Dissertação

Mestrado em Engenharia Informática – Computação Móvel

Identificação, Análise e Avaliação de Linguagens de Programação Adequadas ao Desenvolvimento de Agentes Móveis Multi-Plataforma

---

Paulo Manuel Pinto Reis

Dissertação de Mestrado realizada sob a orientação do Doutor Nuno Alexandre Ribeiro Costa, Professor da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria.

Leiria, *setembro* de *2014*

Esta página foi intencionalmente deixada em branco

À minha Família.

Esta página foi intencionalmente deixada em branco

Agradecimentos

Embora uma dissertação seja um trabalho individual ela contém sempre contributos de natureza diversa que devem ser salientados. Por essa razão, desejo expressar os meus sinceros agradecimentos.

Ao Professor Doutor Nuno Costa, meu orientador, pela competência científica e acompanhamento do trabalho, pela disponibilidade e generosidade reveladas ao longo deste ano de trabalho, assim como pelas críticas, correções e sugestões relevantes feitas durante a orientação.

Ao Professor Doutor António Pereira, pela motivação e conselhos que me deu num momento em que a palavra "desistir" ecoava no meu pensamento.

Ao "camarada" de mestrado Pedro Martins, pela sua generosa entrega pessoal e profissional.

Quero prestar um agradecimento especial a Sílvia Oliveira, pelo incentivo e serenidade nos momentos mais difíceis.

Aos meus pais, que tornaram possível a minha educação. Sem eles não teria chegado aonde cheguei.

Esta página foi intencionalmente deixada em branco

Resumo

Agentes Móveis (AM) são programas autónomos que podem viajar sob o seu próprio controlo dentro de uma grande rede de computadores, de computador para computador, realizando uma tarefa em nome de um utilizador. Podem ainda fornecer uma estrutura conveniente, eficiente e robusta para a implementação de aplicações distribuídas, incluindo aplicações móveis.

Dificuldades de instalação e de configuração, ocorrência de erros não documentados, funcionamento estável apenas para determinados sistemas operativos, entre outros, são algumas das lacunas identificadas nas várias *frameworks* desenvolvidas para suportar aplicações de AM. A estas, a popularização dos dispositivos móveis veio acrescentar a necessidade de compatibilidade com os seus ambientes computacionais.

A identificação de uma ou várias linguagens de programação, potencialmente indicadas para implementação de AM multiplataforma, é o objetivo principal deste trabalho, para o qual foi elaborado um conjunto de testes centrados na análise de suporte ao nível da serialização de dados e do carregamento dinâmico, características intrínsecas ao funcionamento dos AM. A implementação de um algoritmo nas diversas linguagens de programação candidatas, recorrendo a técnicas de desenvolvimento nativo, foi executado em diversos sistemas operativos multitarefa e móveis, tendo sido identificadas algumas das suas principais caracteristicas funcionais, vantagens e desvantagens.

Do estudo levado a cabo, concluí-se que a linguagem Python apresentou o melhor suporte nos testes realizados, estando disponível, assim como o seu interpretador, para as mais diversas plataformas. Python disponibiliza, na sua biblioteca padrão, os módulos necessários a uma implementaçao assente no paradigma de AM, possibilitando a sua utilização como linguagem embutida em outras aplicações, nomeadamente em aplicações móveis.

Palavras-chave:Agentes Móveis, Sistemas de Agentes Móveis, Linguagens de Programação, Serialização de Dados, Carregamento Dinâmico, Desenvolvimento Nativo.

Esta página foi intencionalmente deixada em branco

Abstract

Mobile Agents (MA) are autonomous programs that can move under their own control within a large computer network, from computer to computer, performing a task on behalf of a user. They can provide a convenient, efficient, and robust framework for implementing distributed applications including mobile applications.

Installation and configuration difficulties, undocumented errors, stable operation only for certain operating systems, among others, are some of the identified gaps in the various frameworks developed to support MA applications. To these, the popularity of mobile devices has added the need for compatibility with their computing environments.

The identification of one or more programming languages, potentially suitable for implementing multiplatform MA, is the main objective of this work, for which we designed a set of tests focused on support analysis in terms of data serialization and dynamic loading, intrinsic characteristics of the MA. By implementing an algorithm on different candidates programming languages, using native development techniques, to be executed in multitasking operating systems and mobile operating systems, it was possible to identify some of its main functional characteristics, advantages and disadvantages.

According to the study carried out, it can be concluded that the Python programming language had the best support in the performed tests, being available, as well its interpreter, for various platforms. The Python standard library provides the necessary modules for an implementation based on the MA paradigm, allowing to be used as embedded language in other applications, particularly in mobile applications.

Key-Words: Mobile Agents, Mobile Agents Systems, Programming Languages, Data Serialization, Dynamic loading, Native Development.

Esta página foi intencionalmente deixada em branco

Índice de Figuras

Índice de Tabelas

Esta página foi intencionalmente deixada em branco

Lista de Siglas

ACL	Agent Communication Language
AFME	Agent Factory Micro Edition
AMS	Agent Management System
ANSI	American National Standards Institute
API	Application Programming Interface
ARA	Agents for Remote Action
ART	Android Runtime
CGI	Common Gateway Interface
CORBA	Common Object Request Broker Architecture
CPAN	Comprehensive Perl Archive Network
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DF	Directory Facilitator
DVM	Dalvik Virtual Machine
FIPA	Foundation for Intelligent Physical Agents
GPL	General Public License
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
iOS	iPhone Operation System
J2ME	Java Platform Micro Edition
JADE	Java Agent DEvelopment framework
JAR	Java ARchive
JCL	Job Control Language
JDK	Java Development Kit
JME	Java Micro Edition
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LEAP	Lightweight and Extensible Agent Platform
MASIF	Mobile Agent System Interoperability Facility
MIDP	Mobile Information Device Profile
MTS	Message Transport Service
NDK	Android Native Development Kit
OHA	Open Handset Alliance
OMG	Object Management Group
PDA	Personal Digital Assistant

PHP	Hypertext Preprocessor
RPC	Remote Procedure Calling
SDK	Software Development Kit
SL4A	Scripting Layer for Android
SMA	Sistema de Agentes Móveis
SO	Sistema Operativo
SOMA	Secure and Open Mobile Agent
SPRINGS	Scalable PlatfoRm for movINg Software
TACOMA	Tromsø And COrnell Moving Agents
TCL	Tool Command Language
VM	Virtual Machine

Índice

Capítulo 1

1. Introdução

O elevado desenvolvimento nas tecnologias de sistemas distribuídos e de rede, motivadas pelo uso generalizado do acesso à Internet, veio aumentar a necessidade de serem criadas aplicações com potencialidades de troca eficaz de informação entre redes heterogéneas.

Numa aplicação distribuída, onde sejam necessários dados de diferentes sistemas para executar a mesma tarefa, por vezes é necessário transferir grandes volumes de dados entre os sistemas envolvidos. Neste campo, a mobilidade de código pode ser considerada uma solução alternativa para o projeto e implementação de sistemas distribuídos.

No contexto de mobilidade de código uma importante área de pesquisa é a tecnologia de Agentes Móveis (AM). Genericamente, um pequeno programa migra para um sistema heterogêneo, ou não, executa uma determina tarefa nesse sistema e regressa com resultados obtidos para a origem, sendo possível visitar vários sistemas remotos antes do retorno à origem.

O paradigma da computação distribuída baseado em AM caracteriza-se por levar a computação até ao destino em vez de trocar dados através da rede, tal como acontece no paradigma cliente/servidor. Contudo, nos últimos anos, temos assistido a um abandono deste paradigma da computação em detrimento do modelo cliente/servidor baseado em *Web Services*, por exemplo. Apesar das razões para este aparente abandono não serem muito claras, é um facto que o *software* de apoio à mobilidade de código, existente e ainda ativo, apresenta vários entraves à adoção deste paradigma de computação distribuído, por apresentar dificuldades de instalação e de configuração, ocorrência de erros não documentados e funcionamento estável apenas para determinados sistemas operativos. Outra razão aparente poderá ser o aumento de débito das redes e o baixo custo no acesso à Internet. A somar a todas estas razões acresce ainda o problema da execução em segurança do AM.

Recentemente, a área de computação móvel tem tido a uma crescente e rápida evolução, motivada principalmente pela popularização de um vasto e diversificado conjunto de dispositivos móveis que apresentam enormes capacidades de processamento e conectividade, introduzindo assim novos desafios no desenvolvimento de aplicações e prestação de serviços, podendo os AM, agora, serem utilizados por estas plataformas. A implementação de uma *framework* leve e flexível de suporte a AM multiplataforma (e.g., *Windows*, *Linux*, *Mac OS X*, *Android* e *iOS*) assume-se claramente como algo desafiante. Quer ao nível do potencial computacional daí resultante, principalmente devido à existirem várias *frameworks* de MA mas nenhuma com suporte aos sistemas operativos mais usados, quer ao nível de constrangimentos presentes nos sistemas operativos móveis, nomeadamente, durante a respetiva fase de desenvolvimento de uma aplicação assente neste paradigma.

1.1. Objetivos propostos

O principal objetivo desta dissertação é apresentar um estudo detalhado de identificação, análise e avaliação de Linguagens de Programação potencialmente indicadas ao desenvolvimento de Agentes Móveis multiplataforma.

Neste estudo são também abordadas algumas *frameworks* de desenvolvimento de sistemas de Agentes Móveis (SMA) existentes, para as quais foi realizada a caracterização e análise comparativa. Esta abordagem visou identificar sistemas operativos suportados pelos diversos

SMA como também efetuar um levantamento das linguagens de programação maioritariamente usadas no respetivo desenvolvimento.

O trabalho aqui proposto centra-se num conjunto de testes realizados em diferentes ambientes computacionais: *Windows*, *Linux*, *Mac OS X*, *Android* e *iOS*. Os testes compreendem a avaliação de suporte de duas características chave por parte das linguagens de programação disponíveis: i) capacidade de serialização de dados e ii) carregamento dinâmico de classes/objetos, de modo a possibilitar uma implementação do paradigma de MA.

Na realização deste estudo, optou-se por seguir uma linha de implementação comum às linguagens alvo nos diferentes ambientes computacionais, procurando deste modo estabelecer um cenário de testes em que os resultados obtidos fossem facilmente comparáveis. O desempenho das linguagens, concretamente ao nível de execução de *scripting*, sobrepôs-se à interação com o utilizador e a aspetos mais específicos no desenvolvimento de aplicações, tais como: a facilidade de construção de aplicações, uso de recursos de arrastar e soltar (*drag and drop*) no desenho de GUI, etc..

Devido a optar-se por um desenvolvimento nativo, não se recorreu à utilização de *frameworks* de abstração de plataforma, em particular de ferramentas de desenvolvimento *cross-platform* existentes para *Android* e *iOS*. Estas ferramentas apresentam a vantagem de reduzir custos e aumentar a rapidez com que os aplicativos são desenvolvidos. São geralmente bastante simples de usar por serem baseadas em linguagens comuns para *scripting*, nas quais se inclui o CSS, HTML e JavaScript. No entanto, as ferramentas de desenvolvimento *cross-platform* para aplicações móveis tem algumas desvantagens. Em primeiro lugar, os sistemas operativos móveis são atualizados com frequência. Sempre que um sistema operativo móvel recebe uma nova atualização, o aplicativo também deve ser atualizado para ser compatível com o novo sistema. Além disso, a aplicação gerada tem probabilidade de ser mais lenta a executar, traduzindo-se numa menor experência de utilização por parte do utilizador. O tempo de renderização de código também pode ser maior, uma vez que cada sistema operativo precisa de um conjunto separado de código necessário à operação.

Para além do objetivo principal, pretende-se que esta dissertação sirva de introdução e referência àqueles que desejem aprofundar o estudo de sistemas de AM, em particular na implementação de uma *framework* de suporte a AM multiplataforma. Assim, a intenção é

redigir um texto relativamente auto-contido e suficientemente abrangente, que apresente um conjunto de informação pesquisada sobre a área de estudo em questão. É também disponibilizado um conjunto de anexos com informação útil para complementar a compreensão da informação presente no texto.

1.2. Estrutura do documento

Para além do presente capítulo onde se enquadra o presente estudo e se efetua a definição dos objetivos propostos, este documento encontra-se dividido em cinco capítulos.

O capítulo 2 apresenta o trabalho relacionado na área de AM e *frameworks* existentes. O estado da arte inclui a investigação e recolha de informação sobre conceitos teóricos, padrões para implementação e sobre as principais *frameworks* existentes para implementação e gestão de AM. No final do capítulo é apresentado um resumo comparativo que inclui o estado atual de disponibilização/distribuição das *frameworks* e sistemas operativos suportados.

O capítulo 3 apresenta conceitos gerais e um estudo das linguagens de programação Java, Python, Lua e Perl, maioritariamente usadas no desenvolvimento de AM. É realizada uma análise comparativa das linguagens alvo do estudo e uma visão geral sobre a respetiva usabilidade em sistemas operativos móveis, pois, *Android* e *iOS* são plataformas visadas neste estudo, sendo por isso necessário encontrar uma linguagem que seja também comum a ambas.

Posteriormente, no capítulo 4, são apresentados cenários de testes de serialização de dados e de carregamento dinâmico de código, os respetivos objetivos e os resultados obtidos em cada ambiente computacional.

Por último, no capítulo 5, são apresentadas as conclusões sobre todo o trabalho desenvolvido no âmbito desta dissertação e o trabalho futuro em perspectiva.

Fazem parte deste documento os seguintes anexos:

- Anexo A: código fonte implementado para a linguagem Java;
- Anexo B: código fonte implementado para a linguagem Python;
- Anexo C: código fonte implementado para a linguagem Lua;
- Anexo D: código fonte implementado para a linguagem Perl.

Capítulo 2

2. Revisão da literatura

Neste capítulo são apresentados trabalhos relacionados com o tema abordado nesta dissertação. Inicialmente serão apresentados conceitos gerais sobre Agentes de Software, mobilidade de código e Agentes Móveis (AM), sendo também feita uma breve descrição sobre a implementação de Sistemas de AM. Segue-se a apresentação dos padrões existentes para a implementação de AM. São ainda referidos alguns trabalhos e conceitos sobre *frameworks* existentes e um resumo comparativo das mesmas. No final é apresentada uma síntese sobre o estado de arte aqui apresentado.

2.1. Agente de Software

Em termos computacionais, um *agente* pode ser definido como um *software* capaz de executar uma tarefa complexa em nome de um utilizador. Entre outras definições encontradas, um Agente de Software distingue-se de um normal programa para computador devido à capacidade de observar e avaliar o estado atual do ambiente onde é executado, decidir como agir com base nessas informações e, em seguida, executar uma ação correspondente [1].

De uma forma geral, um utilizador pode especificar um conjunto de tarefas e objetivos que o agente deve realizar (e.g., pesquisar, filtrar e obter informação; comprar ou vender um determinado produto; etc.). O agente será responsável por suportar a delegação dessas tarefas, gerir a respetiva complexidade, adaptar-se ao utilizador, adaptar-se ao seu meio, aprender de forma adaptativa, etc…

Existem inúmeras definições e interpretações para o termo agente e todas apresentam em comum a característica funcional deste não estar sujeito ao controlo contínuo e direto do utilizador durante a sua execução [1][2][3][4].

Para tipificar um agente existe também um conjunto mínimo de características comuns. Podemos, deste modo, caracterizar um agente como [1][2][3][5]:

- Autónomo, pode funcionar como um processo autónomo que tem controlo sobre as suas ações e o seu estado;

- Comunicativo (ou Sociável), não só com o utilizador mas também com outros agentes ou outros processos de *software*;

- Percetivo, por ter a capacidade de perceber e responder a alterações no seu ambiente;

- Pró-ativo, não responde simplesmente ao ambiente mas toma a iniciativa exibindo assim um comportamento dirigido a um objetivo;

- Reativo, porque tem a capacidade de reagir às mudanças que sente no ambiente (estímulos);

- Inteligente, devido a usar técnicas do campo da inteligência artificial que o capacita com um bom grau de inteligência e bom senso, o que lhe confere personalidade credível e um estado emocional;

- Temporalmente contínuo, porque ao contrário de outros programas que executam uma ou mais tarefas e terminam, um agente está continuamente em atividade.

Para Franklin e Graesser [1], autonomia, pró-atividade, reatividade e continuidade temporal são características essenciais num agente. De facto, a maioria das discussões sobre agentes concentram-se em torno da sua autonomia, reatividade, pró-atividade, sociabilidade, inteligência, mobilidade e auto-organização.

São vastas as vantagens da sua utilização em diversos ambientes [6]. Alguns exemplos mencionados incluem:

- Na redução do trabalho humano, em que muitas tarefas podem ser executadas mais rapidamente e com mais precisão por um agente;

- Ao lidar com a sobrecarga de informações, pois o agente pode pesquisar automaticamente através de grande quantidade de informação não-estruturada disponível na Web;

- Fornecer ajuda automatizada para um enorme número de utilizadores sem formação numa determinada tarefa/ferramenta.

Normalmente são bem adequados para utilização em aplicações que envolvem a computação distribuída, na automação de processos e fluxos de trabalho, no comércio eletrónico, em aplicações Internet, etc...

Os Agentes de Software, quanto à sua mobilidade, podem ser classificados como *agentes estáticos* ou *agentes móveis*. Agentes estáticos alcançam o seu objetivo através da execução numa única máquina. Os agentes móveis migram, de um computador para outro na rede, sendo executados em várias máquinas. Os agentes estáticos quando necessitam de interagir ou de alguma informação num local remoto, utilizam um mecanismo de mensagens para um agente nesse local com o pedido para a ação. Numa situação semelhante, um agente móvel transfere-se para o local remoto e invocar aí a execução da ação.

2.2. Mobilidade de Código

O termo *mobilidade* é usado para indicar uma mudança de localização realizado pelas entidades de um sistema. Inicialmente esta mudança de localização (*mover*) destinava-se a ser

realizada apenas sobre dados simples, tendo evoluído para outras finalidades, tais como, mover o controlo de execução ou mover o código de execução.

Genericamente, a mobilidade de código permite que um excerto de um programa (código móvel) seja enviado pela rede para ser executado num computador remoto.

O conceito de execução remota teve origem na década de 60 com a linguagem JCL (*Job Control Language*), que permitia, a partir de minicomputadores, submeter *batch jobs* (tarefas executadas sem necessidade de interação com o utilizador) para serem executados em computadores de grande porte [7]. O código movia-se para a máquina e a execução passava a ser local.

Um outro exemplo, de 1984, é o envio de um *shell script* para ser executado numa máquina remota através do comando *rsh* do sistema operativo UNIX. Em 1995, devido ao crescimento das tecnologias para Internet e do sucesso da linguagem Java, na forma de *applets* para dinamização de páginas HTML ou utilizando a tecnologia Jini [8], que possibilita a capacidade de "Plug and Work" a dispositivos de uma rede, a mobilidade de código teve um crescimento promissor.

A mobilidade de código é definida como a capacidade de uma aplicação ser transferida entre diferentes localizações de uma rede e continuar a sua execução a partir do ponto anterior à ocorrência da migração [9]. A partir desta ideia, têm sido criados vários paradigmas de programação no desenvolvimento de *software* para computação distribuída, entre os quais o modelo Cliente/Servidor, *Remote Procedure Call*[1] (RPC) e Agentes Móveis [10].

A mobilidade pode ser implementada em dois modos diferentes: *execução remota* ou por *migração*. Execução remota é o mecanismo que constrói um novo Agente para um local especificado, enquanto migração é o mecanismo que continua a execução corrente de um Agente num local diferente.

[1] *Remote Procedure Call* é uma tecnologia de comunicação entre processos que permite a um programa de computador chamar um procedimento num outro espaço de endereçamento (geralmente num outro computador ligado por uma rede).

Um Agente Móvel (AM) é uma forma específica de código móvel. É definido como um objeto que possuí comportamento, estado e localização, podendo optar por migrar entre computadores em qualquer momento durante a sua execução. Deste modo, quando decide deslocar-se, guarda o seu próprio estado de execução, transporta esse estado e continua a sua execução no próximo *host*.

Em analogia, ao aceder a uma página Web, o utilizador não está realmente a visualizar os conteúdos dessa página diretamente no servidor, mas sim a aceder a uma cópia descarregada para visualização no cliente. No caso do AM, o seu movimento é realizado através da duplicação de dados. Sendo um programa computacional que se desloca entre várias máquinas (computadores), tem a capacidade de conservar o seu estado, ou seja, guardar os valores assumidos pelas suas estruturas de dados, incluindo o contexto de execução.

Durante o tempo de vida do AM, o processo de migração consiste em desativar o agente, capturar seu estado, transportar o agente para um novo local, restaurar o estado do agente, e continuar com a execução do agente. Estratégias diferentes de migração podem ser empregues. Na mobilidade de código é feita a distinção entre duas formas [11] [12]:

- *Mobilidade fraca*, quando é feita a migração de código e dados do agente, ou seja, este apenas guarda a componente estática do seu estado de execução. Assim, a execução do agente na máquina destino será reiniciada sempre no ponto pré-definido antes da transferência.

- *Mobilidade forte*, quando para além da migração de código do agente, dados e estado de execução, também é guardada a componente dinâmica (pilha, registos da máquina, etc.). Deste modo, a execução do agente continua, na máquina destino, a partir da última instrução executada na máquina origem.

2.3. Agente Móvel

Genericamente, um AM é um processo computacional que, de forma autónoma, é capaz de se transferir de uma máquina para outra, através de diferentes arquiteturas de sistemas e diferentes plataformas [3] e continuar a sua execução.

Das diversas definições encontradas, a mais citada pertence ao relatório de pesquisa elaborado por Chess, Harrison e Kershenbaum [13], segundo o qual, "Agentes Móveis são programas tipicamente escritos numa linguagem de *script*, que podem ser executados a partir de um computador cliente e transportados para um computador remoto para nova execução".

Segundo Reddy [2], "Um Agente Móvel é constituído pelo código de programa e o estado de execução do programa (o valor atual de variáveis, próxima instrução a ser executada, etc.), (…) que é guardado e transferido para a nova localização, juntamente com o código do Agente Móvel, para que a execução possa ser retomada no novo destino (*host*)."

Para Milojicic [14], os AM são abstrações de *software* que podem migrar através da rede móvel e que representam os utilizadores em diversas tarefas.

Excetuando a mobilidade, do ponto de vista meramente funcional, existem diferenças que os distinguem dos restantes agentes. Destacam-se seguidamente as mais expressivas:

- Os AM podem tomar algumas decisões por conta própria. Significa que são livres para escolher o próximo anfitrião e quando essa migração deve ocorrer. Estas decisões são tomadas no interesse do utilizador, como tal, são transparentes para este;

- Os AM podem aprender com as experiências e adaptarem-se ao meio ambiente. Podem monitorizar o tráfego em grandes redes e aprender sobre os pontos de conflito na rede. Com base nas experiências do agente na rede, este pode escolher as melhores rotas para chegar ao próximo destino (*host*).

Pode-se, deste modo, encontrar uma interpretação para o conceito de AM: programa em execução, que viaja de um computador para outro invocando serviços locais, e, cumprindo alguma tarefa em nome de um utilizador. Observado de um modo mais arquitetural, um AM contêm: um estado, que podem ser os valores das variáveis de instância ou o seu estado de execução; um código de implementação e de *interface*, que pode ser transferido com o agente, obtido na sua nova localização, ou descarregado a pedido através da rede; um identificador único; e, credenciais de segurança.

É na área da computação distribuída que o interesse científico se tem manifestado na utilização deste modelo de programação, tendo sido encontrados diversos trabalhos na literatura consultada [15][5][2][16], nos quais se identificam as principais razões e vantagens. Estas incluem a melhoria na latência e largura de banda, na robustez e tolerância a falhas, na instalação dinâmica de *software* e na execução assíncrona e autónoma.

A necessidade de serviços extras nas máquinas onde os AM irão ser executados, assim como as preocupações com a segurança, com a robustez e com a interoperabilidade, são algumas das desvantagens identificadas na sua utilização.

É também importante que os AM apresentem flexibilidade suficiente para executarem as suas tarefas em diferentes ambientes de execução, em particular nos dispositivos móveis, de modo que a execução dos agentes não seja afetada por características particulares desses ambientes.

Historicamente, em 1994, o termo "agente móvel" foi introduzido pela linguagem de programação *Telescript*[2] que desenvolveu o conceito de mobilidade ao nível da linguagem de programação [14] [17].

Implementação de Agentes Móveis

Para ser possível implementar serviços baseados em AM é necessária a existência de uma plataforma (*framework*) de *software*, capaz de proporcionar um mínimo de serviços básicos, tais como a capacidade de comunicação entre os agentes e da migração para outros computadores. Esta plataforma pode ser vista como o sistema operativo dos AM.

Na investigação sobre AM, do impacto e interesse que a tecnologia despertou no final da década de 1990 até aos primeiros anos do presente milénio, resultou um vasto e diversificado número de *frameworks*. No entanto, a falta de padrões internacionais era uma barreira para o desenvolvimento de tecnologias para SMA, impedindo a interoperabilidade entre os sistemas.

Com o aparecimento da FIPA (*Foundation for Intelligent Physical Agents*) em 1995, e com as primeiras especificações concluídas em 2002, resultou um grande incentivo para novos

[2] *Telescript* é uma linguagem de programação criada para facilitar o desenvolvimento de programas móveis em redes de computadores, sendo considerada a primeira implementação comercial de um AM.

esforços de criação de *frameworks* genéricas que garantissem a interoperabilidade entre SMAs.

A este período seguiu-se um mais moderado. Atualmente os AM ainda são um importante foco de atenção [18], mas algumas dúvidas surgem, por exemplo, sobre a sua aplicabilidade e desempenho [19]. No entanto, existe um grande interesse de pesquisa na área de sistemas baseados em agentes que agregam as características de mobilidade (*agentes móveis*) e de aprendizagem (*agentes inteligentes*) num mesmo componente de *software*, o qual é definido como *agente móvel inteligente*.

2.4. Padrões para implementação de Agentes Móveis

A *interoperabilidade* é uma característica importante dos SMA. Por interoperabilidade entenda-se a troca de informação e/ou dados através de computadores. É também a capacidade de comunicar, executar programas através de várias unidades funcionais, utilizando-se linguagens e protocolos comuns. No caso dos AM, é importante que os agentes troquem informação e serviços entre si, como também consigam estabelecer formas de negociação e raciocínio. Deste modo, a interoperabilidade entre os vários sistemas de agentes é um objetivo essencial.

A tecnologia de AM começou a ser objeto de pesquisa antes da popularização/massificação da Internet, no final da década de 1990. Nessa época, as diversas *frameworks* existentes para desenvolvimento de AM apresentavam, entre elas, diferenças na prespetiva de implementação e arquitetura, o que dificultava a sua interoperabilidade. Surge assim a necessidade de definir especificações para tornar possível a interação entre os mais diversos sistemas.

Das diversas iniciativas, destaca-se o papel desempenhado pelo *Object Management Group* (OMG) [20] e pela *Foundation for Intelligent Physical Agents* (FIPA) [21] na padronização de MA. No entanto, devido à falta de colaboração entre estas duas organizações, o resultado final foi o desenvolvimento de padrões paralelos e concorrentes [22].

Padrão MASIF

O *OMG* é um consórcio internacional que aprova padrões abertos para aplicações orientadas a objetos e para ambientes distribuídos. Em 1997 apresentou um padrão, denominado MASIF (*Mobile Agent System Interoperability Facilities*) [23], com o propósito de alcançar a interoperabilidade entre plataformas de MA de diferentes fabricantes.

O padrão MASIF especifica um conjunto de conceitos e *interfaces* básicas, procurando deste modo facilitar o desenvolvimento futuro de sistemas de agentes móveis. Esta especificação tem como objetivo permitir que um AM possa deslocar-se entre agências com perfil semelhante (partilhando a mesma linguagem, tipo de agência, tipo de autenticação e método de serialização) através de um conjunto de interfaces CORBA[3] normalizadas.

Uma das críticas apontadas ao padrão MASIF é de não cobrir todas as funcionalidades requeridas para o desenvolvimento de um sistema de agentes mas apenas as relacionadas com a interação entre estes. Para além disso, apenas orienta para interoperabilidade entre sistemas de agentes escritos em Java, o que segundo Kotz [16] *(…) conduz a uma situação em que o agente não pode migrar para a máquina desejada, mas apenas para uma máquina vizinha na qual está em execução o sistema de agente "certo"*.

Padrão FIPA

A FIPA é uma fundação internacional vocacionada para a criação de padrões concretos de comunicação que tornem possível a implementação de agentes abertos e interoperáveis. Desde 2005 que é uma organização da *IEEE Computer Society*.

O trabalho de especificação da FIPA destina-se a facilitar a interoperabilidade entre sistemas de agentes, uma vez que, para além da linguagem de comunicação, especifica também quais os agentes principais envolvidos na gestão de um sistema, a ontologia necessária para a interação entre sistemas, e ainda, o nível de transporte dos protocolos. Possuindo uma arquitetura focalizada na troca de mensagens e interoperabilidade, a especificação FIPA

[3] CORBA (*Common Object Request Broker Architecture*), é uma especificação da OMG para o desenvolvimento de aplicações distribuídas baseadas no modelo orientado a objetos, independentes do sistema operativo e da linguagem de programação adotada para a construção do *software*.

2002-37 [24] dá particular atenção a uma linguagem comum para comunicação entre agentes, denominada ACL - *Agent Communication Language*.

A FIPA também define uma arquitetura de sistema de agentes, não necessariamente móveis, contendo os seguintes elementos fundamentais [25]:

- AMS (*Agent Management System*): Agente responsável pelos serviços de páginas brancas e de gestão da plataforma (autenticação e registo dos agentes);

- DF (*Directory Facilitator*): Agente que disponibiliza o serviço de páginas amarelas;

- MTS (*Message Transport Service*): Serviço de transporte de mensagens que fornece os canais para a troca de mensagens entre os agentes, dentro e fora da plataforma.

Especificamente, o AMS mantém uma lista dos nomes e endereços de todos os agentes da plataforma; o DF mantém uma lista com as descrições de serviços prestados por agentes da plataforma, e eventualmente de outras plataformas, conjuntamente com os nomes dos agentes que os prestam; o MTS encarrega-se de distribuir mensagens recebidas na plataforma para qualquer dos seus agentes e de contactar o MTS de outra plataforma, garantindo assim a distribuição de mensagens enviadas por um dos seus agentes para agentes de outra plataforma.

Uma plataforma FIPA, geralmente caracterizada como *FIPA-compilant*, tem um único AMS e um único MTS, mas pode ter diversos DFs.

2.5. Frameworks para desenvolvimento de agentes móveis

Um sistema de AM consiste numa infraestrutura de *software* que suporta o paradigma de AM, ou seja, que define os mecanismos de mobilidade e de comunicação entre os agentes, proporcionando também segurança e proteção de dados [13].

Esta infraestrutura inclui protocolos, regras para a mobilidade, como ainda, direções e diretórios com informação sobre todos os *hosts* disponíveis. Permite assim a integração dos

agentes com os *hosts*, a utilização dos serviços disponíveis nesses *hosts* e a negociação de serviços com outros agentes e objetos.

A comunicação em tempo real para obter informação sobre o estado do agente, quando este se desloca entre vários pontos numa rede, nas suas tarefas de recolha de dados num *host*, é essencial para decidir o futuro comportamento do agente. Para solicitar tal informação a um agente, a sua localização tem que ser conhecida, o que implica que o agente tem de ser localizado pelo sistema [26]. Na infraestrutura, a entidade que disponibiliza diversos serviços para migração, comunicação e acesso a recursos para os agentes num *host* é designado por *Servidor de Agentes*.

Normalmente, no desenvolvimento de um destes sistemas, opta-se por uma arquitetura *multiagentes* (*Multi-agent*), no sentido de existirem vários agentes a efetuar tarefas distintas e a interagir uns com os outros, conferindo um controlo distribuído ao sistema. Como estão bem definidas as responsabilidades a cada agente, no caso de falha de um ou mais agentes, o sistema pode continuar a funcionar. A facilidade em adicionar mais agentes potencia a escalabilidade do sistema.

Numa arquitetura de *Agente Único* (*Single agent*) todas as tarefas são atribuídas a um agente. A complexidade do sistema aumenta consoante a complexidade da tarefa atribuída, podendo também ocorrer uma sobrecarga do sistema. Opta-se por este desenvolvimento quando se pretende ter um controlo centralizado num determinado domínio.

