Pontifícia Universidade Católica
do Rio de Janeiro

Fernando de Abreu e Lima Alves

Estendendo o Luaproc: Suporte para aplicações em ambientes móveis

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC-Rio.

Orientador: Profª. Noemi de La Rocque Rodriguez

Rio de Janeiro
Julho de 2018

Pontifícia Universidade Católica
do Rio de Janeiro

Fernando de Abreu e Lima Alves

Estendendo o Luaproc: Suporte para aplicações em ambientes móveis

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Profª. Noemi de La Rocque Rodriguez
Orientador
Departamento de Informática – PUC-Rio

Profª. Ana Lucia de Moura
Departamento de Informática – PUC-Rio

Prof. Markus Endler
Departamento de Informática – PUC-Rio

Prof. Marcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 20 de Julho de 2018

Fernando de Abreu e Lima Alves

Graduou-se em Engenharia de Computação na Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Participou do desenvolvimento do sistema OCTOPUS junto ao Laboratório de Inteligência Computacional Aplicada da PUC–Rio, hoje em funcionamento na PETROBRAS. Trabalha como pesquisador e desenvolvedor na área de métodos de apoio a decisão.

Ficha Catalográfica

Alves, Fernando de Abreu e Lima

Estendendo o Luaproc: Suporte para aplicações em ambientes móveis / Fernando de Abreu e Lima Alves; orientador: Noemi de La Rocque Rodriguez. – Rio de janeiro: PUC-Rio, Departamento de Informática , 2018.

v., 90 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática .

Inclui bibliografia

1. Informática – Teses. 2. Luaproc;. 3. Concorrência;. 4. Android;. 5. Broker;. 6. Filas de Mensagens;. I. Rodriguez, Noemi de La Rocque. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática . III. Título.

CDD: 004

Agradecimentos

Gostaria de agradecer principalmente a minha orientadora Prof. Noemi Rodriguez pela atenção, carinho e apoio ao decorrer do Mestrado.

Também gostaria de agradecer meus amigos, colegas e familiares pelo apoio, compreensão e incentivo durante este trabalho.

Por fim, gostaria de agradecer o CNPq e a Pontifícia Universidade Católica do Rio de Janeiro pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Resumo

Alves, Fernando de Abreu e Lima; Rodriguez, Noemi de La Rocque. **Estendendo o Luaproc: Suporte para aplicações em ambientes móveis**. Rio de Janeiro, 2018. 90p. Dissertação de Mestrado – Departamento de Informática , Pontifícia Universidade Católica do Rio de Janeiro.

Cada vez mais os aparelhos móveis estão se aperfeiçoando, com aumentos em suas capacidades de processamento e memória. Essa tendência acaba tornando o processamento móvel uma alternativa interessante. Este trabalho visa explorar esse mundo mobile e o seu potencial através do paralelismo, tanto localmente, na forma de exploração multicore, quanto distribuida, na forma de exploração multidispositivo. Exploramos isto através de uma biblioteca de paralelismo da linguagem de programação Lua, chamada Luaproc. Propomos um novo modelo de comunicação para esta biblioteca, para incluir esse cenário multidispositivo e combinar as facilidades de um serviço de enfileiramento de mensagens com o suporte para paralelismo já existente. Apresentamos algumas aplicações da biblioteca desenvolvida, avaliando sua utilização e desempenho em diferentes cenários.

Palavras-chave

Luaproc; Concorrência; Android; Broker; Filas de Mensagens;

Abstract

Alves, Fernando de Abreu e Lima; Rodriguez, Noemi de La Rocque (Advisor). **Extending Luaproc: Support for applications in mobile environments**. Rio de Janeiro, 2018. 90p. Dissertação de mestrado – Departamento de Informática , Pontifícia Universidade Católica do Rio de Janeiro.

Mobile devices are undergoing constant increases in their processing and memory capabilities. This tendency is making mobile processing an interesting alternative. This work aims to support the programmer in exploring this potential by using parallelism, both local, in the form of multicore exploitation, as well as distributed, in the form of multidevice exploration. We explored this through a parallel library for the Lua programming language, called Luaproc. We propose an extension to this library and its communication model, to include this multidevice scenario and combine the facilities of a message queuing service with the existing facilities for multicore programming. We then present some applications to show different use cases with distribution and their performance.

Keywords

Luaproc; Concurrency; Android; Broker; Message Queueing;

Sumário

Lista de figuras

1
Introdução

Uma tendência crescente em computação é a das plataformas mobile. Com avanços frequentes, essas tem se tornado cada vez mais interessantes como uma forma alternativa de processamento. Além da disseminação e do poder de processamento, uma das coisas que também torna o mobile atraente é a versatilidade e potencial na criação de programas/aplicativos, assim podendo realizar das mais diversas tarefas, desde lazer até processamento de dados sensoriais/científicos.

Com a crescente oferta de computadores multicore, vem crescendo também o interesse por bibliotecas que explorem o paralelismo. A biblioteca Luaproc(1) permite a exploração de paralelismo em Lua, oferecendo um modelo *m x n*, onde o programador pode trabalhar com diferentes números de threads de aplicação e threads de sistema. Isto torna fácil a paralelização conceitual de programas sem necessariamente ter que pagar o custo de criar e gerenciar várias threads de sistema operacional. No entanto, essa biblioteca não oferece suporte a aplicações distribuídas, que é algo desejável atualmente com o avanço do uso de modelos/aplicações distribuídas e portanto a comunicação entre os aparelhos envolvidos. As áreas de jogos, sensoriamento e chat são alguns exemplos de aplicações beneficiadas por essa forma de comunicação.

A proposta deste trabalho é explorar o uso da biblioteca Luaproc no ambiente móvel, estendendo a biblioteca para permitir a comunicação entre diferentes dispositivos. Como parte do trabalho, estudamos diferentes alternativas para essas extensões tanto em termos de interface como implementação.

1.1
Objetivo do trabalho

Nosso objetivo é oferecer uma plataforma que combine facilidades para explorar o paralelismo com o suporte para programação de aplicações distribuídas. O modelo atual de comunicação do Luaproc é o de troca de mensagens, em que o sistema não permite compartilhamento de memória e oferece canais de comunicação. Vamos estender este modelo de canais para múltiplos dispositivos.

Investigamos uma arquitetura e interface apropriadas tendo em mente o

modelo atual de comunicação oferecido pelo Luaproc, para criar uma interface semelhante para comunicação entre dispositivos.

1.2 Por que Lua?

Lua tem a reputação de ser robusta e ter um bom desempenho apesar de ser uma linguagem interpretada. Em comparação com outras linguagens scripts, como Python, Lua utiliza menos memória e tem um interpretador mais rápido(2). Alguns casos de destaque são softwares famosos como Adobe's Photoshop Lightroom, o middleware Ginga[1], e jogos como World of Warcraft[2], Angry Birds, Dark Souls, entre outros.

Lua também permite rodar os seus scripts dinamicamente. Isto é útil para a manutenção de seu código, já que mudanças em Lua não necessitam da recompilação de seu código. Além de facilitar a escalabilidade de uma aplicação, pois é suportado uma carga dinâmica de scripts para execução podendo ser ajustado conforme a necessidade. Também é possível montar uma string, representando um código válido Lua, em tempo de execução e executá-lo.

Lua é pequena (menos de 1MB) e requer apenas um compilador C padrão, tornando-a portátil e embutível. Isso é um incentivo para utilizar Lua no Android pois é simples de compilá-la através do CMake[3] e não ocupa muito espaço. Um exemplo de software que tira proveito disso é a Engine de jogos 2D Corona[4], baseado em Lua e focado no desenvolvimento de aplicações mobile.

Lua facilita a implementação do mecanismo de segurança *Sandboxing*. Esta técnica permite limitar o ambiente para apenas utilizar as funções que você permitir, gerando assim um ambiente protegido. Isto é útil para garantir a segurança de seu aplicativo, protegendo-o de códigos maliciosos.

1.3 Comunicação e Luaproc

Neste trabalho apresentaremos um modelo de comunicação para múltiplos dispositivos em nossa extensão Luaproc. Diversas classes de aplicações podem se beneficiar desse novo modelo de comunicação conjugada com o paralelismo Luaproc. Alguns exemplos seriam:

– Aplicações científicas com processamento distribuídas.

[1] `http://www.ginga.org.br/`
[2] `http://wowwiki.wikia.com/wiki/Lua`
[3] `https://cmake.org/`
[4] `https://coronalabs.com/`

– Jogos com comunicação entre jogadores (ou que simplesmente precisem realizar um *broadcast* de informações para sua base de jogadores).

– Aplicações que envolvem compartilhamento de dados.

– Aplicações de coleta, organização e análise de dados via sensoriamento distribuído.

Nosso novo modelo leva em consideração as peculiaridades da comunicação mobile. No ambiente de dispositivos móveis, onde há muitas desconexões, falhas e latência, o modelo síncrono original do Luaproc não é apropriado. Devido à essas incertezas optamos por seguir um modelo assíncrono de comunicação.

Para realizar a comunicação também integramos ao nosso modelo um serviço de gerênciamento de mensagens via *broker*, que permite o desacoplamento entre os diferentes dispositivos móveis.

1.4 Organização do trabalho

O capítulo 2 apresenta um resumo sobre a biblioteca Luaproc, além de uma análise de alguns protocolos de troca de mensagens para realizar nossa comunicação multidispositivo. O capítulo 3 descreve sucintamente alguns trabalhos nas plataformas móveis, em que focamos no paralelismo e/ou comunicação apresentados pelos mesmos. O capítulo 4 descreve o processo de transporte da biblioteca Luaproc para o Android, além de nossa solução para executar scripts Lua (com dependências de módulos externos, como o próprio Luaproc). O capítulo 5 apresenta nosso modelo de comunicação multidispositivo proposto para a extensão da biblioteca Luaproc, detalhando suas características e arquitetura. O capítulo 6 apresenta nossos resultados de avaliação da biblioteca Luaproc estendida em testes comparativos com outras bibliotecas de comunicação Android, além da implementação de alguns aplicativos. Finalmente, o capítulo 7 apresenta nossas conclusões e sugestões para trabalhos futuros.

2
Conceitos básicos

Neste capítulo, apresentamos inicialmente a biblioteca Luaproc. A seguir discutimos diferentes protocolos de comunicação por filas de mensagens, relevantes para a escolha do protocolo de comunicação para nossa extensão da biblioteca Luaproc.

2.1
Luaproc

Luaproc é uma biblioteca de extensão para programação concorrente, desenvolvida em C por Alexandre Skyrme(2). Ela oferece ao usuário uma forma de programação paralela em Lua por meio de seus processos Lua, ou LP's (Lua Processes). Esses processos Lua funcionam como threads de aplicação que podem ser rodados por uma certa quantidade de threads de sistema operacional (ou kernel threads), definidas pelo usuário. Assim, o ganho paralelo ocorre ao definir a quantidade ideal de threads de sistema, ou *workers*, para a máquina utilizada. Por exemplo, em um Octa-core o ideal tipicamente seriam 8 ou 16 *workers* (já que um número maior resultaria em uma competição por tempo de CPU).

Os processos Lua, não compartilham memória. Isto é intencional pois este modelo de paralelismo evita diversas condições de corrida (data races) e a necessidade de sincronização entre os LP's, assim facilitando o entendimento do programa e minimizando as possibilidades de bugs. A biblioteca oferece troca de mensagens síncrona entre os LP's através de canais de comunicação, em que um LP pode enviar uma cópia de seus dados ao outro. Dessa forma, os tipos de valores transmitidos pelas mensagens são restringidos a números, booleanos, strings e nil.

Apesar da linguagem Lua não oferecer suporte integrado à programação concorrente, ela inclui o conceito de co-rotina, que dá apoio à concorrência não preemptiva(1). As primitivas `resume` e `yield` são usadas para ativar uma co-rotina e para abrir mão do controle, retornando à co-rotina ativadora.

Cada LP é uma co-rotina, composta por código Lua, e está encapsulada em um Lua State, que por sua vez é uma instância do interpretador Lua, ou seja, é uma estrutura que guarda o estado do interpretador Lua. Dessa

forma, cada Lua State tem seu próprio conjunto de variáveis globais, e consequentemente cada LP também.

Cada *worker* é uma thread da biblioteca POSIX, também conhecida como pthread. A biblioteca Luaproc mantém uma fila FIFO de processos Lua, em que cada um é selecionado e executado por um *worker* repetidamente. Porém os LP's não são preemptivos, o que significa que eles só param a execução através de um comando explícito, `yield`, ou se a execução for concluída. Na troca de mensagens entre LP's, pode ocorrer um `yield` implícito caso esteja faltando um LP para completar essa troca, liberando o *worker* para executar outros LP's.

O paralelismo em Luaproc está diretamente relacionado com a quantidade de *workers*. Em particular, se houver apenas um *worker*, o programa executará em apenas um processador. Se um LP fizer uma chamada bloqueante (como por exemplo uma operação de E/S), o *worker* também ficará bloqueado.

2.2 Comunicação por Filas de Mensagens

No mundo da computação móvel, os dispositivos estão sujeitos a falhas e quedas de conexão. A comunicação nesse ambiente torna necessária a realização de tarefas complementares, como o tratamento a falhas e o gerenciamento da entrega de mensagens. Um serviço de enfileiramento de mensagens nos permite delegar boa parte dessas tarefas, facilitando o desacoplamento entre dispositivos. Sendo assim, pesquisamos alguns protocolos de troca de mensagens para utilizar esse tipo de serviço.

