networks and Applications, pp. 129-140.

Kurose, J. F., Ross, K. W. (2012). Computer Networking: A top-down approach featuring the Internet. Addison-Wesley Reading, vol 6.

Qi H., Gani, A. (2012). Research on mobile cloud computing: Review, trend and perspectives. Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on, pp.195-202.

Marinelli, E. E. (2009). Hyrax: Cloud Computing on Mobile Devices using MapReduce. MS thesis, Carnegie Mellon University, Pittsburgh, USA.

Mednieks, Z., Dornin, L., Meike, G. B., Nakamura, M. (2012). Programming Android. O'Reilly Media, Inc.

Sanaei, Z., Abolfazli, S., Gani, A., Buyya, R. (2013). Heterogeneity in Mobile Cloud Computing: Taxonomy and Open Challenges. Communications Surveys & Tutorials, IEEE , vol. PP, no. 99, pp. 1-24.

Satyanarayanan, M. (2001). Pervasive computing: vision and challenges. Personal Comunications, IEEE, pp. 10-17.

Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N. (2009). The Case for VM-Based *Cloudlet*s. In Mobile Computing. Pervasive Computing, IEEE, vol. 8, pp. 14-23.

Shiraz, M., Gani, A., Khokhar, R. H., Buyya, R. (2013). A Review on Distributed Application Processing *Framework*s in Smart Mobile Devices for Mobile Cloud Computing. Communications Surveys & Tutorials, IEEE , vol.PP, no.99, pp. 1-20.

Thurlow, R. (2009). RFC5531: Remote Procedure Call Protocol Specification Version 2 (RPC).

Whitechapel, A., McKenna, S. (2013). Windows Phone 8 Development Internals. Pearson Education.

Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z., Yang, L. (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software CodeSign and System Synthesis, pp. 105-114.