Num denominado *Sistema Multiagentes*, para além dos agentes e servidores de agentes, existem componentes auxiliares, tais como, um diretório de nomes usado para a localização de recursos e servidores do sistema, e, nem sempre presente, um protocolo criptográfico que lhe confere a parte confiável. Estes sistemas fornecem uma *framework* sobre a qual as aplicações de AM podem ser desenvolvidas e geridas. A *framework* pode disponibilizar um conjunto de serviços, os quais permitem a implementação de um ou mais tipos de arquiteturas de sistemas de agentes (idealmente, todos os tipos de arquiteturas).

As *frameworks* para implementação e gestão de AM existentes foram desenvolvidas por grupos de pesquisa ou por empresas privadas, sendo a maioria delas implementadas em Java. Algumas já não têm qualquer tipo de suporte, ou desapareceram por completo, enquanto

outras continuam a ser usadas em diversos laboratórios de investigação e em alguns produtos comerciais. Também se verifica que nem todas seguem os padrões FIPA.

Telescript [27], *Agent Tcl / D'Agents* [28], *Mole* [29], *Aglets* [30], *ARA* [31], *Ajanta* [32], *Voyager* [33], *Grasshopper* [34], *Concordia* [35], *SOMA* [36], *JADE* [37], *FIPA-OS* [38], *ZEUS* [39], *TACOMA* [40], *SPRINGS* [41], *JADE-LEAP* [42] *Mobile-C* [43] e *Agent Factory Micro Edition* [44], são algumas das mais referidas na literatura pesquisada.

Uma caracterização geral de cada uma destas *frameworks* é apresentada seguidamente, numa ordem cronológica de desenvolvimento.

Telescript [27]

A tecnologia *Telescript* foi criada pela empresa *General Magic*, em 1994, para o desenvolvimento de sistemas de AM [45], sendo considerada como a primeira implementação comercial nesta área. O ambiente de desenvolvimento consistia numa linguagem de programação remota orientada a objetos e numa plataforma que permitia a criação de aplicações distribuídas em rede.

A natureza proprietária e a necessidade de aprendizagem de uma linguagem completamente nova foi um dos motivos de fracasso desta *framework*. Mais tarde, a crescente popularidade da Internet motivou a *General Magic* a reimplementar, com recurso a linguagens e protocolos de rede abertos, um sistema de AM baseado em Java denominado *Odyssey*.

Agent Tcl / D'Agents [16]

Agent-Tcl, posteriormente denominado *D'Agents*, foi desenvolvido no *Darmouth College* em 1995. O objetivo inicial era o de construir um sistema capaz de fornecer um mecanismo de comunicação entre agentes, fácil de usar, possuir mecanismos de segurança e ter como principal linguagem de desenvolvimento uma linguagem de *script*, e que pudesse ser estendido para aceitar facilmente outras linguagens [46].

A arquitetura do *Agent Tcl* é baseada no modelo de servidor de *Telescript* e suporta uma versão modificada da linguagem Tcl, a sua implementação da linguagem de *script* de alto

nível. *D'Agents* possui suporte para as linguagens Tcl, Java e Scheme. O código fonte do sistema está disponível para uso não comercial. Como projeto terminou em 2003.

Mole [29]

Projetado pela *Universidade de Stuttgart*, foi o primeiro sistema de AM desenvolvido na linguagem Java. A primeira versão foi concluída em 1995, tendo sido constantemente melhorada nos anos seguintes, mas atualmente já não tem qualquer suporte. Caracteriza-se por oferecer um ambiente estável para o desenvolvimento e utilização de AM na área de aplicações distribuídas. O modelo deste sistema baseia-se principalmente nos conceitos de agentes e destinos.

Aglets [30]

A plataforma Aglets é baseada na linguagem Java (a origem do seu nome vem da aglutinação das palavras *AGent* e *appLET*) e foi inicialmente desenvolvida pela *IBM Tokyo Research Laboratories*, em 1995. Posteriormente, em 2001, a IBM disponibilizou o código fonte do sistema sob licença pública. Apesar de ter prestado um grande contributo na área dos AM, tendo sido uma das plataformas mais populares, presentemente a sua utilização é bastante baixa. Aglets é hoje um projeto *open source* desenvolvido com a designação *Aglet Software Development Kit*.

Os agentes neste sistema são objetos criados na linguagem Java, denominados *aglets*, fazendo uso da capacidade multiplataforma, dessa linguagem, para criar uma arquitetura simples de suporte à migração e execução de código entre *hosts* interligados através da Internet. O ambiente de execução requer a utilização de um servidor de agentes próprio, o *Tahiti*, que fornece um interface gráfico ao utilizador para a criação, monitorização e eliminação dos agentes.

ARA [31]

Agents for Remote Action, ou simplesmente ARA, foi desenvolvido na *Universidade de Kaiserslautern*, em 1997, tendo como grande objetivo adicionar mobilidade às linguagens de programação existentes. A arquitetura deste sistema, semelhante ao *Agent Tcl*, é composta

por um núcleo e por interpretadores para cada linguagem suportada. No núcleo, estão concentradas as funcionalidades independentes da linguagem, como a mobilidade e comunicação. Já as questões dependentes da linguagem, como a captura e a restauração do estado de um agente, são resolvidas pelos interpretadores.

Ajanta [32]

Trata-se de um trabalho desenvolvido na *Universidade de Minnesota*, iniciado em 1997 e que atualmente não tem qualquer suporte. No *Ajanta*, um servidor de agentes é executado sobre uma máquina virtual Java, que deve estar presente em qualquer nó da rede, no qual se pretenda suportar aplicações baseadas em agentes. Cada servidor, além de executar agentes visitantes, oferece primitivas para comunicação, migração, controlo e monitorização.

Voyager [33]

Voyager foi inicialmente desenvolvido e comercializado pela *ObjectSpace*, em 1997, atualmente pertence à *Recursion Software, Inc*.. *C*arateriza-se por ser uma plataforma para o desenvolvimento de sistemas distribuídos, utilizando técnicas tradicionais e de AM. É baseado em Java e possui recursos para AM, mantendo a capacidade de criar e mover objetos remotamente, o que tornou esta plataforma bastante popular, na época, apesar de não possuir uma interface gráfica para a criação de agentes. Atualmente, apesar da sua constante evolução, consiste numa solução comercial.

Grasshopper [34]

Grasshopper apresenta-se como a primeira plataforma de acordo com o padrão MASIF e foi desenvolvida pela empresa *IKV++ GmbH*, em 1998. Implementada em Java, é uma plataforma de desenvolvimento e de execução de agentes, construída em cima de um ambiente de processamento distribuído, conseguindo uma integração do paradigma cliente/servidor com a tecnologia de AM.

Concordia [35]

Concordia é um produto comercial, desenvolvido em 1998, pela *Mitsubishi Electric ITA*, destinado ao desenvolvimento e gestão de aplicações de AM que se estendam a qualquer sistema que suporte Java. Um sistema de agentes em Concordia é constituído por uma série de servidores executados sobre máquinas virtuais Java. Cada servidor possui uma série de componentes, todos escritos em Java, que são responsáveis por funcionalidades como a mobilidade dos agentes, comunicação, administração, gestão de segurança e persistência, entre outras.

SOMA [36]

SOMA, abreviatura para *Secure and Open Mobile Agent*, é um projeto desenvolvido na *Universidade de Bologna*, em 1998. Tem como objetivos principais a segurança e a interoperabilidade. É implementado em Java e fornece um conjunto de serviços tradicionais para AM e dois componentes responsáveis por garantir a segurança e a interoperabilidade com outras aplicações e serviços, independente do estilo de programação adotado.

JADE [37]

JADE (*Java Agent DEvelopment framework*) foi inicialmente desenvolvido conjuntamente pela *Telecom Ita* e pela *Universidade de Parma*, em 1998, tornando-se *open source* desde 2000. É provavelmente a *framework* mais largamente utilizada para o desenvolvimento de MA.

Totalmente implementada em Java, é um ambiente para desenvolvimento de aplicações baseadas em agentes, conforme as especificações da FIPA, para a interoperabilidade entre sistemas multiagentes. Fornece uma biblioteca de classes para a implementação de agentes, um conjunto de ferramentas gráficas para administrar e monitorizar as atividades dos agentes em execução, como ainda uma plataforma distribuída sobre a qual os agentes serão executados.

FIPA-OS [38]

FIPA-OS é uma plataforma de desenvolvimento de sistemas multiagentes, desenvolvida inicialmente pela *Nortel Networks*, em 1999, tendo sido posteriormente distribuída sob uma licença *open source* [47]. Totalmente desenvolvida em Java, foi desenhada com um conjunto de componentes para a implementação de agentes compatíveis com os padrões FIPA. Foram desenvolvidas duas versões: a versão Standard FIPA-OS e a versão MicroFIPA-OS. Esta última era destinada a ser executada em dispositivos móveis que suportassem a especificação J2ME, vulgarmente conhecidos como PDAs.

ZEUS [39]

ZEUS é um *open source* desenvolvido pela *British Telecommunications Labs*, em 1999, caracterizando-se principalmente por ser um ambiente de desenvolvimento para agentes colaborativos. O seu objetivo central é o de facilitar o desenvolvimento rápido de aplicações multiagentes, abstraindo no ambiente de desenvolvimento os componentes e princípios comuns de alguns dos sistemas existentes. É puramente implementado em Java, o que o torna compatível com a maioria das plataformas de *hardware*, sendo também compatível com os padrões FIPA.

TACOMA [40]

O TACOMA (*Tromsø And COrnell Moving Agents*) é um projeto com o objetivo de estudar as questões científicas e de engenharia levantadas no paradigma de programação de AM. Foi desenvolvido de 1995 a 2002, tendo apresentado uma série de protótipos que forneciam um suporte básico para agentes escritos na linguagem Tcl. Mais tarde, o sistema foi estendido para suportar outras linguagens (C, Perl, Python, Scheme, C++ ou Java).

SPRINGS [41]

SPRINGS (*Scalable PlatfoRm for movINg Software*) foi desenvolvido pela *Universidade de Saragoza*, em 2005. Esta plataforma incide sobre a escalabilidade e a confiabilidade em cenários com um número moderado e elevado de AM. O seu desenvolvimento foi inspirado

pelas características de outras plataformas, como o Voyager e o Grasshopper. A sua principal desvantagem é que não suporta a comunicação de agentes segundo o padrão FIPA.

JADE-LEAP [42]

JADE-LEAP (*Lightweight and Extensible Agent Platform*) é uma versão modificada e reduzida da *framework* JADE para suportar aplicações móveis e sem fios. Desenvolvida a partir de 2005, surge pela necessidade de permitir a implantação de agentes em dispositivos móveis de baixo custo e com recursos limitados de memória e de processamento.

LEAP estende a arquitetura do JADE usando um conjunto de perfis que permitem que seja configurado para várias máquinas virtuais Java (JVMs). A sua arquitetura é modular e contém componentes para a gestão do ciclo de vida dos agentes e para o controlo da heterogeneidade dos protocolos de comunicação. Contém ainda módulos suplementares que permitem a execução em dispositivos móveis que suportam a tecnologia JME (com as especificações MIDP e CDC), a *framework* .NET da Microsoft e a plataforma *Android*. Contudo, estes agentes apenas garantem um suporte de mobilidade fraca [48].

Mobile C [43]

Mobile-C foi desenvolvido na *Universidade da Califórnia-Davis*, a partir de 2006, apresenta-se como uma plataforma multiagente compatível com padrão FIPA para suportar AM escritos em C/C++, especialmente usado em sistemas inteligentes de mecatrónica ou em sistemas embebidos. Apesar de ser uma plataforma multiagente de propósito geral, foi desenhada para atender especificamente as aplicações com restrições de tempo real e de recursos de *hardware*. Uma das suas particularidades é a incorporação de um interpretador de C/C++, o *Ch*, para apoiar a execução de código do AM.

Agent Factory Micro Edition [44]

O *Agent Factory Micro Edition* (AFME) foi desenvolvido na *University College Dublin*, em 2006, para permitir a criação de agentes para execução em dispositivos móveis com recursos limitados. AFME é vagamente baseado noutra *framework* existente, a *Agent Factory*, que suporta uma abordagem estruturada para o desenvolvimento de sistemas de agentes

inteligentes. Compatível com os padrões FIPA, o AFME possibilita a utilização de linguagens de desenvolvimento de agentes. Possui suporte a migração fraca, exigindo que o destino para onde o agente se deseja deslocar, já possua o seu código fonte.

2.6. Resumo comparativo das frameworks

Durante o levantamento do estado de arte relativo às *frameworks*, cujas principais características foram anteriormente apresentadas, foi dado particular atenção a um conjunto de informação relevante para este estudo.

Esta consistia em identificar a licença de utilização, o estado atual de disponibilização/distribuição, o desenvolvimento segundo padrões FIPA, e, qual a linguagem de programação usada no código do agente. Um resumo comparativo das *frameworks* analisadas é apresentado na Tabela 1.

Tabela 1 – Resumo comparativo de *frameworks* para sistemas multiagentes

Plataforma	**Licença**	**FIPA-compilant**	**Linguagem de código do agente**	**Ativo**
Aglets	IBM Public	Não (MASIF)	Java	Sim
Agent Factory ME	LGPLv2	Sim	AFAPL2, AgentSpeak	Sim
Ajanta	Código Livre	Não	Java	Não
ARA	Código Livre	Não	Tcl, C/C++	Não
Concordia	Código Livre	Não	Java	Não
D'Agents/Agent Tcl	BSD	Não	Tcl, Scheme, Java	Não
FIPA-OS	LGPL	Sim	Java	Não
Grasshopper	Código Livre	Sim	Java	Não
JADE	LGPL	Sim	Java	Sim
JADE-LEAP	LGPL	Sim	Java	Sim
Mobile-C	Código Livre	Sim	C/C++	Sim
Mole	Código Livre	Sim	Java	Não
SOMA	Código Livre	Não (MASIF)	Java	Não
SPRINGS	Código Livre	Não	Java	Sim

Plataforma	Licença	FIPA-compilant	Linguagem de código do agente	Ativo
TACOMA	Código Livre	Não	Tcl, Perl, Python, Scheme, C/C++	Não
Telescript	Comercial	Não	Tcl	Não
Voyager	Comercial	Não	Java	Sim
ZEUS	Código Livre	Sim	Java	Não

Pela observação da tabela, verifica-se que existe um predomínio de *frameworks* de código livre, ou, com licenças bem definidas de *open source*, para além de maioritariamente terem sido desenvolvidas em Java.

Para as *frameworks* desenvolvidas segundo os padrões FIFA e nas quais foi identificada alguma atividade recente, aprofundou-se mais o respetivo estudo visando saber quais os sistemas operativos alvo e quais as plataformas móveis suportadas. Esta análise teve como base a documentação disponibilizada pelo respetivo fabricante. O resultado é apresentado na Tabela 2.

Tabela 2 – Frameworks ativas e FIPA-Compilant

Plataforma	Sistemas Operativos alvo	Plataformas móveis
Aglets	Solaris, Windows, AIX, OS/2	Não suportado
Agent Factory ME	Não suportado	Dispositivos J2ME
JADE	Qualquer SO que suporte Java	Não suportado
JADE-LEAP	Qualquer SO que suporte Java	Dispositivos J2ME, Windows Mobile, Android
Mobile-C	Windows, Solaris, Linux, Mac OS X, LinuxPPC, FreeBSD, QNX	Dispositivos J2ME
ZEUS	Windows, Solaris	Não suportado

Na observação dos resultados, verifica-se que praticamente todas as *frameworks* estão relacionadas com o suporte à tecnologia Java existente, ou possível de existir, no sistema operativo alvo ou na plataforma móvel onde será executada.

2.7. Síntese

De acordo com a revisão da literatura, várias *frameworks* foram desenvolvidas (ou começaram a ser desenvolvidas) para suportar aplicações de AM, tendo sido concebidas com diversos objetivos, caracterizando-se por diferente qualidade e maturidade. Apesar do interesse que a tecnologia de AM provocou, a maioria das *frameworks* não estão, atualmente, disponíveis, ou tornaram-se obsoletas ou foram descontinuadas.

As primeiras *frameworks* foram desenvolvidas para redes locais e globais, tendo sido maioritariamente implementadas ou integradas com Java, o que lhes confere portabilidade e alguma garantia de segurança. Verifica-se ainda que, na sua maioria, foram desenvolvidas para suportar apenas AM em Java, existindo poucas que tenham sido estendidas para suportar agentes escritos noutras linguagens. A este período, compreendido entre os finais da década de 1990 e os primeiros anos de 2000, segui-se-lhe um mais moderado, não obstante a tecnologia de AM estar devidamente padronizada.

Atualmente, a massificação da variedade de dispositivos móveis, assim como o desenvolvimento de tecnologias de comunicação sem fios, motivam a necessidade de novas soluções para gestão, interação e partilha de informação. Nesse sentido, tem surgido várias propostas de implementação de soluções [49][50][51][52] que exploram os conceitos da tecnologia de AM aplicadas no campo de redes sem fios e de redes de sensores, no entanto ao nível de *frameworks* não se tem verificado algum desenvolvimento notório.

Apesar de existir investigação nesta área, a tecnologia de AM tem falhas na consolidação, isto é, ainda não estão bem visíveis as vantagens na sua aplicação, e também em termos de segurança, já que o AM executa em *hosts* externos. Não obstante, o decréscimo do preço no acesso à Internet e o aumento de débito/largura de banda disponíveis poderão ser também razões para o aparente abandono deste paradigma. Contudo poderão existir aplicações, mesmo que confinadas a redes locais ou domínios de empresas, onde o paradigma dos AM faça sentido. Um exemplo disso poderá ser as aplicações de apoio/seguimento a pessoas.

Capítulo 3

3. Linguagens de programação alvo

Este capítulo apresenta, inicialmente, um conjunto de conceitos gerais sobre linguagens de programação, seguindo-se uma visão geral sobre a execução de linguagens de *script* nos sistemas operativos móveis *Android* e *iOS,* justificada pelos condicionalismos existentes nestes ambientes. Depois é apresentado o estudo das linguagens potencialmente indicadas para o desenvolvimento de AM multiplataforma. No final é realizada uma análise comparativa das linguagens alvo deste estudo.

3.1. Conceitos gerais sobre linguagens de programação

Uma linguagem de programação consiste num conjunto de regras sintáticas e semânticas usadas para desenvolver um programa de computador [53]. O conjunto de palavras, composto de acordo com essas regras, constitui o *código fonte* de um programa que permite comunicar as instruções para um computador. As instruções especificam, precisamente, que ações devem ser tomadas em várias circunstâncias, quais os dados a manipular e como esses dados são armazenados ou transmitidos.

Existem inúmeras linguagens de programação, cada uma delas usando diferentes conceitos e abstrações. As diferentes conceções de programação, tais como funcional, imperativa e orientada a objetos, apresentadas a seguir, designam-se por "paradigma de programação".

No paradigma de programação funcional, os programas são vistos como conjuntos de funções. A computação é tratada como uma aplicação e avaliação de funções em vez de variáveis e atribuições, procurando, ao máximo, imitar as funções matemáticas. Assim, um programa surge da interação entre as funções: o resultado de uma função é passado como parâmetro para outras. Haskell e Lisp são exemplos de linguagens funcionais.

O paradigma de programação imperativa descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa. Os programas são sequências de comandos, semelhantes aos usados no funcionamento interno dos computadores. A partir desse conceito, derivam diversos paradigmas secundários. O principal é a programação estruturada ou procedimental (*procedural* em inglês), onde os comandos são organizados em grupos, chamados "funções", que podem ser invocados em momentos diferentes. Exemplos de linguagens imperativas incluem C, Pascal, Basic e Assembler.

No paradigma da programação orientada a objetos realiza-se a análise, projeto e desenvolvimento baseado na composição e interação entre diversas unidades de *software* denominadas como "objetos". Os programas são conjuntos de objetos de diferentes classes a interagir através de mensagens que ativam métodos nos objetos destino. Uma classe define o comportamento de seus objetos (através de métodos) e os estados possíveis destes objetos (através de atributos). Como exemplos de linguagens de programação que suportam a orientação a objetos, encontramos Java, Objetive-C, Ruby, Eiffel, entre muitas outras que possuem algum suporte a este paradigma.

Linguagens compiladas e linguagens interpretadas

As linguagens de programação são frequentemente tipificadas como compiladas ou interpretadas. O código fonte de um programa pode ser convertido em linguagem de máquina, por *compilação*, ou é traduzido e executado instrução a instrução, através de um processo denominado *interpretação* [54].

Na compilação, o código fonte do programa é lido e traduzido para linguagem de máquina por um outro programa, o *compilador*, sendo gerado um ficheiro binário que pode ser executado diretamente pelo *hardware* da plataforma alvo.

Em contrapartida, um programa escrito numa linguagem interpretada não é convertido num ficheiro executável. É executado recorrendo a um outro programa, o *interpretador*, que lê o código fonte e o traduz diretamente instrução a instrução, durante a sua execução. O interpretador pode também, para além das instruções do programa, receber comandos para controlar o funcionamento do programa.

Geralmente, um programa compilado apresenta a vantagem de ser executado mais rapidamente que um programa interpretado. Por outro lado, o interpretador tem a vantagem de não necessitar de passar por um processo de compilação, durante o qual as instruções para linguagem de máquina são geradas, processo que pode consumir muito tempo caso o programa seja extenso. Embora obtendo-se um programa mais lento, algumas linguagens interpretadas tem outra vantagem: a portabilidade. Como não é gerado um código de máquina, mas sim um código intermediário que será interpretado por uma máquina virtual, pode-se obter a portabilidade do código se a máquina virtual for desenvolvida para diversas plataformas (ou computadores diferentes).

Linguagens de *script*

As linguagens de *script* são geralmente linguagens de programação interpretadas que oferecem uma maior facilidade de instalação, compilação, carga e execução dinâmica comparativamente com as linguagens tradicionais. Apresentam também uma maior agilidade de programação devido a serem tendencialmente mais simples e flexíveis.

Uma das características importantes deste tipo de linguagens é a possibilidade de execução com carregamento dinâmico de código. Trata-se de um mecanismo que permite a um programa de computador, em tempo de execução, carregar uma biblioteca (ou um ficheiro binário) para a memória, recuperar os endereços de funções e variáveis contidas nessa biblioteca, executar as funções ou aceder às variáveis, e por fim, libertar a biblioteca da memória. Esta característica é bastante importante para o suporte a AM já que estes, ao moverem-se, terão que ser instanciados novamente no destino. Todas estas linguagens são

interpretadas, porém, nem todas as linguagens interpretadas são linguagens de *script*. Atualmente, muitas não são interpretadas diretamente mas sim compiladas para um *bytecode* que é interpretado numa máquina virtual, tal como acontece com a linguagem Java.

Máquina Virtual

O termo "máquina virtual" surgiu nos anos 60, como método de construção de um computador abstrato dentro de um computador real [55]. Podemos definir uma máquina virtual (ou VM, *Virtual Machine*) como um tipo de programa de computador usado para criar um ambiente virtual, produzido através de um processo referido como "virtualização".

Uma VM apresenta-se como um ambiente computacional, no qual um sistema operativo ou um programa pode ser instalado e executado, comportando-se deste modo como um outro computador. Na sua utilização, é possível executar um sistema operativo (e respetivas aplicações) sobre um outro, usar uma aplicação de uma outra plataforma, executar múltiplos sistemas operativos, entre demais tarefas.

Para além de poder flexibilizar uma plataforma complexa de trabalho, o recurso a uma VM apresenta a vantagem da independência de plataforma, já que uma aplicação pode ser executada em qualquer VM que garanta o suporte ao sistema operativo usado, sem se preocupar com o *hardware* e *software*. Esta independência da plataforma promove a portabilidade e, consequentemente, a mobilidade, característica importante no paradigma de AM [56]. Atualmente, devido aos baixos custos, são uma alternativa usada em vários sistemas de computação.

3.2. Execução de linguagens de script nas plataformas Android e iOS

O contexto de desenvolvimento de aplicações móveis é muito diferente do que é realizado para uma aplicação *desktop* ou para uma aplicação Web, pelo que se justifica uma breve abordagem neste documento. Vários aspetos devem ser tomados em consideração, principalmente os relacionados com as restrições de *hardware* e *software* existentes nos diversos dispositivos móveis e, em particular, no *Android* e no *iOS*.

3.2.1. Android

O *Android* [57] é um sistema operativo de código aberto baseado no *kernel* do *Linux* para dispositivos móveis. Foi desenvolvido pelo grupo *Open Handset Alliance* (OHA) do qual faz parte a Google.

Para desenvolvimento de aplicações é disponibilizado o *Android Software Development Kit* (SDK), que contém as ferramentas e bibliotecas necessárias para construir e testar aplicações para este sistema. Os testes podem ser realizados através de um simulador de um dispositivo com *Android*, também disponibilizado no SDK, que, apesar das limitações, permite simular as diversas versões do sistema. É também disponibilizada uma biblioteca que permite a criação e execução de *scripts* e interpretadores interativos diretamente no dispositivo, a *Scripting Layer for Android* (SL4A) [58], existindo uma distribuição própria que possibilita a sua descarga e instalação.

A SL4A fornece uma infraestrutura que permite a interoperação dos motores das linguagens de *script* com a API do *Android*, através de chamadas de procedimento remoto para um servidor implementado como uma aplicação nativa *Android*. Os *scripts* podem oferecer processos de interação através de uma *interface* gráfica ou usados através do tradicional modo de execução em terminal. São suportadas as linguagens *Python*, *Perl*, *Ruby*, *Lua*, *BeanShell*, *JavaScript* e *PHP*, cujo respetivo interpretador pode ser descarregado como uma normal aplicação (*apk*). A linguagem Tcl deixou de ser suportada.

3.2.2. iOS

O *iPhone Operation System* (*iOS*) é um sistema operativo proprietário da Apple para dispositivos móveis [59]. A versão atual é o iOS 8 lançado a 17 de setembro de 2014.

Baseado no sistema operativo *Mac OS X*, possui um *kit* de desenvolvimento de software (iOS SDK) que permite o desenvolvimento de aplicações para os dispositivos *iPod Touch*, *iPhone* e *iPad*. No desenvolvimento de aplicações para estas plataformas é necessário um computador Apple Macintosh com processador Intel, a ferramenta Xcode e o iOS SDK.

O *Xcode* é o ambiente de desenvolvimento integrado que usa a linguagem de programação Objective-C [88] na criação de aplicações nativas para o *iOS*. É uma ferramenta de desenvolvimento rápido e permite, entre outras funcionalidades, desenhar rapidamente interfaces gráficas a partir de um conceito de *storyboards* e simular o ambiente das aplicações sem recorrer a um dispositivo físico [60]. Suporta a API *CocoaTouch*, entre outros recursos essenciais para o desenvolvimento de aplicações. Possui um simulador próprio, o *iOS Simulator*, que permite realizar diversos testes sem a necessidade da presença real de um dispositivo. Apesar do seu uso ser gratuito, o XCode está disponível apenas para o sistema operativo *Mac OS X*.

Alegando razões de segurança e estabilidade, a Apple não permitia linguagens de *script* nos seus dispositivos móveis, como também limitava a execução de aplicações em segundo plano. Até abril de 2010, o contrato de licença para o *iPhone Developer Program* [61], no ponto 3.3.2, declarava que "Nenhum código interpretado pode ser descarregado ou usado numa aplicação, exceto o código que é interpretado e executado por APIs documentadas da Apple e interpretador embutido(s)."

Posteriormente, em setembro de 2010, foi efetuada uma revisão ao referido contrato [62], que se mantêm em vigor, na qual a Apple passa a permitir o uso de linguagens de *script*. No mesmo ponto, prevê-se agora que "Uma aplicação não pode descarregar ou instalar código executável. Código interpretado só pode ser utilizado numa aplicação, se todos os *scripts*, código e interpretadores estejam incluídos (*packaged*) na aplicação e não descarregado.".

3.2.3. Ferramentas de abstração de plataforma para desenvolvimento de aplicações móveis

Destinadas ao desenvolvimento de aplicações móveis, nos últimos anos tem aparecido um elevado número ferramentas de abstração de plataforma, também designadas por *frameworks* de desenvolvimento *Cross-Platform*, das quais se destacam, entre outras, *MoSync* [63], *PhoneGap* [64], *Codename One* [65], *Appcelarator Titanium* [66], *Corona SDK* [67], *Sencha Touch* [68], *RhoMobile* [69].

Este desenvolvimento consiste na técnica de escrever uma única base de código para aplicativos que venham a ser utilizados em diferentes sistemas operativos móveis. A parte visual é normalmente desenvolvida em HTML 5 e CSS e é renderizada num *browser*

embutido no SDK padrão da linguagem nativa (Objective-C do iOS, Java do Android, etc.) tornando assim possível o desenvolvimento para várias plataformas.

O desenvolvimento nestas ferramentas, geralmente, diminui consideravelmente o custo de um projeto, principalmente quando são várias as plataformas envolvidas. Porém, pelo facto de criarem uma aplicação genérica, o acesso a recursos nativos torna-se mais trabalhoso e frequentemente mais problemático. Deste modo, são frequentemente utilizadas no desenvolvimento de aplicativos simples que não necessitam tanto de recursos nativos.

3.3. Estudo das linguagens de programação potencialmente indicadas para Agentes Móveis

Os AM podem ser vistos como código, estado e atributos. O código é um programa que define o comportamento do agente e este deve ser executado da mesma forma em qualquer máquina da rede [70]. Isto significa que deve existir uma independência da plataforma onde o AM é executado logo, as linguagens de programação mais adequadas para o desenvolvimento de AM são as interpretadas diretamente, ou, as compiladas para uma linguagem intermediária baseada num interpretador que possa ser facilmente transportado.

Assim, a escolha das linguagens de programação sobre as quais se elaborou este estudo incidiu na sua capacidade de execução em multiplataforma, para além da utilização no desenvolvimento de *frameworks* existentes (apresentadas no capítulo 2). A capacidade do código escrito poder ser usado em diversas plataformas, mantendo-se praticamente inalterado, como a possibilidade de um desenvolvimento rápido, possibilitando, facilmente, a criação de programas destinados a tarefas específicas, determinaram a identificação das linguagens alvo. *Java*, *Python*, *Perl*, *Lua* e *Tcl* são as linguagens que melhor se enquadram nestes critérios.

Segue-se uma apresentação do propósito de cada linguagem, visão geral, distribuições existentes e facilidade/constrangimentos de execução em sistemas móveis (*Android* e *iOS*).

3.3.1. Java

A linguagem Java[4] foi desenvolvida no início da década de 1990, por James Gosling na *Sun Microsystems*, atualmente uma subsidiária da *Oracle*. Imediatamente após o seu lançamento, ganhou rapidamente uma grande aceitação pela indústria de *software* e é globalmente aceite como a linguagem de programação para a Web.

Foi inicialmente projetada como um ambiente de execução completo, pretendendo promover a comunicação entre computadores e restantes equipamentos digitais, sendo por isso capaz de fornecer uma interface consistente e abstrata, independentemente da plataforma de *hardware* e de *software* que lhe estejam subjacentes [71]. Esta independência de plataforma é conseguida através da utilização de uma máquina virtual Java (JVM), que emula a sua própria plataforma computacional e funciona como um interpretador de código java, chamado *bytecode*.

Em cada plataforma que se pretenda executar um programa em Java é necessário um interpretador. Assim, o programa é compilado para um código intermédio (*bytecode*), código esse que é gerado não para um processador específico mas sim para um processador "abstrato", a JVM. Praticamente todas as plataformas dispõem de interpretadores Java, desde *smart cards* até servidores, o que permite a universalidade de um programa compilado em *bytecodes*.

Principais características da linguagem Java

A linguagem Java é uma linguagem de alto nível, orientada a objetos, em que os programas consistem na manipulação de classes, objetos, atributos e métodos. Com a exceção dos tipos básicos da linguagem, a maior parte dos elementos de um programa Java são objetos. É disponibilizada uma grande quantidade de classes e métodos nas bibliotecas de classes, podendo ser facilmente utilizadas pelo programador.

Uma das características mais conhecidas é a sua portabilidade, ou seja, a capacidade de execução em ambientes heterogêneos. Como o compilador transforma o código fonte em *bytecodes*, ao executar o programa a JVM local traduz o código intermediário para a

[4] Página oficial: http://www.oracle.com/technetwork/java/index.html.

linguagem da máquina na qual será executado o programa. Como os *bytecodes* gerados são entendidos por qualquer JVM, independentemente do ambiente em que foram gerados, os programas Java podem ser executados em qualquer sistema que possua uma JVM. Outra característica importante da JVM é a libertação automática de memória alocada dinamicamente, o que evita problemas de *memory-leaks*, que acontecem noutras linguagens como o C, usadas por programadores indisciplinados, ou por simples esquecimento de libertação de memória. A libertação automática de memória, na linguagem Java, está a cargo do componente *Garbage Collector*.