Consideramos os protocolos AMQP[5], ZMQ (ou ZeroMQ)[6] e MQTT[7]. Todos utilizam enfileiramento de mensagens para armazenar mensagens não consumidas, mas variam com relação à confiabilidade da transmissão e forma de conexão. Em geral, o gerenciamento das mensagens é feito por um nó central, conhecido como *Broker*. Assim, a conexão entre os diversos nós é indireta e intermediada pelo *broker*. Pesquisamos sobre tolerância a falhas oferecidas pelas implementações dos *brokers* nesses protocolos (por exemplo, RabbitMQ[8] e ActiveMQ[9]), como desconexões de clientes e falhas/quedas de servidor. No caso do ZeroMQ, porém, a conexão é direta e deve-se conhecer o endereço de destino. Como a utilização de um *broker* é essencial para o desacoplamento espacial que desejamos e para a delegação de responsabilidades (como garantias

[5] `https://www.amqp.org/`
[6] `http://zeromq.org/`
[7] `http://mqtt.org`
[8] `http://www.rabbitmq.com/`
[9] `http://activemq.apache.org/`

de entrega), optamos por desconsiderar o protocolo ZeroMQ como candidato na extensão Luaproc.

Também estudamos o envio de mensagens e as devidas garantias oferecidas por cada protocolo. Os protocolos AMQP e MQTT oferecem diferentes configurações de confiabilidade de envio, dentre as quais 3 se destacam:

- Envio sem confirmação: Mais eficiente, porém sem nenhuma garantia de envio.
- Envio com confirmação: Garante que a mensagem é entregue pelo menos uma vez, podendo haver duplicatas
- Envio exato: Menos eficiente, mas garante que todas mensagem chegue sem duplicatas.

O MQTT se diferencia dos protocolos anteriores por ter sido criado tendo em mente redes não confiáveis (alta chance de queda) ou de alta latência(3). Já o AMQP se diferencia por oferecer alta configurabilidade e flexibilidade para troca de mensagens(3). Com base nisso, uma aplicação com alta chance de queda de conexão pode se beneficiar da escolha do protocolo MQTT. Já para o caso de uma aplicação com conexão confiável e necessidade de uma transmissão de mensagem diferenciada, talvez a melhor escolha seja utilizar o AMQP. Porém um estudo comparativo entre AMQP e MQTT(4), que simulou ambientes instáveis, concluiu que ambos são eficientes nesse cenário de instabilidade. Contudo, no mesmo estudo o AMQP se mostrou melhor com relação à segurança enquanto MQTT em relação à eficiência (consumo energético). Outra diferença foi a inversão da ordem das mensagens acumuladas pelo AMQP, devido ao gerenciamento LIFO, enquanto MQTT sempre garante a entrega em ordem, o que pode ser relevante em alguns casos. Assim, a escolha do protocolo dependerá de diferentes critérios e da aplicabilidade visada.

2.2.1 Implementações dos protocolos para Android

Nesta subseção apresentamos algumas bibliotecas que implementam os protocolos vistos na Seção 2.2. Foram realizados testes comparativos entre elas e serão apresentados na Subseção 2.2.2.

- Paho MQTT

  Os responsáveis pelo projeto Eclipse Paho[10] oferecem uma versão do cliente MQTT para Android, que utilizamos em nossos testes (Subseção 2.2.2). Também há diferentes implementações de clientes MQTT

[10] `https://www.eclipse.org/paho/`

para sua biblioteca. Em particular, nos interessamos pela implementação em C como candidata para a extensão da biblioteca Luaproc.

– RabbitMQ para Android

Os responsáveis pelo RabbitMQ oferecem um cliente Java de sua biblioteca, podendo ser integrado com o Android através do Gradle (um sistema de automação de compilação open source).

Apesar de haver uma implementação em C do cliente RabbitMQ[11], acabamos desconsiderando-a em favor da implementação da Paho pois esta última oferecia uma API melhor e também por nosso interesse pelo protocolo MQTT.

– ZeroMQ para Android

Os responsáveis pela biblioteca ZeroMQ também oferecem um cliente Java, chamado JeroMQ[12], podendo ser integrado com o android através do Gradle. Há uma implementação em C da biblioteca, porém como mencionado na Seção 2.2 desconsideramos pela ausência de um *broker* nativo.

2.2.2 Teste entre bibliotecas

Foi realizado um teste comparativo entre as diversas bibliotecas de troca de mensagem oferecidas para o Android, no caso: Paho MQTT Java, Paho MQTT C, RabbitMQ para Android e JeroMQ (ZeroMQ Java, compatível com Android).

O teste se resume em realizar um *ping-pong* de mensagens entre dois celulares cuja entrega é intermediada por um servidor (*broker*). Dessa forma os celulares aguardam receber uma mensagem para então responder. O tempos medidos, mostrados na Figura 2.1, são relativos ao celular que inicia com um envio.

O teste foi realizado em um Samsung SM-J700M, versão Android 6.0.1 API 23, 1.5BG de RAM e processador ARM Cortex-A53 (Octa-Core 64bit) em conjunto com um emulador Nexus 5 do Android Studio executados em uma máquina com 16GB RAM e processador Intel i7-4770 (Octa-Core 64bit).

[11] `https://github.com/alanxz/rabbitmq-c`
[12] `https://github.com/zeromq/jeromq`

	1000 Msgs			
	RabbitMQ	ZeroMQ	Paho Java	Paho C
Média (segundos)	5,65	18,40	40,34	6,96

	10000 Msgs			
	RabbitMQ	ZeroMQ	Paho Java	Paho C
Média (segundos)	70,91	173,18	430,17	70,46

Figura 2.1: Tempo de execução do *ping-pong* de mensagens.

O desempenho ruim do ZeroMQ pode ser atribuido ao *broker*, já que foi necessário implementá-lo e esta implementação foi simplória (sendo apenas um servidor *single thread* que recebe/envia mensagens entre duas portas).

Já o caso do cliente Paho Java aparenta ser um problema da portabilidade da biblioteca para o Android, pois esta implementação utiliza estruturas diferentes na versão Android com relação à versão original para Java, ou do serviço MQTT da Paho para o Android. Na página online do cliente Android da Paho[13] é mencionado que estão tentando gerar a biblioteca no formato AAR (*Android Archive Library*) para Android, unindo todas as dependências em um único arquivo. Isto pode sugerir que a versão atual, em formato JAR e dividido em duas dependências, pode ser a causa do desempenho ruim.

As demais bibliotecas se mostraram com desempenho bem semelhante.

[13] `https://www.eclipse.org/paho/clients/android/`

3
Trabalhos relacionados

Neste capítulo apresentamos alguns trabalhos que se propuseram a tirar proveito da plataforma mobile, explorando diferentes formas de paralelismo.

3.1
Corona Labs

Corona Labs oferece um framework multiplataforma de desenvolvimento mobile, chamado Corona SDK, integrado à linguagem Lua acima de uma camada C++/OpenGL para aplicações gráficas (como jogos, em geral). Luaproc também está disponível como um plugin[14] podendo ser integrado de forma semelhante a um módulo em Lua (através do comando `require`). A popularidade do Corona demonstra o sucesso da integração Lua e mobile e que há demanda pelo uso de Lua nos celulares.

Para a comunicação entre dispositivos, pode-se utilizar a biblioteca LuaSocket[15], que fornece suporte para as camadas de transporte TCP e UDP. Este SDK também disponibiliza troca de mensagens através de *push notifications*, que são mensagens exibidas na interface do celular no estilo *pop-up* (como por exemplo, mensagens de chat recebidas recentemente). Pode-se realizar isto de forma nativa (com a função `showPopup()`[16]) ou através de plugins (como OneSignal[17]).

Em geral, os *push notifications* atendem às necessidades dos jogos (que são a maioria dos casos) porém não permitem ao programador processar a mensagem. Caso o programador queira realizar outra forma de comunicação (como por exemplo, um chat entre 2 clientes) ele deverá implementá-lo. Isto pode ser feito com o auxilio de plugins também, como o *framework multiplayer* da Photon[18].

Assim, através do Corona há a possibilidade que tirar proveito do paralelismo multicore dos celulares, além de implementar a troca de mensagens entre aparelhos. Porém isto é feito de maneira não integrada, sendo necessário

[14] `https://marketplace.coronalabs.com/corona-plugins/luaproc`
[15] `https://github.com/diegonehab/luasocket`
[16] `http://docs.coronalabs.com/api/library/native/showPopup.html`
[17] `https://marketplace.coronalabs.com/corona-plugins/onesignal`
[18] `https://marketplace.coronalabs.com/corona-plugins/photon-cloud`

incluir plugins ou implementar a própria comunicação. Isto pode ser uma inconveniência ao usuário, pois ele deve usar de seu tempo para buscar o plugin ideal para suas necessidades. Caso não haja tal plugin, o usuário terá então que implementar ele mesmo a comunicação ou paralelismo.

3.2 ZOOMM Engine

A ZOOMM(5) é uma engine de browser para aparelhos móveis multicore. Ela foi criada com a motivação de que maior parte do uso dos celulares é gasto nos browsers. A ZOOMM propõe uma nova arquitetura de browser paralela a fim de diminuir o tempo e consumo energético do carregamento dos browsers.

A Engine atinge isso ao explorar a concorrência multicore para esconder a latência da rede, através de *prefetching* de recursos, e melhorar o desempenho. Em seus testes, feitos em um aparelho HTC Jetstream com processador Snapdragon Dual-Core, o tempo de carregamento dos browsers (das páginas CNN, BBC, Yahoo, Guardian, NYT, Facebook, Engadget e QQ) chegou a ser reduzido em torno da metade (aceleração de aproximadamente 2x).

Isto demonstra como a exploração da paralelização nos aparelhos móveis pode resultar em um bom retorno de desempenho, que é algo que queremos possibilitar da forma mais completa (tanto local quanto distribuída) com nossa extensão.

3.3 MARE SDK

MARE (Multicore Asyncronous Runtime Environment)(6) é um SDK de programação paralela, voltado para desenvolvedores Android. Foi criado com a motivação de que a programação paralela é difícil, com a necessidade de gerenciamento de diversas threads e da refatoração de algoritmos para serem paralelizáveis. Outra motivação foi o padrão atual de celulares multicore, justificando o uso desses *cores* adicionais na forma de paralelismo. MARE se propõe como a biblioteca para resolver essas dificuldades, facilitando a programação multicore, especialmente para a programação mobile.

No caso do Luaproc, oferece-se um modelo de programação paralela em Lua por meio de seus processos Lua. Já MARE oferece um modelo orientado a tarefas,em que o usuário apenas dispara as tarefas que precisem ser feitas em paralelo sem se preocupar com sincronismo. Supostamente ela é capaz de melhorar o desempenho de forma quase linear além de economizar diversas linhas de código devido à sua API simples[19].

[19] `https://www.qualcomm.com/news/onq/2014/02/25/qualcomm-mare-making-multicore-programming-easie`

Os desenvolvedores realizaram um teste[20] com a engine de física Bullet[21] em conjunto com a engine gráfica OGRE[22] (Object-oriented Graphics Rendering Engine) com o objetivo de paralelizar os *hot-spots* das porções seriais de Bullet, utilizando MARE. Os resultados demonstraram uma saída de aproximadamente 2x mais FPS (Frames Per Second).

Os resultados demonstram a relevância da programação paralela mobile e as vantagens de sua utilização/exploração.

3.4 COMPSs-Mobile Framework

A COMPSs-Mobile(7) é um framework para aplicações MCC (Mobile Cloud Computing, computação móvel em nuvem) paralelas. Parte da motivação de sua criação está na grande disseminação de dispositivos móveis, como *smartphones* e *tablets*, gerando um interesse em explorar esse potencial computacional na forma de computação móvel distribuída.

Os autores propõem unir a computação móvel em nuvem com detecção e paralelização automática de trechos de código, oferecendo ao desenvolvedor um modelo sequencial de programação. Isto ocorre em tempo de compilação, em que a aplicação é modificada para inserir um conjunto de invocações que, em tempo de execução, gerencia seu particionamento e sua implementação na infraestrutura subjacente. Assim, o programador pode escrever suas aplicações de forma sequencial sem a preocupação com detalhes de infraestrutura e paralelismo. No entanto, os autores reconhecem as dificuldades da computação móvel, como instabilidade causada pela latência e queda de conexão, que podem comprometer o desempenho. Eles tratam disso ao dar prioridade às tarefas (ou *tasks*) dos dispositivos móveis cujos dados de entrada já foram coletados (já que as outras tarefas não podem iniciar sem seus dados de entrada).

Para avaliar o framework, os autores executaram a aplicação *HeatSweeper*. O aplicativo *HeatSweeper* é um *workflow* de várias soluções, cujo objetivo é encontrar o posicionamento ideal de 1 a N fontes de calor na superfície de um corpo sólido para reduzir o tempo de aquecimento. Para tal, o aplicativo executa um algoritmo de busca intensivo procurando a melhor combinação de localizações para as fontes de calor.

[20]Infelizmente não é mencionado explicitamente o aparelho em que os testes foram executados. Porém, na apresentação é comparado um celular *single-core* de 2010 com um celular de processador Snapdragon Quad-Core de 2013 (ano da apresentação). Assim, talvez o teste foi realizado nesse último celular.

[21]`http://bulletphysics.org/wordpress/`

[22]`https://www.ogre3d.org/`

Os resultados dos seus testes mostraram a diminuição do tempo de execução e consumo energético, demonstrando a relevância da computação móvel distribuída, que é algo que queremos possibilitar com nossa extensão Luaproc.

4
Transporte da biblioteca Luaproc para o ambiente Android

Neste capítulo, descrevemos o processo de migração da biblioteca Luaproc para a plataforma de desenvolvimento Android Studio[23].

4.1
Lua no Android

No primeiro momento, buscamos simplesmente utilizar Lua no Android para rodar scripts autosuficientes, ou seja, executar scripts Lua que não necessitem de módulos externos ou outros scripts para a sua execução.