A linguagem Java carrega o código (classes) dinamicamente, à medida das necessidades. Para isso inclui um mecanismo responsável por encontrar e carregar uma classe na memória da JVM em tempo de execução. Normalmente as classes só são carregadas a pedido. Caso um nome de classe existente num programa não possa ser resolvido localmente ou a classe seja explicitamente requerida pela aplicação, a JVM utiliza uma classe especializada chamada *ClassLoader* para o efeito. Outra funcionalidade importante da linguagem é ela possuir reflexão computacional. Reflexão é o processo onde se consegue, em tempo de execução, informação (métodos, atributos e construtores) de uma classe.

O Java foi concebido para ser seguro quando executado através de uma rede de computadores, mesmo caso o programa seja de autoria desconhecida. Como não permite a utilização de apontadores, torna impossível aceder diretamente à memória. A linguagem implementa ainda um mecanismo robusto para manipular exceções, esperadas ou não.

Distribuições de Java

A versão mais recente da linguagem, Java 8, foi lançada a 18 de março de 2014. Em termos de distribuições, a linguagem está disponível no *Java Development Kit* (JDK), destinado ao desenvolvimento de aplicações que comporta, para além da JVM, um compilador e outras ferramentas úteis; e no *Java Runtime Environment* (JRE), que possui apenas o necessário para se executar uma aplicação Java (a JVM e as APIs), voltado ao utilizador final. A Oracle disponibiliza a maioria das distribuições Java gratuitamente para praticamente todos os sistemas operativos, casos do *Windows*, *Linux*, *Mac OS X* e *Solaris* [72]. Documentação

sobre a tecnologia Java pode ser encontrada na página da Oracle[5] para além de inúmera literatura publicada.

Java em plataformas móveis

A capacidade de execução Java em dispositivos móveis é geralmente integrada pelo fabricante do dispositivo, não estando disponível para descarga ou instalação pelo utilizador.

Tecnicamente, a linguagem Java não é suportada em *Android* [73], o que significa que não é possível executar ficheiros JAR ou visitar *sites* com conteúdo Java. Para executar um ficheiro JAR, será necessário ter um acesso *root* e depois instalar um emulador.

O sistema operativo *Android* tem como base o *kernel Linux* e, à semelhança deste, existem determinadas operações que necessitam de privilégios de super utilizador para que possam ser executadas. O denominado processo de "root-ing" varia de dispositivo para dispositivo, existindo aplicações específicas para o efeito.

Em *Android*, apesar das aplicações nativas serem escritas na linguagem Java, a máquina virtual utilizada não é a JVM. É utilizada uma implementação desta, denominada máquina virtual *Dalvik* (DVM), otimizada para execução em dispositivos móveis [74]. A DVM não interpreta Java *bytecodes* mas sim *dexcode*, obtido usando a ferramenta *dx* que transforma os arquivos *.class* compilados com um compilador Java normal em arquivos *.dex*, estes específicos para execução na DVM.

A última versão de *Android*, lançada a 31 de outubro de 2013, denominado por *Android 4.4 KitKat*, para além de manter a DVM, apresenta, a título experimental, uma nova máquina virtual, a *Android Runtime* (ART), candidata a futuramente substituir a DVM. No caso dos sistemas operativos da Apple embora seja possível executar uma JVM no sistema *Mac OS X*, a Apple não permite que a JVM seja executada em qualquer *iPhone* ou *iPad* [62]. Uma solução possível, que não envolva um processo de adicionar recursos extras, seja por meio da instalação de *software* não autorizado ou pela ativação de funcionalidades bloqueadas, é adaptar e compilar o código Java para uma aplicação nativa do *iOS*, o sistema operativo

[5] Página de documentação oficial: http://docs.oracle.com/javase/.

usado neste tipo de dispositivos móveis. As aplicações nativas para o *iOS* e *Mac OS X* são escritas na linguagem de programação Objective-C.

3.3.2. Python

Python[6] é uma linguagem de programação de alto nível, orientada a objeto, interpretada e interativa [75], disponível para vários sistemas operativos. Foi inicialmente desenvolvida por Guido van Rossum em 1991 no *Stichting Mathematisch Centrum* (Holanda). O seu propósito era, segundo o autor [76], ser uma linguagem fácil e intuitiva como também tão poderosa quanto as maiores competidoras; de código aberto, para que qualquer um possa contribuir para o desenvolvimento; possuir código tão inteligível quanto o idioma inglês; ser adequada para tarefas "diárias", permitindo um tempo de desenvolvimento mais curto.

Na realidade, Python é parte da fórmula vencedora para a produtividade, qualidade de *software* e manutenção em muitas empresas e instituições de todo o mundo [77]. É tipicamente usada em aplicações Web, desenvolvimento de jogos e como linguagem de *script* para administração de sistemas. É extensível, ou seja, caso haja necessidade de algoritmos críticos estes podem ser escritos em C ou C++ e utilizados em Python. Esta facilidade de integração, principalmente com linguagem C, faz com que Python seja atrativa para ser embutida em aplicações de maior porte. É também frequentemente usada para o desenvolvimento rápido de aplicações.

Formalmente, a linguagem de programação Python é uma especificação, existindo diversas implementações. Todas as suas versões são de código aberto com licença GPL, possuindo deste modo um modelo de desenvolvimento comunitário, sendo a especificação da linguagem mantida pela *Python Software Foundation*[7].

Características da linguagem Python

A linguagem Python possui uma sintaxe clara e concisa o que favorece a legibilidade do código fonte, sendo o uso da indentação para definir blocos uma das suas características mais visíveis. Esta é uma particularidade a ter em atenção pois a indentação é requerida na

[6] Página oficial: http://www.python.org/.
[7] Página da *Python Software Foundation*: http://www.python.org/psf/.

escrita do código. Também a pouca utilização de caracteres especiais torna a linguagem muito parecida com um algoritmo executável.

Em Python não é necessário declarar as variáveis a usar, nem definir o seu tipo. A própria sintaxe do dado que está a ser armazenado identifica o tipo da variável. Outra característica importante é que suporta múltiplos paradigmas de programação. Possui diversas estruturas de alto nível e uma vasta coleção de módulos, como também diversas *frameworks* desenvolvidas por terceiros que podem ser usadas em ambos os paradigmas. Em Python os módulos são coleções de funções que são tratadas como objetos. Dividir programas em módulos facilita o reaproveitamento e localização de falhas no código.

O modelo de programação imperativa suportado pode ser usado na criação rápida de programas simples, existindo várias estruturas de dados complexas, como listas e dicionários, para facilitar o desenvolvimento de algoritmos complexos. A programação orientada a objetos é totalmente suportada, possibilitando o desenvolvimento de grandes projetos de *software*.

Para carregar código Python é usada a declaração *import* [78], sendo necessário identificar o módulo, o que se designa por importação absoluta. Importar módulos em tempo de execução também é possível com a utilização de uma função padrão (*__import__*) ou através de uma biblioteca padrão (*importlib*) [79].

Distribuições de Python

O Python está definido em duas especificações. *Python 2*, atualmente na versão 2.7.8, é que tem mais suporte e módulos de terceiros. A especificação *Python 3*, cuja versão 3.4.1 foi lançada em 18 de março de 2014, está em permanente desenvolvimento e tem conseguido apresentar versões estáveis. No entanto, ainda não é totalmente compatível com alguns módulos de terceiros desenvolvidos na especificação anterior.

Num sistema *Linux* ou *Mac OS X*, o interpretador Python já vem instalado por padrão, sendo apenas necessário escrever o comando "*python*" através do programa de terminal. No sistema *Windows*, o interpretador deve ser descarregado e instalado, existindo também um utilitário para fazer o papel de terminal e de editor Python, denominado IDLE.

A linguagem fornece também implementações para *.NET* (*IronPython*), JVM (*Jython*) para além de Python (*PyPy*). Para um sistema operativo não suportado, basta que exista um compilador C disponível nesse sistema para gerar o Python a partir do respetivo código fonte.

Python em plataformas móveis

O desenvolvimento de aplicações Python para o sistema *Android* é possível recorrendo ao uso do SL4A [58] (a camada de *script* deste sistema operativo) sendo necessário instalar uma distribuição do interpretador – *PythonForAndroid_r5.apk* ou *Python3ForAndroid_r6.apk* - no dispositivo [80].

Nativamente não é possível desenvolver aplicações em Python para o sistema *iOS*. Uma solução passa por efetuar um processo de embutimento (*embedding*) do interpretador, ou, por adaptar e compilar o código Python para uma aplicação nativa do iOS. Neste segundo caso, existem diversas aplicações nativas desenvolvidas por terceiros que permitem o uso da linguagem diretamente nos dispositivos móveis.

Python for iOS [81] é uma aplicação de uso comercial, que funciona como um interpretador de Python em qualquer dispositivo com *iOS*. *Pythonista* [82], também uma aplicação de uso comercial, possibilita construir aplicações nativas em Python no *iOS*, possuindo algumas bibliotecas personalizadas para desenho no ecrã, áudio e interações sensíveis ao toque.

Kivy [83] é uma biblioteca de código aberto desenvolvida em Python para programar em Python. Trata-se de uma *framework* criada para o rápido desenvolvimento de aplicações móveis, focalizada na criação de aplicações multi-toque, caracterizando-se ainda por ser multiplataforma (*Android*, *iOS*, *Linux*, *Mac OS X* e *Windows*).

3.3.3. Perl

Perl[8] (*Practical Extraction and Report Language*) é uma linguagem de programação de uso geral desenvolvida por Larry Wall na década de 80. É uma linguagem de alto nível, interpretada e multiplataforma.

[8] Página oficial: http://www.perl.org/.

A linguagem surgiu para facilitar a manipulação de textos, colmatando as necessidades que os comandos UNIX, *awk* e *sed*, não conseguiam resolver. Devido a ser uma linguagem de código aberto, possuindo uma licença GPL, originou a sua expansão para as mais diversas áreas de computação. Com a evolução da Internet, em particular da *World Wide Web*, a linguagem recebeu um enorme impulso quando passou a ser utilizada na programação de *Common Gateway Interface* (CGI).

A linguagem Perl é utilizada em tarefas, desde encontrar expressões em textos longos, até ao desenvolvimento de comércio eletrónico (*e-commerce*) ou mesmo na geração dinâmica de imagens. É também bastante utilizada em finanças e em bioinformática, onde é reconhecida pelo desenvolvimento rápido de aplicações e pela capacidade de lidar com grandes conjuntos de dados. Outras utilizações incluem a administração de sistemas, desenvolvimento Web, programação de redes ou desenvolvimento de *interfaces* gráficas. Trata-se de uma linguagem interpretada cuja maior vantagem é sua adaptação à manipulação de cadeias de caracteres (*strings*). Além disso, as suas funcionalidades de manipulação de ficheiros, diretórios e bases de dados transformaram o Perl na linguagem preferida para a escrita de interfaces CGI.

O interpretador Perl é gratuito e está disponível para vários sistemas operativos. O código fonte do interpretador, em linguagem C, também está disponível, o que significa ser possível compilar o interpretador para qualquer sistema operativo.

Características da linguagem Perl

Uma das principais características do Perl é facilitar a manipulação de textos e processos [84]. Os programas são simples ficheiros de texto ASCII que contêm comandos na sintaxe da linguagem, necessitando de um interpretador Perl para serem executados. São parcialmente compilados para detetar erros antes da interpretação e de modo a simplificar a tarefa do interpretador. Outras características apontadas à linguagem são a sua facilidade de uso, suporte para programação estruturada como também para programação orientada a objetos, e, uma das maiores coleções de módulos escritos por terceiros. Os módulos são fciheiros de código Perl usados para armazenar funções e objetos de forma a estender a funcionalidade da linguagem. Um desses módulos, o *AutoLoader*, permite o carregamento dinâmico de código (sub-rotinas) a pedido.

O Perl possui uma série de tipos de dados fundamentais: *scalar* (escalares), *array* (vetor) e *Hash* (vetor associativo). Porém, é considerada uma linguagem não tipificada, ou seja, as variáveis não são restringidas a usar um único tipo. Também não exige a declaração de tipos de dados sendo capaz de escolher que tipo utilizar, dinamicamente, para cada variável, podendo ainda alterá-lo durante a compilação ou a execução do programa. Em Perl, o operador é o elemento que define como os operandos serão utilizados, o que significa que a linguagem adota a conversão automática entre tipos.

O Perl possui um pequeno conjunto de construções primitivas que são combinadas de diversas formas para formar estruturas de controlo. O mecanismo de gestão de memória automático é implementado através de contagem de referência (quando a contagem de referência é zero, o objeto é removido). Possibilita a interpretação dinâmica de código, o que significa que a linguagem é capaz de executar excertos de código criados dinamicamente, no mesmo ambiente de execução do programa.

Apenas com a versão Perl 5 começou a existir suporte nativo à POO. No entanto, este é bastante minimalista e para usá-lo é necessário um bom conhecimento de sub-rotinas (*perlsub*), referências (*perlref*) e módulos/pacotes (*perlmod*), como recomenda a própria página de manual *perlootut* [85].

Uma alternativa é utilizar o Moose [86], uma extensão ao sistema POO do Perl 5, que possibilita ao programador pensar mais na abstração dos dados e da classe e não sobre como a classe deve ser implementada.

Distribuições de Perl

O Perl 5.18 é a atual versão estável, distribuída sob duas licenças: a *Artistic License* e a GPL. O principal centro de distribuição de código fonte, módulos e documentação de Perl é o *Comprehensive Perl Archive Network* (CPAN)[9]. O Perl 6[10] é uma nova especificação da linguagem cujo desenvolvimento foi iniciado em 2000, existindo diversas implementações mas nenhuma ainda considerada estável.

[9] Página oficial: http://www.cpan.org/.
[10] Página oficial: http://www.perl6.org/.

As distribuições para *UNIX/Linux*, como também para *Mac OS X*, trazem o Perl 5.x como padrão do sistema, sendo recomendado realizar a descarga de módulos específicos para melhorar o desempenho na construção de *scripts*. Para *Windows* existem diversas distribuições que permitem uma instalação automática.

Perl em plataformas móveis

No sistema *Android* é possível a execução de Perl através da SL4A. O interpretador Perl disponibilizado - *perl_for_android_r1.apk* - é datado de 25 de agosto de 2010 [87].

No sistema *iOS* da Apple a execução direta de *scripts* não é permitida. A solução passa por adaptar e compilar o código Perl para uma aplicação nativa do *iOS*.

3.3.4. Lua

Lua[11] é uma linguagem de programação interpretada, criada em 1993 por Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes [88], na Pontifícia Universidade Católica do Rio de Janeiro, no Brasil.

Foi desenvolvida com o intuito de estender, através de *scripts*, aplicações que usam outras linguagens de programação como o C ou o C++. Possibilita estender programas escritos em Java, C#, Smalltalk, Fortran, Ada, e mesmo outras linguagens de *script*, como Perl e o Ruby. Apesar de poder ser utilizada por si só, foi concebida para ser fácil a sua incorporação numa outra aplicação em geral. Possui uma API simples e bem documentada com operações primitivas e adotou o modelo de pilha e não de registos, o que a torna ainda mais genérica e independente de processadores específicos.

Lua é utilizada no desenvolvimento de sistemas de Intranet, na construção de páginas para a Web, em aplicações de robótica e no desenvolvimento de jogos eletrónicos [89]. Como foi implementada numa interseção entre ANSI C e ANSI C++ é garantida a sua portabilidade, pois as bibliotecas de Lua podem ser executadas em qualquer plataforma que tenha um compilador compatível com o padrão ANSI.

[11] Página oficial: http://lua.org/.

Apesar de ser uma linguagem de programação imperativa, oferece suporte para abstração de dados, necessária no paradigma da programação orientada a objetos, proporcionando mecanismos flexíveis que permitem ao programador construir qualquer modelo adequado para a aplicação a desenvolver [90]. Lua conta com uma única implementação principal, mantida pelos autores na página oficial da linguagem, existindo ainda diversas distribuições mantidas por terceiros, pois é distribuída no regime de *software* livre.

Características da linguagem Lua

A linguagem Lua possui uma sintaxe simples e de fácil aprendizagem, existindo um sólido manual de referência e vários livros sobre a linguagem. Apesar de ser uma linguagem desenhada com a finalidade de estender funcionalidades de aplicações, pode ser utilizada como uma linguagem de propósito geral. É interpretada a partir de *bytecodes* e possui gestão automática de memória. Ao ser utilizado autonomamente, o interpretador é fornecido com a distribuição padrão. Este interpretador, *lua*, inclui todas as bibliotecas padrão como também a biblioteca de depuração de erros (*debugging*). A linguagem faz a verificação de tipo de dados em tempo de execução do programa (tipificação dinâmica). Possui os mecanismos usuais para controlo de fluxo, sendo capaz de executar excertos de código criados dinamicamente no mesmo ambiente de execução do programa.

Lua não é uma linguagem de POO, mas fornece um pequeno conjunto de mecanismos com os quais se pode construir recursos de um nível superior. Diferentes sistemas de objetos foram desenvolvidos em Lua, incluindo os sistemas mais tradicionais, bem como sistemas sem classes.

Distribuições de Lua

A versão atual da linguagem é Lua versão 5.2, datada de 11 de novembro de 2013. A distribuição oficial é disponibilizada em formato de código fonte, pelo que é necessário efetuar a respetiva compilação. No entanto, existem distribuições em formato executável (*binaries*) [91], como um interpretador *online*[12] que permite um primeiro contacto com a linguagem.

[12] Lua Demo: http://www.lua.org/cgi-bin/demo/.

Para o sistema operativo *Windows*, a distribuição *Lua for Windows* [92] contém um pacote completo da versão 5.1, que incluí o interpretador com as bibliotecas padrão, um editor e várias bibliotecas extras. Para *Linux*, dada a diversidade de distribuições deste sistema operativo, além da possibilidade de compilação do código fonte, algumas dessas distribuições vêm com o pacote Lua instalado por defeito. Em outras será necessário a respetiva instalação. Para o sistema operativo *Mac OS X* é possível compilar diretamente o código-fonte, desde que a máquina já tenha instaladas as ferramentas de desenvolvimento em C. Outra opção é usar um sistema gestor de pacotes [93].

Lua em plataformas móveis

O interpretador *lua* não é fornecido com o sistema *Android* sendo possível, através da SL4A, a sua execução usando uma distribuição disponível - *lua_for_android_r1.apk* - datada de agosto de 2010 [87]. Outra opção é integrar, ou portar, o interpretador através da ferramenta *Android Native Development Kit* (NDK) [94]. O NDK é um conjunto de ferramentas que permite usar bibliotecas escritas em C ou C++ para utilização nas aplicações para *Android*. Duas práticas comuns da utilização do NDK são para o aumento do desempenho da aplicação e aproveitar código C existente transferindo-o para este ambiente computacional. *AndroLua* [95] e *Kahlua* [96] são dois exemplos encontrados.

O desenvolvimento de aplicativos em *iOS*, com Lua, é possível recorrendo a *frameworks* de abstração de plataforma, entre as quais se destaca o *Corona SDK* [67] e *Gideros Studio* [97]. Também a integração do interpretador *lua* num aplicativo *iOS* é viável.

3.3.5. Tcl

Tcl (*Tool Command Language*) é uma linguagem de programação criada por John Ousterhout [98], em 1988, na Universidade da Califórnia (Berkeley). Atualmente é de utilização livre e mantida pela comunidade *open source*[13].

A necessidade de criar uma linguagem de comando integrada com outros sistemas operativos, e em particular com o UNIX, motivou o seu desenvolvimento, pretendendo ser uma

[13] Página oficial: http://www.tcl.tk/.

linguagem simples, genérica e extensível. Ao ser combinada com a biblioteca gráfica Tk é fornecido um conjunto de componentes para criação de *interfaces* gráficas.

O Tcl é usado em aplicações Web e de escritório, programação de redes, desenvolvimento de sistemas embebidos, programação de propósito geral, administração de sistemas, trabalhos com base de dados, entre outras [99].

É essencialmente uma linguagem de *script* dinâmica, interpretada, de âmbito geral e disponibilizada nos sistemas operativos UNIX e Linux, possuindo interpretadores para um grande número de outros sistemas operativos.

Cada interpretador, em execução, deve ser considerado como uma máquina virtual separada e com o seu próprio estado interno, que é, consequentemente, modificado à medida que recebe comandos. Os comandos podem ser lidos a partir de um ficheiro contendo um *script* ou por terminal de linha de comandos, permitindo interagir com o interpretador de maneira dinâmica.

Para ser executado numa plataforma específica, é necessário um interpretador para a respetiva plataforma. Os mais usados são o *tclsh* e o *wish*. O primeiro é utilizado para os *scripts* que são executados em modo de terminal e o último para aplicações com uma *interface* gráfica do utilizador, geralmente baseadas em Tk.

Características da linguagem Tcl

A linguagem Tcl caracteriza-se pela fácil integração com outras linguagens de programação, casos do C/C++ e Java, como também pela sua extensibilidade. Neste caso, a partir de um conjunto básico de primitivas implementadas em C/C++, é possível construir extensões da linguagem.

Os programas Tcl consistem em comandos [100]. Um comando é formado por palavras separadas por espaços. Os comandos são separados por uma nova linha ou pelo caracter ponto-e-vírgula. É possível o programador criar novos comandos ou procedimentos, denominados *procs*, que funcionam como comandos internos. Dentro dos comandos, para além de *scripts* em Tcl, são suportadas construções usando outras linguagens.

A linguagem Tcl só possui um tipo de dados, *strings*, ou cadeias de caracteres. Contudo, alguns comandos, especialmente os que realizam cálculos, interpretam os seus argumentos como valores numéricos ou booleanos. Não existe na linguagem o conceito de passagem de parâmetros por referência sendo necessário a utilização de variáveis globais. A linguagem Tcl não é uma linguagem de base orientada a objetos, apesar de possuir extensões que o permitem, como o *incrTcl*, *XOTcl* ou *TclOO*. A partir da versão Tcl 8.6 o suporte *TclOO* vem incorporado no interpretador.

Algumas lacunas são identificadas na linguagem, tais como, a não existência de um mecanismo de gestão automática de memória, o não possuir um modelo de segurança e não suportar a migração de código. Estas lacunas podem ser ultrapassadas usando o *Safe-Tcl* [101], o qual permite a execução local de programas Tcl com origem noutros computadores de uma rede. Esta capacidade é disponibilizada pela instalação de interpretadores da linguagem. Na execução de programas Tcl de origem desconhecida podem ser estabelecidas restrições (e.g., o acesso a dados ou a ficheiros locais) o que permite estabelecer um modelo de segurança. As restrições impostas podem ser ultrapassadas pela execução de chamadas de segurança a funções monitorizadas pelo interpretador, permitindo a execução de ações necessárias, mas em segurança.

Distribuições de Tcl/Tk

Tcl/Tk 8.6, datada de 20 de dezembro de 2012, é a mais recente versão estável. A distribuição mais usada é a da empresa *ActiveState*[14] que fornece edições gratuitas e comerciais do seu pacote Tcl para múltiplas plataformas. Estas distribuições incluem bibliotecas e extensões para além do núcleo Tcl. O *SourceForge*[15] contém distribuições para múltiplas plataformas mas não inclui as extensões e bibliotecas. *KitCreator*[16] fornece distribuições personalizáveis para descarga, podendo opcionalmente incluir outras bibliotecas e extensões.

Tcl/Tk não está disponível para as plataformas *Android* e *iOS*.

[14] ActiveState Software Inc : http://www.activestate.com/activetcl/.
[15] Página do projeto Tcl/Tk: http://tcl.sourceforge.net/.
[16] Página do projeto KitCreator: http://kitcreator.rkeene.org/.

3.4. Análise comparativa das linguagens

Com base nas características identificadas em cada linguagem, foram elaboradas duas tabelas comparativas, que seguidamente se apresentam.

Na Tabela 3 é identificada a capacidade de execução nas diferentes plataformas, enquanto na Tabela 4 são identificadas algumas das características funcionais de cada linguagem relevantes para o estudo.

Tabela 3 - Linguagens de *script* e a execução multiplataforma

Plataforma / Linguagem	*Windows*	*Linux*	*Mac OS X*	*Android*	*iOS*
Java	Sim, requer instalação	Sim, requer instalação	Sim, requer instalação	Sim, usando um emulador	Não
Python	Sim, requer instalação	Sim, nativo	Sim, nativo	Sim, através de SL4A	Sim, embutida em aplicação nativa
Perl	Sim, requer instalação	Sim, nativo	Sim, nativo	Sim, através de SL4A	Sim, embutida em aplicação nativa
Lua	Sim, requer instalação	Sim, requer instalação	Sim, requer instalação	Sim, através de SL4A	Sim, embutida em aplicação nativa
Tcl	Sim, requer instalação	Sim, requer instalação	Sim, requer instalação	Não	Não

Verifica-se que as linguagens alvo necessitam todas de uma instalação no sistema *Windows*. As linguagens Python e Perl estão incluídas na distribuição padrão dos sistemas operativos *Linux* e *Mac OS X*, sendo necessário proceder à respetiva instalação das linguagens Java, Lua e Tcl. No sistema *Android*, existe um suporte para a execução das linguagens Python, Perl e Lua, recorreando ao SL4A, no entanto, esta biblioteca não suporta a linguagem Java e a linguagem Tcl. Ainda no sistema operativo *Android*, a linguagem Java é possível de ser

executada usando um emulador. Não existe qualquer suporte no sistema iOS para as linguagens Java e Tcl, sendo possível embutir, numa aplicação nativa, o respetivo interpretador das linguagens Python, Perl e Lua.

Tabela 4 – Linguagens de *script* e as características funcionais

Característica / Linguagem	Carregamento dinâmico de código	Gestão automática de Memória	Suporte nativo a POO	Verificação de tipo de dados	Tratamento de Exceções
Java	Sim	Sim	Sim	Estática	Sim
Perl	Sim	Não	Sim	Estática e Dinâmica	Sim
Python	Sim	Sim	Sim	Dinâmica	Sim
Lua	Sim	Sim	Sim, usando extensões	Dinâmica	Não
Tcl	Sim	Não	Sim, usando extensões	Dinâmica	Sim

Todas as linguagens alvo apresentam a característica de carregamento dinâmico de código, como também possibilitam um suporte, nativo ou usando extensões, à programação orientada a objetos. A gestão automática de memória é realizada nas linguagens Java, Python e Lua. A capacidade de verificar dinamicamente o tipo de dados, não é realizada pela linguagem Java, ao contrário das restantes linguagens. Finalmente, verifica-se que apenas a linguagem Lua não possui algum mecanismo para tratamento de exceções, o qual, terá de ser codificado.

Apesar da informação já apresentada, ainda não é possível identificar com exatidão qual a linguagem ou linguagens mais indicadas para desenvolvimento de AM multiplataforma. Será necessário efetuar um conjunto de testes de desempenho e capacidades de cada linguagem para aferir, entre outros, a simplicidade, disponibilização de API prontas a usar e manipulação do código em tempo de execução.

É no entanto possível aferir que, devido a não ser suportada a sua execução em *Android* e em *iOS*, a linguagem Tcl não é adequada para o fim pretendido.

Capítulo 4

4. Testes de avaliação das linguagens

Este capítulo apresenta os testes realizadas às linguagens de programação que, até ao momento, apresentam características adequadas aos objetivos propostos neste estudo. Assim, para Java, Python, Lua e Perl é apresentada uma análise ao resultado do respetivo teste de avaliação do suporte de serialização e de carregamento dinâmico, obtido na execução nos sistemas operativos *Windows*, *Linux* e *MAC OS X* e nos sistemas móveis *Android* e *iOS*.

O exemplo de programa adotado para cada teste não se propõe a apresentar a melhor solução, mas sim a expor diferentes aspetos de cada linguagem e servir como modelo comparativo entre estas. O código gerado para cada linguagem está optimizado para apresentar o mínimo de linhas possível, não contendo, por vezes, como recomendado, a verificação do valor armazenado nas variáveis usadas, evitando assim referências vazias (*null*). Para uma melhor leitura e compreensão dos resultados, todo o código fonte gerado para cada linguagem é remetido para os anexos a este documento.

4.1. Cenário de testes

Na análise de desempenho das linguagens foram usados os seguintes ambientes computacionais:

- *Windows*: *Windows 7 Home Premium 64 bits*
- *Linux*: máquina virtual com a distribuição *Ubuntu 32 bits 12.04.3 LTS*
- *Mac OS X*: máquina virtual com a versão *OS X 10.7.5 "Lion"*
- *Android*: versão 4.2.2 kernel 3.4.5
- *iOS*: Apple iPad 16GB iOS 7.1.1 e *iOS Simulator*

Na codificação das diversas versões do programa, para cada linguagem, foram verificados em cada ambiente computacional, os seguintes requisitos:

- Por omissão, o Java não vem instalado nas diferentes distribuições dos sistemas operativos, pelo que para executar aplicações desenvolvidas nesta linguagem foi necessário proceder à instalação do JDK, versão 1.7.0_45, adequado a cada sistema operativo;

- Para executar um programa em Python é necessário um interpretador da linguagem, disponível por defeito nas distribuições *Linux* e no *Mac OS X*. Foi necessário efetuar a respetiva instalação para os sistemas *Windows*, na versão 2.7.6; o mesmo acontece para um programa em *Perl,*tendo sido instalada a versão 5.14.2;

- A linguagem *Lua* não é distribuída, por padrão, nos diferentes sistemas operativos pelo que foi efetuada a instalação do respetivo interpretador, versão 5.2.0;

- Para ser possível a criação e execução de *scripts* no sistema *Android*, foi necessário instalar no dispositivo a SL4A [58] e os interpretadores das linguagens disponibilizados para esta biblioteca;

- Para testar num dispositivo com *iOS* é necessário obter uma licença de desenvolvimento[17].

[17] https://developer.apple.com/programs/ios/

4.2. Teste ao suporte de serialização

No contexto de armazenamento e transmissão de dados, a serialização é o processo que permite guardar um objeto num meio de armazenamento (ficheiro de computador ou *buffer* de memória) ou transmiti-lo através de uma ligação de rede, seja em formato binário ou em formato de texto. O conjunto de *bytes* serializado pode posteriormente ser usado para recriar um objeto com o mesmo estado interno que o original.

O teste de serialização envolve a utilização de um objeto, denominado *MyObject*, cuja estrutura (*X* e *Nome*) e comportamento é representado na Figura 1. A variável de instância *X* contém um valor numérico, podendo ser incrementada, decrementada ou atribuída diretamente. Para a variável *Nome* optou-se por atribuir o nome da máquina onde o programa é executado. Os dados serializados são guardados num ficheiro de texto (*myObject.ser*).

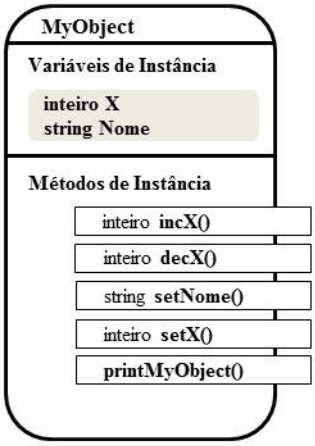

Figura 1 - Estrutura e comportamento do objeto MyObject

Os métodos de instância verificam o seguinte:

- ser possível criar objetos com
  - valor inicial de *X* igual a 0 (zero);
  - conteúdo inicial de *Nome* igual a “foo”;
- ser possível incrementar e decrementar o objeto
  - de uma unidade;
  - de um dado valor passado por argumento;
- ser possível atribuir
  - um conteúdo a *Nome*;
- ser possível obter uma representação textual do objeto.

Foram testadas as características de instanciação, serialização e de-serialização. Foi considerado pertinente que os procedimentos necessários para realizar cada uma destas características possam ser realizados indistintamente durante a execução do programa. Assim, vai-se obter um maior número de situações possíveis de análise de desempenho na execução do programa em diferentes ambientes computacionais (e.g., de-serializar no *Linux* dados serializados em *Windows*).

A análise de desempenho de cada linguagem incidiu sobre:

- Correta compilação do código fonte;
- Instanciação e inicialização do objeto;
- Serialização e de-serialização de dados no mesmo ambiente computacional;
- Serialização e de-serialização entre os diferentes ambientes computacionais;
- Necessidade de utilização de módulos/bibliotecas adicionais.

O algoritmo seguinte foi usado para a implementação em cada uma das linguagens:

```
Algoritmo "MainTestMyObject"
Var    op: inteiro
       workObj: MyObject

Procedimento Inicializar (e: MyObject)
Var    localhostname: texto
Início
       Ler(localhostname)
       e.setNome ← localhostname
       printMyObject (e)
Fim-procedimento

Procedimento Serializar (e: MyObject)
Var    fileOut: ficheiro
Início
       fileOut← e
       Escrever(fileOut)
Fim-procedimento

Procedimento Desserializar : MyObject
Var    e: MyObject
       fileIn: ficheiro
Início
       Ler(fileIn)
       e ← fileIn
       printMyObject (e)
       Retorna e
Fim-procedimento

Procedimento Menu
Inicio
       Escreve("---Tarefas---")
       Escreve("Escolha uma opcao:")
       Escreve("  1) Inicializar")
       Escreve("  2) Serializar")
       Escreve("  3) Desserializar")
```

```
        Escreve("  4) Incrementar")
        Escreve("  5) Decrementar")
        Escreve("  6) Apresentar")
        Escreve("  0) Sair")
Fim-procedimento

Início
        op = - 1
        workObj← MyObject
        printMyObject (workObj)
        Enquanto op != 0
                Menu
                Ler(op)
                Escolha
                        Caso op = 1
                                Inicializar(workObj)
                        Caso op = 2
                                Serializar(workObj)
                        Caso op = 3
                                workObj ← Desserializar()
                        Caso op = 4
                                incX(1)
                        Caso op = 5
                                decX(1)
                        Caso op = 6
                                printMyObject(workObj)
                Fim escolha
        Fim-enquanto
Fim-algoritmo.
```

No projeto desenvolvido em *iOS*, para cada linguagem, optou-se pela execução sequencial do algoritmo proposto, ou seja, todas as características a testar são executadas uma após a outra. Como o *script* é executado em segundo plano não existe qualquer interação com o utilizador, pelo que o menu de tarefas apesar de exibido na vista da aplicação não produz qualquer efeito.

Isto obrigou a que fossem atribuídos valores à variável de controlo do menu para simular todos os processos a testar. A alternativa a esta opção seria incluir um componente gráfico adequado (e.g. *Bar Buttom Item* ou *Picker View*) para receber a opção do utilizador e fazer chamadas ao procedimento no ficheiro de *script* pretendido ou criar vários *scripts*, um para cada tarefa [18].

4.2.1. Java

O código fonte para a implementação em Java consta no Anexo A (Programa 1 e Programa 2), no qual foram usadas bibliotecas de classe disponibilizadas pela linguagem.

[18] Entrar em detalhes sobre o que precisava ser alterado no código fonte de modo a ser possível a interação com o utilizador em dispositivos iOS está para além do âmbito deste estudo.

Para ser possível a serialização de um objeto (Classe) deve-se explicitamente implementar a *interface* "java.io.Serializable". Esta interface é apenas de marcação, pois não possuí qualquer método a ser implementado, serve apenas para que a JVM saiba que aquela determinada Classe está preparada para ser serializada.

A serialização dos objetos é feita chamando, na classe "java.io.ObjectOutputStream", o método "writeObject". A de-serialização do objeto é feita invocando, na mesma classe, o método "readObject".

O Programa 2 foi compilado e executado corretamente no ambiente *Windows* (Figura 2.), *Linux* (Figura 3.) e *Mac OS X* (Figura 4.), pelo que as tarefas de instanciação, inicialização, serialização e de-serialização foram concluídas com sucesso, assim como a de-serialização de dados provenientes de outros ambientes computacionais.

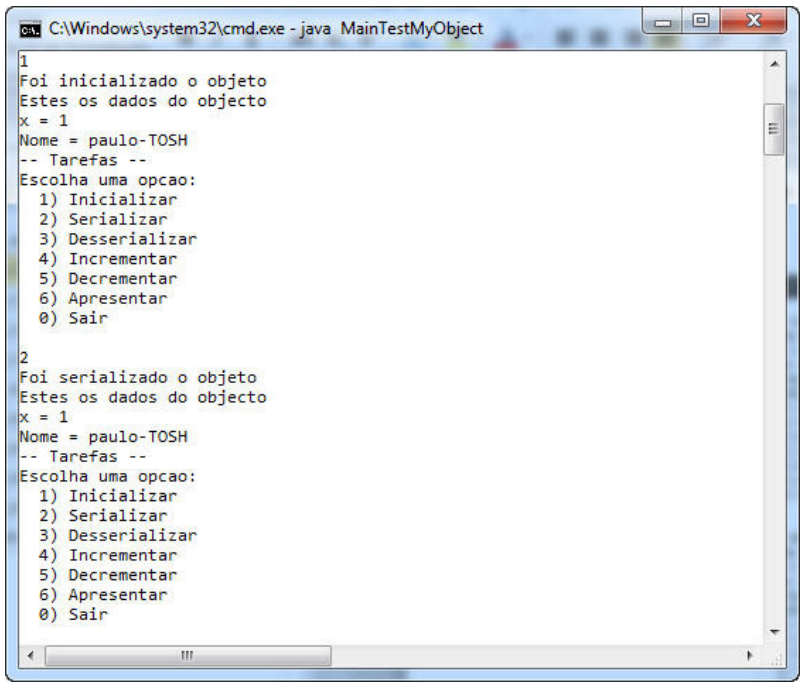

Figura 2 – Execução do programa de serialização em Java no ambiente Windows

Na Figura 2 é possível verificar o resultado das tarefas de inicialização e de serialização, com a respetiva representação textual do objecto, tendo o programa obtido, para a variável de instância *Nome*, o *hostname* do ambiente *Windows* onde o programa foi executado.

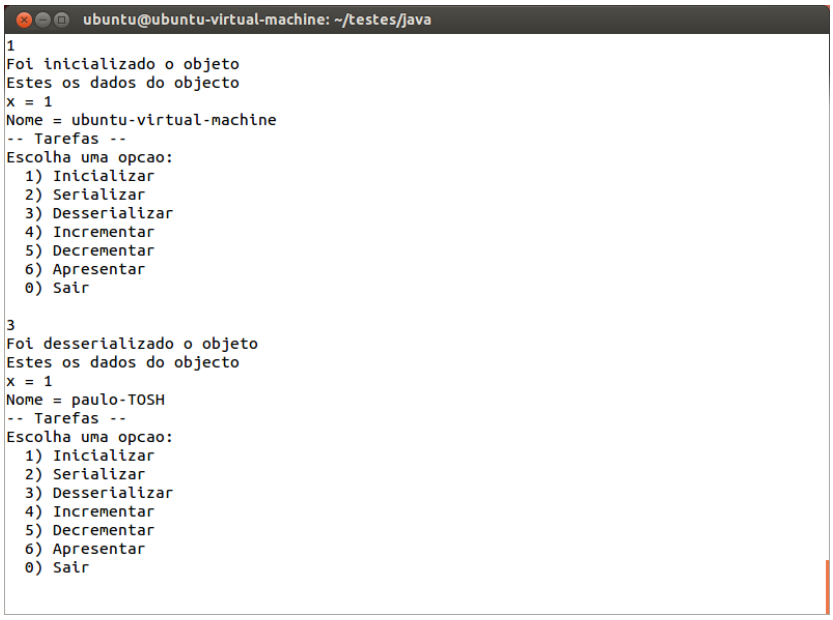

Figura 3 – Execução do programa de serialização em Java no ambiente Linux

Na Figura 3 são apresentas as tarefas de inicialização do objeto e de-serialização de dados no ambiente *Linux*. Para obter este resultado, o ficheiro serializado no *Windows* foi copiado para o ambiente atual de execução, antes da chamada ao programa.

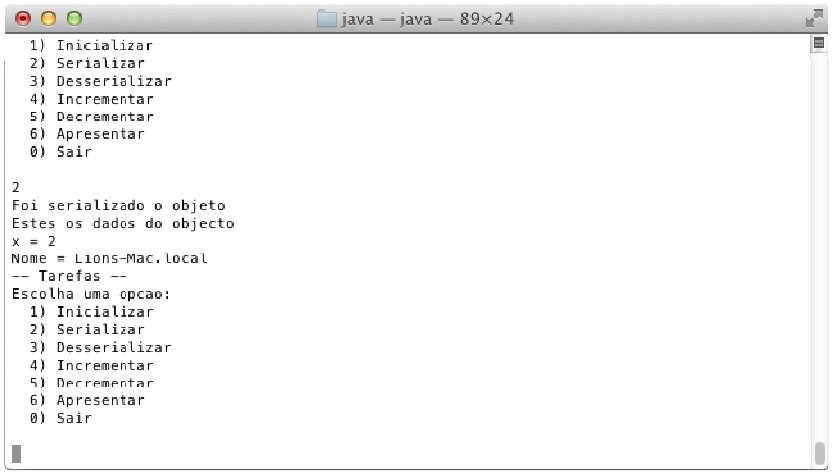

Figura 4 – Execução do programa de serialização em Java no ambiente Mac OS X

Na Figura 4 é exibido o resultado da execução, no ambiente Mac OS X, da tarefa de serialização, seguida da respetiva representação textual do objeto anteriormente incrementado.

Devido às restrições existentes nos sistemas *Android* e em *iOS*, referidas anteriormente em 3.3.1., sobre a execução de Java em plataformas móveis, não foi realizado o teste nesses ambientes.

4.2.2. Python

O código fonte usado para o desenvolvimento em Python consta no Anexo B: o Programa 4 cria a classe *MyObject*; o Programa 5 produz o teste de serialização para execução em *Windows*, *Linux*, *Mac OS X* e *Android*; os programas 6, 7 e 8 fazem parte do projeto que possibilita a execução numa aplicação *iOS*.

De notar que, na escrita do código, a não existência de um controlo do tipo *switch-case statement* na linguagem pelo que é usada uma sequência de blocos *if-else*.

Em Python existem diversos mecanismos disponíveis para serialização de dados. A escolha vai depender do tipo de aplicação exigida. Praticamente qualquer objeto em Python, ou até mesmo código, pode ser convertido para uma representação *string*. O objeto, enquanto estiver representado como uma *string*, pode ser armazenado num ficheiro ou transferido pela rede. No exemplo foi usado o *pickle* [102], módulo padrão para serialização de objetos num ficheiro.

O programa foi executado corretamente nos ambientes *Windows* (Figura 5.), *Linux* (Figura 6.) e *Mac OS X* (Figura 7.), tendo sido concluídas com sucesso a instanciação, inicialização, serialização e de-serialização. Os dados serializados em outros ambientes computacionais foram devidamente obtidos.

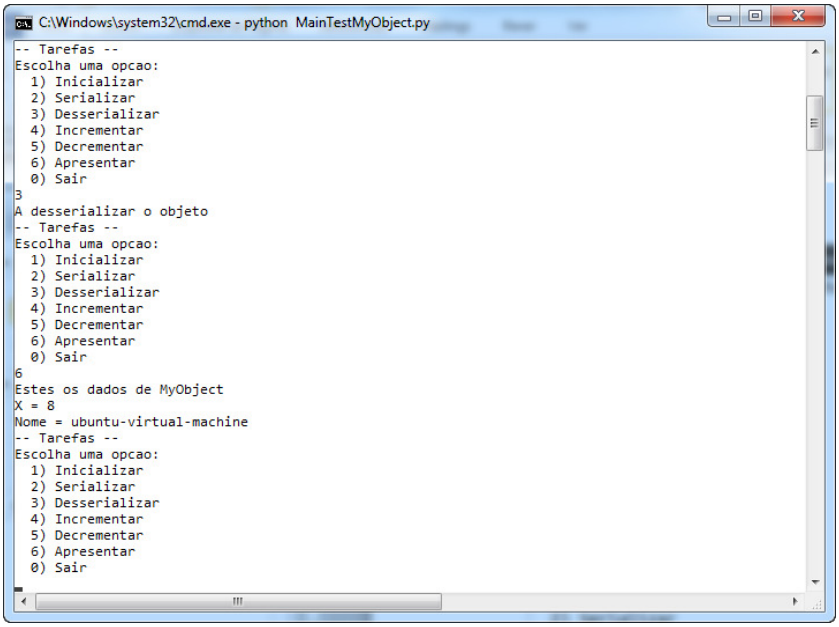

Figura 5 – Execução do programa de serialização em Python no ambiente Windows

A Figura 5 apresenta o resultado obtido na tarefa de-serialização de dados no ambiente *Windows*, utilizando um ficheiro serializado no *Linux*, ficheiro esse que foi copiado para o ambiente atual de execução, antes da chamada ao programa.

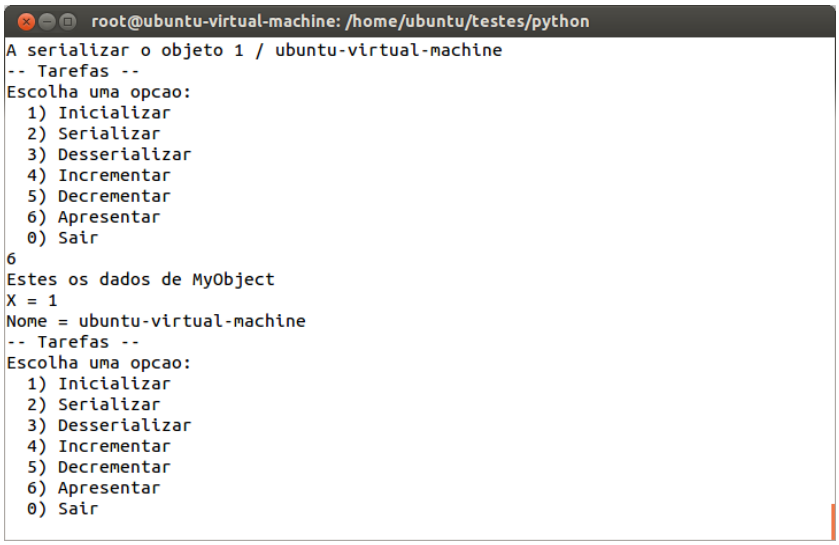

Figura 6 – Execução do programa de serialização em Python no ambiente Linux

A Figura 6 exibe a execução da tarefa de serialização em *Linux*, seguido-se a representação textual do objeto no seu estado atual.

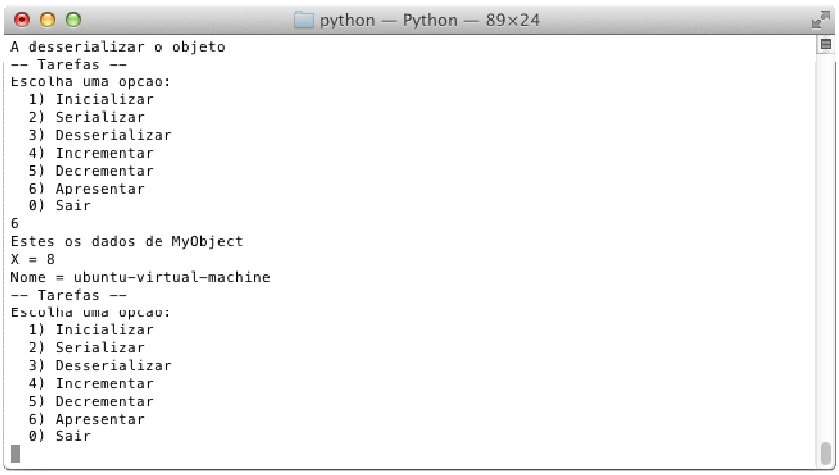

Figura 7 – Execução do programa de serialização em Python no ambiente Mac OS X

Na Figura 7 é apresentada a tarefa de de-serialização de dados no ambiente *Mac OS X*, cujo resultado é obtido através de um ficheiro previamente serializado no *Linux*, o qual foi copiado para o ambiente atual de execução.

O programa foi executado corretamente no ambiente *Android* (Figura 8.), realizando todas as tarefas pretendidas, sem necessitar de alguma edição do código original. O interpretador para a linguagem, Py4A [103], usado conjuntamente com a biblioteca SL4A, não requereu qualquer configuração adicional.

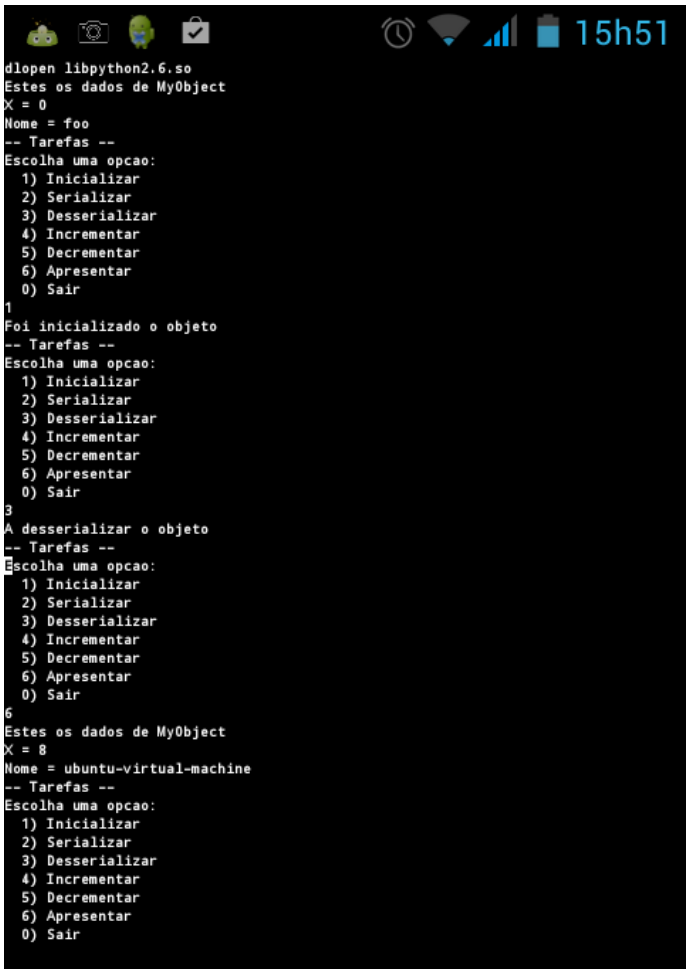

Figura 8- Execução do programa de serialização em Python no ambiente Android com SL4A

Na Figura 8 são apresentadas as tarefas de inicialização e de-serialização de dados no ambiente *Android*. O resultado da de-serialização é obtido através de um ficheiro serializado no *Linux*, posteriormente copiado para o ambiente atual de execução, antes da chamada do programa.

Para testar a execução em *iOS* optou-se por criar uma aplicação nativa, na qual foi incorporado (*embedding*) o interpretador para possibilitar a chamada e execução de código Python [104]. A inclusão do interpretador é feita manualmente, devido ao Xcode, a partir da versão 5.0 [105], deixar de disponibilizar no seu SDK a "Python.Framework", a qual inclui as

bibliotecas padrão da linguagem. Este processo consiste em localizar e adicionar ao projeto a pasta da *framework* existente na máquina local, e, seguidamente, fazer um *build* para criar as dependências necessárias. Depois disto, a biblioteca de inclusão "Python.h" será reconhecida no projeto.

A aplicação criada, comportando uma única vista, requer codificação adicional ao programa principal "*main.m*" (Anexo B, Programa 6). Essa codificação consiste na indicação da diretiva de inclusão da *framework* e na utilização de funções associadas à inicialização do interpretador [106]. Também, para possibilitar carregar o módulo *MyObject*, imprescindível no teste, foi codificado um mecanismo de localização dentro do diretório atual da aplicação [107], tal como mostra o excerto de código seguinte.

```
Py_SetProgramName(argv[0]);
Py_Initialize();
PySys_SetArgv(argc, argv);
PyRun_SimpleString("import os, sys \n"
                   "#print sys.argv, \"\\n\".join(sys.path)\n"
                   "#print os.getcwd()\n"
                   "import MyObject\n");
```

No programa "*Viewcontroller.m*" (Anexo B, Programa 7), onde se encontra a implementação necessária à aplicação, foi adicionado o código que permite criar variáveis globais para armazenar a localização do diretório atual, do diretório do utilizador e o nome do dispositivo, como também as funções para a execução do código e finalização do interpretador.

Por razões de segurança e de privacidade, o *iOS* instala, para cada aplicação, uma área isolada do *File System* designada por *sandbox*, onde é armazenada a aplicação e os respetivos dados [108]. Assim, através de métodos e funções disponibilizadas pelo Objetive-C, é obtida essa localização, a qual é usada na criação de objetos de memória [109], concretamente, para criar as variáveis globais a usar no *script* a executar, tal como mostra o código seguinte.

```
PyObject *main = PyImport_AddModule("__main__");
PyObject *globals = PyModule_GetDict(main);
PyObject *value = PyString_FromString(cDocumentsDirectory);
PyDict_SetItemString(globals, "mypath", value);
PyObject *value2 = PyString_FromString(cDevHostName);
PyDict_SetItemString(globals, "appHostName", value2);
```

A execução do código é realizada criando um objeto de ficheiro Python que é usado como parâmetro pela função *PyRun_SimpleFile*.

```
FILE* fd = fopen(prog, "r");
PyRun_SimpleFile(fd, prog);
```

O programa de teste de serialização (Anexo B, Programa 8) necessitou de ser editado para incluir a variável global que localiza o diretório de dados, tendo também sido removida a função responsável pela exibição do menu de tarefas proposto ao utilizador, pelas razões referidas no ponto 4.2., relativamente à execução neste sistema operativo móvel. Após estas alterações, foi executado corretamente (Figura 9.), recorrendo ao simulador do dispositivo, realizando todas as tarefas pretendidas.

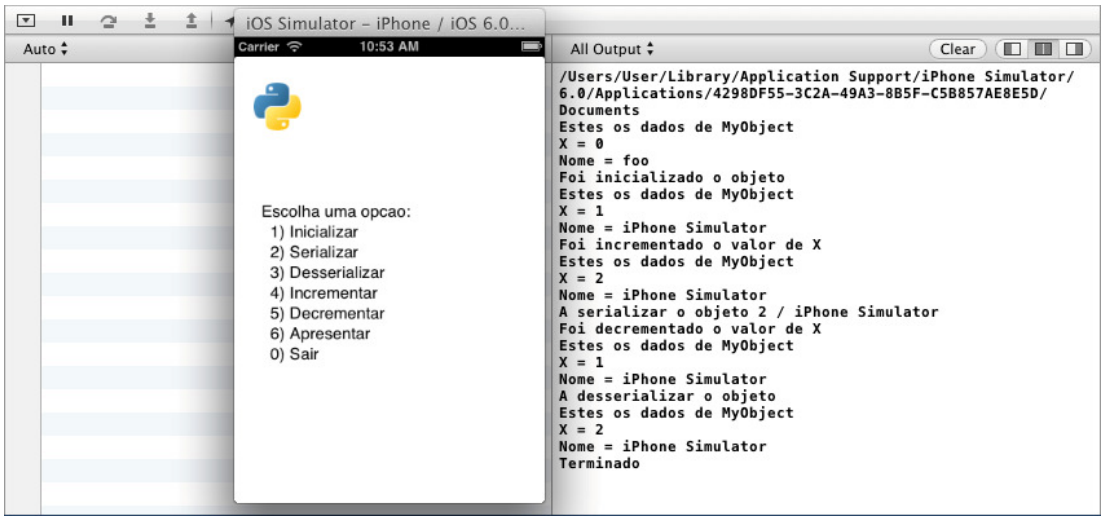

Figura 9 – Execução do programa de serialização em Python no simulador de ambiente iOS

A Figura 9 apresenta a vista criada para a aplicação, exibida no simulador de iOS, a qual apresenta uma área com o *output* gerado durante a execução do programa em modo de depuração.

4.2.3. Lua

O código fonte para implementação em Lua consta no Anexo C: o Programa 10 cria a classe *MyObject*; o Programa 11 produz o teste de serialização para execução em *Windows*, *Linux*, *Mac OS X*; o Programa 12, idem para *Android*; o Programa 13 e Programa 14 fazem parte do projeto que possibilita a execução numa aplicação *iOS*.

Não sendo uma linguagem puramente orientada a objetos, Lua fornece mecanismos para a implementação de classes e herança, através da utilização de *tabelas* e *metatabelas*. Uma *tabela* (ou *array* associativo) é a única forma disponível para a estruturação de dados e pode ser usada para representar registos, objetos e módulos, sendo constituída por um conjunto de chaves e pares de dados onde os dados são referenciados por chave. *Metatabela* é uma tabela que controla o comportamento de outras estruturas de dados.

Em Lua não é fornecido qualquer processo integrado para serializar uma tabela devido aos vários requisitos necessários para obter esse efeito, pelo que terá de ser codificado o processo pretendido. Assim, dos diversos módulos existentes para serialização, não incluídos na distribuição padrão, optou-se pelo *serpent* [110].
Para obter o nome da máquina[19] a guardar na variável de instância *Nome*, foi necessário recorrer à biblioteca de extensão – *luasocket* - não incluída na distribuição padrão [111].

O programa (Anexo C, Programa 11) foi executado corretamente nos ambientes *Windows* (Figura 10.), *Linux* (Figura 11.) e *Mac OS X* (Figura 12.), tendo as tarefas de instanciação, inicialização, serialização e de-serialização sido todas concluídas com sucesso. Verificou-se ainda que a de-serialização de dados serializados em outros ambientes computacionais foi realizada corretamente.

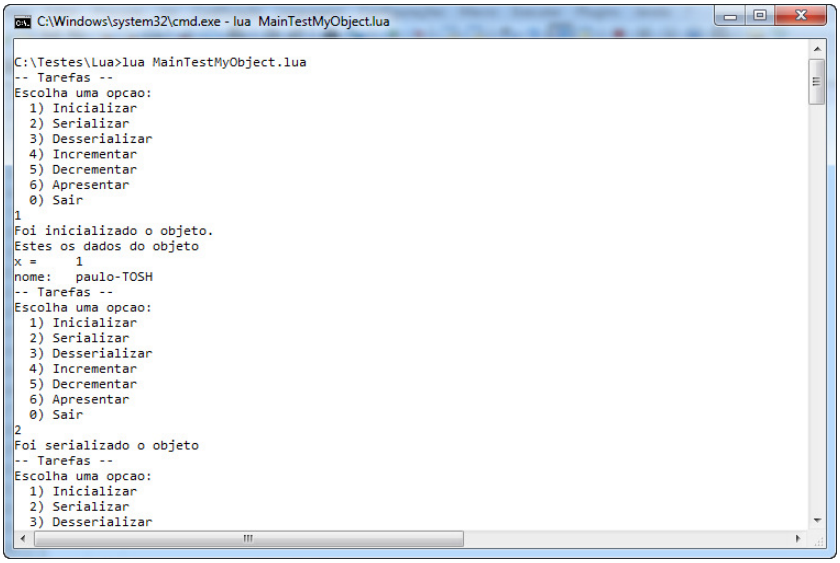

Figura 10 – Execução do programa de serialização em Lua no ambiente Windows

[19] Não sendo muito relevante para o teste em questão, optou-se por manter a coerência entre todas as linguagens na procura de um processo para fornecer esta informação.

A Figura 10 apresenta a inicialização do objeto no ambiente *Windows* e a atribuição da identificação do *hostname* à variável de instância *Nome*.

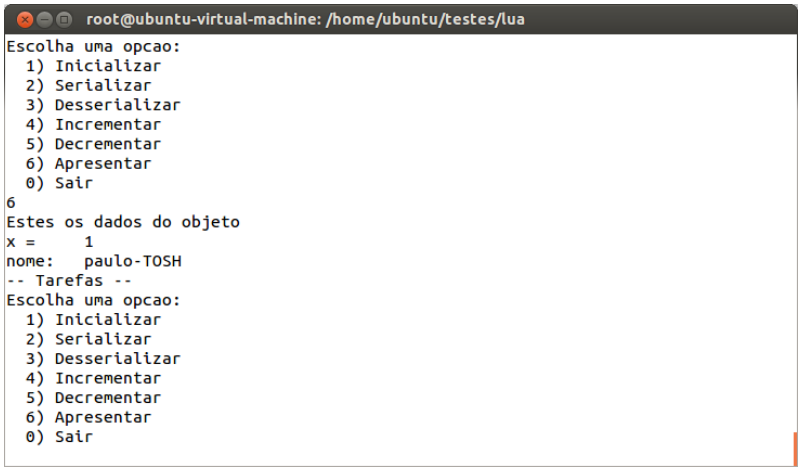

Figura 11 – Exemplo de execução do programa de serialização em Lua no ambiente Linux

A Figura 11 apresenta, no ambiente *Linux*, a representação textual do objeto obtido através da de-serialização de um ficheiro gerado em outro sistema operativo (*Windows*). Observa-se, neste caso, que o valor da variável de instância *Nome* indentifica a máquina usada para testes no sistema *Windows*.

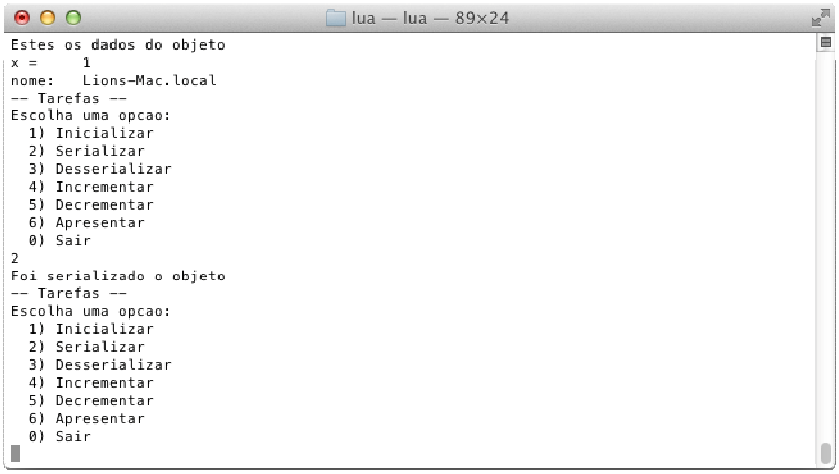

Figura 12 – Exemplo de execução do programa de serialização em Lua no ambiente Mac OS X

A Figura 12 apresenta a tarefa de inicialização e respetiva representação textual do objeto durante uma execução em ambiente *Mac OS X*.

Na execução no ambiente *Android*, recorrendo ao SL4A, é necessário incluir e indicar a localização de *serpent* [110], que é a biblioteca utilizada para as tarefas de serialização, como

ainda efetuar algumas correções ao código original. Esta edição consistiu na inclusão das seguintes linhas de código, de modo a obter acesso à API do dispositivo e configurar o caminho dos módulos a carregar:

```
require "android"
package.path = package.path .. ";../?.lua"
```

Verificou-se que, em caso de erro de sintaxe ou de carregamento adicional de módulos e respetivas dependências, o interpretador utilizado, *lua_for_android*, não devolve qualquer informação, simplesmente termina o *script*, pelo que é recomendável adicionar algum código para tratar as possíveis exceções. Depois destas alterações, foi possível executar o programa (Anexo C, Programa 12), tendo sido realizadas com sucesso todas as tarefas pretendidas (Figura 13.).

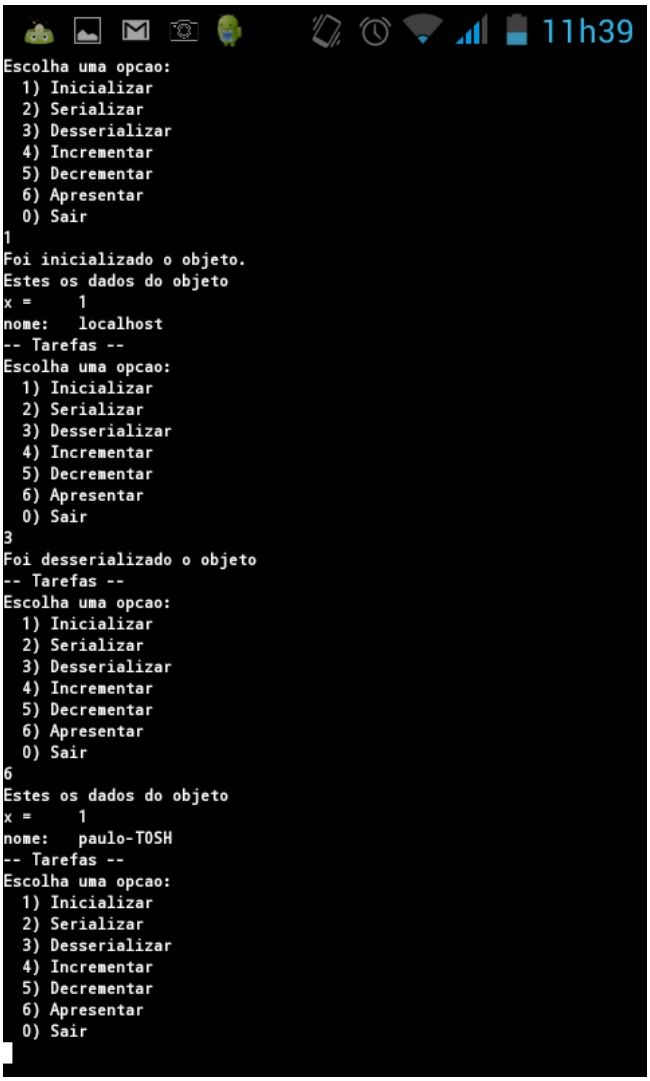

Figura 13 – Exemplo de execução de serialização em Lua no ambiente Android com SL4A

A Figura 13 apresenta, durante a execução do programa em *Android*, a inicialização do objeto e a tarefa de-serialização, com a respetiva representação textual. A de-serialização é efetuada a partir de um ficheiro gerado em *Windows* que anteriormente foi copiado para o dispositivo.