As tentativas iniciais de utilizar Lua em Android envolveram a utilização de ferramentas que possibilitam a manipulação das estruturas de Lua em Java, como Luaj[24] e JnLua[25]. Porém acabamos abandonando essa opção e optando pela simplicidade de utilizar Lua diretamente através do CMake do Android Studio, uma ferramenta que possibilita a compilação de código C e C++ em bibliotecas nativas. Assim, fomos capazes de utilizar Lua como uma biblioteca nativa e fazer uso de sua API C, graças ao NDK (Native Development Kit) do Android Studio, que possibilita o uso de código C e C++ em aplicativos Android. Desta forma, é possível rodar os scripts de Lua primeiramente carregando o código (através da função `luaL_loadbuffer` da API) e em seguida executando esse código (através da função `lua_pcall` da API).

4.1.1
Android Assets

O diretório *Assets* permite armazenar diversos arquivos (que serão compilados para a aplicação). Através do *AssetManager* podemos navegar nesse diretório da mesma forma que em um sistema de arquivos normal, usando URIs (Uniform Resource Identifier), e ler arquivos como streams de bytes. Assim, utilizamos esta pasta como um repositório para os scripts Lua e binários/módulos Lua que serão utilizados.

[23] `https://developer.android.com/studio/`
[24] `https://sourceforge.net/projects/luaj/`
[25] `https://github.com/danke-sra/jnlua-android`

4.1.2 Memória Interna e o Lua *package.loaders*

Devido ao acesso dos nossos arquivos Lua e binários serem feitos pelo *AssetManager*, não existe um caminho fixo (o qual podemos referenciar) para dentro da pasta *Assets*. Isto se torna um problema quando um script tiver alguma dependência, por exemplo um script que utiliza outro script Lua (através do `require`). Assim, é necessário copiar os mesmos para a memória interna do Android, a partir da qual teremos acesso a esses caminhos.

Para simplificar e facilitar a busca dos arquivos e binários Lua, convencionamos que todos estes devem estar dentro de uma pasta chamada "files". Dessa forma, conseguimos separar os arquivos relevantes para Lua de possíveis outros desnecessários (como, por exemplo, imagens), além de definir uma pasta raiz dos arquivos Lua. Uma vez definido esse diretório raiz, podemos então complementar o *package loader* de Lua de forma que este seja capaz de encontrar e carregar as bibliotecas (através do `require` de Lua). Isto é necessário, pois por padrão Lua procura em diretórios fixos (como por exemplo em Linux no diretório "/usr/local/").

4.2 Wrapper Lua

A fim de encapsular a execução dos scripts Lua e a cópia dos arquivos Lua para memória interna, criamos um Wrapper em Java responsável por chamar o código nativo que realizará estas funções. Esta cópia é feita pelo Wrapper somente na primeira instanciação da classe.

4.2.1 Utilizando o Wrapper

Desenvolvemos nosso Wrapper de forma que fosse bem simples executar um script Lua. Assim, basta copiar o script que se deseja rodar para dentro do diretório "assets/files/", instanciar o Wrapper e passar o caminho do script dentro desse diretório. Também implementamos a opção de passar argumentos de entrada para o script Lua a ser executado, através do Wrapper. Como exemplo, o código a seguir irá executar o script no diretório "assets/files/test/script1.lua" e configurar o argumento de entrada `arg[1]` com o valor "arg1" (e `arg[0]` como "script1.lua"):

```
wrapper(‘‘test/script1.lua’’,‘‘arg1’’)
```

4.3 Luaproc no Android

Concluída a etapa de utilizar Lua no Android, partimos para a importação do módulo Luaproc e execução de um script que utilize-a no seu código (através do `require` de Lua). Para a execução de scripts que utilizem a biblioteca Luaproc, a solução foi integrar o Luaproc com o *wrapper* através do `luaL_requiref` da API C. Porém para o caso de outras dependências (como outros scripts Lua), foi necessário modificar seu código fonte pois os processos Lua criados não compartilham memória. Isto significa que a referência para os arquivos Lua na memória interna Android tem que ser refeita para cada LP criado. Para solucionar esse problema, armazenamos este caminho para então adicioná-lo ao *package loader* de cada LP novo criado.

5
Modelo proposto

Neste capítulo, apresentamos nosso modelo de troca de mensagens entre processos distribuídos para a biblioteca Luaproc. Discutiremos sobre a interface, a arquitetura e a implementação do mesmo.

O propósito deste modelo é disponibilizar Luaproc na plataforma Android de forma que seja possível combinar as facilidades de um serviço de enfileiramento de mensagens com o suporte para processamento concorrente e paralelo já existente na biblioteca. Essa combinação permite a exploração de paralelismo em aplicações distribuídas, que exigem comunicação entre dispositivos, e também a simples paralelização entre múltiplos dispositivos.

Para a interface utilizamos um padrão semelhante aos tópicos/canais da biblioteca Luaproc. Qualquer mensagem transmitida terá como destino uma fila nomeada (ou *tópico*). Para a implementação, após estudar os diferentes protocolos de enfileiramento (discutido com detalhes na Seção 2.2), optamos pelo protocolo MQTT.

5.1
Interface MQ Luaproc

Para a nova interface de troca de mensagens, que chamamos de "MQ", tentamos nos manter fiéis para com a interface oferecida pelo Luaproc. Dessa forma, nos inspiramos na sua API de troca de mensagens entre processos (IPC) e de criação/destruição de canais. Assim, oferecemos uma API semelhante, com operações de envio/recebimento de mensagens, além de realizar/desfazer o registro em tópicos (semelhante aos canais Luaproc). No entanto, vale ressaltar que a semântica da nova comunicação não segue o mesmo sincronismo da comunicação original (entre LP's através dos canais). O envio de informações é apenas sincronizado com o servidor, assim não é possível garantir a sincronia de dois LP's executando em máquinas distintas, um enviando e outro recebendo, da mesma forma como é feito originalmente.

O gerenciamento dos tópicos é feito pelo *broker*, o que significa que o usuário não precisa se preocupar com a criação ou destruição dos mesmos. O usuário pode assumir que o servidor vai tratar corretamente os pedidos de registro e publicações de mensagens, por exemplo. Somente strings são

trafegadas pela rede, dessa forma os tipos aceitos no envio de mensagens em Lua são apenas números e strings. Todas as operações MQ que enviam informações, ou pedidos, são sincronizadas com o *broker*, o que quer dizer que os LP's correspondentes estarão bloqueados até que as operações sejam confirmadas pelo mesmo. No caso do recebimento de mensagens, há a opção da operação ser síncrona ou assíncrona. A primeira bloqueia o LP até o recebimento de uma mensagem e a outra não.

- mqconnect(configuration)
  Conecta ao servidor com as configurações fornecidas. O processo é bloqueado até o término (sucesso ou falha).
- mqdisconnect()
  Desconecta do servidor. O processo é bloqueado até o término (sucesso ou falha).
- mqregister(topic)
  Registra o cliente ao tópico fornecido. O processo é bloqueado até o término (sucesso ou falha).
- mqunregister(topic)
  Cancela o registro do cliente MQTT ao tópico fornecido, além de eliminar as mensagens armazenadas deste tópico. O processo é bloqueado até o término (sucesso ou falha).
- mqsend(message, topic)
  Envia a mensagem fornecida ao tópico escolhido. O processo é bloqueado até o término (sucesso ou falha).
- mqreceive(topic, [asynchronous])
  Se há uma mensagem armazenada do tópico escolhido, retorna-a. Caso contrário, se o booleano "asynchronous" for verdadeiro retorna-se nil e uma mensagem de erro, senão o processo Lua é bloqueado até o recebimento de uma mensagem.

5.2 Implementação

Tendo em vista a baixa confiabilidade de conexão do mundo mobile escolhemos o protocolo MQTT, já que ele foi criado para ambientes de alta latência e sujeitos a quedas de conexão. Outro fator influenciador foi a implementação em C da Paho MQTT ter se mostrado mais estável do que a implementação em C do RabbitMQ[26].

[26] `https://github.com/alanxz/rabbitmq-c`

5.2.1 Eclipse Paho MQTT C client

Realizamos a extensão da biblioteca Luaproc através da implementação C do protocolo MQTT feita pela Eclipse Paho. Como a implementação da biblioteca Luaproc é em C, isto facilitou a integração com a biblioteca cliente MQTT.

5.2.2 Arquitetura

A arquitetura original de Luaproc consiste em uma ou mais threads de sistema operacional (*workers*) responsáveis pelo processamento e gerenciamento dos processos Lua, implementados por co-rotinas. Os *workers* adquirem os LP's a partir de uma fila de processos prontos. Cada LP é então executado por um *worker* até que o processo termine ou realize um `yield`, o que pode ocorrer em 3 casos (ilustrado na Figura 5.1):

- A execução de um `yield` explícito no script Lua
- A execução de um `yield` implícito através da função `send` (quando não há um processo esperando para receber)
- A execução de um `yield` implícito através da função `receive` (quando não há um processo esperando para enviar)

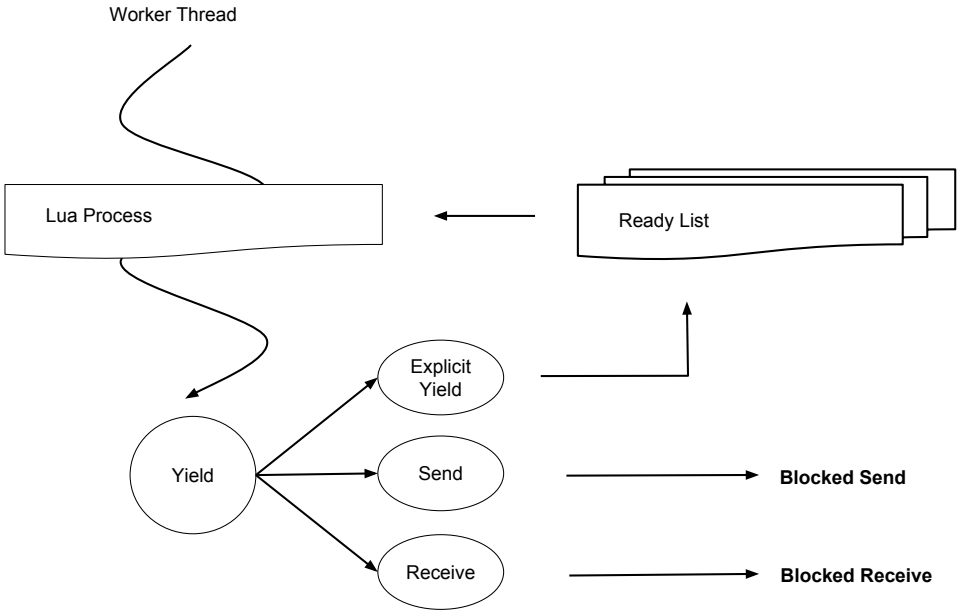

Figura 5.1: Tratamento de um *yield* feito por um processo Lua.

No caso de um processo Lua realizar um `send`/`receive`, se houver um processo Lua esperando para completar um `receive`/`send`, a operação é completada imediatamente. Caso não exista, a chamada `send`/`receive` coloca o processo Lua em uma fila de bloqueados e realiza um `yield`, liberando a thread atual para executar outro LP. Apenas quando aparecer um outro processo para casar com ele a operação será finalizada, retornando-o para a fila de processos prontos. A estrutura dos canais pode ser vista na Figura 5.2.

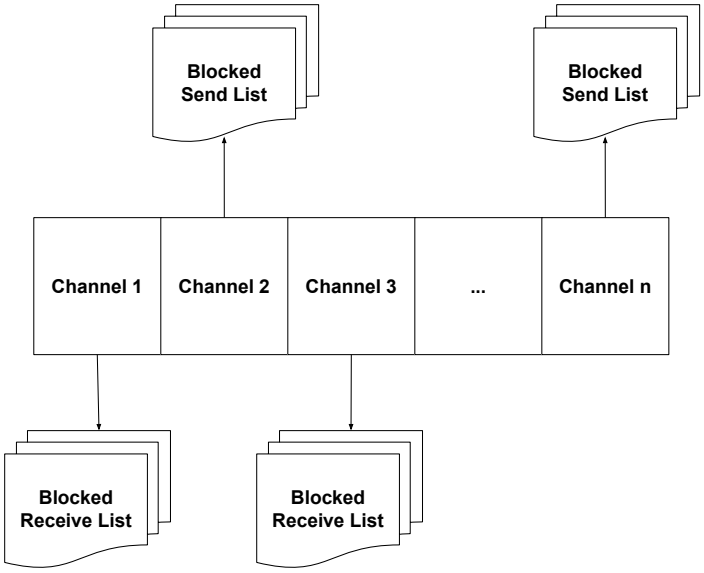

Figura 5.2: Canais Luaproc com suas respectivas listas de LP's bloqueados por envio e por recebimento.

A arquitetura original de Luaproc também prevê os seguintes estados em que os processos Lua podem se encontrar:

- LUAPROC_STATUS_IDLE
  Estado ocioso (inicial)
- LUAPROC_STATUS_READY
  Estado pronto para executar.
- LUAPROC_STATUS_BLOCKED_SEND
  Estado bloqueado ao tentar enviar uma mensagem.
- LUAPROC_STATUS_BLOCKED_RECV
  Estado bloqueado ao tentar receber uma mensagem.
- LUAPROC_STATUS_FINISHED
  Estado de execução concluída.

Com a arquitetura existente, apesar das operações `send`/`receive` serem síncronas, o *worker* não fica bloqueado (graça ao `yield`). Assim, nesse trabalho nos preocupamos em estender a arquitetura de forma a não bloquear a thread

de SO (*worker*) em chamadas síncronas às funções da interface MQ. Fizemos uso das chamadas assíncronas da biblioteca para manter um comportamento semelhante ao `send`/`receive` da biblioteca original, em que realizamos o `yield` quando necessário de maneira a obter um sincronismo (em Lua) sem comprometer/bloquear o *worker*.