Lua pode ser usada como linguagem embutida numa aplicação nativa para *iOS*. As técnicas existentes para o efeito variam entre usar a própria linguagem para construir toda a aplicação, recorrendo a um *kit* de desenvolvimento [97][112], normalmente multiplataforma, ou fazer uso da linguagem como um componente para *scripting* da aplicação. Esta última foi a opção seguida para o teste realizado. Nesse sentido foi descarregado e descompacto o ficheiro da distribuição da linguagem, foi renomeada a pasta "src" para "lua" e foram eliminados os ficheiros "makefile", "lua.c" e "luac.c", apenas necessários para executar programas em linha de comando. O processo fica concluído depois de arrastar todo o conteúdo da pasta para um projeto Xcode e de se certificar que a opção "*create external build system*" foi desmarcada. Entre outras opções, foi escolhido o modelo "Single View Application" para criar uma aplicação simples com uma única vista, o respetivo controlador e *storyboard*. Depois de compilar o projeto, é possível interagir com o interpretador usando uma API C disponibilizada pela linguagem Lua.

No ficheiro "ViewController.h", da aplicação, é necessário incluir as bibliotecas de Lua e declarar uma variável apontador para armazenar numa estrutura em C, denominada *lua_State*, o estado do interpretador.

```
#import <UIKit/UIKit.h>
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
@interface ViewController : UIViewController{
    lua_State *L;
}
@end
```

No ficheiro "ViewController.m" (Anexo C, Programa 13), onde se encontra a implementação necessária à aplicação, para o método *viewDidLoad*, foi adicionado o código que permite invocar a estrutura de estado do interpretador (*luaL_newstate*) e abrir as bibliotecas básicas de Lua (*luaL_openlibs*). Seguidamente, é importante limpar todo o armazenamento temporário

existente na pilha de memória (*lua_settop*), já que toda a passagem de dados de C para Lua e de Lua para C é feita através deste mecanismo.

Como referido anteriormente, por razões de segurança e de privacidade, o *iOS* usa um processo de *sandbox*, responsável por armazenar a aplicação e os respetivos dados [108]. Assim, cada aplicação tem uma pasta "Documents" para a qual pode ler/escrever sendo possível, através da função *NSSearchPathForDirectoriesInDomains* [113], obter a respetiva localização da pasta.

Como Lua requer todo o caminho de um ficheiro para o ler ou escrever, é necessário determinar exatamente onde este reside. Através de um estudo mais pormenorizado da API C fornecida, foi possível encontrar uma solução. Assim, o código seguinte permite criar uma estrutura de dados (*AppFolders*) de modo a armazenar numa variável global (*docPath*) a localização da pasta "Documents" da aplicação. Depois, o valor da variável deverá ser concatenado com o nome do ficheiro a manipular.

```
   NSArray *paths = NSSearchPathForDirectoriesInDomains (NSDocumentDirectory,
NSUserDomainMask, YES);
   NSString *documentsDirectory = [paths objectAtIndex:0];
   lua_createtable(L, 0, 2);
   lua_pushstring(L, [documentsDirectory UTF8String]);
   lua_setfield( L, -2, "docPath" );
   lua_setglobal(L, "AppFolders");
```

Quando é chamada a função *require*, o interpretador procura uma série de locais predefinidos para o *script* exigido. Devido ao processo de *sandbox* nenhum desses locais predefinidos fazem parte dos recursos da aplicação, pelo que a função nunca vai encontrar o que procura. Será assim necessário incluir a localização da pasta da aplicação nas variáveis globais do interpretador. Para o efeito foi necessário codificar a função *setLuaPath*, que recebe como argumento um valor fornecido pelo método *mainBundle* da classe *NSBundle* [114], permitindo manipular e ajustar a variável global *package.path*.

```
int setLuaPath( NSString* path, lua_State *L )
{
    lua_getglobal( L, "package" );
    lua_getfield( L, -1, "path" );
    NSString * cur_path = [NSString stringWithUTF8String:lua_tostring( L, -1 )];
```

```
    cur_path = [cur_path stringByAppendingString:@";"];
    cur_path = [cur_path stringByAppendingString:path];
    cur_path = [cur_path stringByAppendingString:@"/?.lua"];
    lua_pop( L, 1 );
    lua_pushstring( L, [cur_path UTF8String]);
    lua_setfield( L, -2, "path" );
    lua_pop( L, 1 );
    return 0;
}
...
   NSString *luaFilePath = [[NSBundle mainBundle] pathForResource:@"MainTestMyObject"
ofType:@"lua"];
   setLuaPath([luaFilePath stringByDeletingLastPathComponent], L);
```

A última codificação necessária nesta implementação compreende a chamada ao ficheiro a executar. Este, através da função *luaL_loadfile*, é carregado e compilado, devolvendo o interpretador um código diferente de zero em caso de erro. A verificação desse código, através da função *luaL_error*, permite obter uma mensagem de erro mais objetiva e descriminada. Finalmente, a execução do *script* é feita usando a função *lua_pcall*, ou seja, em modo protegido, para possibilitar que o interpretador consiga detetar e devolver o respetivo código de um eventual erro.

```
   int err;
   err = luaL_loadfile(L, [luaFilePath cStringUsingEncoding:[NSString
defaultCStringEncoding]]);
   if (0 != err) {
         luaL_error(L, "cannot compile lua file: %s", lua_tostring(L, -1));
         return;
   }
   err = lua_pcall(L, 0, 0, 0);
   if (0 != err) {
         luaL_error(L, "cannot run lua file: %s", lua_tostring(L, -1));
         return;
   }
   lua_close(L);
```

A aplicação foi executada corretamente (Figura 14.) recorrendo ao simulador do dispositivo iOS, tendo sido concluídas, com sucesso, todas as tarefas pretendidas. Também se verificou que os dados serializados em outros ambientes computacionais foram devidamente des-serializados. Devido ao processo de *sandbox*, esta tarefa obrigou que o ficheiro serializado

fosse previamente copiado para a pasta “Documents” da aplicação, através da utilização do terminal do sistema.

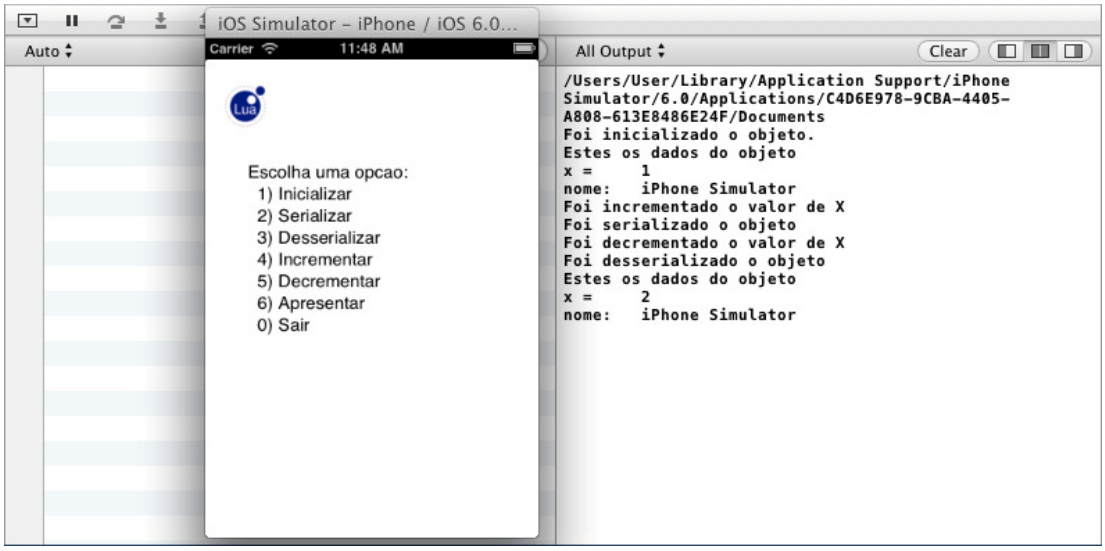

Figura 14 – Exemplo de execução de serialização em Lua no simulador de ambiente iOS

A Figura 14 apresenta a vista criada para a aplicação, exibida no simulador de iOS, assim como o *output* gerado, por cada tarefa, durante a execução do programa em modo de depuração.

4.2.4. Perl

O código fonte para implementação em Perl consta no Anexo D: o Programa 16 cria a classe *MyObject*; o Programa 17 apresenta o teste de serialização para execução em *Windows*, *Linux*, *Mac OS X*; o Programa 18, idem para *Android*; o Programa 19 e Programa 20 fazem parte do projeto que possibilita a execução numa aplicação *iOS*.

Perl não fornece nenhuma sintaxe especial para a construção de um objeto. Os objetos são apenas estruturas de dados (*hashes*, matrizes, escalares, *filehandles*, etc.) que são explicitamente associados a uma classe em particular.

As classes são definidas através de *packages* (pacotes), sendo criadas como referências pertencentes a um determinado pacote, através do recurso à função *bless*, tornando assim possível chamar funções para essas referências, ou seja, os métodos.

Existem vários módulos disponíveis para serialização de dados, tendo sido usado, no exemplo, o módulo *Storable* [115] que faz parte da distribuição padrão.

O programa foi executado nos ambientes *Windows* (Figura 15.), *Linux* (Figura 16.) e *Mac OS X* (Figura 17.), realizando com sucesso as tarefas de instanciação, inicialização e serialização.

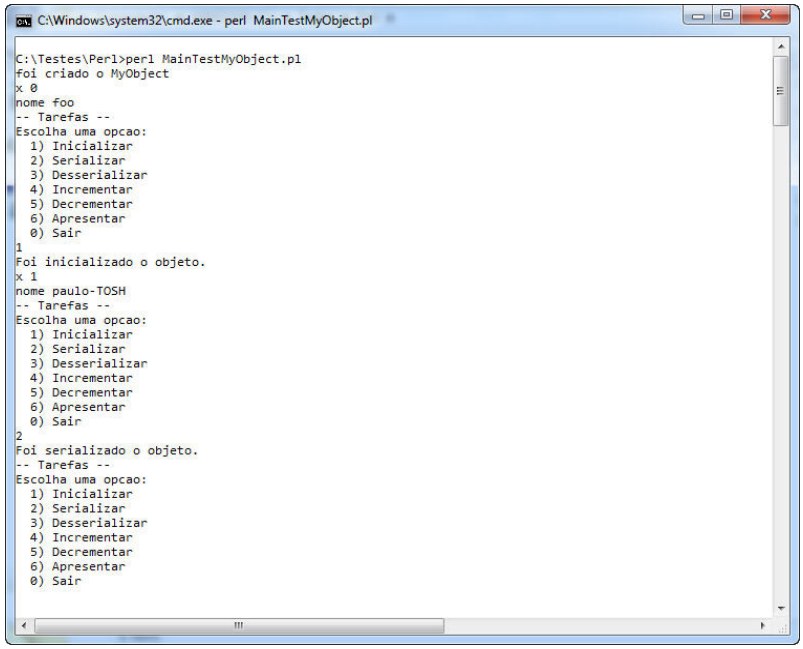

Figura 15 – Execução do programa de serialização em Perl no ambiente Windows

Na Figura 15 são apresentadas as tarefas de inicialização e serialização no ambiente *Windows*.

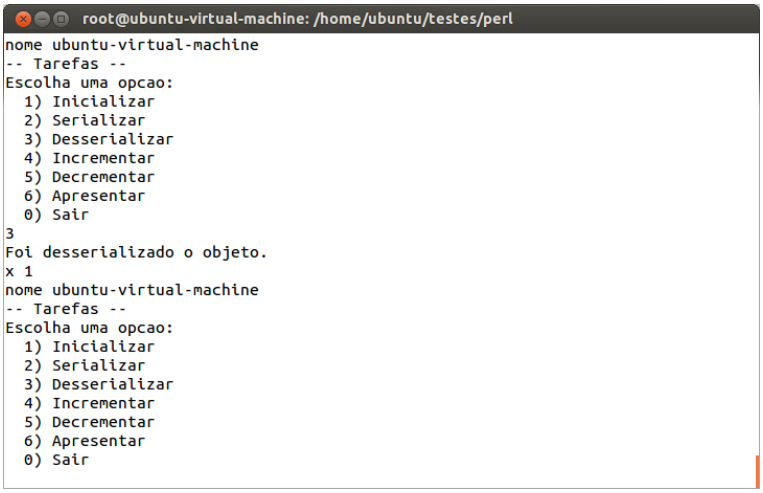

Figura 16- Execução do programa de serialização em Perl no ambiente Linux

Na Figura 16 é exibida a tarefa de de-serialização e a representação textual do objeto durante uma execução em ambiente *Linux*, tendo sido, primeiramente, o objeto inicializado e incrementado.

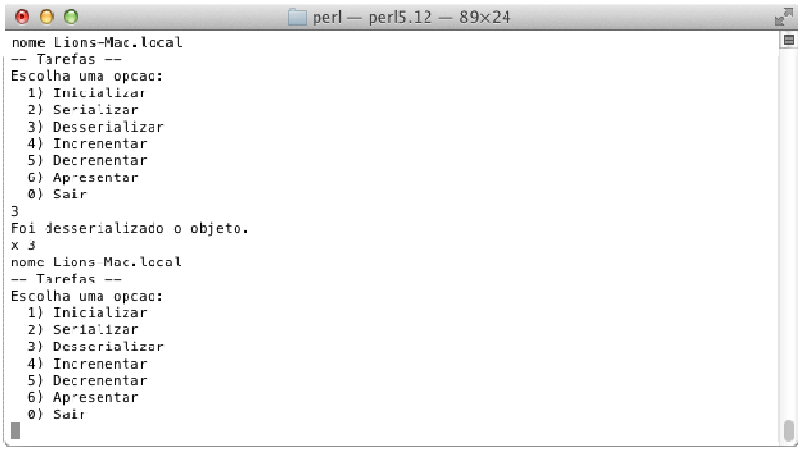

Figura 17 – Execução do programa de serialização em Perl no ambiente Mac OS X

Na Figura 17 é dado a observar a representação textual do objeto após as tarefas de inicialização, vários incrementos e execução da serialização no ambiente *Mac OS X*.

Verificou-se que a serialização de dados num sistema de 64 *bits* não pode ser lida por um sistema de 32 *bits*, sucedendo o mesmo na escrita em 64 *bits* e correspondente tentativa de leitura em 32 *bits*, originando um erro, como a Figura 18 ilustra. Esta incompatibilidade entre arquiteturas está referenciada [116] na documentação do módulo *Storable*.

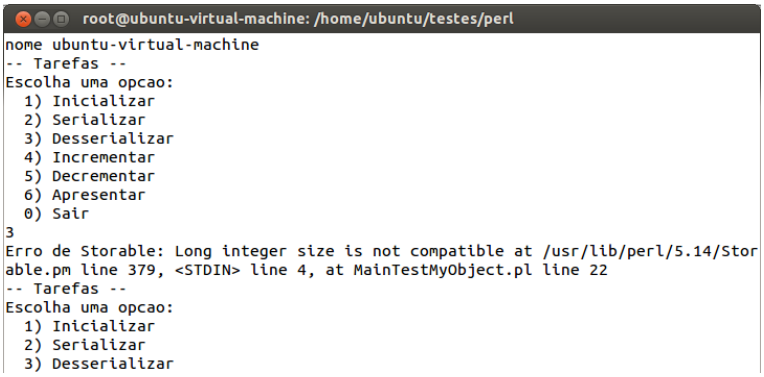

Figura 18 - Erro no processo de-serialização em Perl entre arquiteturas de 32 e 64 bits

Para a execução em ambiente *Android* foi instalado o interpretador de Perl (versão 5.10) [117], *perl_for_android*, atualmente disponível para utilizar com a biblioteca SL4A. Foi

necessário editar o código fonte do programa e incluir a seguinte linha para aceder à API do dipositivo:

```
use Android;
```

Devido ao processo de compilação usado na sua construção, este interpretador apenas disponibiliza módulos puramente escritos na linguagem Perl. No entanto, certos módulos exigem funcionalidades, extendidas, através de bibliotecas dinâmicas, escritas em código C, que necessitam de ser compiladas e carregadas para serem usadas pelo interpretador. Uma biblioteca compilada é designada pelo Perl como o "loadable object".

O módulo *Storable* possui a sua própria biblioteca dinâmica, escrita em C, que não está portada para o interpretador usado na SL4A. Deste modo, não é possível realizar as tarefas de serialização exigidas no teste (Figura 19.), limitando a execução do programa em *Android* à instanciação e inicialização do objeto, como também à chamada dos respetivos métodos.

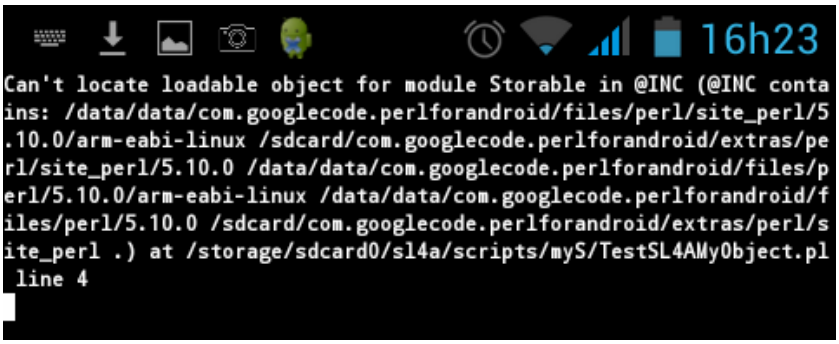

Figura 19 - Erro de dependência de biblioteca dinâmica de módulo Perl em ambiente Android

A Figura 19 apresenta o erro de dependência de biblioteca dinâmica do módulo *Storable*, produzindo uma mensagem com a informação das localizações possíveis onde foi procurada a biblioteca requerida. Isto, porque, a linguagem Perl possui uma variável especial, @INC, equivalente à variável PATH da *shell* de comandos de um sistema operativo, que contém uma lista de diretórios a partir da qual os módulos Perl e bibliotecas podem ser carregadas.

À semelhança do realizado para as linguagens Python e Lua, optou-se também por incorporar o interpretador Perl numa aplicação nativa *iOS*. A inclusão do interpretador é feita, manualmente, através da máquina local e consiste em localizar e adicionar a pasta “CORE”, que contém as bibliotecas padrão da linguagem, num novo projeto Xcode. Seguidamente

deve-se compilar o projeto para criar as dependências necessárias. A inclusão do cabeçalho "perl.h" nos ficheiros onde se pretende utilizar a linguagem completa o processo.

A funcionalidade mínima necessária para executar um *script* Perl dentro de uma aplicação *iOS* consiste em algumas linhas de código: chamadas de função que alocam o interpretador, analisar o *script*, executar o *script* e finalizar o interpretador. Essas linhas são apresentadas a seguir.

```
#include "perl.h"

static PerlInterpreter *my_perl=NULL;
perl_construct(my_perl);
perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
perl_run(my_perl);
perl_destruct(my_perl);
perl_free(my_perl);
```

Tudo isto poderia ser conseguido pela edição do respetivo programa principal "main.m". No entanto, é necessário saber qual o diretório da aplicação e dos dados definidos pela *sandbox* do iOS [108]. Também é necessário utilizar outros módulos Perl.

Assim, no ficheiro "ViewController.h" (Anexo D, Programa 19) é feita a codificação necessária à implementação da aplicação. No seu ambiente normal de execução, o Perl irá automaticamente inicializar os módulos conforme a necessidade, mas isto não acontece quando é incorporado. Para que o interpretador consiga comunicar com outros módulos, entre os quais o *MyObject*, usado para o efeito do teste, terá de ser feita uma chamada ao programa [118] para forçar a inicialização de todos os módulos necessários.

```
static void xs_init (pTHX);
EXTERN_C void boot_DynaLoader (pTHX_ CV* cv);
EXTERN_C void xs_init(pTHX)
{
    char *file = __FILE__;
    newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, file);
}
. . .

perl_parse(my_perl, xs_init, 2, my_argv, NULL);
```

O diretório da aplicação e de dados, assim como o nome do dispositivo, são obtidos através métodos e funções disponibilizados pelo Objetive-C. Fazendo uso da função *eval_pv* [119] é possível validar cadeias de carateres em código Perl de modo a serem usadas como variáveis

dentro de uma função do *script*. Será, deste modo, criada a variável *mypath*, destinada a guardar a localização da pasta de dados assim como a variável *mydevice* para conter o nome do dispositivo.

```
    const char *cDocumentsDirectory = [documentsDirectory
cStringUsingEncoding:NSUTF8StringEncoding];
    const char *cDevHostName = [devHostName cStringUsingEncoding:NSUTF8StringEncoding];
    . . .
    NSString *myeval = [NSString stringWithFormat:@"\"$mypath='%s';\"", cDocumentsDirectory
];
    NSString *mydevice = [NSString stringWithFormat:@"\"$mydevice='%s';\"", cDevHostName ];
    . . .
    eval_pv((const char *)myeval, TRUE);
    eval_pv((const char *)mydevice, TRUE);

    call_argv("main", G_DISCARD | G_NOARGS, args);
```

O *script* a executar (Programa 20) inclui alterações necessárias para ajustar a localização do ficheiro que guarda os dados seralizados (fazendo uso da variável *mypath*). Foi codificada uma nova função, *prepara()*, que faz uso do módulo padrão *FinBin* [120] para compor o ambiente de execução.

A aplicação foi executada corretamente (Figura 20.), recorrendo ao simulador do dispositivo iOS, tendo sido realizadas, com sucesso, as tarefas de instanciação, inicialização e serialização. A de-seralização entre diferentes arquiteturas, como referida anteriormente nos testes dos outros sistemas operativos, gera erro. Também, à semelhança de Lua, o ficheiro serializado foi previamente copiado para a pasta "Documents" da aplicação, através da utilização do terminal do sistema.

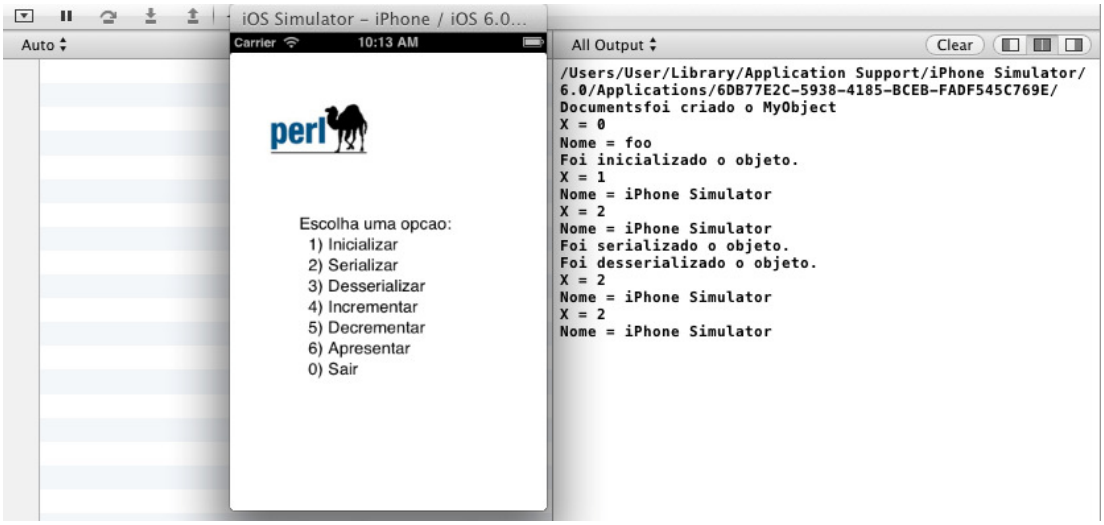

Figura 20 – Execução do programa de serialização em Perl no simulador de ambiente iOS

A Figura 20 apresenta a vista criada para a aplicação, exibida no simulador de iOS, assim como o *output* gerado, em cada tarefa, na execução do programa em modo de depuração.

4.3. Teste ao suporte de carregamento dinâmico

O carregamento dinâmico (*dynamic loading*) é um mecanismo pelo qual um programa de computador pode, em tempo de execução, carregar uma biblioteca ou código na memória, recuperar os endereços de funções e variáveis contidas na biblioteca, executar essas funções ou aceder a essas variáveis, e finalmente libertar a biblioteca da memória.

O teste realizado neste estudo consiste na utilização do objeto *MyObject*, definido anteriormente no ponto *4.2.*, que será carregado dinamicamente num programa em execução. De referir que o objeto estará na mesma localização (pasta) do programa e não será usado qualquer mecanismo de importação existente em cada linguagem. No programa, será declarada e instanciada uma variável para referenciar o objeto usando os métodos disponibilizados. Finalmente, serão executados os métodos que permitirão atribuir valores aos atributos da variável e a sua impressão textual.

A análise de suporte de cada linguagem incidiu sobre:

- correta compilação do código fonte;
- instanciação e inicialização do objeto;
- apresentação textual dos métodos disponibilizados;
- execução dos métodos após o carregamento dinâmico;
- necessidade de utilização de módulos/bibliotecas adicionais.

O seguinte algoritmo serviu como base para a implementação em cada uma das linguagens:

```
Algoritmo "LoadTestMyObject"
Var    classNameToBeLoaded: texto
       myClass: MyObject

Início
       Ler(classNameToBeLoaded)
       Declarar variável myClass para referenciar classNameToBeLoaded
       Instanciar myClass
       Obter lista de métodos de myClass
       Imprimir lista de métodos de myClass
       Atribuir valor 100 ao atributo X de myClass
       Atribuir valor 'localhost' ao atributo Nome de myClass
       Decrementar o atributo X de myClass
       Imprimir valores de myClass
Fim-algoritmo.
```

4.3.1. Java

O código fonte em Java para execução em *Windows*, *Linux* e *Mac OS X* consta no Anexo A (Programa 3).

A linguagem Java oferece, através da *Reflexion API (java.lang.reflect)* [121], a possibilidade de programaticamente obter a informação de uma classe, em tempo de execução. Este processo, designado por reflexão ou introspeção, consiste na capacidade de um objeto fornecer informação sobre si mesmo, permitindo saber quais são os seus métodos, propriedades e construtores, como também executar métodos das classes que foram carregadas dinamicamente.

O programa de teste foi executado corretamente no *Windows* (Figura 21.), no *Linux* (Figura 22.) e *Mac OS X* (Figura 23.). Devido à linguagem não ser suportada em *Android* e *iOS*, não foi possível realizar o teste nesses sistemas.

Figura 21 – Execução do carregamento dinâmica de Java em ambiente Windows

A Figura 21 ilustra uma execução do programa em ambiente *Windows*, sendo possível visualizar que a linguagem Java apresenta uma informação detalhada sobre cada um dos métodos do objeto, na qual se inclui o tipo de dados de entrada e de saída. Verifica-se também o valor devolvido por cada um dos métodos, após o carregamento dinâmico da classe *MyObject*, ou após a execução das tarefas de incrementação, decrementação e representação textual, exigidas no teste.

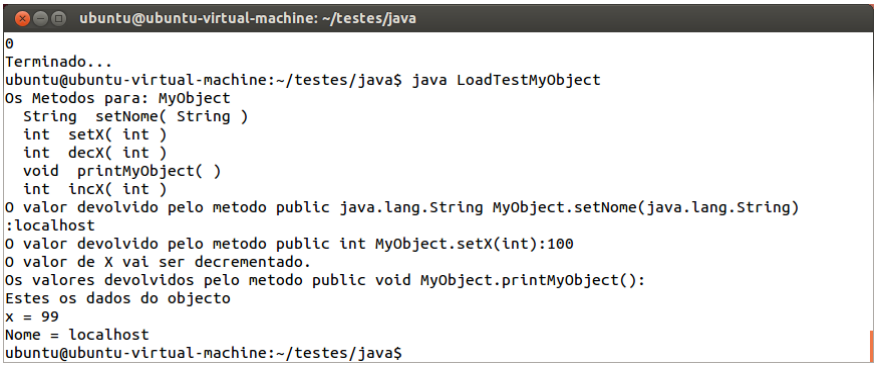

Figura 22 – Execução do carregamento dinâmico de Java em ambiente Linux

A Figura 22, *idem*, mas para a execução em ambiente *Linux*.

Figura 23 – Execução do carregamento dinâmico de Java em ambiente Mac OS X

A execução em ambiente *Mac OS X* das tarefas, exigidas no teste, é ilustrada pela Figura 23.

4.3.2. Python

O código fonte Python para teste de carregamento dinâmico consta no Anexo B (Programa 9).

Para carregar dinamicamente uma classe deve-se, primeiro, fazer a importação do módulo onde ela 'reside'. Ao contrário da diretiva *import* que carrega toda uma *package*, a função *__import__()* [122] recebe uma *string* como argumento (o nome do módulo) e devolve um objeto representando o módulo.

O módulo *inspect* [123], disponibilizado pela linguagem, permite identificar e apresentar as suas estruturas internas, tais como o âmbito de variáveis, métodos e atributos.

O programa de teste foi executado corretamente em *Windows* (Figura 24.), *Linux* (Figura 25.), *Mac OS X* (Figura 26.), *Android* (Figura 27.) e *iOS* (Figura 28.), não requerendo edição adicional para adaptabilidade a qualquer um destes ambientes computacionais.

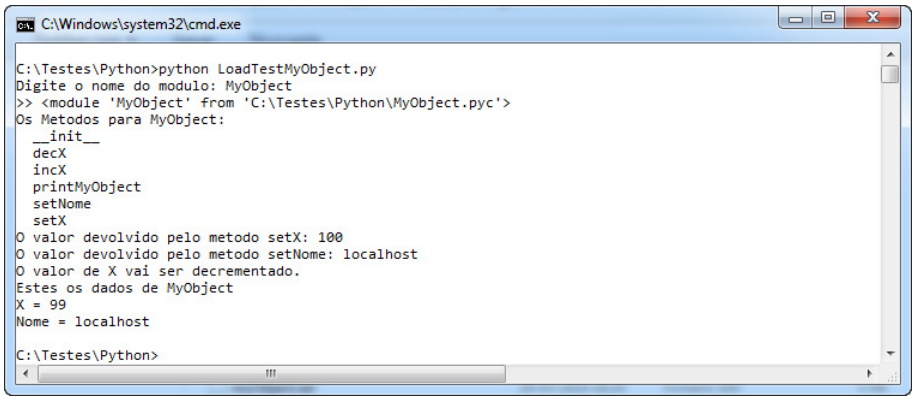

Figura 24 – Execução do carregamento dinâmico de Python em ambiente Windows

A Figura 24 apresenta a execução do programa em ambiente *Windows*, sendo possível visualizar os métodos do objeto, como o resultado obtido na atribuição de valores às variáveis de instância. De notar que, em Python, não é fornecida a informação do tipo de dados de entrada e/ou de saída, devido à tipagem dinâmica existente nesta linguagem. Também se verifica que é exibido um método especial, "__init__", não definido em *MyObject*.

Em Python, novos objetos são criados a partir das classes através de atribuição. O objeto é uma instância da classe, que possui características próprias. Quando um novo objeto é criado, o construtor da classe é executado. Este construtor é um método especial, chamado "__new__". Após a chamada ao construtor, o método "__init__" é chamado para inicializar a nova instância.

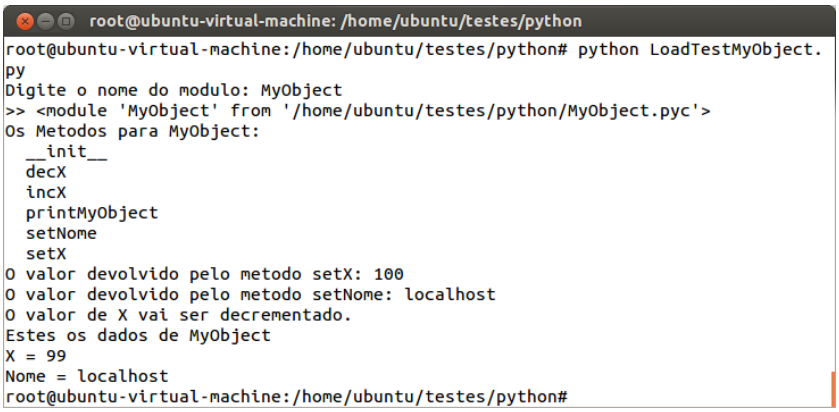

Figura 25 – Execução do carregamento dinâmico de Python em ambiente Linux

A Figura 25 ilustra a execução do programa em ambiente *Linux* e o resultado obtido é semelhante ao apresentado para *Windows*.

Figura 26 – Exemplo de execução do carregamento dinâmico de Python em ambiente Mac OS X

A execução do programa em ambiente *Mac OS X*, ilustrado pela Figura 26, não apresenta qualquer diferença ao resultado obtido, até ao momento, nos outros sistemas operativos.

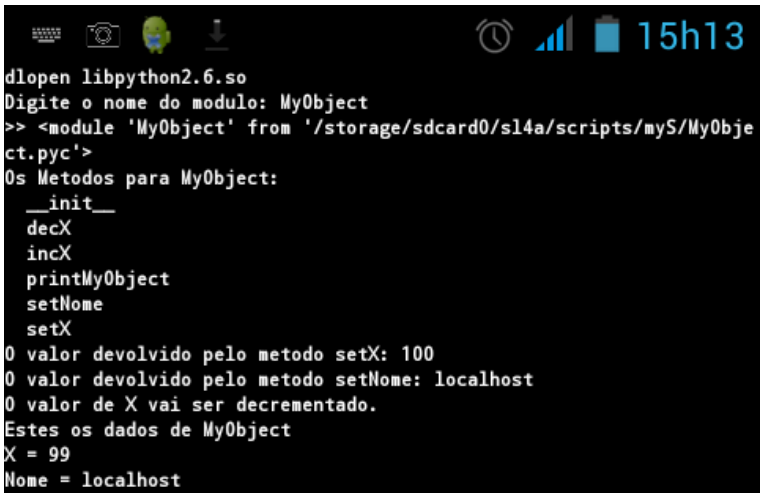

Figura 27 – Execução do carregamento dinâmico de Python em ambiente Android com SL4A

A execução em ambiente *Android*, como ilustra a Figura 27, segue o padrão anterior, alcançando todos os objetivos exigidos no teste de carregamento dinâmico.

De referir que, o código gerado na implementação do teste de carregamento dinâmico para a execução em *Windows*, *Linux*, *Mac OS X* e *Android*, é igual para todos estes ambientes computacionais, o que confere um elevado grau de portabilidade da linguagem Python.

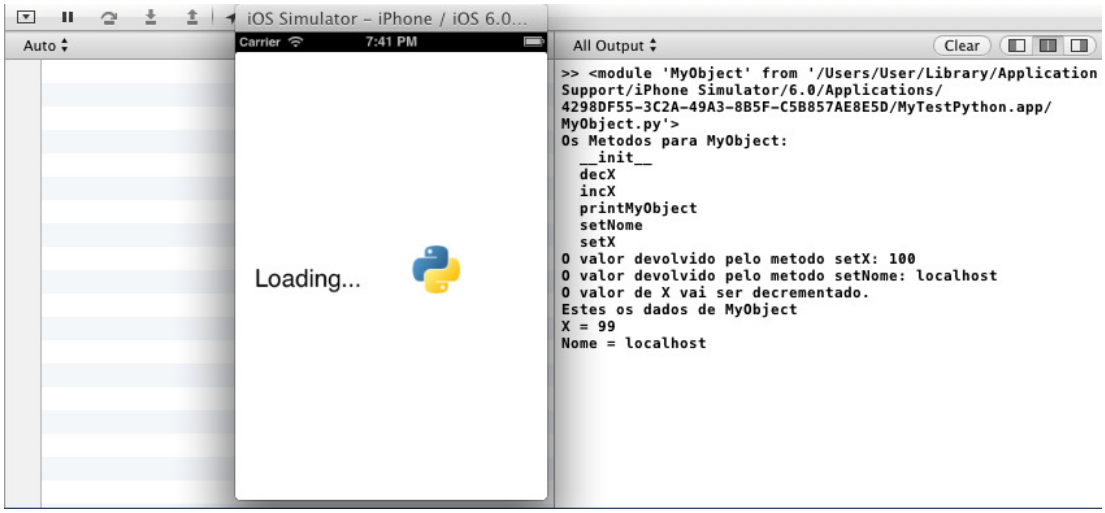

Figura 28 – Execução do carregamento dinâmico de Python em ambiente iOS

A Figura 28 apresenta a execução em ambiente *iOS*, na qual se inclui a vista criada para a aplicação, exibida no simulador de iOS, como também é possível visualizar o *output* gerado com a informação textual dos métodos e resultados obtidos, durante a execução do programa em modo de depuração.

Para a implementação de código do programa de teste, no sistema *Android* recorreu-se novamente ao uso da SL4A; no *iOS* fez-se a adaptabilidade da aplicação anteriormente criada, mais concretamente no ficheiro "ViewController.m" (Anexo B, Programa 7, linha 43) para fazer a chamada ao *script* a executar.

```
   const char * prog = [
            [[NSBundle mainBundle] pathForResource:@"LoadTestMyObject" ofType:@"py"]
cStringUsingEncoding: NSUTF8StringEncoding];
```

4.3.3. Lua

O código fonte para o teste de carregamento dinâmico em Lua consta no Anexo C. O Programa 15 produz o teste de carregamento dinâmico para execução em *Windows*, *Linux* e *Mac OS X*; o Programa 16, *idem* para *Android*; o Programa 17 possibilita a execução numa aplicação *iOS*.

Em Lua existem várias formas para efetuar o carregamento de código externo, seja uma biblioteca escrita em C/C++, como também código escrito noutra ou na própria linguagem.

As funções existentes são *loadstring*, *load*, *loadfile* e *dofile*, sendo que a última também executa o código [124].

A função *dofile* só deve ser usada quando existe a certeza que o arquivo a executar não produzirá erros, caso contrário, todo o programa *Lua* irá falhar. Fazendo uso da função *loadfile*, que executa em modo protegido, quando não existem erros o código do ficheiro é compilado e devolvido como uma função.

Em *Lua*, um objeto é meramente uma tabela, contendo campos com os seus dados (variáveis de instância) e operações (métodos). Também não é declarado o tipo das variáveis, uma vez que o tipo é determinado dinamicamente, dependendo do valor que ela armazena. Assim, para determinar quais os métodos existentes em *MyObject*, foi necessário usar um ciclo *for*, genérico, que usa a função *pairs* para percorrer as chaves da metatabela que definem o objeto. No entanto, não existe nenhuma forma de obter o nome desta metatabela dado que, quando o programa é compilado para *bytecode* o nome nunca mais é usado.

O programa de teste foi executado corretamente em *Windows* (Figura 29.), *Linux* (Figura 30.) e *Mac OS X* (Figura 31.).

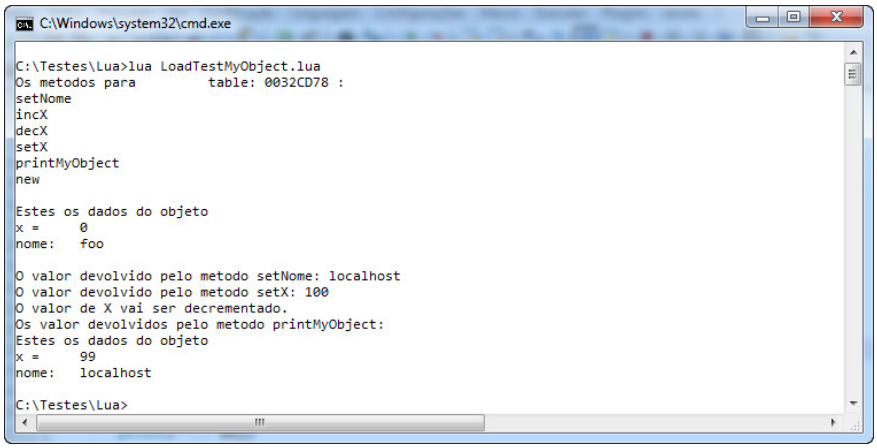

Figura 29 – Execução do carregamento dinâmico de Lua em ambiente Windows

A Figura 29 apresenta a execução do programa em ambiente *Windows*, na qual se verifica que a tabela, usada pela linguagem Lua para definir o objeto, exibe as operações (os métodos da classe) que lhe estão associadas. O método "new()" está incluído no código que implementa o *MyObject* (Anexo C, Programa 10), funcionando como um construtor de classe. Deste modo,

justifica-se a exibição textual dos valores das variáveis de instância, antes da tarefa de inicialização (*x=0*; *nome: foo*).

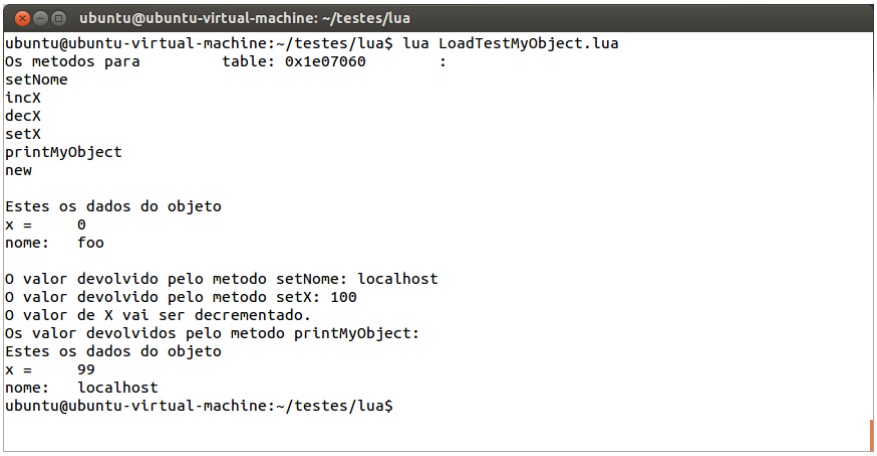

Figura 30 – Execução do carregamento dinâmico de Lua em ambiente Linux

A execução do programa em ambiente *Linux* é ilustrada pela Figura 30, tendo-se obtido resultados iguais ao ambiente *Windows* e, neste caso, uma informação textual dos métodos idêntica ao programa reproduzido em *Windows*.

Figura 31 – Execução do carregamento dinâmico de Lua em ambiente Mac OS X

A Figura 31 apresenta a execução do programa em ambiente *Mac OS X* e, neste caso, apesar de os resultados obtidos serem iguais, verifica-se que a informação textual dos métodos é diferente. Isto resulta de os pares (conjunto chaves e de valores) da tabela serem armazenados de uma forma arbitrária e, consequentemente, a tabela não é ordenada sempre da mesma forma. Para o efeito de uma pré-visualização ordenada dos métodos, será necessário codificar uma função que empregue um algoritmo de ordenação.

No sistema *Android*, foi necessária a edição do código (Programa 16) para adequar a sua execução no dispositivo com recurso à SL4A. Tal como no teste de serialização (Ponto 4.2.3), para além da inclusão das linhas de código que permitem obter acesso à API C e configurar o caminho de módulos utilizados, foi necessário indicar a localização exata da classe *MyObject* que se pretende carregar. Como a função *loadfile* faz uma procura no diretório currente da aplicação e a SL4A possui uma pasta "Scripts", onde realmente os programas residem, caso não seja feito este ajuste é originado um erro de execução.

```
local f, s = loadfile ("Scripts/MyObject.lua")
```

Após as alterações, o programa foi devidamente executado.

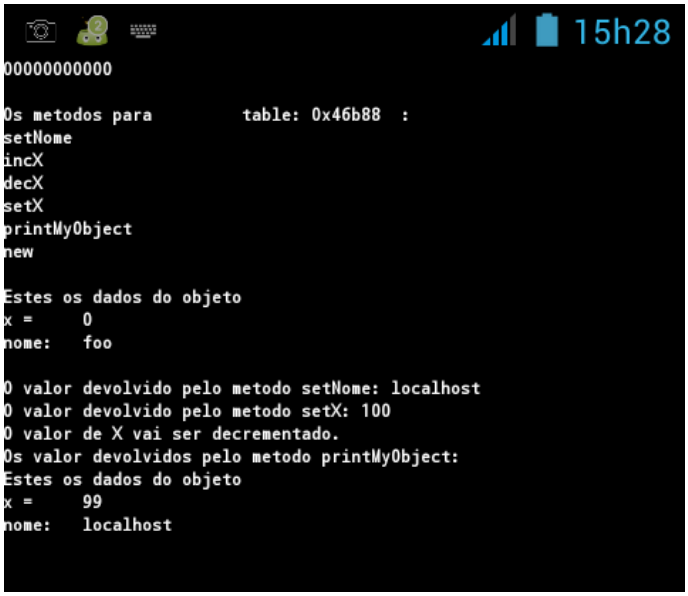

Figura 32 – Execução do carregamento dinâmico de Lua em ambiente Android com SL4A

Um exemplo de execução do programa em ambiente Android com recurso à SL4A é ilustrado pela Figura 32, tendo sido obtido os resultados pretendidos para o teste.

Para a aplicação nativa criada em iOS, com a edição da linha de código que efetua a chamada ao *script* existente no "ViewController.m", é possível executar o teste.

```
    NSString *luaFilePath = [[NSBundle mainBundle] pathForResource:@"LoadTestMyObject"
ofType:@"lua"];
```

Após a alteração, o programa (Programa 17) foi devidamente executado.

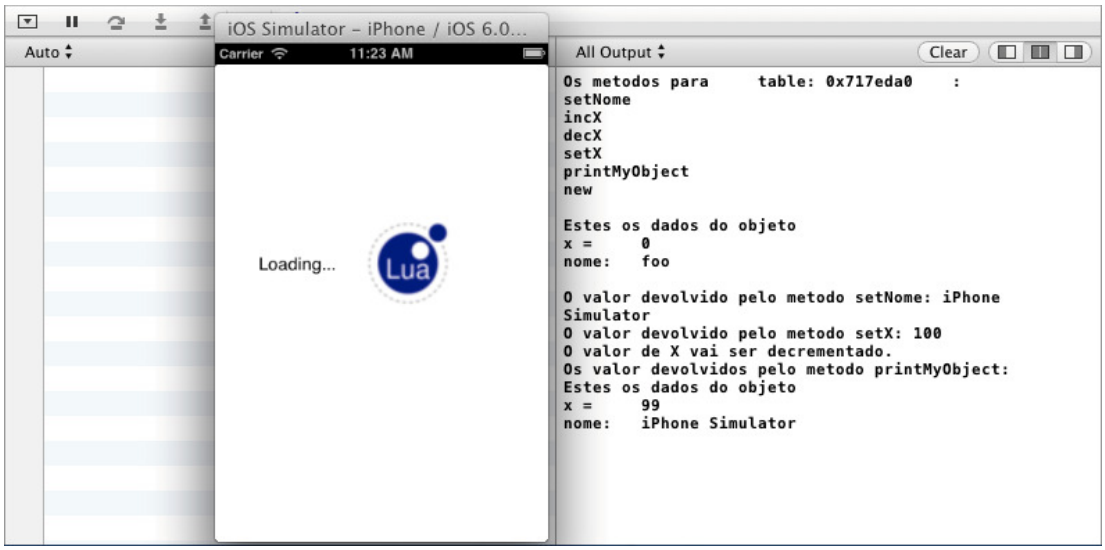

Figura 33 – Execução do carregamento dinâmico de Lua no ambiente iOS

A Figura 33 apresenta a execução em ambiente *iOS*, na qual se inclui a vista criada para a aplicação, exibida no simulador de iOS, como também é possível visualizar o *output* gerado com a informação textual dos métodos e resultados obtidos, durante a execução do programa em modo de depuração.

4.3.4. Perl

O código fonte para o teste de carregamento dinâmico em Perl consta no Anexo D. O Programa 21 produz o teste de carregamento dinâmico para execução em *Windows*, *Linux* e *Mac OS X*; o Programa 22, *idem* para *Android*; o Programa 23 possibilita a execução numa aplicação *iOS.*

A capacidade de carregar dinamicamente um módulo foi conseguida através do recurso ao módulo *Module::Load* [125] disponibilizado na distribuição padrão.

Apesar de existirem várias formas de obter informação sobre a classe carregada, recorreu-se ao módulo *Class::Inspector* [126] para o efeito. Este módulo proporciona uma listagem ordenada alfabeticamente pelo nome dos métodos.

O programa de teste foi executado corretamente em *Windows* (Figura 34.), *Linux* (Figura 35.) e *Mac OS X* (Figura 36.).

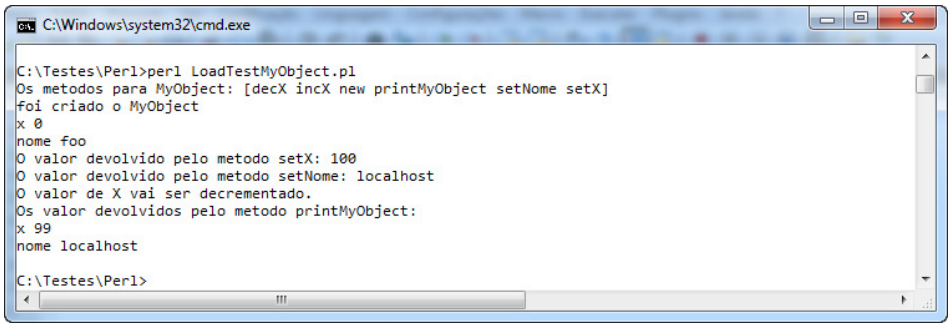

Figura 34 – Execução do carregamento dinâmico de Perl em ambiente Windows

Na Figura 34 é possível visualizar a execução do programa em ambiente *Windows*, denotando-se a listagem ordenada da informação textual dos métodos, onde se inclui o método "new()" codificado para intanciar a classe *MyObject* (Anexo D, Programa 16). Deste modo, justifica-se a exibição, no ecrã do programa, dos valores das variáveis de instância antes da tarefa de inicialização (*x 0*; *nome foo*).

Figura 35 – Execução do carregamento dinâmico de Perl em ambiente Linux

Na Figura 35 pode-se visualizar um exemplo de execução do programa em ambiente *Linux*, tendo-se obtido resultados iguais aos produzidos em *Windows*.

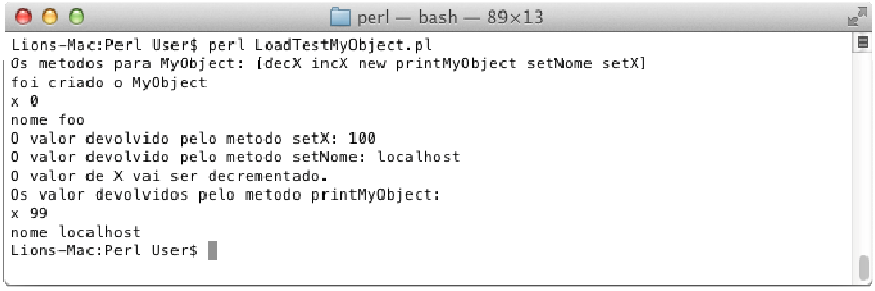

Figura 36 – Execução do carregamento dinâmico de Perl em ambiente Mac OS X

A Figura 36 ilustra um exemplo de execução no ambiente *Mac OS X*, também com resultados iguais aos anteriormente verificados nos outros sistemas operativos.

Pelas razões descritas na secção 4.2.4., o programa não foi executado em *Android* devido à necessidade de compilação de bibliotecas dinâmicas que extendem funcionalidades dos módulos utilizados (Figura 37.).

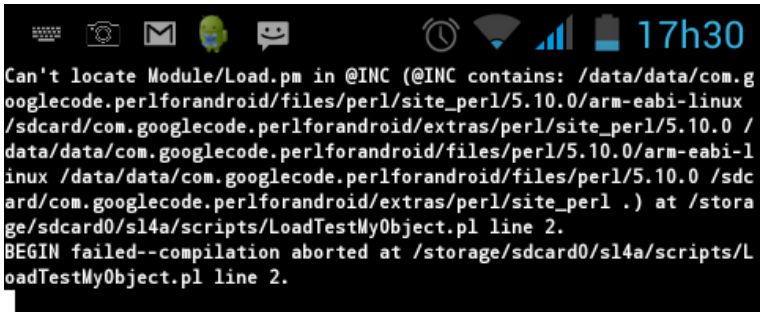

Figura 37 - Erro de carregamento de extensão de módulo Perl em ambiente Android

A Figura 37 exibe o erro produzido, em ambiente *Android*, relativamente à falha na localização da biblioteca dinâmica requerida pelo módulo *Module::Load*.

Para a aplicação nativa criada em *iOS*, procedeu-se à edição da linha de código que efetua a chamada ao *scrip*t existente no "ViewController.m".

```
    const char * prog = [
                         [[NSBundle mainBundle] pathForResource:@"LoadTestMyObject"
ofType:@"pl"] cStringUsingEncoding: NSUTF8StringEncoding];
```

Após a alteração, o programa (Programa 23) implementado, este é devidamente executado.

Figura 38 – Execução do carregamento dinâmico de Perl no ambiente iOS

A Figura 38 apresenta a execução em ambiente *iOS*, fazendo uso do simulador de iOS. É também possível visualizar o *output* gerado com a informação textual dos métodos e resultados obtidos, durante a execução do programa em modo de depuração.

4.4. Análise dos resultados

Os resultados obtidos nos testes expostos anteriormente foram resumidos na Tabela 5, apresentando o desempenho das tarefas definidas previamente, por ambiente computacional, sendo exibido o nome da linguagem em caso de sucesso.

Tabela 5 - Quadro resumo da análise de desempenho das linguagens

Teste de desempenho das linguagens por ambiente computacional:	*Windows*	*Linux*	*Mac OS X*	*Android*	*iOS*
Compilação do código fonte? Interpretação? Compilação e/ou interpretação?	Java Python Lua Perl	Java Python Lua Perl	Java Python Lua Perl	Python Lua Perl	Python Lua Perl
Instanciação e inicialização de objetos?	Java Python Lua Perl	Java Python Lua Perl	Java Python Lua Perl	Python Lua Perl	Python Lua Perl
Invocação dos métodos dos objetos?	Java Python Lua Perl	Java Python Lua Perl	Java Python Lua Perl	Python Lua Perl	Python Lua Perl
Serialização e de-serialização de dados gerados no mesmo ambiente computacional?	Java Python Lua Perl	Java Python Lua Perl	Java Python Lua Perl	Python Lua	Python Lua Perl
Serialização e de-serialização de dados entre diferentes ambientes computacionais?	Java Python Lua	Java Python Lua	Java Python Lua	Python Lua	Python Lua

Teste de desempenho das linguagens por ambiente computacional:	*Windows*	*Linux*	*Mac OS X*	*Android*	*iOS*
Necessidade de instalação de módulos e/ou bibliotecas adicionais não incluídas na distribuição padrão?	Lua	Lua	Lua	Lua	Lua
Carregamento dinâmico de classe/objeto?	Java Python Lua Perl	Java Python Lua Perl	Java Python Lua Perl	Python Lua	Python Lua Perl
Invocação dos métodos do objeto após o carregamento dinâmico?	Java Python Lua Perl	Java Python Lua Perl	Java Python Lua Perl	Python Lua	Python Lua Perl
Necessidade de edição de código fonte original para a adaptabilidade ao ambiente computacional?				Lua Perl	Lua Perl

Nos testes realizados verifica-se que a linguagem Python apresenta um notável desempenho, disponibilizando, na sua distribuição padrão, todos os módulos e extensões requeridas para a correta execução dos testes, além do código implementado ser portável para todos os ambientes computacionais testados.

O Python tem a capacidade de ser embutido (*embedded*) numa outra aplicação, tendo sido esta a opção tomada para possibilitar a realização dos testes para dispositivos com *iOS*. No entanto, é necessário algum trabalho adicional, como seja o de fornecer uma *framework* C/Objetive C que possibilite iniciar a aplicação e que inclua o interpretador.

A linguagem Lua também apresenta um elevado desempenho nos testes realizados. Considerada pelos seus autores como poderosa, rápida e leve, esta adjetivação confirma-se na sua utilização prática. A portabilidade de código entre os diversos ambientes computacionais é facilmente conseguida com pequenas alterações relacionadas, maioritariamente, com a localização relativa de módulos e outros recursos utilizados (ficheiros, pastas) em sistemas

operativos móveis. Também se verifica que é facilmente incorporada numa outra aplicação, como se verificou em *iOS*, fornecendo uma API C cuja manipulação é simples e eficaz.

O ponto menos forte de Lua prende-se com a disponilização de algumas funcionalidades requeridas que não fazem parte da sua distribuição padrão, entre estas, o método de serialização. Existindo módulos de terceiros para suprir esta necessidade, é necessário ter em atenção a compatibilidade com a versão da distribuição padrão usada podendo, em alguns casos, revelar-se alguma complexidade na respetiva instalação.

O desempenho obtido com Perl em função dos testes realizados e nos objetivos perseguidos, foi fraco, tendo apresentado uma má funcionalidade no que respeita à serialização de dados entre diferentes arquiteturas. A existência de bibliotecas dinâmicas para extender as funcionalidades de alguns dos módulos disponibilizados na distribuição padrão, necessitam de ser compiladas antes do carregamento pelo interpretador, sendo isto conseguido automaticamente e de modo transparente em ambientes computacionais de disponham de um compilador nativo de C. No caso de *Android* verificou-se que esta operação não foi possível devido ao interpretador disponível para a SL4A não incorporar essas bibliotecas ou outros ficheiros escritos em C.

A incorporação numa outra aplicação, em particular numa aplicação para dispositivos com *iOS*, está devidamente teorizada na documentação da linguagem Perl mas não faz uma explicação concreta sobre os argumentos das funções usadas, requerendo assim algum estudo na devida implementação prática.

O desempenho de Java é alto quando executado em ambientes computacionais tradicionais, como no caso de *Windows*, *Linux* e *Mac OS X*, mas o mesmo não se verifica perante sistemas operativos móveis. Em *Android* não existe um suporte da SL4A para a linguagem Java. Em *iOS*, devido às restrições impostas pela Apple, não foi possível implementar uma solução semelhante à utilizada nas restantes linguagens.

Após esta análise, para os objetivos do estudo, a linguagem mais indicada será Python e o modelo de implementação, como apresentado na Tabela 6, passará pelo seguinte:

Tabela 6 - Modelo de implementação da linguagem Python

Ambiente computacional:	*Requisitos necessários:*
Windows	- Instalação do interpretador da linguagem Python, versão 2.7.6; - Configuração de variáveis de ambiente e de sistema; - Desenvolvimento de aplicações com código Python; - Instalação do pacote Py2exe para conversão de um módulo Python num executável
Linux	- Desenvolvimento de aplicações com código Python.
Mac OS X	- Desenvolvimento de aplicações com código Python.
Android	- Instalação da biblioteca SL4A no dispositivo móvel; - Instalação do interpretador de Python, Py4a Release 6, no disposivo móvel; - Implementação do código Python em aplicação nativa; - Desenho de interfaces (Webviews).
iOS	- Carregamento da *framework* Python de forma embutida; - Implementação do código Python em aplicação nativa; - Desenho de vistas da aplicação.

Os sistemas operativos *Linux* e *Mac OS X* disponibilizam uma distribuição padrão da linguagem Python, sendo requerida a instalação do interpretador em *Windows*. Para a implementação em *Android*, é necessário descarregar e instalar, para o dispositivo móvel, a biblioteca SL4A, podendo o interpretador Py4A ser instalado através da biblioteca. É possível efetuar o desenvolvimento de *scripts* diretamente no dispositivo *Android* mas, para aplicações com um maior grau de complexidade e exigência de interação com o utilizador, este desenvolvimento deverá ser feito num sistema operativo (*Windows* ou *Linux*) com recurso a um IDE. O desenvolvimento para *iOS* é realizado numa máquina com o sistema operativo *Mac OS X*, fazendo uso da ferramenta de desenvolvimento Xcode, sendo necessário localizar a *framework* de Python para embutir numa aplicação nativa.

Capítulo 5

5. Conclusões e trabalho futuro

A capacidade de um AM se mover através de uma rede de larga escala torna possível o desenvolvimento de aplicações e serviços mais dinâmicos quando comparados com os desenvolvidos em outros modelos (cliente/servidor e RPC), podendo fornecer uma estrutura conveniente, eficiente e robusta para a implementação de aplicações distribuídas, incluindo aplicações móveis.

Partindo do estudo do paradigama de AM, no qual, entre outras, se procurou obter informação relevante sobre *frameworks* existentes, este trabalho apresenta um estudo de identificação, análise e avaliação de Linguagens de Programação potencialmente indicadas ao desenvolvimento de Agentes Móveis multiplataforma.

A constituição de um cenário de testes conseguiu expressar o desempenho das linguagens Java, Python, Lua e Perl, relativamente ao respetivo suporte de serialização de dados e de carregamento dinâmico (características fundamentais para a implementação de AM) no conjunto de ambientes computacionais: *Windows*, *Linux*, *Mac OS X*, *Android* e *iOS*.

Para além da segurança, não incluída no âmbito deste trabalho, a serialização e o carregamento dinâmico são um dos mecanismos mais importantes para o desenvolvimento de AM, já que permitem enviar um objeto para um ficheiro, ou através da rede, e voltar a instanciar, preservando os dados na altura da serialização.

Assim, a capacidade de execução de cada uma das linguagens nos diferentes ambientes computacionais revela-se de grande importância na procura de uma solução que permita implementar um *middleware* de suporte a AM multiplataforma.

Os resultados obtidos permitem aferir que Python e Lua apresentam desempenhos adequados a tal objetivo. As distribuições existentes para os tradicionais sistemas operativos multitarefas permitem o desenvolvimento de aplicações com um elevado grau de fiabilidade, para além de fazer uso do mesmo código fonte. A possibilidade de incorporação (*embedding*) do respetivo interpretador numa aplicação nativa para sistemas operativos móveis, permite realizar o desenvolvimento recorrendo a uma API C disponibilizada por cada linguagem.

Python apresenta, em relação a Lua, a vantagem de disponibilizar um vasto conjunto de bibliotecas padrão, permitindo assim a realização de tarefas mais exigentes e específicas. No caso de Lua, será necessário identificar uma biblioteca de terceiros compatível com a distribuição usada ou criar uma especificamente para o efeito pretendido.

A grande vantagem de Lua em relação a Python prende-se com a sua flexibilidade e simplicidade quando usada como linguagem de extensão. Para além da facilidade de incorporação, a sua API C é bastante intuitiva e de fácil aprendizagem quando comparada com a fornecida por Python.

Considerando os objetivos deste estudo e os testes realizados, Python é a linguagem indicada para o desenvolvimento de AM multiplataforma. Para além disso, a linguagem Python e o seu interpretador estão disponíveis para as mais diversas plataformas (Linux, FreeBSD, Solaris, MacOS X, Windows), podendo o interpretador ser gerado, com recurso a um compilador C a partir do seu código-fonte, para um sistema operativo não suportado. A biblioteca padrão de Python inclui módulos para processamento de texto e expressões regulares, protocolos de rede, acesso a serviços do sistema operativo, criptografia, *interface* gráfica, etc., existindo uma grande variedade de extensões adicionais para qualquer tipo de aplicação que se pretenda

desenvolver. É possível usar Python como linguagem embutida em aplicações de maior porte, ou em aplicações nativas para ambientes móveis (*Android* e *iOS*), devido à sua facilidade de integração com C.

A opção de desenvolver aplicações nativas fazendo uso das linguagens de programação candidatas expostas neste documento, apresenta algumas limitações a este estudo, para o qual uma análise de ferramentas de abstração multiplataforma será decerto um tópico de trabalho futuro. Também os apectos de segurança e constrangimentos específicos, em particular dos sistemas operativos móveis, motivam a uma obrigatória atenção. Finalmente, o próximo passo a seguir deverá ser a implementação de um micro-ambiente de AM para averiguar/provar a escolha da linguagem feita neste estudo.