Com a introdução das novas operações MQ, foram acrescentados dois novos estados `LUAPROC_STATUS_BLOCKED_MQ_SEND` e `LUAPROC_STATUS_BLOCKED_MQ_RECEIVE` (semelhantes aos estados bloqueados acima). Assim, os processos Lua podem ficar bloqueados aguardando uma operação MQ terminar. Quando estão neste estado eles são colocados em uma fila de processos bloqueados por MQ, conforme a Figura 5.3.

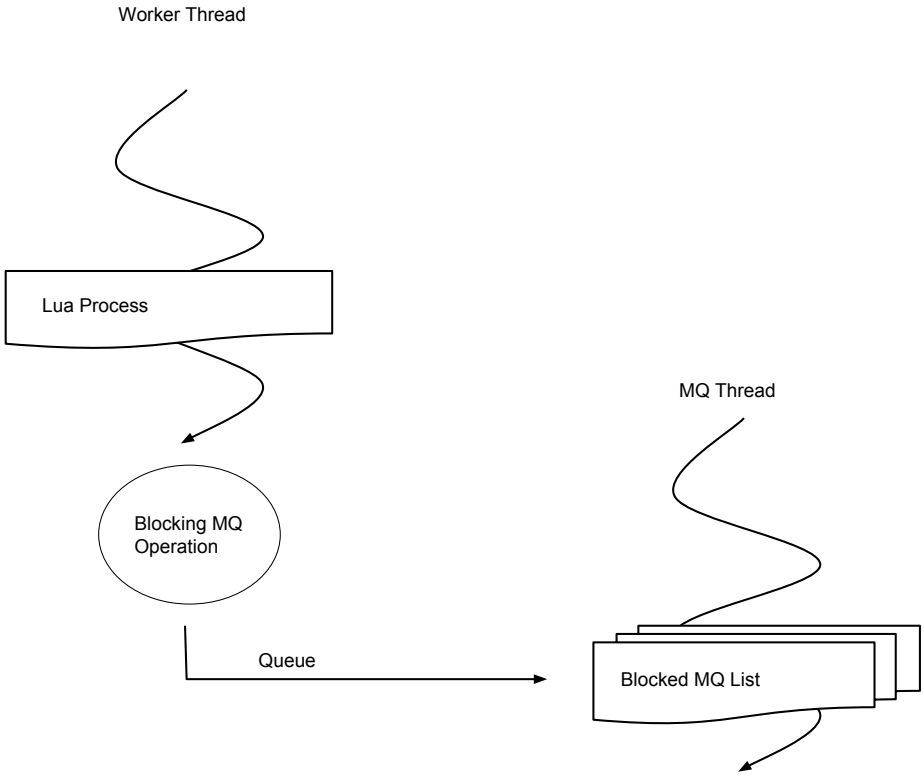

Figura 5.3: Procedimento de bloqueio por operação MQ.

Em nossa extensão, dedicamos uma thread de SO (Sistema Operacional) para o gerenciamento dos processos Lua bloqueados por operações MQ, chamada de MQ_Worker. Uma única *thread* independente - a MQ_Worker - gerencia os processos Lua da lista de bloqueados (por MQ).

A biblioteca Paho define a figura de um cliente MQTT. Este cliente é uma estrutura da biblioteca Paho que é utilizada nas operações de comunicação com

o servidor/*broker*. Pode-se pensar nele como a representação de uma conexão com o servidor. Assim, o cliente MQTT é o responsável pela comunicação com o servidor/*broker*, já que ele é utilizado em todas as operações de comunicação. Para tratar dessa comunicação, a biblioteca cria duas threads de SO quando ocorre uma conexão: uma thread para envio e outra para o recebimento de informações[27]. Isto totaliza então 3 threads auxiliares na aplicação, como ilustrado na Figura 5.4: nosso *MQ_Worker*, que gerencia os LP's bloqueados (por MQ); a thread Paho de envio, que envia informações ao servidor (como mensagens, pedidos de inscrições à tópicos, etc); e a thread Paho de recebimento, que recebe as mensagens do servidor.

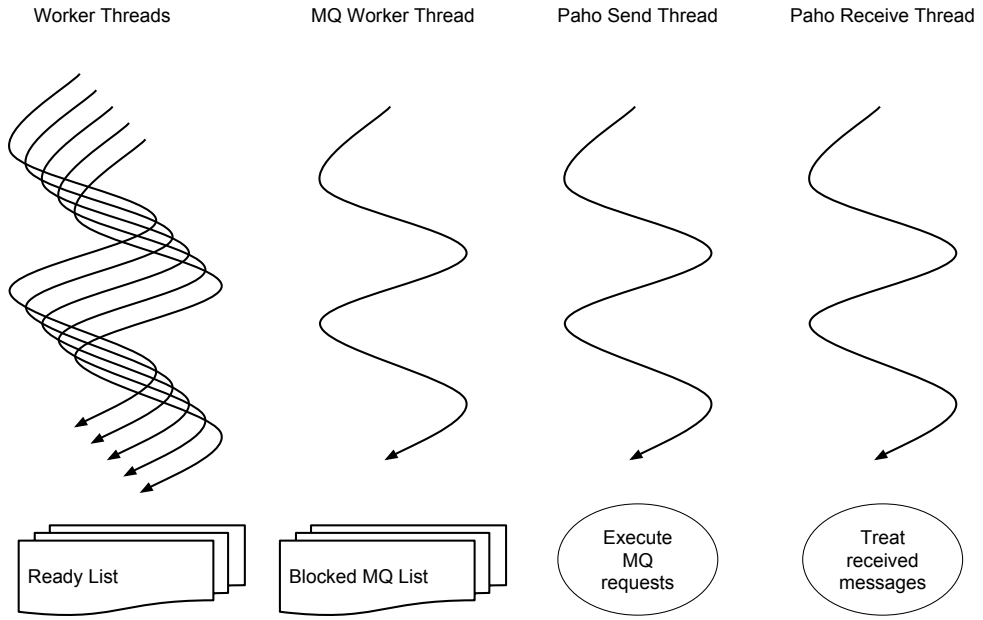

Figura 5.4: Threads da aplicação: os *workers* e as threads auxiliares.

Na arquitetura do nosso modelo, um detalhe importante é que há apenas um cliente MQTT por aplicativo, atendendo a todos os pedidos dos processos Lua. Essa escolha de utilizar somente um cliente foi feita considerando que cada aplicativo poderia distribuir suas mensagens recebidas dentre seus processos Lua. Outra razão para essa escolha foi que se colocássemos uma conexão por LP

[27]Apesar de não termos encontrado essa informação na documentação oficial, através de nossas observações e analise do código fonte constatamos que são criadas essas duas threads pela biblioteca Paho.

poderíamos sobrecarregar o sistema de comunicação do dispositivo. Também, ao multiplexar a comunicação dos diversos LP's, visamos uma economia de energia. Caso implementássemos a arquitetura utilizando vários clientes, o trafego das múltiplas conexões iria acarretar em um maior consumo energético pelo celular e consequentemente um menor tempo útil do aparelho. Assim, ao evitar o uso de múltiplas conexões evitamos então gastos adicionais de energia. A Figura 5.5 mostra essa relação entre o cliente e os processos Lua.

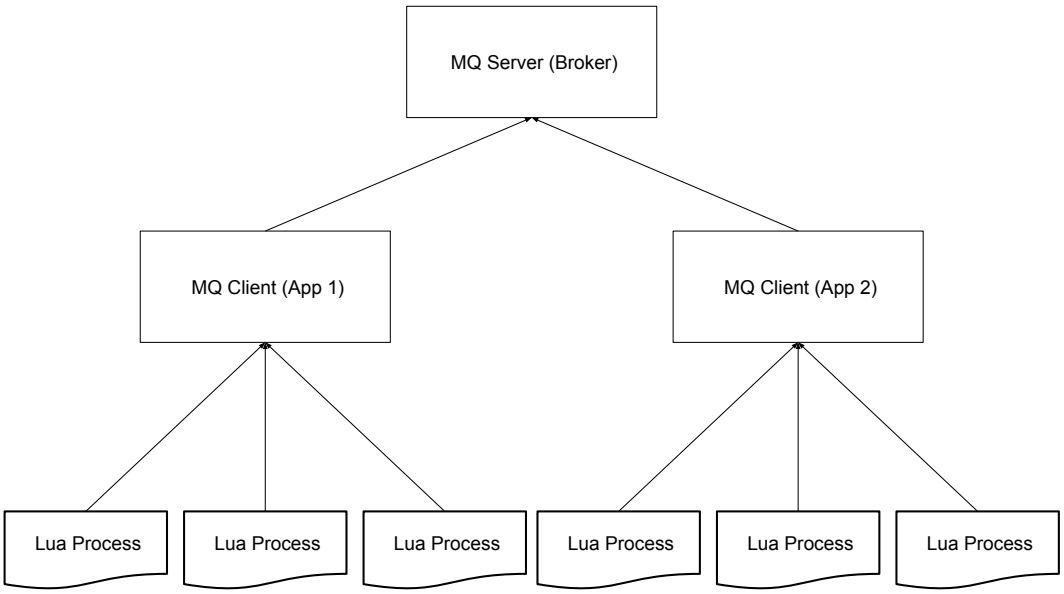

Figura 5.5: Arquitetura do Cliente MQTT.

5.2.3 Comunicação e concorrência

Toda a comunicação com o servidor MQTT ocorre utilizando um único cliente MQTT central (como visto na Subseção 5.2.2). Assim, os processos Lua não se comunicam diretamente com o servidor e apenas realizam requisições. As operações que então forem feitas pelos LP's serão realizadas assincronamente pelas threads criadas pela biblioteca Paho. Em outras palavras, as chamadas à interface MQ são implementadas como requisições a uma única thread da Paho (de envio). Isto ocorre em todas as operações da interface MQ com exceção do recebimento de mensagens. Isto porque o recebimento real das

mensagens é feito pela outra thread Paho, de recebimento, que ao receber uma mensagem do *broker* executa uma *callback* de recebimento que armazena a mensagem localmente. Assim, a chamada `mqreceive` apenas verifica localmente a existência de alguma mensagem armazenada e não interage com o cliente MQTT. Este funcionamento pode ser visualizado na Figura 5.6.

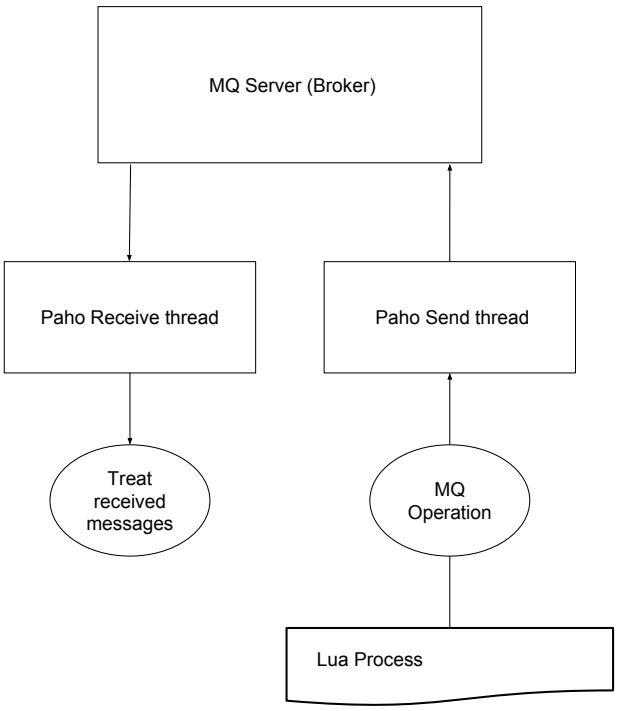

Figura 5.6: Comunicação com o servidor, tratada pelas threads da Paho.

Caso uma operação MQ (executada assincronamente pela thread Paho) finalize antes do término da função que a disparou, o processo Lua segue sua execução normalmente. Caso contrário, o processo Lua é bloqueado de forma a liberar o *worker* executando-o(através de um `yield`). Desta forma conseguimos atingir um sincronismo no código Lua, que faz a requisição MQ, sem comprometer o *worker* (ao bloqueá-lo até a conclusão da requisição). Neste último caso, o LP é então posto em uma fila de processos bloqueados por operações MQ e permanece na mesma até a conclusão da operação.

O nosso *MQ_Worker* (responsável pelo gerenciamento dos processos Lua bloqueados) é criado ao se conectar com o servidor e destruído ao se desconectar. Após uma operação MQ finalizar (com sucesso ou falha), o devido LP é então retornado para a fila de processos prontos (pelo *MQ_Worker*), onde será executado novamente pelos *workers*. A sincronização dos pedidos MQ

paralelos é realizada pelas threads da biblioteca Paho, como por exemplo 2 LP's enviando mensagens para o servidor ao mesmo tempo. Porém a sincronização do gerenciamento dos estados dos processos Lua bloqueados é feita por nosso *MQ_Worker*, utilizando *mutexes* para garantir a corretude da lista dos processos Lua (semelhante à implementação original dos *workers*). No caso particular de desconexão, utilizamos também uma variável de condição para sinalizar ao *MQ_Worker* que o mesmo deve finalizar. Assim, o *MQ_Worker* espera a finalização dos processos Lua bloqueados por envio (já que estes podem estar finalizando, e se não for o caso há um timeout definido que garante o término). Já para os LP's esperando o recebimento de mensagens, estes são liberados imediatamente na desconexão, retornando um código de erro.

Mensagens recebidas de canais nos quais ocorreram inscrições são armazenadas localmente e são consumidas pelo primeiro LP que executar um pedido de recebimento. Em outras palavras, um LP que executar um `mqreceive` vai consumir e liberar a mensagem (se houver) da fila local do tópico escolhido. Dessa forma, o broadcast feito pelo *broker* não se aplica aos diversos LP's que podem ter se inscritos no mesmo tópico. Modelamos dessa forma tendo em mente que seria a melhor forma de distribuir possíveis tarefas a serem recebidas, ou de processar dados paralelamente, já que cada LP poderia processar uma mensagem independentemente. Caso seja necessário que vários LP's recebam o mesmo dado, é possível compartilhar a mensagem através dos canais Luaproc. Dessa forma, esse modelo não prejudica esse último caso de uso.

5.2.4 Alternativa de implementação cogitada

Em um primeiro estudo, cogitamos aproveitar a estrutura orientada a *callbacks* do biblioteca da Eclipse Paho. A ideia era usar as *callbacks* de *operação bem sucedida* para alterar o estado do processo Lua relevante de *bloqueado* de volta para *pronto*. Em outras palavras, quando uma operação MQ finalizar e então chegar o momento executar sua *callback* de sucesso, seria neste instante que pretendiamos alterar o estado do LP para pronto. O problema dessa abordagem, porém, foi sincronizar o `yield` do LP com essa *callback* assíncrona de sucesso. Por exemplo, seria possível a operação assíncrona finalizar logo antes de ser realizado o `yield` do LP para bloqueá-lo, assim deixando o processo em limbo (pois seu estado nunca seria alterado). Cogitamos realizar uma sincronização por meio de *flags* intermediárias porém esta ideia foi abandonada por sua falta de clareza para uma futura manutenabilidade de código.

6
Avaliação

Para fins de avaliação, testamos a implementação do nosso modelo (Paho MQTT Lua) contra a biblioteca Paho MQTT C a fim de medir a sobrecarga de Lua, além também do desempenho. Também implementamos alguns aplicativos utilizando nossa biblioteca Luaproc com a extensão MQTT. Quase todas as aplicações que desenvolvemos trocam informações pontuais e realizaram o processamento localmente (tirando proveito do paralelismo local da biblioteca original). Porém realizamos um exemplo que utilizou processamento distribuído com a nossa implementação do problema do Caixeiro Viajante, descrita na Seção 6.3.

Os testes foram realizados em um Samsung SM-J700M, versão Android 6.0.1 API 23, 1.5BG de RAM e processador ARM Cortex-A53 (Octa-Core 64bit). Nos testes com mais de um aparelho, além do Celular J700M foram utilizados também emuladores Nexus 5 do Android Studio executados em uma máquina com 16GB RAM e processador Intel i7-4770 (Octa-Core 64bit)

6.1
Sobrecarga de Lua

A versão Paho MQTT Lua desenvolvida por nós utiliza a implementação C da Paho e nessa seção pretendemos avaliar o desempenho e sobrecarga de nossa biblioteca em comparação com a biblioteca Paho MQTT C (sem Lua).

6.1.1
MQTT C vs MQTT Lua

Realizamos o mesmo teste de *ping-pong* da Subseção 2.2.2 entre as versões Lua e C, cuja medidas podem ser vistas na Figura 6.1. As bibliotecas se mostraram com desempenho bem semelhantes, porém a versão Lua se mostrou um pouco mais rápida. Apesar da diferença ser pequena, acreditamos que ela ocorreu provavelmente devido ao uso dos emuladores, que pode ter criado alguma distorção nos tempos de execução.

	1000 Msgs	
	Paho Lua	Paho C
Média (segundos)	6,93	6,96

	10000 Msgs	
	Paho Lua	Paho C
Média (segundos)	66,92	70,46

Figura 6.1: Tempo de execução do *ping-pong* de mensagens entre Lua e C.

Também realizamos alguns testes comparando apenas as versões Paho Lua e Paho C. Neste teste em particular, pareamos o celular J700M com outro celular ao invés de um emulador, no caso um LGE LG-D686, versão Android 4.4.2 API 19, 1GB de RAM e processador ARM Cortex-A9 (Dual-Core 32bit).

Nelas realizamos medições de uma comunicação unidirecional, com um dispositivo apenas enviando e outro apenas recebendo. O objetivo desse teste é analisar o tempo do comunicação dos aparelhos sem a influência da espera de uma resposta (como no teste *ping-pong*). Os resultados estão ilustrados na Figura 6.2.

	10000 Msgs			
	Send		Receive	
	Paho Lua	Paho C	Paho Lua	Paho C
Média (segundos)	17,06	12,92	21,09	18,18

Figura 6.2: Tempo do envio e recebimento unidirecional de mensagens.

Os resultados mostram um desempenho melhor pela versão C sobre a versão Lua. Acreditamos que a diferença de tempo entre as medidas das versões ocorreu pela sobrecarga do gerenciamento dos processos Lua Bloqueados. Ou seja, pelo tempo de inserção na lista de bloqueados, na verificação de completude da operação MQ e finalmente o tempo de retorno à lista de execução. Isto não foi um problema no teste anterior (*ping-pong*) devido ao tempo de espera gerado pelo recebimento síncrono da resposta do outro aparelho, que acreditamos que escondeu essa sobrecarga.

6.2 Aplicativos

Foram feitos 4 aplicativos no total, cada um deles explorando de forma diferente a biblioteca desenvolvida:

1. Aplicativo de chat
2. Aplicativo do jogo "Hare and hounds" com sugestão de jogada
3. Aplicativo de compartilhamento de fotos
4. Aplicativo de busca e reconhecimento facial

O primeiro aplicativo é uma implementação de *chatroom*, realizando troca de mensagens textuais. Um usuário então recebe e envia mensagens. para um determinado tópico, e as mensagens trocadas são exibidos na tela de formas diferentes dependendo se a mensagem for do próprio usuário ou não. Uma execução do aplicativo pode ser vista na Figura 6.3.

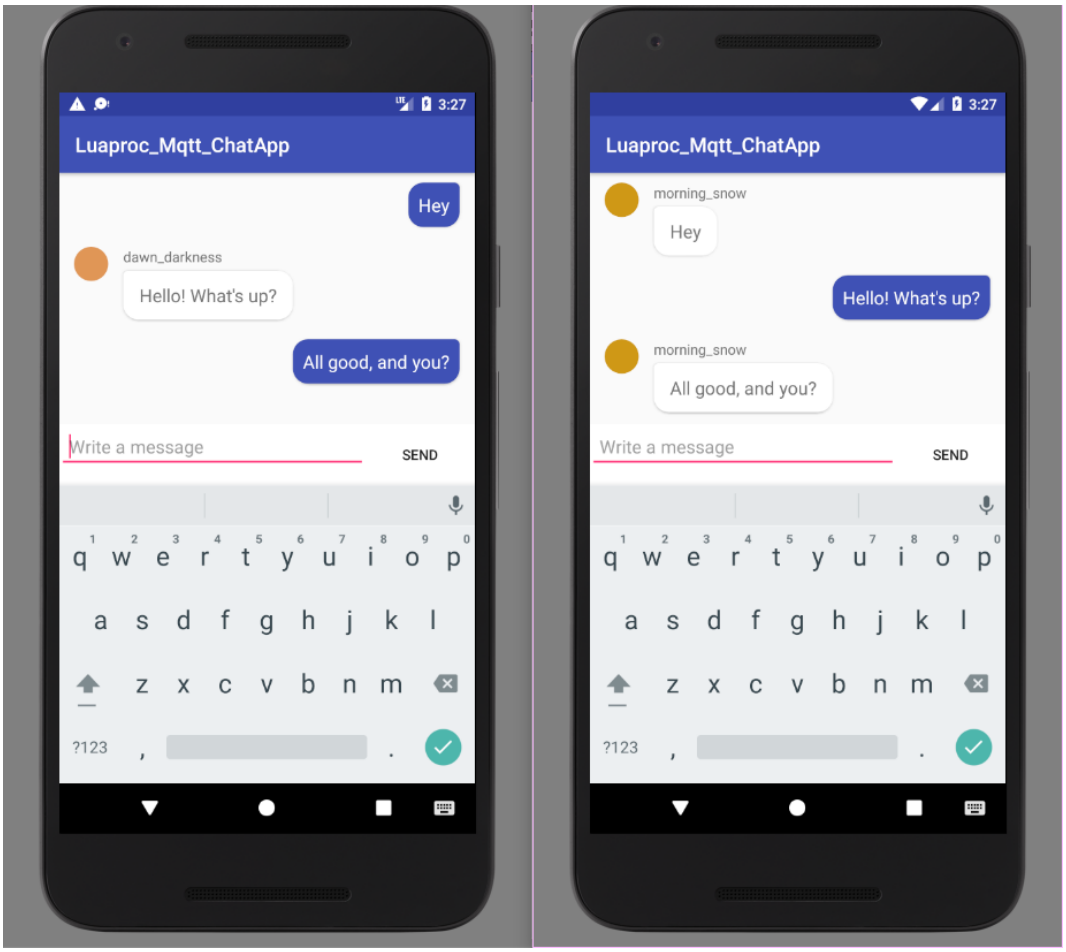

Figura 6.3: Aplicativo Chat com dois usuários.

O objetivo deste aplicativo foi exemplificar um caso de uso em que a biblioteca pode ser usada somente para realizar a comunicação, ou troca de mensagens. Em outras palavras, não necessariamente precisa-se ter um problema paralelizável para utilizar a biblioteca. Luaproc pode ser utilizado para a simples conveniência de realizar trocas de mensagens em Lua. Um detalhe relevante foi que neste aplicativo se mostrou importante o uso do recebimento assíncrono de mensagens para não travar a interface gráfica do celular. A Seção C.1 do apêndice mostra o código do aplicativo.

O segundo aplicativo é uma implementação do jogo "Hare and hounds"[28], um jogo de tabuleiro para dois jogadores podendo escolher ou o lado da lebre (hare) ou o lado dos cães (hounds). O objetivo do jogo para os cães consiste em prender a lebre, enquanto para a lebre consiste em ultrapassar os cães (já que eles não podem se mover para trás). A comunicação das jogadas entre os jogadores é feita pelo tópico do jogo, e para cada jogada é calculada uma sugestão de jogada para o jogador. Essa sugestão é computada localmente através do algoritmo de Poda Alpha-beta (ou *Alpha–beta pruning*)(8), como pode ser visto na Figura 6.4.

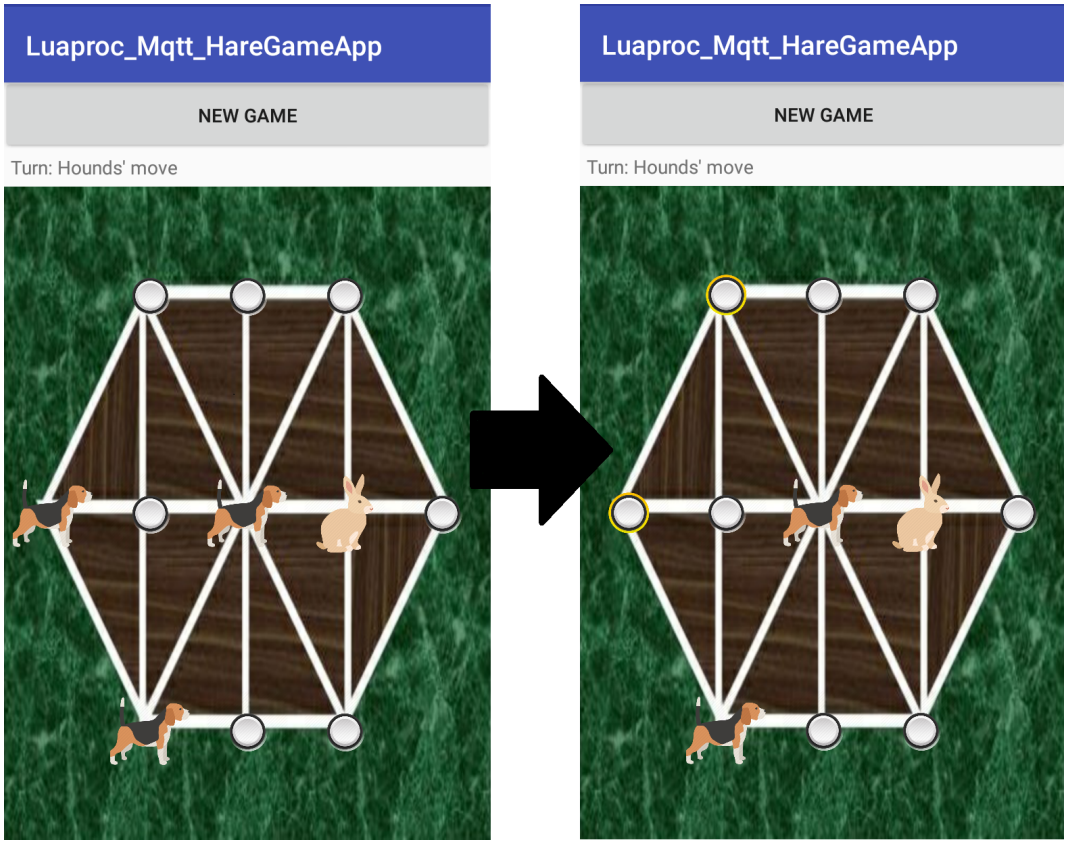

Figura 6.4: Antes e depois da sugestão de jogada.

[28] `https://en.wikipedia.org/wiki/Hare_games#Hare_and_Hounds`

O objetivo deste aplicativo foi exemplificar a exploração do paralelismo local com troca de informações entre aparelhos, já que a sugestão é calculada localmente de forma paralela depois de receber o tabuleiro atual do oponente.