Esta página foi intencionalmente deixada em branco

Referências

[1] S. Franklin and A. Graesser, “Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents,” … *agents III agent Theor. Archit. …*, 1997.

[2] P. M. Reddy, “Mobile agents: Intelligent Assistants on the Internet,” *Resonance*, vol. 7, no. 7, pp. 35–43, Jul. 2002.

[3] O. Etzioni and D. Weld, “Intelligent agents on the internet: Fact, fiction, and forecast,” *IEEE Expert*, pp. 1–14, 1995.

[4] M. Wooldridge and N. Jennings, “Intelligent agents: Theory and practice,” *Knowl. Eng. …*, 1995.

[5] D. Lange, “Mobile objects and mobile agents: the future of distributed computing?,” *ECOOP’98—Object-Oriented Program.*, 1998.

[6] B. Tarr, D. Nebesh, and S. Foster, “Introduction to mobile agent systems and applications,” *Dep. Def. Present. …*, pp. 1–76, 2000.

[7] D. Chess, B. Grosof, and C. Harrison, “Itinerant agents for mobile computing,” *Pers. …*, vol. 20010, 1995.

[8] J. Waldo, “The Jini architecture for network-centric computing,” *Commun. ACM*, vol. 42, no. 7, 1999.

[9] A. Carzaniga, G. Picco, and G. Vigna, “Designing distributed applications with mobile code paradigms,” … *19th Int. Conf. …*, 1997.

[10] A. Fuggetta, G. Picco, and G. Vigna, “Understanding code mobility,” *Softw. Eng. IEEE …*, vol. 354768, 1998.

[11] G. Pietro Picco, “Mobile agents: an introduction,” *Microprocess. Microsyst.*, vol. 25, no. 2, pp. 65–74, Apr. 2001.

[12] W. Vieira and L. Camarinha-Matos, “Agentes móveis na operação remota,” *Jornadas da Eng. …*, no. 1, 2000.

[13] D. Chess, C. Harrison, and A. Kershenbaum, “Mobile agents: Are they a good idea?,” *Mob. Object Syst. Towar. …*, vol. 19887, no. 88465, 1997.

[14] D. Milojicic, “Mobile agent applications,” *IEEE Concurr.*, pp. 80–89, 1999.

[15] D. Lange and M. Oshima, “Seven good reasons for mobile agents,” *Commun. ACM*, vol. 42, no. 3, pp. 88–89, 1999.

[16] D. Kotz and R. Gray, “Mobile Agents and the Future of the Internet,” *Oper. Syst. Rev.*, vol. 33, no. August, pp. 7–13, 1999.

[17] D. Wong, N. Paciorek, and D. Moore, “Mobile Agents,” vol. 42, no. 3, 1999.

[18] G. Samaras, “Mobile agents: What about them? Did they deliver what they promised? Are they here to stay?,” *Environments*, 2004.

[19] G. Vigna, “Mobile agents: ten reasons for failure,” *IEEE Int. Conf. Mob. Data Manag. 2004. Proceedings. 2004*, pp. 298–299, 2004.

[20] OMG, “Object Management Group, Inc.” [Online]. Available: http://www.omg.org/. [Accessed: 14-Nov-2013].

[21] J. Odell and M. Nodine, “The Foundation for Intelligent Physical Agents,” *http://www. fipa. org*, 2012. [Online]. Available: http://www.fipa.org/. [Accessed: 14-Nov-2013].

[22] C. Georgousopoulos and O. F. Rana, “An approach to conforming a MAS into a FIPA-compliant system,” *Proc. first Int. Jt. Conf. Auton. agents multiagent Syst. part 2 - AAMAS ’02*, p. 968, 2002.

[23] G. FOKUS and I. B. M. Corporation, “Mobile Agent System Interoperability Facilities Specification,” 1997. [Online]. Available: http://www.omg.org/cgi-bin/doc?orbos/97-10-05. [Accessed: 13-Nov-2013].

[24] FIPA, “FIPA Communicative Act Library Specification”. Report 00037.,” 2002. [Online]. Available: http://www.fipa.org/specs/fipa00037/.

[25] FIPA, “FIPA Specifications Grouped by Category,” 2005. [Online]. Available: http://fipa.org/repository/bysubject.html. [Accessed: 15-Nov-2013].

[26] J. Baumann, “A comparison of mechanisms for locating mobile agents,” no. August, 1999.

[27] P. Domel, “Mobile Telescript agents and the Web,” *Compcon’96.'Technologies Inf. …*, 1996.

[28] Darmouth College, “D’Agents Project,” 2003. [Online]. Available: http://agent.cs.dartmouth.edu/. [Accessed: 11-Nov-2013].

[29] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer, “Mole–Concepts of a mobile agent system,” *world wide web*, no. August, 1998.

[30] D. Lange, M. Oshima, G. Karjoth, and K. Kosaka, “Aglets: Programming mobile agents in Java,” *Worldw. Comput. Its …*, 1997.

[31] H. Peine and T. Stolpmann, “The architecture of the Ara platform for mobile agents,” *Mob. Agents*, no. 1219, 1997.

[32] a. R. Tripathi and N. M. Karnik, “Mobile agent programming in Ajanta,” *Distrib. …*, pp. 190–197, 1999.

[33] B. G. Glass and C. T. O. Objectspace, “Overview of Voyager : ObjectSpace ’ s Product Family for State-of-the-Art Distributed Computing,” 1999.

[34] C. Baumer, M. Breugst, S. Choy, and T. Magedanz, “Grasshopper—A universal agent platform based on OMG MASIF and FIPA standards,” … *Work. Mob. Agents* …, pp. 99–111, 1999.

[35] D. Wong, N. Paciorek, T. Walsh, and J. DiCelie, “Concordia: An infrastructure for collaborating mobile agents,” *Mob. Agents*, pp. 1–12, 1997.

[36] P. Bellavista, A. Corradi, and C. Stefanelli, “Protection and interoperability for mobile agents: a secure and open programming environment,” *IEICE Trans.* …, 2000.

[37] F. Bellifemine, A. Poggi, and G. Rimassa, “Developing Multi-agent Systems with JADE,” pp. 89–103, 2001.

[38] S. Poslad, P. Buckle, and R. Hadingham, “The FIPA-OS agent platform: Open source for open standards,” … *Intell. Agents Multi-Agents*, 2000.

[39] H. Nwana, D. Ndumu, L. Lee, and J. Collis, “ZEUS: a toolkit and approach for building distributed multi-agent systems,” *Proc. third* …, pp. 1–39, 1999.

[40] D. Johansen, “A tacoma retrospective,” *Softw. Pract.* …, pp. 1–23, 2002.

[41] S. Ilarri, R. Trillo, and E. Mena, “SPRINGS: A scalable platform for highly mobile agents in distributed computing environments,” … *Symp. World Wireless, Mob.* …, pp. 633–637, 2006.

[42] A. Moreno, A. Valls, and A. Viejo, *Using JADE-LEAP implement agents in mobile devices*. 2003.

[43] B. Chen, H. Cheng, and J. Palen, “Mobile C: a mobile agent platform for mobile C/C++ agents,” *Softw. Pract.* …, no. July, pp. 1711–1733, 2006.

[44] C. Muldoon and G. O’Hare, “Agent factory micro edition: A framework for ambient applications,” … *Sci. 2006*, no. Cldc, 2006.

[45] J. Tardo and L. Valente, “Mobile agent security and Telescript,” *COMPCON ’96. Technol. Inf. Superhighw. Dig. Pap.*, pp. 58–63, 1996.

[46] R. Gray, “Agent Tcl: A transportable agent system,” … *ciKM Work. Intell. Inf. Agents* …, 1995.

[47] emorphia Ltd., “FIPA-OS sourceforge.net,” 2003. [Online]. Available: http://fipa-os.sourceforge.net/index.htm.

[48] G. Caire and F. Pieri, “Leap user guide,” 2006.

[49] C.-L. Wu, C.-F. Liao, and L.-C. Fu, “Service-Oriented Smart-Home Architecture Based on OSGi and Mobile-Agent Technology,” *IEEE Trans. Syst. Man Cybern. Part C (Applications Rev.*, vol. 37, no. 2, pp. 193–205, Mar. 2007.

[50] S. Nasution and P. Hartel, “Trust Level and Routing Selection for Mobile Agents in a Smart Home,” … *Simulation, 2010* …, 2010.

[51] R. Hans, “Location Monitoring In Mobile Agents System: A Survey,” vol. 72, no. 7, pp. 14–19, 2013.

[52] T. Leppänen, J. Riekki, M. Liu, E. Harjula, and T. Ojala, “Internet of Things Based on Smart Objects,” 2014.

[53] A. Fischer and F. Grodzinsky, *The anatomy of programming languages*, no. September. 1993.

[54] K. A. Louden, K. C., & Lambert, *Programming languages: principles and practices*, Third edit. Cengage Learning, 2011.

[55] M. Rosenblum, “The reincarnation of virtual machines,” *Queue*, no. August, 2004.

[56] N. Costa, A. Pereira, and C. Serodio, “Virtual Machines Applied to WSN’s: The state-of-the-art and classification,” *Syst. Networks* …, 2007.

[57] A. O. S. Project, “Android, the world’s most popular mobile platform.” [Online]. Available: http://developer.android.com/about/index.html. [Accessed: 09-Jan-2014].

[58] Android Open Source Project, “android-scripting.” [Online]. Available: https://code.google.com/p/android-scripting/. [Accessed: 13-Jan-2014].

[59] Apple Inc., “iOS Dev Center.” [Online]. Available: https://developer.apple.com/devcenter/ios/index.action. [Accessed: 06-Jan-2014].

[60] N. Fonseca, C. Reis, C. Silva, L. Marcelino, and V. Carreira, *Desenvolvimento em iOS iPhone, iPad e iPod Touch*. FCA, 2012.

[61] Apple Inc., “iPhone Developer Program License Agreement.” pp. 1–28, 2010.

[62] Apple Inc., “iOS Developer Program License Agreement,” 2013.

[63] MoSync AB, “MoSync - Cross-platform SDK and HTML5 tools for mobile app development.” [Online]. Available: http://www.mosync.com/. [Accessed: 15-Jun-2014].

[64] Adobe Systems Inc., “PhoneGap - free and open source framework for quickly building cross-platform mobile apps.” [Online]. Available: http://phonegap.com/. [Accessed: 15-Jun-2014].

[65] Codename One, “Codename One - Reinventing Mobile Development.” [Online]. Available: http://www.codenameone.com/. [Accessed: 16-Jun-2014].

[66] Appcelerator Inc., “Appcelerator Titanium - Mobile Development Environment.” [Online]. Available: http://www.appcelerator.com/titanium/. [Accessed: 17-Jun-2014].

[67] Corona Labs Inc., “Corona SDK.” [Online]. Available: http://coronalabs.com/products/corona-sdk/. [Accessed: 16-Jun-2014].

[68] Sencha Inc., “Sencha Touch - Build Mobile Web Apps with HTML5.” [Online]. Available: http://www.sencha.com/products/touch. [Accessed: 18-Jun-2014].

[69] I. Motorola Solutions, “RhoMobile Suite.” [Online]. Available: http://rhomobile.com/. [Accessed: 18-Jun-2014].

[70] A. Lingnau, O. Drobnik, and P. Dömel, “An HTTP-based infrastructure for mobile agents,” *Fourth Int. World Wide Web …*, 1995.

[71] J. Gosling and H. McGilton, *The Java language environment*, no. May. 1995.

[72] Oracle, “Java Downloads for All Operating Systems.” [Online]. Available: https://www.java.com/en/download/manual.jsp. [Accessed: 08-Jan-2014].

[73] Oracle, “How do I get Java for Mobile device?” [Online]. Available: https://www.java.com/en/download/faq/java_mobile.xml. [Accessed: 21-Apr-2014].

[74] G. Inc., “Dalvik Technical Information.” [Online]. Available: http://source.android.com/devices/tech/dalvik/index.html. [Accessed: 08-Jan-2014].

[75] G. Van Rossum, “Python Programming Language.,” *USENIX Annual Technical Conference*, 2007. [Online]. Available: http://www.python.org/. [Accessed: 30-Nov-2013].

[76] G. Rossum, “Computer Programming for Everybody (Revised Proposal) A Scouting Expedition for the Programmers of Tomorrow,” no. August 1999, 1999.

[77] python.org, “OrganizationsUsingPython,” 2013. [Online]. Available: https://wiki.python.org/moin/OrganizationsUsingPython. [Accessed: 16-Jan-2014].

[78] python.org, “The Python Language Reference: The import system,” 2014. [Online]. Available: http://docs.python.org/3/reference/import.html. [Accessed: 17-Jan-2014].

[79] python.org, “The Python Standard Library: importlib – An implementation of import,” 2014. [Online]. Available: http://docs.python.org/3/library/importlib.html#module-importlib. [Accessed: 17-Jan-2014].

[80] A. O. S. Project, “python-for-android.” [Online]. Available: https://code.google.com/p/python-for-android/. [Accessed: 17-Jan-2014].

[81] Jonathan Hosmer, “Python for iOS - A Python IDE for the iOS,” 2012. [Online]. Available: http://www.pythonforios.com/. [Accessed: 17-Jan-2014].

[82] Omz:software, “Pythonista,” 2012. [Online]. Available: http://omz-software.com/pythonista/. [Accessed: 17-Jan-2014].

[83] Kivy.org, “Kivy - Open source Python library.” [Online]. Available: http://kivy.org/#home. [Accessed: 18-Jan-2014].

[84] L. Wall, “The Perl programming language,” 1994.

[85] “perlootut - Object-Oriented Programming in Perl Tutorial.” [Online]. Available: http://perldoc.perl.org/perlootut.html. [Accessed: 12-Jan-2014].

[86] Infinity Interactive, “Moose - A Postmodern Object System for Perl.” [Online]. Available: http://moose.iinteractive.com/en/about.html. [Accessed: 12-Jan-2014].

[87] A. O. S. Project, “android-scripting Download List.” [Online]. Available: https://code.google.com/p/android-scripting/downloads/list. [Accessed: 13-Jan-2014].

[88] R. Ierusalimschy, L. De Figueiredo, and W. Celes, “The evolution of an extension language: A history of Lua,” … *Lang. pages B–14–* …, 2001.

[89] Paul Merrell, “Where Lua Is Used.” [Online]. Available: https://sites.google.com/site/marbux/home/where-lua-is-used. [Accessed: 19-Jan-2014].

[90] R. Ierusalimschy, “The evolution of Lua,” … *third ACM SIGPLAN* …, pp. 2–1–2–26, 2007.

[91] lua-users.org, “Lua Binaries,” 2014. [Online]. Available: http://lua-users.org/wiki/LuaBinaries. [Accessed: 19-Jan-2014].

[92] “Lua for Windows.” [Online]. Available: https://code.google.com/p/luaforwindows/. [Accessed: 20-Jan-2014].

[93] T. M. Project, “MacPorts - Available Ports.” [Online]. Available: http://www.macports.org/ports.php?by=name&substr=lua. [Accessed: 20-Jan-2014].

[94] Android Open Source Project, “Android NDK.” [Online]. Available: http://developer.android.com/tools/sdk/ndk/index.html. [Accessed: 20-Jan-2014].

[95] M. Kottman, “Lua and LuaJava ported to Android,” 2012. [Online]. Available: https://github.com/mkottman/AndroLua. [Accessed: 21-Jan-2014].

[96] K. Karlsson, “kahlua - a Lua Implementation for Java,” 2013. [Online]. Available: https://github.com/krka/kahlua2. [Accessed: 21-Jan-2014].

[97] Yakut Caddesi, “Gideros Studio.” [Online]. Available: http://giderosmobile.com/. [Accessed: 29-May-2014].

[98] J. Ousterhout and K. Jones, *Tcl and the Tk toolkit*. Addison-Wesley Publishing Company, Inc., 1994.

[99] T. D. Xchange, “Uses for Tcl/Tk.” [Online]. Available: http://www.tcl.tk/about/uses.html. [Accessed: 28-Dec-2013].

[100] P. Raines and J. Tranter, *TCL/TK in a Nutshell*, First Edit. O’Reilly, 2009.

[101] J. Ousterhout, J. Levy, and B. Welch, *The safe-tcl security model*. 1998, pp. 1–18.

[102] Python Software Foundation, “pickle — Python object serialization.” [Online]. Available: http://docs.python.org/2/library/pickle.html. [Accessed: 03-Feb-2014].

[103] google project hosting, “python-for-android - Py4A.” [Online]. Available: https://code.google.com/p/python-for-android/. [Accessed: 07-May-2014].

[104] G. Van Rossum and F. L. Drake, “Extending and Embedding Python,” 2009.

[105] Apple Inc., “Technical Note TN2328: Changes To Embedding Python Using Xcode 5.0,” *IOS Developer Library*, 2013. .

[106] Python Software Foundation, “Embedding Python in Another Application.” [Online]. Available: https://docs.python.org/2/extending/embedding.html. [Accessed: 21-Jun-2014].

[107] Python Software Foundation, “os — Miscellaneous operating system interfaces.” [Online]. Available: https://docs.python.org/2/library/os.html#os-file-dir. [Accessed: 21-Jun-2014].

[108] Apple Inc., “The iOS Environment - iOS App Programming Guide,” 2013. [Online]. Available: https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/TheiOSEnvironment/TheiOSEnvironment.html. [Accessed: 29-May-2014].

[109] Python Software Foundation, “Common Object Structures.” [Online]. Available: https://docs.python.org/2/c-api/structures.html#PyObject. [Accessed: 21-Jun-2014].

[110] P. Kulchenko, “pkulchenko / serpent,” 2012. [Online]. Available: https://github.com/pkulchenko/serpent/blob/master/src/serpent.lua. [Accessed: 30-Jan-2014].

[111] D. Nehab, “LuaSocket - Network support for the Lua language,” 2007. [Online]. Available: http://w3.impa.br/~diego/software/luasocket/. [Accessed: 30-Jan-2014].

[112] Corona Labs Inc, “Corona SDK.” [Online]. Available: http://coronalabs.com/products/corona-sdk/. [Accessed: 30-May-2014].

[113] Apple Inc., “Foundation Functions Reference Contents - NSSearchPathForDirectoriesInDomains,” 2014.

[114] Apple Inc., “NSBundle Class Reference Contents,” 2013.

[115] Perl.org, “Storable - persistence for Perl data structures.” [Online]. Available: http://perldoc.perl.org/Storable.html. [Accessed: 02-Feb-2014].

[116] CPAN Search, “Storable - persistence for Perl data structures.” [Online]. Available: http://search.cpan.org/~ams/Storable-2.45/Storable.pm#64_bit_data_in_perl_5.6.0_and_5.6.1. [Accessed: 21-Apr-2014].

[117] google project hosting, “perldroid.” [Online]. Available: https://code.google.com/p/perldroid/. [Accessed: 07-May-2014].

[118] perldoc.perl.org, “perlembed - Internals and C language interface.” [Online]. Available: http://perldoc.perl.org/perlembed.html. [Accessed: 03-Jul-2014].

[119] perldoc.perl.org, “perlapi - Internals and C language interface.” [Online]. Available: http://perldoc.perl.org/perlapi.html#eval_pv. [Accessed: 05-Jul-2014].

[120] R. Garcia-Suarez, “FindBin - Locate directory of original perl script,” *CPAN Search*. [Online]. Available: http://search.cpan.org/~rgarcia/perl-5.9.5/lib/FindBin.pm. [Accessed: 05-Jul-2014].

[121] Oracle, “Package java.lang.reflect.” [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html. [Accessed: 08-Apr-2014].

[122] Python Software Foundation, “Python - Built-in Functions.” [Online]. Available: https://docs.python.org/2/library/functions.html?highlight=__import__#__import__. [Accessed: 21-Apr-2014].

[123] Python Software Foundation, “inspect — Inspect live objects.” [Online]. Available: https://docs.python.org/2.7/library/inspect.html. [Accessed: 21-Apr-2014].

[124] R. Ierusalimschy, L. Figueiredo, and W. Celes, “Lua 5.1 Reference Manual - Functions and Types,” *lua.org*. [Online]. Available: http://www.lua.org/manual/5.1/manual.html#3.7. [Accessed: 17-Apr-2014].

[125] perldoc.perl.org, “Module::Load.” [Online]. Available: http://perldoc.perl.org/Module/Load.html. [Accessed: 07-Apr-2014].

[126] A. Kennedy, “Class::Inspector,” *CPAN Search*. [Online]. Available: http://search.cpan.org/dist/Class-Inspector/lib/Class/Inspector.pm. [Accessed: 08-Apr-2014].

Anexos

Anexo A: código fonte em Java

Programa 1 – implementação da classe *MyObject*

```
 1. // programa: MyObject.java
 2. import static java.lang.System.out;
 3. import java.io.Serializable;
 4.
 5. public class MyObject implements Serializable {
 6.   private static final long serialVersionUID = 1L;
 7.   MyObject() {  x = 0;    nome = "foo";  }
 8.   int x;
 9.   String nome;
10.   public MyObject (int startX, String startNome) {
11.                         x = startX;
12.                         nome = startNome;
13.         }
14.   public int incX (int incr) { x += incr; return x;}
15.   public int decX (int incr) { x -= incr; return x;}
16.   public int setX (int valor) { x = valor; return x;}
17.   public String setNome (String novoNome) { nome = novoNome; return nome;}
18.   public void printMyObject () {
19.                         out.println("Estes os dados do objeto");
20.                         out.println("X = " + x);
21.                         out.println("Nome = " + nome);
22.     }
23. }
```

Programa 2 – implementação do programa de teste de serialização para Windows, Linux e Mac OS X

```
 1. // Programa: MainTestMyObject.java
 2. import java.io.FileInputStream;
 3. import java.io.FileOutputStream;
 4. import java.io.IOException;
 5. import java.io.ObjectInputStream;
 6. import java.io.ObjectOutputStream;
 7. import java.util.Scanner;
 8.
 9. public class MainTestMyObject {
10.     public static void inicializar(MyObject e) {
11.         try {
12.              String localhostname = java.net.InetAddress.getLocalHost().getHostName();
13.              e.setX(1);
14.              e.setNome(localhostname);
15.              System.out.println("Foi inicializado o objeto ");
16.              e.printMyObject();
17.         }
18.         catch(IOException i)     {
19.                 i.printStackTrace(); }
20.     }
21.
22.     public static void serializar(MyObject e) {
23.         try {
24.                 FileOutputStream fileOut = new FileOutputStream("myObject.ser");
25.                 ObjectOutputStream out = new ObjectOutputStream(fileOut);
26.                 out.writeObject(e);
27.                 out.close();
28.                 fileOut.close();
29.                 System.out.println("Foi serializado o objeto ");
30.                 e.printMyObject();
31.         }
32.         catch(IOException i)   {
33.                 i.printStackTrace();  }
34.     }
35.
36.     public static MyObject desseralizar() {
37.         MyObject e = null;
38.         try {
39.                 FileInputStream fileIn = new FileInputStream("myObject.ser");
40.                 ObjectInputStream in = new ObjectInputStream(fileIn);
41.                 e = (MyObject) in.readObject();
42.                 in.close();
43.                 fileIn.close();
44.                 System.out.println("Foi desserializado o objeto ");
45.         }
46.         catch(IOException i)     {
47.                 i.printStackTrace();
48.                 return null;
49.             }
50.         catch(ClassNotFoundException c)  {
51.                 System.out.println("Classe MyObject nao encontrada!");
52.                 c.printStackTrace();
53.                 return null;
54.             }
55.         return e;
56.     }
57.     private static Scanner input = new Scanner(System.in);
58.     public void display() {
59.         System.out.println("-- Tarefas --");
60.         System.out.println(
```

```
61.          "Escolha uma opcao: \n" +
62.          "  1) Inicializar\n" +
63.          "  2) Serializar\n" +
64.          "  3) Desserializar \n" +
65.          "  4) Incrementar \n" +
66.          "  5) Decrementar \n" +
67.          "  6) Apresentar \n" +
68.          "  0) Sair\n ");
69.      }
70.
71.      public static void main(String[] args) {
72.          MyObject workObj = new MyObject();
73.          workObj.printMyObject();
74.          MainTestMyObject menu = new MainTestMyObject();
75.          while (true) {
76.                  menu.display();
77.                  int op = input.nextInt();
78.                  input.nextLine();
79.                  switch (op) {
80.                          case 1:
81.                                  inicializar(workObj);
82.                                  break;
83.                          case 2:
84.                                  serializar(workObj);
85.                                  break;
86.                          case 3:
87.                                  workObj =desseralizar();
88.                                  break;
89.                          case 4:
90.                                  workObj.incX(1);
91.                                  System.out.println("Foi incrementado o valor de X ");
92.                                  break;
93.                          case 5:
94.                                  workObj.decX(1);
95.                                  System.out.println("Foi decrementado o valor de X ");
96.                                  break;
97.                          case 6:
98.                                  workObj.printMyObject();
99.                                  break;
100.                         case 0:
101.                                 System.out.println("Terminado...");
102.                                 System.exit(1);
103.                                 break;
104.                         default:
105.                                 System.out.println("Opcao Invalida.");
106.                                 break;
107.                         }
108.             }
109.     }
110. }
```

Programa 3 – implementação do programa de teste de carregamento dinâmico para Windows, Linux e Mac OS X

```
1.  // LoadTestMyObject.java
2.  import java.lang.reflect.InvocationTargetException;
3.  import java.lang.reflect.Method;
4.
5.  public class LoadTestMyObject {
6.           public static void main(String[] args) {
```

```
7.                  try {
8.                          ClassLoader myClassLoader = ClassLoader.getSystemClassLoader();
                            String classNameToBeLoaded = "testingJava.samples.MyObject";
9.                          Class<?> myClass = myClassLoader.loadClass(classNameToBeLoaded);
10.                         Object myInstance = myClass.newInstance();
11.                         Method[] methods = myClass.getDeclaredMethods();
12.                         System.out.println("Os Metodos para: " + myClass.getName());
13.
14.                 for (Method m: methods) {
15.                        // imprimir tipo do metodo
16.                        System.out.print("  " + m.getReturnType().getSimpleName());
17.
18.                        // imprimir  nome do metodo
19.                        System.out.print("  " + m.getName() + "( ");
20.
21.                        // imprimir parametros do metodo
22.                        Class<?>[] classes = m.getParameterTypes();
23.                        for (int i=0; i<classes.length; i++) {
24.                                if (i < (classes.length - 1) ) {
25.                                   System.out.print(classes[i].getSimpleName() + ", ");
26.                                } else {
27.                                   System.out.print(classes[i].getSimpleName() + " ");
28.                        }
29.                     }
30.                         System.out.println(")");
31.                  }
32.                 String nomeParameter = "localhost";
33.                 int xParameter = 100;
34.
35.                 Method myMethod1 = myClass.getMethod("setNome", new Class[] { String.class });
36.
37.                 String returnValue1 = (String) myMethod1.invoke(myInstance,
     new Object[] { nomeParameter });
38.
39.                 System.out.println("Valor devolvido pelo metodo " + myMethod1 + ":"
     + returnValue1);
40.
41.                 Method myMethod2 = myClass.getMethod("setX", new Class[] { int.class });
42.
43.                 int returnValue2 = (int) myMethod2.invoke(myInstance,
    new Object[] { xParameter });
44.
45.                 System.out.println("O valor devolvido pelo metodo " + myMethod2 + ":" +
    returnValue2);
46.
47.                 System.out.println("O valor de X vai ser decrementado.");
48.                 Method myMethod3 = myClass.getMethod("decX", new Class[] {int.class});
49.                 myMethod3.invoke(myInstance, new Object[] { 1 });
50.
51.                 Method myMethod0 = myClass.getMethod("printMyObject", new Class[] {} );
52.                 System.out.println("O valor devolvido pelo metodo " + myMethod0 + ":");
53.                 myMethod0.invoke(myInstance, new Object[] {  });
54.                 }
55.                 catch (InvocationTargetException e) {
56.                    e.printStackTrace();
57.                 } catch (Exception e) {
58.                 System.out.println("Exception: " + e.toString());
59.             }
60.         }
61. }
```

Anexo B: código fonte em Python

Programa 4 – implementação da classe *MyObject*

```
1.  # Programa: MyObject.py
2.  class MyObject:
3.      def __init__(self):
4.          self.x = 0
5.          self.nome = "foo"
6.
7.      def incX (self, valor):
8.          self.x = self.x + valor
9.          return self.x
10.
11.     def decX (self, valor):
12.         self.x = self.x - valor
13.         return self.x
14.
15.     def setX(self, valor):
16.         self.x = valor
17.         return self.x
18.
19.     def setNome(self, texto):
20.         self.nome = texto
21.         return self.nome
22.
23.     def printMyObject(self,x,nome):
24.         print("Estes os dados de " + self.__class__.__name__)
25.         print("X = " + str(self.x))
26.         print("Nome: " + self.nome)
27.
28. def getClass():
29.     return MyObject
```

Programa 5 – implementação do programa de teste de serialização para Windows, Linux, Mac OS X e Android

```
1.  # Programa: MainTestMyObject.py
2.  from MyObject import MyObject
3.  import socket
4.  import pickle
5.
6.  def inicializa(self):
7.      self.x = novo.setX(1)
8.      self.nome = socket.gethostname()
9.      print("Foi inicializado o objeto ")
10.
11. def serializa(self):
12.     print("A serializar o objeto " + str(self.x) + " / " + self.nome)
13.     f = open('myObject.ser', 'w')
14.     pickle.dump(self, f)
15.     f.close
16.
17. def desserializa():
18.     print("A desserializar o objeto ")
19.     try:
20.         f= open('myObject.ser', 'r')
```

```
21.         unpickledlist = pickle.load(f)
22.         return unpickledlist
23.     except:
24.         print("O arquivo myObject.ser nao foi encontrado")
25.
26. def menu():
27.     print("-- Tarefas --")
28.     print("Escolha uma opcao:")
29.     print("  1) Inicializar")
30.     print("  2) Serializar")
31.     print("  3) Desserializar")
32.     print("  4) Incrementar")
33.     print("  5) Decrementar")
34.     print("  6) Apresentar")
35.     print("  0) Sair")
36.     return input ("")
37.
38. if __name__ == '__main__':
39.     workObj = MyObject()
40.     workObj.printMyObject(workObj.x, workObj.nome)
41.
42.     loop = 1
43.     op = 0
44.     while loop == 1:
45.         op = menu()
46.         if op == 1:
47.             inicializa(workObj)
48.         elif op == 2:
49.             serializa(workObj)
50.         elif op == 3:
51.             workObj = desserializa()
52.         elif op == 4:
53.             workObj.x = workObj.incX(1)
54.         elif op == 5:
55.             workObj.x = novo.decX(1)
56.         elif op == 6:
57.             workObj.printMyObject(workObj.x, workObj.nome)
58.         elif op == 0:
59.             loop = 0
60.
61.     print ("Terminado")
```

Programa 6 – implementação do programa *main.m* em projeto para iOS

```
1.  //
2.  //  main.m
3.  //  MyTestPython
4.  //
5.
6.  #import <UIKit/UIKit.h>
7.
8.  #import "IPLAppDelegate.h"
9.  #include "Python.h"
10.
11. int main(int argc, char *argv[])
12. {
13.     Py_SetProgramName(argv[0]);
14.     Py_Initialize();
15.     PySys_SetArgv(argc, argv);
16.     PyRun_SimpleString("import os, sys \n"
17.                        "#print sys.argv, \"\\n\".join(sys.path)\n"
18.                        "#print os.getcwd()\n"
```

```
                        "import MyObject\n");

    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([IPLAppDelegate class]));
    }
}
```