A Figura 6.5 mostra o tempo de execução da sugestão de jogada desse aplicativo, comparando a sua execução serial com uma versão paralela. Assim, conseguimos ter uma ideia do ganho obtido ao tirar proveito do paralelismo Luaproc (após feita a comunicação entre os aparelhos). Variamos também a versão paralela utilizando 8 e 4 *workers*, como pode ser visto na Figura 6.5.

	Jogo Hare and Hounds					
	Início de jogo			**Final de jogo**		
	8 Workers	4 Workers	Serial	8 Workers	4 Workers	Serial
Média (segundos)	13,92	15,18	48,07	0,84	0,86	1,64
aceleração	3,45	3,17	1,00	1,94	1,91	1,00

Figura 6.5: Tempo do cálculo da sugestão do jogo Hare and Hounds.

Medimos os tempos em duas situações: o início e final do jogo. Isto pois eles representam duas configurações, uma com muitas possibilidades de jogadas e outra com poucas. A quantidade de LP's criadas são de acordo com as jogadas immediatas possíveis (ou seja, apenas para a primeira jogada de cada possibilidade e não todas até o término), que no caso foram 8 e 7 LP's respectivamente para o início e final do jogo. Nota-se que houve um benefício no jogo Hare'n'Houds, porém não houve uma aceleração linear com aumento de *workers*. A utilização de 8 *workers* resultou em uma aceleração muito semelhante à obtida com 4 workers, apesar de estarmos utilizando um dispositivo com 8 *cores*. Isto se dá pelo balanceamento de tarefas de cada processo Lua, pois não necessariamente cada configuração do tabuleiro terá o mesmo custo para fazer o cálculo do algoritmo alpha-beta. Dessa forma, um LP está com uma carga grande de trabalho (responsável por maior parte do tempo) enquanto os outros terminam rapidamente. Nota-se também que ao final do jogo, quando há poucas possibilidades tanto de configurações de tabuleiros quanto de jogadas possíveis, a aplicação não usufrui tanto do paralelismo devido ao pouco trabalho (caindo também sua aceleração).

Uma dificuldade deste aplicativo em particular foi sincronizar os jogadores, por conta da comunicação em *broadcast*. Por se tratar de um jogo de tabuleiro entre dois jogadores, não é desejado nem necessário compartilhar ou receber jogadas com outro jogador além do oponente. A solução para isso foi dar a cada jogador um tópico único para que só fosse usado esse tópico (pelo oponente) durante o jogo. Assim, o jogador que estiver procurando um jogo

realizará um *broadcast* de seu tópico a um tópico público. Os oponentes então receberão esse tópico e responderão através do mesmo com os seus respectivos tópicos. Para então completar o *matchmaking*, o jogador inicial irá acusar o recebimento para o primeiro tópico que receber e o jogo começará. Para os demais oponentes que podem estar aguardando uma resposta, foi definido um *timeout* para recomeçar caso eles não tenham sido escolhidos. A Seção C.2 do apêndice mostra o código do aplicativo.

O terceiro aplicativo trata do envio e recebimento de fotos. Um usuário pode escolher entre enviar ou receber fotos de um determinado tópico. Ao enviar, o usuário pode escolher manualmente um número determinado de fotos ou uma pasta contendo as fotos. Daí, as fotos escolhidas são serializadas e enviadas ao tópico adequado que então será recebido pelos usuários que escolheram receber fotos desse tópico.

O objetivo deste aplicativo foi exemplificar a exploração do paralelismo do envio de informações, já que cada foto é serializada e enviada de forma paralela e independente. Além disso, vários dispositivos poderem receber essas fotos, bastando apenas estarem inscritos no tópico correspondente.

Medimos o tempo de execução do envio das fotos desse aplicativo, comparando a sua execução serial e com uma versão paralela. Mais uma vez, variamos também a versão paralela utilizando 8 e 4 *workers*, como pode ser visto na Figura 6.6.

	Compartilhamento de fotos					
	10 fotos			60 fotos		
	8 Workers	4 Workers	Serial	8 Workers	4 Workers	Serial
Média (segundos)	18,50	19,13	49,05	21,21	34,32	142,09
aceleração	2,65	2,56	1,00	6,70	4,14	1,00

Figura 6.6: Tempo do compartilhamento de fotos.

Foi criado um LP por foto para tratar da serialização, o que significa 10 LP's na primeira medida e 60 LP's na segunda. No caso do aplicativo de compartilhamento de fotos, quando são enviadas poucas fotos nota-se algo similar ao caso do app do jogo Hare'n'Hounds. O desempenho não melhora muito ao utilizar mais *workers* pois há uma foto "pesada" (em torno de alguns MB enquanto há outras na ordem de KB) que se tornou o gargalo da execução. Ao aumentar a quantidade de fotos estamos gerando mais trabalho de forma a compensar esse desbalanceamento, deixando mais evidente essa observação. Com mais fotos, notamos que a aceleração com 8 *workers* ficou melhor do que a execução com 4 *workers* (em contraste com o caso com menos fotos em que

quase não houve diferença ao aumentar os *workers* de 4 para 8). A Seção C.3 do apêndice mostra o código do aplicativo.

O quarto aplicativo segue a ideia de "Encontrar pessoas perdidas"(9). O propósito da aplicação seria buscar pessoas (em geral crianças) perdidas de forma distribuída (em vários dispositivos). O usuário que estiver procurando, por exemplo, uma criança, utilizaria uma foto da mesma para enviar e buscar, através de reconhecimento facial[29] , dentre as fotos da Galeria dos outros usuários do aplicativo. O reconhecimento facial se divide em duas etapas, primeiro identificando as faces da foto e em seguida comparando-as e agrupando as faces cujo modelo classifica como sendo a mesma pessoa. Isto está ilustrado na Figura 6.7. Durante o desenvolvimento deste aplicativo apareceu a necessidade de utilizar a funcionalidade do recebimento assíncrono (`mqrecieve`), para encontrar os dispositivos disponíveis.

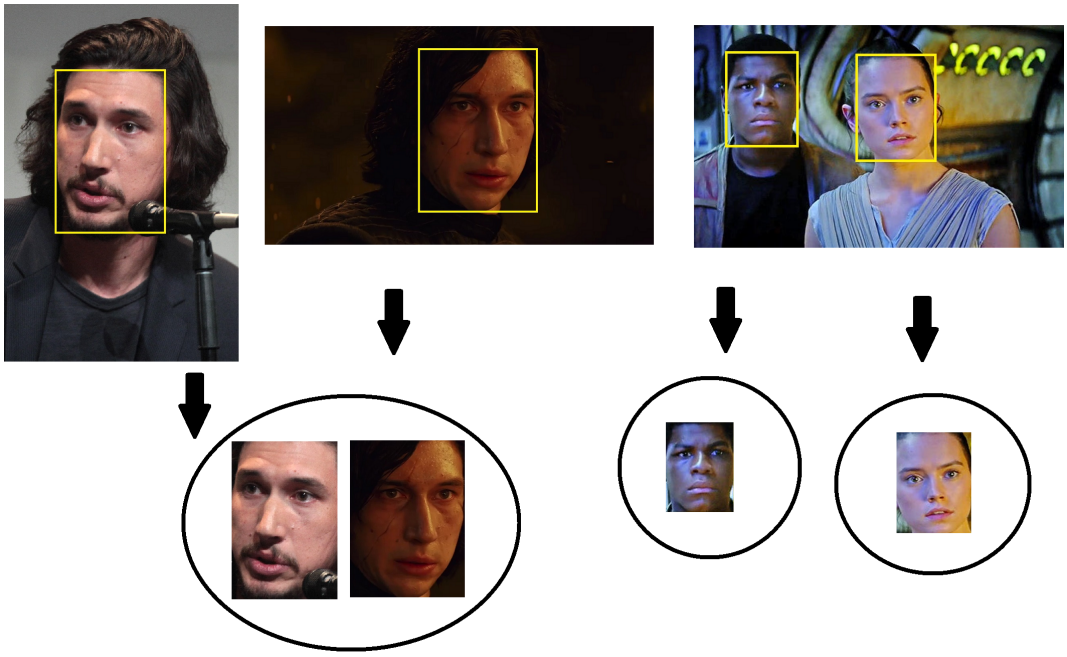

Figura 6.7: Clusterização por reconhecimento facial.

O objetivo deste aplicativo foi exemplificar a exploração da comunicação distribuída combinada com paralelismo no processamento local, já que vários aparelhos podem receber a foto de referência (para a busca) e procurar paralelamente dentre suas fotos locais. Assim, realizando uma forma de busca distribuída.

Medimos o tempo de execução da busca por reconhecimento facial desse aplicativo, comparando a sua execução serial com uma versão paralela.

[29]Para realizar o reconhecimento, utilizamos a biblioteca C++ "dlib" (`https://github.com/davisking/dlib`). Criamos um módulo Lua a partir de seu código fonte para então ser chamado em conjunto com Luaproc (para paralelizar a busca).

Novamente variando também a versão paralela utilizando 8 e 4 *workers*, como pode ser visto na Figura 6.8.

	Busca por reconhecimento facial					
	Encontrando			Não encontrando		
	8 Workers	4 Workers	Serial	8 Workers	4 Workers	Serial
Média (segundos)	105,25	123,17	230,40	30,04	48,55	152,91
aceleração	2,19	1,87	1,00	5,09	3,15	1,00

Figura 6.8: Tempo de busca no conjunto de fotos.

Testamos este aplicativo com uma base com 10 fotos e rodamos em duas configurações de teste: uma encontrando a pessoa e outra sem encontrar a pessoa. Isto foi feito pois ao encontrar a pessoa, o aplicativo retorna a foto desta pessoa ao solicitante e, como visto com o app de compartilhamento de fotos, este processo pode ser custoso. Assim, sem encontrar a pessoa, podemos avaliar simplesmente a busca desconsiderando o tempo de envio da foto. Foram criados 10 LP's, um para cada foto da base. De forma semelhante ao caso do app de compartilhamento de fotos, o uso de mais *workers* não gerou muito impacto devido a um gargalo gerado pelo LP que está enviando a foto da pessoa encontrada. Porém quando isto não ocorre nota-se uma aceleração significativa com relação ao aumento da quantidade de *workers*. A Seção C.4 do apêndice mostra o código do aplicativo.

6.3 Teste do Caixeiro Viajante

Realizamos um teste com o problema clássico do Caixeiro Viajante(10), com o objetivo de medir o ganho possível com processamento distribuído. Modelamos o problema com um dispositivo Produtor (que enviará o trabalho a ser feito) e outros aparelhos Consumidores (que executarão o trabalho, retornando o resultado). Medimos os tempos de duas estratégias para a execução distribuída. A primeira utilizou tópicos fixos e conhecidos para os Consumidores, assim o Produtor fica alternando entre elas para enviar o trabalho. A segunda utilizou um tópico de Consumidores livres (ou *idle*) de forma que o Produtor descobre os Consumidores disponíveis para então enviar trabalho a eles. Esta segunda implementação foi feita para garantir o balanceamento das tarefas. A Figura 6.9 mostra os tempos de execução que obtivemos para as diferentes estratégias e diferentes números de Consumidores.

	Caixeiro Viajante							
	Tópicos Fixos				Tópico de Balanceamento			
Consumidores	1	2	3	4	1	2	3	4
Média (segundos)	44,37	24,59	23,16	18,07	80,87	42,39	30,46	22,61
Aceleração	0,95	1,71	1,81	2,32	0,52	0,99	1,38	1,86
Aceleração relativa	1,00	1,80	1,92	2,46	1,00	1,91	2,65	3,58

	Caixeiro Viajante - Standalone
Média (segundos)	41,95

Figura 6.9: Tempo da resolução do problema do Caixeiro Viajante.

Os testes foram executados com 14 cidades e uma quantidade fixa de 2 *workers* por Consumidor. A quantidade de LP's está relacionado à quantidade de cidades. Como se trata de um grafo completo e a cidade de origem não conta para o problema, a quantidade de LP's da versão *standalone* é então igual a 13. Para a versão distribuída esta quantidade depende do número de Consumidores e da estratégia utilizada. No caso da estratégia de tópicos fixos, a quantidade é simplesmente o resultado da distribuição, por exemplo para 4 Consumidores o primeiro criará 4 LP's e os demais 3 LP's. No caso da estratégia de balanceamento, não é possível dizer com certeza a quantidade criada por cada Consumidor pois dependerá da disponibilidade de cada um de receber as tarefas.

Além da aceleração com relação à versão *standalone*, também medimos a aceleração relativa, que diz respeito ao ganho da própria implementação com o aumento de Consumidores. Utilizando o tempo da execução *standalone* como base, podemos comparar com a execução distribuída com apenas 1 Consumidor para ter uma ideia da sobrecarga da comunicação pela rede. Nota-se que a segunda versão se mostrou bem mais lenta, devido à necessidade de sincronização para a comunicação, já que o Produtor deve esperar um Consumidor avisar que está livre para então enviar trabalho. Por outro lado, essa implementação obteve uma boa aceleração relativa em contraste com a primeira versão (dos tópicos fixos), que em particular se mostrou bem desbalanceada quando utilizamos 3 Consumidores (mostrando pouco ganho). Apesar da versão balanceada obter uma aceleração relativa próxima do ganho linear, ela ainda foi mais lenta que a versão dos tópicos fixos, o que demonstra a importância da sobrecarga de comunicação.

7
Conclusão e trabalhos futuros

Neste trabalho estendemos a biblioteca Luaproc, oferecendo suporte à programação paralela e distribuída dentro do mesmo módulo, assim possibilitando o usuário tirar proveito tanto do paralelismo multicore quanto multidispositivo. Também combinamos as facilidades de um serviço de enfileiramento de mensagens de forma que o usuário desfrute do desacoplamento resultante, tornando o gerenciamento das mensagens uma preocupação a menos.

Apresentamos um modelo de comunicação multidispositivo para a biblioteca Luaproc. Para implementar este modelo, escolhemos o protocolo MQTT de comunicação e utilizamos a implementação da Eclipse Paho em nossa extensão Luaproc.

Implementamos também uma forma de execução de scripts Lua através de nosso Wrapper Lua, oferecendo uma maneira simples do usuário Android utilizar código Lua de dentro da plataforma.

Através de nossos testes na Subseção 2.2.2 e Subseção 6.1.1 constatamos que, apesar da camada extra de execução do interpretador Lua, a biblioteca se manteve competitiva com a própria versão C da Paho, além das concorrentes como a versão Android do RabbitMQ.

Para trabalho futuros, a biblioteca poderia ser estendida para incluir outros protocolos além do MQTT, como AMQP, de forma a oferecer uma maior flexibilidade ao usuário para escolher seu protocolo de preferência. Outra possibilidade seria a de realizar um estudo para avaliar outras formas de gerenciamento dos LP's bloqueados por MQ, de modo a tirar melhor proveito do tempo de CPU. Poderíamos reavaliar nossa tentativa (vista na Subseção 5.2.4) de utilizar as *callbacks* da Eclipse Paho para reinserir os LP's na fila de execução de forma a "aposentar" nosso *MQ_Worker*.

Referências bibliográficas

[1] A. SKYRME; N. RODRIGUEZ; R. IERUSALIMSCHY. **Exploring lua for concurrent programming**. Journal of Universal Computer Science, 2.2, 2008.

[2] A. SKYRME; N. RODRIGUEZ. **Um modelo alternativo para programação concorrente em lua**. Dissertação de mestrado, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2008.

[3] ANDY PIPER. **Choosing your messaging protocol: Amqp, mqtt, or stomp**. VMware Blogs, 2018. Acesso em: Julho de 2018.

[4] JORGE E. LUZURIAGA; MIGUEL PEREZ; PABLO BORONAT; JUAN CARLOS CANO; CARLOS CALAFATE; PIETRO MANZONI. **A comparative evaluation of amqp and mqtt protocols over unstable and mobile networks**. In: CONFERêNCIA IEEE, 2015.

[5] CALIN CASCAVAL; SETH FOWLER ;PABLO MONTESINOS ORTEGO; WAYNE PIEKARSKI; MEHRDAD RESHADI; BEHNAM ROBATMILI; MICHAEL WEBER; VRAJESH BHAVSAR. **Zoomm: a parallel web browser engine for multicore mobile devices**. In: 18TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2013.

[6] CALIN CASCAVAL. **Keynote talk: Parallel programming for mobile computing**. In: 22ND INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 2013.

[7] F. LORDANEMAIL; ROSA M. BADIA. **Compss-mobile: Parallel programming for mobile cloud computing**. Journal of Grid Computing, 15, 2017.

[8] DONALD E. KNUTH; RONALD W. MOORE. **An analysis of alpha-beta pruning**. Artificial Intelligence, 6, 1975.

[9] CRISTIAN BORCEA; XIAONING DING; NARAIN GEHANI; REZA CURTMOLA; MOHAMMAD A KHAN; HILLOL DEBNATH. **Avatar: Mobile distributed computing in the cloud**. In: CONFERêNCIA IEEE, 2015.

[10] GILBERT LAPORTE; SILVANO MARTELLO. **The selective travelling salesman problem**. Discrete Applied Mathematics - Southampton conference on combinatorial optimization, 26):193–207, 1990.

A
Interface Luaproc

- mqconnect(table configuration)
  Conecta ao servidor com as configurações da tabela “configuration”. É esperado um campo obrigatório “host” (endereço do broker), e opcionalmente os campos “port” (porta em que se deve conectar cujo default é “1883”) e “client_id” (nome/id do client mqtt cujo default é um id gerado aleatoriamente). Retorna 1 se bem sucedido, caso contrário retorna uma indicação de erro.
- mqdisconnect()
  Desconecta do servidor (broker) caso conectado. Se não conectado, apenas retorna. Retorna 1 se bem sucedido, caso contrário retorna uma indicação de erro.
- mqregister(string topic)
  Registra o cliente ao tópico fornecido. As mensagens publicadas para esse tópico serão recebidas e armazenadas. Retorna 1 se bem sucedido, caso contrário retorna uma indicação de erro.
- mqunregister(string topic)
  Cancela o registro do cliente ao tópico recebido, além de eliminar as mensagens armazenadas deste tópico. Retorna 1 se bem sucedido, caso contrário retorna uma indicação de erro.
- mqsend(string message, string topic)
  Envia a mensagem ao tópico escolhido, caso conectado. Retorna 1 se bem sucedido, caso contrário retorna uma indicação de erro.
- mqreceive(string topic, [boolean asynchronous])
  Bloqueia o processo até a chegada de uma mensagem, retornando-a. Caso o booleano “asynchronous” seja verdadeiro e não haja uma mensagem disponível no momento da chamada, retorna uma indicação de erro.

B
Processo para a utilização da extensão Luaproc para Android

B.1
Gerando o pacote AAR

1. Deve-se primeiro selecionar a arquitetura desejada através do filtro de compilação
   No arquivo "build.gradle", Há a possibilidade de escolher o filtro adequado para a arquitetura desejada. Como por exemplo, "x86" ou "armeabi-v7a".

2. Assim o próximo passo é iniciar a geração do pacote AAR da biblioteca no modo "release"

   Terminada essa etapa, o pacote se encontrará no diretório "build/outputs/aar" do Android Studio.

B.2
Importando o pacote AAR Luaproc em seu aplicativo

1. Importando o pacote AAR
   O pacote Luaproc deve ser importado como um novo módulo em seu aplicativo. Assim, deve-se selecionar a opção de criar um novo módulo em seu aplicativo, em seguida selecionando a opção de importar um pacote JAR/AAR.

2. Configurando seu App para utilizar o pacote AAR
   Com o pacote importado, deve-se adicioná-lo como uma dependência de seu aplicativo. Para isso, deve-se entrar nas configurações de seu app (clicando com o botão direito para selecionar *Module Settings*). Em seguida, adicionando o pacote Luaproc na aba de dependências (*Dependencies*).

B.3
Alternativa para utilizar Luaproc Android em seu aplicativo

Caso desejado, pode-se importar o projeto Luaproc diretamente em seu aplicativo ao invés do pacote AAR. Para isso deve-se usar a opção de importação *Import Module*.

Em seguida, deve-se fornecer o caminho do projeto Luaproc para finalizar a importação. Os próximos passos são os mesmos da importação de um pacote AAR, ou seja deve-se adicioná-lo como uma dependência de seu aplicativo. Para isso, deve-se entrar nas configurações de seu app (clicando com o botão direito para selecionar *Module Settings*). Em seguida, adicionando o pacote Luaproc na aba de dependências (*Dependencies*).

B.4
Utilizando o pacote AAR Luaproc em seu aplicativo

Para executar os scripts Lua, serão necessários alguns passos que serão descritos a seguir:

1. Criar a pasta *Assets* do Android Studio
   Esta pasta pode ser criada automaticamente pelo Android Studio, clicando com o botão direito em seu aplicativo e selecionando essa opção. Dentro desta pasta deve-se criar outra pasta "files", que deve conter os scripts Lua que se deseja executar (além dos módulos Lua, utilizados pelo "require" de Lua). Todos os arquivos fora desta pasta serão ignorados, porém não há problema em utilizar sub-pastas dentre dela. Deve-se ter em mente que "files" age como a pasta raiz e isto deve ser refletido nos "requires" dos scripts Lua.

2. Utilizando o LuaScriptWrapper
   Os scripts Lua serão rodados através deste Wrapper, que irá executar o script enxergando a pasta "files" (do *Assets*) como seu diretório de execução. Basta passar o caminho do script (dentro de "assets/files/"), opcionalmente com argumentos de entrada (que serão armazenadas na variável global "arg").

C
Código dos aplicativos

C.1
Aplicativo de chat

– **connect.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local host = arg[1] or "37.187.106.16" --http://test.mosquitto.org
local port = arg[2] or "1883"
local client_id = arg[3]

--process to connect
luaproc.newproc([=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
table = require "table"

local config = {}
config.host = ']=].. tostring(host) ..[=['
config.port = ']=].. tostring(port) .. [=['
config.client_id = (']=].. tostring(client_id) ..[=[' ~=
'nil' and ']=].. tostring(client_id) .. [=[' or nil)

print("luaproc.mqconnect:")
retcode, errmsg = luaproc.mqconnect( config )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

]=])
```

```
luaproc.wait()

print("Done calling '"..arg[0].."'")
```

– **disconnect.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

--process to disconnect
luaproc.newproc([=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print

print("luaproc.mqdisconnect:")
retcode, errmsg = luaproc.mqdisconnect( )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

]=])
luaproc.wait()

print("Done calling '"..arg[0].."'")
```

– **register.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local topic = arg[1]

--process to register
luaproc.newproc([=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
```

```
table = require "table"

print("luaproc.mqregister:")
local topic = "]=]..topic..[=["
print("    topic: "..tostring(topic))
retcode, errmsg = luaproc.mqregister( topic )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

]=])
luaproc.wait()

print("Done calling '"..arg[0].."'")
```

– **unregister.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local topic = arg[1]

--process to unregister
luaproc.newproc([=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
table = require "table"

print("luaproc.mqunregister:")
local topic = "]=]..topic..[=["
print("    topic: "..tostring(topic))
retcode, errmsg = luaproc.mqunregister( topic )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

]=])
luaproc.wait()
```

```
print("Done calling '"..arg[0].."'")
```

– **send.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local topic = arg[1]
local payload = arg[2] --"<name>;<color>;<text>"

--process to send
luaproc.newproc([=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
table = require "table"

print("luaproc.mqsend:")
local topic = "]=]..topic..[=["
local msg = "]=]..payload..[=["
print("    topic: "..tostring(topic))
print("    msg: "..tostring(msg))
retcode, errmsg = luaproc.mqsend( topic, msg )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

]=])
luaproc.wait()

print("Done calling '"..arg[0].."'")
```

– **receive.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local topic = arg[1]
```

```
local outputFolder = arg[2]

--process to receive
luaproc.newproc([=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
table = require "table"
io = require "io"

print("luaproc.mqreceive:")
local topic = "]=]..topic..[=["
print("    topic: "..tostring(topic))
recmsg, errmsg = luaproc.mqreceive( topic , true )
print("    recmsg: "..tostring(recmsg))
print("    errmsg: "..tostring(errmsg))

if recmsg ~= nil then
local receivefilepath = "]=]..outputFolder..[=["
local receivefile = io.open(receivefilepath, "w")
receivefile:write(recmsg)
print("wrote recmsg on '"..receivefilepath.."' file")
end
]=])
luaproc.wait()

print("Done calling '"..arg[0].."'")
```

C.2 Aplicativo do jogo "Hare and hounds"

– **findplayer.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local otherPlayerResultFilePath = arg[1]
local mytype = arg[2]
```

```
local session_topic = arg[3] or "game_01_test" --game ID to
trade messages with single player

if otherPlayerResultFilePath == nil or mytype == nil then
error("Must define which player you are. Usage: <output-file-path>
<'hare' or 'hounds'> <game-session-id>")
end

--To simplify, "hounds" players will send game requests and
"hare" players will try to receive game requests
--The hare will receive the "hounds" game session ID then
respond with his game ID and await an "ack" for MAX_TRIES
times
--If no response then will assume someone else got the game
or "hounds" player disconnected and "reset"
local isHounds = mytype == "hounds"
session_topic = (isHounds and session_topic.."_HOUNDS") or
session_topic.."_HARE"

print("My session topic is: "..session_topic)

local hound_code = [=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
table = require "table"
local socket = require "socket"
io = require("io")

local mygame_topic = "]=]..session_topic..[=["

--topic to receive messages from Hare
print("luaproc.mqregister:")
local topic = mygame_topic
print("    topic: "..tostring(topic))
retcode, errmsg = luaproc.mqregister( topic )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))
```

```
local game_found = false
while not game_found do
--send my game ID to some Hare player
topic = "Luaproc_Mqtt_HareGameApp_Topic_HARE"
print("luaproc.mqsend:")
local msg = mygame_topic
print("    topic: "..tostring(topic))
print("    msg: "..tostring(msg))
retcode, errmsg = luaproc.mqsend( topic, msg )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

--receive Hare game ID from my personal topic
print("luaproc.mqreceive:")
topic = mygame_topic
print("    topic: "..tostring(topic))
recmsg, errmsg = luaproc.mqreceive( topic, true )--async
receive
print("    recmsg: "..tostring(recmsg))
print("    errmsg: "..tostring(errmsg))

if recmsg == nil then --no response yet
socket.sleep(1)
else
game_found = true

print("    Opponent session topic: "..tostring(recmsg))

--send "ack" to hare personal topic
local topic = recmsg
print("luaproc.mqsend:")
local msg = mygame_topic --"lets_start"
print("    topic: "..tostring(topic))
print("    msg: "..tostring(msg))
retcode, errmsg = luaproc.mqsend( topic, msg )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

--save Hare game ID to file (so java side can recover it)
```

```
local receivefilepath = "]=]..otherPlayerResultFilePath..[=["
local receivefile = io.open(receivefilepath, "w")
receivefile:write(recmsg)
print("wrote '"..recmsg.."' on '"..receivefilepath.."'
file")

--Unregister to delete messages (clean the topic buffer)
print("luaproc.mqunregister:")
topic = mygame_topic
print("    topic: "..tostring(topic))
retcode, errmsg = luaproc.mqunregister( topic )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