Programa 7 – implementação do programa *IPLViewController.m* em projeto para iOS

```
//
//  IPLViewController.m
//  MyTestPython
//

#import "IPLViewController.h"
#include "Python.h"

@interface IPLViewController ()

@end

@implementation IPLViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSString * appFolderPath = [[NSBundle mainBundle] resourcePath];
    Py_SetPythonHome((char *)[appFolderPath UTF8String]);

    NSString *devHostName = [[UIDevice currentDevice] name];
    NSArray *paths = NSSearchPathForDirectoriesInDomains (NSDocumentDirectory, NSUserDomainMask,
YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];

    const char *cDocumentsDirectory = [documentsDirectory
cStringUsingEncoding:NSUTF8StringEncoding];

    const char *cDevHostName = [devHostName cStringUsingEncoding:NSUTF8StringEncoding];

//    NSLog(@"cDocumentsDirectory is %s\n", cDocumentsDirectory);
//    NSLog(@"cAppFolderPath is %s\n", cAppFolderPath);
//    NSLog(@"cDevHostName is %s\n", cDevHostName);


    PyObject *main = PyImport_AddModule("__main__");
    PyObject *globals = PyModule_GetDict(main);
    PyObject *value = PyString_FromString(cDocumentsDirectory);
    PyDict_SetItemString(globals, "mypath", value);
    PyObject *value2 = PyString_FromString(cDevHostName);
    PyDict_SetItemString(globals, "appHostName", value2);


    const char * prog = [
                         [[NSBundle mainBundle] pathForResource:@"MainTestMyObject"
ofType:@"py"] cStringUsingEncoding: NSUTF8StringEncoding];

    FILE* fd = fopen(prog, "r");
    PyRun_SimpleFile(fd, prog);

    Py_Finalize();

```

```
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

Programa 8 – implementação do programa de serialização para iOS

```
from MyObject import MyObject
import socket
import pickle

mypath = mypath.replace('\'','')
appHostName = appHostName.replace('\'','')

def inicializa(self):
    self.x = novo.setX(1)
    self.nome = socket.gethostname()
    self.nome = appHostName
    print("Foi inicializado o objeto ")

def serializa(self):
    print("A serializar o objeto " + str(self.x) + " / " + self.nome)
    f = open(mypath+'/myObject.ser', 'w')
    pickle.dump(self, f)
    f.close

def desserializa():
    print("A desserializar o objeto ")
    try:
        f= open(mypath+'/myObject.ser', 'r')
        unpickledlist = pickle.load(f)
        '''unpickledlist.printMyObject(unpickledlist.x, unpickledlist.nome)'''
        return unpickledlist
    except:
        print("O arquivo myObject.ser nao foi encontrado")

if __name__ == '__main__':
    novo = MyObject()
    novo.printMyObject(novo.x, novo.nome)

    loop = 1
    op = 1
    while loop == 1:

        if op == 1:
            inicializa(novo)
            novo.printMyObject(novo.x, novo.nome)
            op=4
        elif op == 2:
            serializa(novo)
            op=5
        elif op == 3:
            novo = desserializa()
            op=6
        elif op == 4:
            novo.x = novo.incX(1)
            print("Foi incrementado o valor de X")
```

```
51.             novo.printMyObject(novo.x, novo.nome)
52.             op=2
53.         elif op == 5:
54.             novo.x = novo.decX(1)
55.             print("Foi decrementado o valor de X")
56.             novo.printMyObject(novo.x, novo.nome)
57.             op=3
58.         elif op == 6:
59.             novo.printMyObject(novo.x, novo.nome)
60.             op=0
61.         elif op == 0:
62.             loop = 0
63.
64.     print ("Terminado")
```

Programa 9 – implementação do programa de teste de carregamento dinâmico

```
1.  // LoadTestMyObject.py
2.  mod_name = "MyObject"
3.  mod_obj = __import__(mod_name, globals(), locals(), [''])
4.  print ">> %s" % mod_obj
5.
6.  class_obj = mod_obj.getClass()
7.  myClass = class_obj()
8.
9.  import inspect
10.
11. print ("Os Metodos para %s" % mod_name + ":")
12. for name, value in inspect.getmembers(myClass, callable):
13.     print " ", name
14.
15. myClass.__init__()
16. myClass.setX(100)
17. print ("O valor devolvido pelo metodo setX: %d " % myClass.x)
18. myClass.setNome('localhost')
19. print ("O valor devolvido pelo metodo setNome: " + myClass.nome)
20. myClass.decX(1)
21. print ("O valor de X vai ser decrementado.")
22. myClass.printMyObject(myClass.x, myClass.nome)
```

Anexo C: código fonte em Lua

Programa 10 – implementação da classe *MyObject*

```
1.  -- Programa: MyObject.lua
2.  MyObject = {
3.    x = 0,
4.    nome = "foo"  }
5.
6.  function MyObject.new (self, o)
7.    o = o or {}
8.    setmetatable (o, self)
9.    self.__index = self
10.   return o
11. end
12.
13. function MyObject:incX (valor)
14.   self.x = self.x + valor
15.   return self.x
16. end
17.
18. function MyObject:decX (valor)
19.   return self.x - valor
20. end
21.
22. function MyObject:setX (valor)
23.   self.x = valor
24.   return self.x
25. end
26.
27. function MyObject:setNome (texto)
28.   self.nome = texto
29.   return self.nome
30. end
31.
32. function MyObject:printMyObject(x, nome)
33.    print("Estes os dados do objeto ")
34.    print("X = ", self.x)
35.    print("Nome: ", self.nome)
36. end
```

Programa 11 – implementação do programa de teste de serialização para Windows, Linux e Mac OS X

```
1.  -- Programa: MainTestMyObject.lua
2.  require "socket"
3.  require "MyObject"
4.
5.  function inicializa(self)
6.     self.x = self:setX(1)
7.     self.nome = socket.dns.gethostname()
8.     print("Foi inicializado o objeto.")
9.     self:printMyObject(self.x, self.nome)
10. end
11.
12. function serializa(self)
```

```
13.       local serpent = require("serpent")
14.       local ser = serpent.dump(novo)
15.       f = io.open("MyObject.ser","w")
16.       f:write(ser)
17.       f:close()
18.       print("Foi serializado o objeto ")
19.  end
20.
21.  function desserializa()
22.       f = io.open("MyObject.ser","r")
23.       for VALOR in f:lines() do
24.           str = VALOR
25.       end
26.       f:close()
27.       local fun2 = loadstring(str)
28.       local _b = fun2()
29.       workObj.x = _b.x
30.       workObj.nome = _b.nome
31.       print("Foi desserializado o objeto ")
32.  end
33.
34.  local op
35.  function menu()
36.      print("-- Tarefas --")
37.      print("Escolha uma opcao:")
38.      print("  1) Inicializar")
39.      print("  2) Serializar")
40.      print("  3) Desserializar")
41.      print("  4) Incrementar")
42.      print("  5) Decrementar")
43.      print("  6) Apresentar")
44.      print("  0) Sair")
45.      io.flush()
46.      op=io.read()
47.      return tonumber(op)
48.  end
49.
50.  local function main()
51.     workObj = MyObject.new(MyObject)
52.     while op ~= 0 do
53.       op=menu()
54.       if op==1 then
55.            inicializa(workObj)
56.       elseif op==2 then
57.               serializa(workObj)
58.           elseif op==3 then
59.                  desserializa()
60.              elseif op==4 then
61.                     workObj.x = workObj:incX(1)
62.                     print("Foi incrementado o valor de X ")
63.                 elseif op==5 then
64.                        workObj.x = novo:decX(1)
65.                        print("Foi decrementado o valor de X ")
66.                    elseif op==6 then
67.                           workObj:printMyObject(workObj.x, workObj.nome)
68.       end
69.     end
70.  end
71.
72.  main()
```

Programa 12 – implementação do programa de teste de serialização para Android

```
-- Programa: TestSL4AMyObject.lua
require "android"
package.path = package.path .. ";../?.lua"
require "socket"
require "MyObject"

function inicializa(self)
   self.x = self:setX(1)
   self.nome = socket.dns.gethostname()
   print("Foi inicializado o objeto.")
   self:printMyObject(self.x, self.nome)
end

function serializa(self)
      local serpent = require("serpent")
      local ser = serpent.dump(novo)
      f = io.open("MyObject.ser","w")
      f:write(ser)
      f:close()
      print("Foi serializado o objeto ")
end

function desserializa()
     f = io.open("MyObject.ser","r")
     for VALOR in f:lines() do
         str = VALOR
     end
     f:close()
     local fun2 = loadstring(str)
     local _b = fun2()
     workObj.x = _b.x
     workObj.nome = _b.nome
     print("Foi desserializado o objeto ")
end

local op
function menu()
    print("-- Tarefas --")
    print("Escolha uma opcao:")
    print("  1) Inicializar")
    print("  2) Serializar")
    print("  3) Desserializar")
    print("  4) Incrementar")
    print("  5) Decrementar")
    print("  6) Apresentar")
    print("  0) Sair")
    io.flush()
    op=io.read()
    return tonumber(op)
end

local function main()
   novo = MyObject.new(MyObject)

   while op ~= 0 do
     op=menu()
     if op==1 then
          inicializa(novo)
     elseif op==2 then
             serializa(novo)
         elseif op==3 then
                desserializa()
             elseif op==4 then
```

```
64.                     novo.x = novo:incX(1)
65.                     print("Foi incrementado o valor de X ")
66.                 elseif op==5 then
67.                        novo.x = novo:decX(1)
68.                        print("Foi decrementado o valor de X ")
69.                    elseif op==6 then
70.                           novo:printMyObject(novo.x, novo.nome)
71.      end
72.    end
73. end
74.
75. if pcall(main) then
76.         print("ok")
77. else
78.         print("erro")
79. end
```

Programa 13 – implementação do programa *IPLViewController.m* em projeto Xcode

```
1.  //
2.  //  IPLViewController.m
3.  //  MyTestLua
4.  //
5.
6.  #import "IPLViewController.h"
7.
8.  @interface IPLViewController ()
9.
10. @end
11.
12. @implementation IPLViewController
13.
14. int setLuaPath( NSString* path, lua_State *L )
15. {
16.     lua_getglobal( L, "package" );
17.     lua_getfield( L, -1, "path" );
18.     NSString * cur_path = [NSString stringWithUTF8String:lua_tostring( L, -1 )];
19.     cur_path = [cur_path stringByAppendingString:@";"];
20.     cur_path = [cur_path stringByAppendingString:path];
21.     cur_path = [cur_path stringByAppendingString:@"/?.lua"];
22.     lua_pop( L, 1 );
23.     lua_pushstring( L, [cur_path UTF8String]);
24.     lua_setfield( L, -2, "path" );
25.     lua_pop( L, 1 );
26.     return 0;
27. }
28.
29. - (void)viewDidLoad
30. {
31.     [super viewDidLoad];
32.
33.     NSString *appFolderPath = [[NSBundle mainBundle] resourcePath];
34.     NSString *devHostName = [[UIDevice currentDevice] name];
35.
36.     L = luaL_newstate();
37.     luaL_openlibs(L);
38.     lua_settop(L, 0);
39.
40.     NSArray *paths = NSSearchPathForDirectoriesInDomains (NSDocumentDirectory, NSUserDomainMask,
    YES);
41.     NSString *documentsDirectory = [paths objectAtIndex:0];
42.
```

```
43.     lua_createtable(L, 0, 2);
44.     lua_pushstring(L, [documentsDirectory UTF8String]);
45.     lua_setfield( L, -2, "docPath" );
46.
47.     lua_pushstring(L, [appFolderPath UTF8String]);
48.     lua_setfield( L, -2, "appPath" );
49.
50.     lua_pushstring(L, [devHostName UTF8String]);
51.     lua_setfield( L, -2, "appHostName" );
52.
53.     lua_setglobal(L, "AppFolders");
54.
55.     luaL_loadstring(L, "print(AppFolders.docPath)");
56.     if (0 != lua_pcall(L,0,0,0)) {
57.         puts(lua_tostring(L, -1));
58.      }
59.
60.     int err;
61.
62.     NSString *luaFilePath = [[NSBundle mainBundle] pathForResource:@"MainTestMyObject"
    ofType:@"lua"];
63.     setLuaPath([luaFilePath stringByDeletingLastPathComponent], L);
64.
65.     err = luaL_loadfile(L, [luaFilePath cStringUsingEncoding:[NSString
    defaultCStringEncoding]]);
66.
67.     if (0 != err) {
68.         luaL_error(L, "cannot compile lua file: %s",lua_tostring(L, -1));
69.         return;
70.     }
71.
72.     err = lua_pcall(L, 0, 0, 0);
73.     if (0 != err) {
74.         luaL_error(L, "cannot run lua file: %s", lua_tostring(L, -1));
75.         return;
76.     }
77.
78. lua_close(L);
79. }
80.
81. - (void)didReceiveMemoryWarning
82. {
83.     [super didReceiveMemoryWarning];
84.     // Dispose of any resources that can be recreated.
85. }
86.
87. @end
```

Programa 14 – implementação do programa de teste de serialização para iOS

```
1.  -- Programa: MainTestMyObject.lua
2.  require "MyObject"
3.  local mypath = AppFolders.docPath
4.
5.  function inicializa(self)
6.     self.x = self:setX(1)
7.     self.nome = AppFolders.appHostName
8.     print("Foi inicializado o objeto.")
9.     self:printMyObject(self.x, self.nome)
10. end
11.
```

```
12. function serializa(self)
13.       local serpent = require("serpent")
14.       local ser = serpent.dump(novo)
15.       local f, err = io.open(mypath .. "/MyObject.ser","w")
16.       if f then
17.           f:write(ser)
18.           f:close()
19.           print("Foi serializado o objeto ")
20.       else
21.           print ("erro:" .. err)
22.       end
23. end
24.
25. function desserializa()
26.      local f, err = io.open(mypath .. "/MyObject.ser","r")
27.      if f then
28.         for VALOR in f:lines() do
29.             str = VALOR
30.         end
31.         f:close()
32.         local fun2 = loadstring(str)
33.         local _b = fun2()
34.         novo.x = _b.x
35.         novo.nome = _b.nome
36.         print("Foi desserializado o objeto ")
37.      else
38.         print ("erro:" .. err)
39.      end
40. end
41.
42. local op
43.
44. local function main()
45.
46.    op=1
47.    novo = MyObject.new(MyObject)
48.    while op ~= 0 do
49.
50.      if op==1 then
51.           inicializa(novo)
52.           op=4
53.      elseif op==2 then
54.              serializa(novo)
55.              op=5
56.          elseif op==3 then
57.                 desserializa()
58.                 op=6
59.              elseif op==4 then
60.                     novo.x = novo:incX(1)
61.                     print("Foi incrementado o valor de X ")
62.                     op=2
63.                 elseif op==5 then
64.                        novo.x = novo:decX(1)
65.                        print("Foi decrementado o valor de X ")
66.                        op=3
67.                    elseif op==6 then
68.                           novo:printMyObject(novo.x, novo.nome)
69.                           op=0
70.      end
71.    end
72. end
73.
74. main()
```

Programa 15 – implementação do programa de teste de carregamento dinâmico para Windows, Linux e Mac OS X

```
 1. -- Programa: LoadTestMyObject.lua
 2. local f, s = loadfile ("MyObject.lua")
 3. if not f then
 4.    print ("Erro ao compilar 'MyObject.lua'\n", s)
 5.    return
 6. end
 7.
 8. f()
 9.
10. myfileobj = dofile ("MyObject.lua")
11. local o = MyObject:new()
12. print ("Os metodos para ", o, ":")
13. for key,value in pairs(getmetatable(o)) do
14.     t = type(value)
15.     if t=='function' then
16.        print("".. key)
17.     end
18. end
19.
20. print ("")
21. obj = MyObject:printMyObject()
22. local xParameter = 100
23. local nomeParameter = "localhost"
24. print ("")
25. o.nome = MyObject:setNome(nomeParameter)
26. print ("O valor devolvido pelo metodo setNome: " .. o.nome)
27. o.x = MyObject:setX(xParameter)
28. print ("O valor devolvido pelo metodo setX: " .. o.x)
29. print ("O valor de X vai ser decrementado.")
30. o.x = MyObject:decX(1)
31. print("Os valor devolvidos pelo metodo printMyObject:")
32. MyObject:printMyObject()
```

Programa 16 – implementação do programa de teste de carregamento dinâmico para Android

```
 1. -- Programa: LoadTestSL4AMyObject.lua
 2. require "android"
 3. package.path = package.path .. ";../?.lua"
 4. local f, s = loadfile ("Scripts/MyObject.lua")
 5. if not f then
 6.    print ("Erro ao compilar 'MyObject.lua'\n", s)
 7.    return
 8. end
 9.
10. f()
11.
12. local function main()
13.
14.     local o = MyObject:new()
15.     print ("Os metodos para ", o, ":")
16.     for key,value in pairs(getmetatable(o)) do
17.         t = type(value)
18.         if t=='function' then
19.            print("".. key)
20.         end
21.     end
22.     print ("")
23.
```

```
24.     obj = MyObject:printMyObject()
25.     local xParameter = 100
26.     local nomeParameter = "localhost"
27.
28.     print ("")
29.     o.nome = MyObject:setNome(nomeParameter)
30.     print ("O valor devolvido pelo metodo setNome: " .. o.nome)
31.     o.x = MyObject:setX(xParameter)
32.     print ("O valor devolvido pelo metodo setX: " .. o.x)
33.     print ("O valor de X vai ser decrementado.")
34.     o.x = MyObject:decX(1)
35.     print("Os valor devolvidos pelo metodo printMyObject:")
36.     MyObject:printMyObject()
37.
38. end
39.
40. if pcall(main) then
41.         print("ok")
42. else
53.         print("erro")
44. end
```

Programa 17 – implementação do programa de teste de carregamento dinâmico para iOS

```
1.  -- Programa: LoadTestMyObject.lua
2.  local mypath = AppFolders.appPath
3.  local f, s = loadfile (mypath .. "/MyObject.lua")
4.  if not f then
5.     print ("Erro ao compilar 'MyObject.lua'\n", s)
6.     return
7.  end
8.
9.  f()
10.
11. local function main()
12.
13.     local o = MyObject:new()
14.
15.     print ("Os metodos para ", o, ":")
16.     for key,value in pairs(getmetatable(o)) do
17.         t = type(value)
18.         if t=='function' then
19.            print(".. key)
20.         end
21.     end
22.     print ("")
23.
24.     obj = MyObject:printMyObject()
25.     local xParameter = 100
26.     local nomeParameter = "localhost"
27.
28.     print ("")
29.     o.nome = MyObject:setNome(nomeParameter)
30.     print ("O valor devolvido pelo metodo setNome: " .. o.nome)
31.     o.x = MyObject:setX(xParameter)
32.     print ("O valor devolvido pelo metodo setX: " .. o.x)
33.     print ("O valor de X vai ser decrementado.")
34.     o.x = MyObject:decX(1)
35.     print("Os valor devolvidos pelo metodo printMyObject:")
36.     MyObject:printMyObject()
37.
39. end
```

Anexo D: código fonte em Perl

Programa 16 – implementação da classe *MyObject*

```
 1. # Programa: MyObject.pm
 2. #!/usr/bin/perl
 3. package MyObject;
 4.
 5. sub new {
 6.     my $class = shift;
 7.     my $self = {
 8.         x => 0,
 9.         nome  => "foo",
10.     };
11.     print "foi criado $class\n";
12.     print "x $self->{x}\n";
13.     print "nome $self->{nome}\n";
14.     bless $self, $class;
15.     return $self;
16. }
17.
18. sub setNome {
19.     my ( $self, $_nome ) = @_;
20.     #my $self = shift;
21.         #my $nome = shift;
22.     $self->{nome} = $_nome if defined($_nome);
23.     return $self->{nome};
24. }
25.
26. sub printMyObject {
27.          my ( $self, $nome ) = @_;
28.          print "X $self->{x}\n";
29.          print "Nome $self->{nome}\n";
30. }
31.
32. sub incX {
33.          my $self = shift;
34.          my $valor = shift;
35.          $self->{x} += $valor;
36.          return $self->{x};
37. }
38.
39. sub decX {
40.          my $self = shift;
41.          my $valor = shift;
42.          $self->{x} -= $valor;
43.          return $self->{x};
44. }
45.
46. sub setX {
47.          my $self = shift;
48.          my $valor = shift;
49.          $self->{x} = $valor;
50.          return $self->{x};
51. }
52. 1;
```

Programa 17 – implementação do programa de teste de serialização para Windows, Linux e Mac OS X

```
 1. # Programa: MainTestMyObject.pl
 2. #!/usr/bin/perl
 3. use MyObject;
 4. use Sys::Hostname;
 5. use Storable;
 6.
 7. sub inicializa{
 8.         my ( $self ) = @_;
 9.         $self->{x} = $self->setX(1);
10.         $self->{nome} = $self->setNome(hostname);
11.         print ("Foi inicializado o objeto.\n");
12.         $self->printMyObject();
13. }
14.
15. sub serializa {
16.         my $result = eval { store( $workObj, 'MyObject.ser' ) || die "nao foi possivel escrever
    o ficheiro\n"; };
17.     print ("Foi serializado o objeto.\n");
18. }
19.
20. sub desserializa {
21.         my $result = eval { retrieve( 'MyObject.ser' ) };
22.         if( defined $@ and length $@ )
23.         { warn "Erro de Storable: $@" }
24.     elsif( not defined $result )
25.         { warn "I/O erro de Storable: $!" }
26.     elsif( not eval { $result->isa( 'MyObject' ) } )
27.         { warn "Nao foi obtido o MyObject\n" }
28.     else {
29.          print ("Foi desserializado o objeto.\n");
30.          $result->printMyObject();
31.     }
32. }
33.
34. sub menu {
35.           print "-- Tarefas --\n".
36.           "Escolha uma opcao:\n".
37.           "  1) Inicializar\n".
38.           "  2) Serializar\n".
39.           "  3) Desserializar\n".
40.           "  4) Incrementar\n".
41.           "  5) Decrementar\n".
42.           "  6) Apresentar\n".
43.           "  0) Sair\n";
44.     $input = <STDIN>;
45.     chomp($input);
46.     return ($input);
47. }
48.
49. my $input = '';
50. $workObj = new MyObject();
51. while ($input ne '0') {
52.         $input = menu();
53.
54.          if($input == 1) {
55.          inicializa($workObj);
56.          } elsif($input == 2) {
57.                  serializa($workObj);
58.                  } elsif($input == 3) {
59.                          desserializa();
```

```
60.                             } elsif($input == 4) {
61.                                     $workObj ->{x} = $workObj ->incX(1);
62.                                     } elsif($input == 5) {
63.                                                 $workObj ->{x} = $workObj ->decX(1);
64.                                             } elsif($input == 6) {
65.                                                     $workObj ->printMyObject();
66.                                                     }
67. }
```

Programa 18 – implementação do programa de teste de serialização para Android

```
 1. # Programa: MainTestMyObject.pl
 2. #!/usr/bin/perl
 3. use Android;
 4. use MyObject;
 5. use Sys::Hostname;
 6. use Storable;
 7.
 8. sub inicializa{
 9.         my ( $self ) = @_;
10.         $self->{x} = $self->setX(1);
11.         $self->{nome} = $self->setNome(hostname);
12.         print ("Foi inicializado o objeto.\n");
13.         $self->printMyObject();
14. }
15.
16. sub serializa {
17.         my $result = eval { store( $workObj, 'MyObject.ser' ) || die "nao foi possivel escrever
    o ficheiro\n"; };
18.     print ("Foi serializado o objeto.\n");
19. }
20.
21. sub desserializa {
22.         my $result = eval { retrieve( 'MyObject.ser' ) };
23.         if( defined $@ and length $@ )
24.         { warn "Erro de Storable: $@" }
25.     elsif( not defined $result )
26.         { warn "I/O erro de Storable: $!" }
27.     elsif( not eval { $result->isa( 'MyObject' ) } )
28.         { warn "Nao foi obtido o MyObject\n" }
29.     else {
30.          print ("Foi desserializado o objeto.\n");
31.          $result->printMyObject();
32.     }
33. }
34.
35. sub menu {
36.           print "-- Tarefas --\n".
37.           "Escolha uma opcao:\n".
38.           "  1) Inicializar\n".
39.           "  2) Serializar\n".
40.           "  3) Desserializar\n".
41.           "  4) Incrementar\n".
42.           "  5) Decrementar\n".
43.           "  6) Apresentar\n".
44.           "  0) Sair\n";
45.     $input = <STDIN>;
46.     chomp($input);
47.     return ($input);
48. }
49.
50. my $input = '';
```

```
$workObj = new MyObject();
while ($input ne '0') {
        $input = menu();

         if($input == 1) {
         inicializa($workObj);
         } elsif($input == 2) {
                 serializa($workObj);
                 } elsif($input == 3) {
                         desserializa();
                         } elsif($input == 4) {
                                 $workObj ->{x} = $workObj ->incX(1);
                                 } elsif($input == 5) {
                                             $workObj ->{x} = $workObj ->decX(1);
                                         } elsif($input == 6) {
                                                 $workObj ->printMyObject();
                                                 }
}
```

Programa 19 – implementação do programa *IPLViewController.m* em projeto Xcode

```
//
//  IPLViewController.m
//  MyTestPerl
//

#import "IPLViewController.h"
#include "EXTERN.h"
#include "perl.h"

static PerlInterpreter *my_perl=NULL;

static void xs_init (pTHX);
EXTERN_C void boot_DynaLoader (pTHX_ CV* cv);
EXTERN_C void xs_init(pTHX)
{
    char *file = __FILE__;
    newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, file);
}

@interface IPLViewController ()

@end

@implementation IPLViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSString * appFolderPath = [[NSBundle mainBundle] resourcePath];
    NSString *devHostName = [[UIDevice currentDevice] name];
    NSArray *paths = NSSearchPathForDirectoriesInDomains (NSDocumentDirectory, NSUserDomainMask,
YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];

    const char *cDocumentsDirectory = [documentsDirectory
cStringUsingEncoding:NSUTF8StringEncoding];
    const char *cAppFolderPath = [appFolderPath cStringUsingEncoding:NSUTF8StringEncoding];
    const char *cDevHostName = [devHostName cStringUsingEncoding:NSUTF8StringEncoding];

    const char * prog = [
```

```
                            [[NSBundle mainBundle] pathForResource:@"MainTestMyObject"
ofType:@"pl"] cStringUsingEncoding: NSUTF8StringEncoding];
41.
42.     NSLog(@"Running: %s", prog);
43.
44.     //NSLog(@"$mydoc= '%s'", cDocumentsDirectory);
45.     //NSLog(@"app: %s", cAppFolderPath);
46.     //NSLog(@"host: %s", cDevHostName);
47.
48.     char *my_argv[]= { "", (char *)prog};
49.
50.     char *args[] = { NULL };
51.     PERL_SYS_INIT3(0,&my_argv,NULL);
52.     my_perl = perl_alloc();
53.     perl_construct(my_perl);
54.     perl_parse(my_perl, xs_init, 2, my_argv, NULL);
55.     PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
56.
57.     NSString *myeval = [NSString stringWithFormat:@"\"$mypath='%s';\"",cDocumentsDirectory ];
58.     NSString *mydevice = [NSString stringWithFormat:@"\"$mydevice='%s';\"",cDevHostName ];
59.
60.     eval_pv((const char *)myeval, TRUE);
61.     eval_pv((const char *)mydevice, TRUE);
62.
63.     call_argv("main", G_DISCARD | G_NOARGS, args);
64.
65.     perl_destruct(my_perl);
66.     perl_free(my_perl);
67.     PERL_SYS_TERM();
68. }
69.
70. - (void)didReceiveMemoryWarning
71. {
72.     [super didReceiveMemoryWarning];
73.     // Dispose of any resources that can be recreated.
74. }
75.
76. @end
```

Programa 20 – implementação do programa de teste de serialização para iOS

```
 1. # Programa: MainTestMyObject.pl
 2. #!/usr/bin/perl
 3.
 4. sub prepara {
 5.     use FindBin;
 6.     use lib $FindBin::Bin;
 7.     use MyObject;
 8.     use Storable;
 9. }
10.
11. use Storable;
12.
13. sub inicializa{
14.     my ( $self ) = @_;
15.     $self->{x} = $self->setX(1);
16.     $self->{nome} = $mydevice;
17.     print ("Foi inicializado o objeto.\n");
18.     $self->printMyObject();
19. }
20.
```

```
21. sub serializa {
22.     my $result = eval { store( $novo, $mypath.'/MyObject.ser' ) || die "nao foi possivel
    escrever o ficheiro\n"; };
23.     print ("Foi serializado o objeto.\n");
24. }
25.
26. sub desserializa {
27.     my $result = eval { retrieve( $mypath.'/MyObject.ser' ) };
28.     if( defined $@ and length $@ )
29.         { warn "Erro de Storable: $@" }
30.     elsif( not defined $result )
31.         { warn "I/O erro de Storable: $!" }
32.     elsif( not eval { $result->isa( 'MyObject' ) } )
33.         { warn "Nao foi obtido o MyObject\n" }
34.     else {
35.          print ("Foi desserializado o objeto.\n");
36.          $novo = $result;
37.          $novo->printMyObject();
38.     }
39. }
40.
41. sub main {
42.
43.     prepara();
44.
45. my $input = '';
46. $novo = new MyObject();
47.
48.  $input=1;
49.
50. while ($input ne '0') {
51.        if($input == 1) {
52.          inicializa($novo);
53.           $input=4;
54.          } elsif($input == 2) {
55.                  serializa($novo);
56.                  $input=5;
57.                  } elsif($input == 3) {
58.                          desserializa();
59.                          $input=6;
60.                          } elsif($input == 4) {
61.                                  $novo->{x} = $novo->incX(1);
62.                                  $novo->printMyObject();
63.                                  $input=3;
64.                                  } elsif($input == 5) {
65.                                          $novo->{x} = $novo->decX(1);
66.                                          $input=3;
67.                                          } elsif($input == 6) {
68.                                                 $novo->printMyObject();
69.                                                 $input=0;
70.                                                  }
71.     }
72. }
```

Programa 21 – implementação do programa de teste de carregamento dinâmico para Windows, Linux e Mac OS X

```
 1.  # Programa: LoadTestMyObject.pl
 2.  #!/usr/bin/perl
 3.  use Module::Load;
 4.  use warnings;
 5.
 6.  my $class = 'MyObject';
 7.  load $class;
 8.
 9.  use Class::Inspector;
10.  my $methods = Class::Inspector->methods( 'MyObject');
11.  print "Os metodos para $class: [@$methods]\n";
12.
13.  my $MyClass = $class->new;
14.  $MyClass->setX(100);
15.  print "O valor devolvido pelo metodo @$methods[5]: $ MyClass->{x}\n";
16.  $MyClass->setNome('localhost');
17.  print "O valor devolvido pelo metodo @$methods[4]: $MyClass->{nome}\n";
18.  print "O valor de X vai ser decrementado.\n";
19.  $MyClass->decX(1);
20.  print "Os valor devolvidos pelo metodo @$methods[3]:\n";
21.  $MyClass->printMyObject();
```

Programa 22 – implementação do programa de teste de carregamento dinâmico para Android

```
 1.  # Programa: LoadTestSL4AMyObject.pl
 2.  #!/usr/bin/perl
 3.  use Module::Load;
 4.  use warnings;
 5.
 6.  my $class = 'MyObject';
 7.  load $class;
 8.
 9.  use Class::Inspector;
10.  my $methods = Class::Inspector->methods( 'MyObject');
11.  print "Os metodos para $class: [@$methods]\n";
12.
13.  my $MyClass = $class->new;
14.  $MyClass->setX(100);
15.  print "O valor devolvido pelo metodo @$methods[5]: $ MyClass->{x}\n";
16.  $MyClass->setNome('localhost');
17.  print "O valor devolvido pelo metodo @$methods[4]: $MyClass->{nome}\n";
18.  print "O valor de X vai ser decrementado.\n";
19.  $MyClass->decX(1);
20.  print "Os valor devolvidos pelo metodo @$methods[3]:\n";
21.  $MyClass->printMyObject();
```

Programa 23 – implementação do programa de teste de carregamento dinâmico para iOS

```
 1. # Programa: LoadTestMyObject.pl
 2. #!/usr/bin/perl
 3. sub main {
 4.     use FindBin;
 5.     use lib $FindBin::Bin;
 6.     use Module::Load;
 7.     use warnings;
 8.
 9.     my $class = 'MyObject';
10.     load $class;
11.
12.     use Class::Inspector;
13.     my $methods = Class::Inspector->methods( 'MyObject');
14.     print "Os metodos para $class: [@$methods]\n";
15.
16.     my $MyClass = $class->new;
17.     $MyClass->setX(100);
18.     print "O valor devolvido pelo metodo @$methods[5]: $ MyClass->{x}\n";
19.     $MyClass->setNome('iPhone Simulator');
20.     print "O valor devolvido pelo metodo @$methods[4]: $MyClass->{nome}\n";
21.     print "O valor de X vai ser decrementado.\n";
22.     $MyClass->decX(1);
23.     print "Os valor devolvidos pelo metodo @$methods[3]:\n";
24.     $MyClass->printMyObject();
25. }
```