--Re-register with clean buffer
print("luaproc.mqregister:")
topic = mygame_topic
print("    topic: "..tostring(topic))
retcode, errmsg = luaproc.mqregister( topic )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

end
end
]=]

local hare_code = [=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
table = require "table"
local socket = require "socket"
io = require("io")

local MAX_TRIES = 10
local mygame_topic = "]=]..session_topic..[=["

print("luaproc.mqregister:")
```

```
local topic = mygame_topic
print("    topic: "..tostring(topic))
retcode, errmsg = luaproc.mqregister( topic )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

print("luaproc.mqregister:")
local topic = "Luaproc_Mqtt_HareGameApp_Topic_HARE"
print("    topic: "..tostring(topic))
retcode, errmsg = luaproc.mqregister( topic )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

local game_found = false

while not game_found do
--receive Hounds game ID
print("luaproc.mqreceive:")
topic = "Luaproc_Mqtt_HareGameApp_Topic_HARE"
print("    topic: "..tostring(topic))
recmsg, errmsg = luaproc.mqreceive( topic )
print("    recmsg: "..tostring(recmsg))
print("    errmsg: "..tostring(errmsg))

--received his game ID...send him mine
print("luaproc.mqsend:")
topic = recmsg
local msg = mygame_topic
print("    topic: "..tostring(topic))
print("    msg: "..tostring(msg))
retcode, errmsg = luaproc.mqsend( topic, msg )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

--await for response, if take too long assume Hounds player
is no longer available
for i=1,MAX_TRIES do
print("luaproc.mqreceive:")
topic = mygame_topic
```

```
print("    topic: "..tostring(topic))
recmsg, errmsg = luaproc.mqreceive( topic, true )--async
receive
print("    recmsg: "..tostring(recmsg))
print("    errmsg: "..tostring(errmsg))

if recmsg == nil then --no response yet
socket.sleep(1)
else
game_found = true

print("    Opponent session topic: "..tostring(recmsg))

--save Hounds game ID to file (so java side can recover it)
local receivefilepath = "]=]..otherPlayerResultFilePath..[=["
local receivefile = io.open(receivefilepath, "w")
receivefile:write(recmsg)
print("wrote '"..recmsg.."' on '"..receivefilepath.."'
file")
end
end
end
]=]

local codeToRun = (isHounds and hound_code) or hare_code
luaproc.newproc(codeToRun)
luaproc.wait()

print("Done calling '"..arg[0].."'")
```

– **callAI.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local outputFilePath = arg[1]
local mytype = arg[2]
local board = arg[3]
local numworkers = arg[4] or "8"
```

```
if outputFilePath == nil or mytype == nil or board == nil
then
error("No output file or player type or board defined.
Usage: <output-file-path> <'hare' or 'hounds'> <board-as-string>")
end

local isHounds = mytype == "hounds"

print("outputFilePath: "..outputFilePath)
print("mytype: "..mytype)
print("board: "..board)
print("isHounds: "..tostring(isHounds))

local serial_code = [=[

local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
table = require "table"
local AI = require"alphabeta"
local UTILS = require"utils"
local io = require "io"

local isHoundPlayer = "]=]..tostring(isHounds)..[=["
local computer = (isHoundPlayer == "true" and UTILS.houndInt)
or UTILS.hareInt
AI.SetAiSide(computer)

local board = "]=]..board..[=["
local suggestedBoard = AI.best_move(board)

local receivefilepath = "]=]..outputFilePath..[=["
local receivefile = io.open(receivefilepath, "w")
receivefile:write(suggestedBoard)
print("wrote board on '"..receivefilepath.."' file")

]=]

luaproc.setnumworkers(numworkers)
```

```
local parallel_code = [=[

local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
table = require "table"
local AI = require"alphabeta"
local UTILS = require"utils"
local io = require "io"

local luaproc_code = [[

local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
string = require "string"
table = require "table"
local AI = require"alphabeta"
local UTILS = require"utils"

--receive board etc
local scoreChannel = luaproc.receive( "return_channel" )
local boardString = luaproc.receive( "board_channel" )
local myside = luaproc.receive( "myside_channel" )

local boardArray = UTILS.BoardStringToIntArray(boardString)
AI.SetAiSide(myside)

--compute score
local score = AI.alpha_beta(boardArray, 10, -9999, 9999,
false)

--return score
luaproc.send( scoreChannel, score )
]]

--'AI.best_move'(inside alphabeta.lua) but reworked to use
luaproc
function parallel_AI_best_move(currentBoardString)
local currentBoardArray = UTILS.BoardStringToIntArray(currentBoardString)
```

```
local children = AI.get_children(currentBoardArray,
AI.computer)-- Get All valid board moves/configurations

local max_index = 0
local max_value = -9999

luaproc.newchannel( "myside_channel" )
luaproc.newchannel( "board_channel" )
luaproc.newchannel( "return_channel" )
local scoreBaseString = "score_channel_"

for key, child_board in ipairs(children) do
luaproc.newproc(luaproc_code)

local scoreChannel = scoreBaseString..tostring(key)
luaproc.newchannel( scoreChannel )
luaproc.send( "return_channel", scoreChannel )

local child_board_string = UTILS.IntArrayToBoardString(child_board)
luaproc.send( "board_channel", child_board_string )

luaproc.send( "myside_channel", AI.computer )

end

for key, _ in ipairs(children) do
local scoreChannel = scoreBaseString..tostring(key)

local score = luaproc.receive( scoreChannel )
if score > max_value then
max_index = key
max_value = score
end
end

return UTILS.IntArrayToBoardString(children[max_index])
end

local isHoundPlayer = "]=]..tostring(isHounds)..[=["
```

```
local computer = (isHoundPlayer == "true" and UTILS.houndInt)
or UTILS.hareInt
AI.SetAiSide(computer)

local board = "]=]..board..[=["
local suggestedBoard = parallel_AI_best_move(board)

local receivefilepath = "]=]..outputFilePath..[=["
local receivefile = io.open(receivefilepath, "w")
receivefile:write(suggestedBoard)
print("wrote board on '"..receivefilepath.."' file")

]=]

--process to receive board and call AI for move suggestion
--luaproc.newproc(serial_code)
luaproc.newproc(parallel_code)

luaproc.wait()

print("Done calling '"..arg[0].."'")
```

– **alphabeta.lua**

```
require"string"
require"table"
local UTILS = require"utils"

--*--*--*--*--*--*--*--*--*--|#/\#|--*--*--*--*--*--*--*--*--*--
--*--*--*--*--*--*--*--*--*--|#\/#|--*--*--*--*--*--*--*--*--*--

local AI = {}

function AI.manhattan_distance(start, finish)
local md = 0
md = md + UTILS.GetPositionColumn(finish) - UTILS.GetPositionColumn(start)
md = md + UTILS.GetPositionRow(finish) - UTILS.GetPositionRow(start)
return md
end
```

```
function AI.cutoff(chosenBoardArray, maxplayer)
local cutoff_value = 0

local hare = UTILS.GetCurrentHarePosition(chosenBoardArray)
local hounds = UTILS.GetCurrentHoundPositionsAiSpecialOrder(chosenBoardArray)

local firstPos = 1--first position (aka row 2, column 1)
local lastPos = 11--first position (aka row 2, column 5)

local hareColumn = UTILS.GetPositionColumn(hare)
local hound1Column = UTILS.GetPositionColumn(hounds[0])
local hound2Column = UTILS.GetPositionColumn(hounds[1])
local hound3Column = UTILS.GetPositionColumn(hounds[2])

if (AI.computer == UTILS.houndInt and maxplayer) or
(AI.computer == UTILS.hareInt and (not maxplayer)) then
--computer on hound side
cutoff_value = (AI.manhattan_distance(hare, hounds[0]) +
AI.manhattan_distance(hare, hounds[1]) +
AI.manhattan_distance(hare, hounds[2])) / 3
if(hareColumn > hound1Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
firstPos) +
AI.manhattan_distance(hare, hounds[0]) - 1
end
if(hareColumn > hound2Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
firstPos) +
AI.manhattan_distance(hare, hounds[1]) - 1
end
if(hareColumn > hound3Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
firstPos) +
AI.manhattan_distance(hare, hounds[2]) - 1
end
if(hareColumn == hound1Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
hounds[0])
end
```

```
if(hareColumn == hound2Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
hounds[1])
end
if(hareColumn == hound3Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
hounds[2])
end
else
--computer on hare side
cutoff_value = AI.manhattan_distance(hare, firstPos) -
AI.manhattan_distance(hare, lastPos)
if(hareColumn > hound1Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
hounds[0]) - 1
end
if(hareColumn > hound2Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
hounds[1]) - 1
end
if(hareColumn > hound3Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
hounds[2]) - 1
end
if(hareColumn == hound1Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
hounds[0])
end
if(hareColumn == hound2Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
hounds[1])
end
if(hareColumn == hound3Column) then
cutoff_value = cutoff_value + AI.manhattan_distance(hare,
hounds[2])
end
end

return cutoff_value
```

```
end

function AI.get_children(boardArray, playerSide)
local children = {}

--Initialize 2 players on board
local hareP = UTILS.GetCurrentHarePosition(boardArray)
local houndsP = UTILS.GetCurrentHoundPositionsAiSpecialOrder(boardArray)

--Choose side for AI to move
if playerSide == UTILS.hareInt then
local validPositions = UTILS.GetValidPositions(hareP,
UTILS.hareInt, boardArray)
for i=UTILS.firstBoardIndex, UTILS.lastBoardIndex do
local sPos = UTILS.specialOrder[i]
if UTILS.contains(validPositions, sPos) then
local boardCopy = UTILS.CopyBoardArray(boardArray)
boardCopy[hareP - 1] = UTILS.emptyInt
boardCopy[sPos - 1] = UTILS.hareInt
table.insert(children, boardCopy)
end
end
else --player = houndInt
for i=UTILS.firstBoardIndex, UTILS.lastBoardIndex do
local sPos = UTILS.specialOrder[i]
for j=0,2 do
local houndPos = houndsP[j]
local validPositions = UTILS.GetValidPositions(houndPos,
UTILS.houndInt, boardArray)
if UTILS.contains(validPositions, sPos) then
local boardCopy = UTILS.CopyBoardArray(boardArray)
boardCopy[houndPos - 1] = UTILS.emptyInt
boardCopy[sPos - 1] = UTILS.houndInt
table.insert(children, boardCopy)
end
end
end
end
--return the possible children for current board
```

```
return children
end

function AI.alpha_beta(boardArray, depth, alpha, beta,
maxplayer)
local children

--Game over in board or reach depth limit
local winner = UTILS.ai_is_game_over(boardArray)
if winner ~= 0 or depth <= 0 then
if winner == 0 then
return AI.cutoff(boardArray, maxplayer)
elseif winner == UTILS.hareInt then
if maxplayer then
return -99
else
return 99
end
else--houndInt
return 99
end
end

if maxplayer then
children  = AI.get_children(boardArray, AI.computer)
for key, child_board in ipairs(children) do
local score = AI.alpha_beta(child_board, depth-1, alpha,
beta, (not maxplayer))
if score > alpha then
alpha = score
end
if alpha >= beta then
return alpha
end
end
return alpha
else
children = AI.get_children(boardArray, AI.human)
for key, child_board in ipairs(children) do
```

```
local score = AI.alpha_beta(child_board, depth-1, alpha,
beta, (not maxplayer))
if score < beta then
beta = score
end
if alpha >= beta then
return beta
end
end
return beta
end
end

function AI.best_move(currentBoardString)
local currentBoardArray = UTILS.BoardStringToIntArray(currentBoardString)
local children = AI.get_children(currentBoardArray,
AI.computer)-- Get All valid board moves/configurations

local max_index = 0
local max_value = -9999

local scores = {}
for key, child_board in ipairs(children) do
scores[key] = AI.alpha_beta(child_board, 10, -9999, 9999,
false)
end

for key, score in ipairs(scores) do
if score > max_value then
max_index = key
max_value = score
end
end
return UTILS.IntArrayToBoardString(children[max_index])
end

function AI.SetAiSide(computer)
AI.computer = computer
AI.human = UTILS.GetOppositeType(computer)
```

```
end

--default values
AI.computer = UTILS.hareInt
AI.human = UTILS.GetOppositeType(AI.computer)

return(AI)
```

– **utils.lua**

```
require"string"
require"table"

local UTILS = {}

--returns a new table that is t1 without the values of t2
function UTILS.removeAll(t1, t2)
local t = {}
for  k1, v1 in ipairs(t1) do
local v1InsideT2 = false
for k2, v2 in ipairs(t2) do
if v1 == v2 then
v1InsideT2 = true
end
end
if not v1InsideT2 then
table.insert(t, v1)
end
end
return t
end

--returns if the table contains the value
function UTILS.contains(t, value)
for  key, val in ipairs(t) do
if val == value then
return true
end
end
return false
```

```
end

--returns if table is empty
function UTILS.isEmpty(t)
if next(t) == nil then
--table is empty
return true
else
return false
end
end

UTILS.firstBoardIndex = 0
UTILS.lastBoardIndex = 10

UTILS.emptyInt = 0
UTILS.houndInt = 1
UTILS.hareInt = 2
UTILS.suggestedInt = 3

--special order that test positions with more degrees of
freedom first
--*          (02)2   (09)5   (05)8
--*  (01)1   (08)3   (04)6   (11)9   (07)11
--*          (03)4   (10)7   (06)10
UTILS.specialOrder = {[0]=1,[1]=2,[2]=4,[3]=6,[4]=8,[5]=10,[6]=11,[7]=3,[8]=5,

function UTILS.GetOppositeType(type)
if(type == UTILS.hareInt) then
return UTILS.houndInt
else
return UTILS.hareInt
end
end

function UTILS.GetPositionRow(position)
if position==2 or position==5 or position==8 then
return 1--1st row
elseif position==4 or position==7 or position==10 then
```

```
return 3--3rd column
else--2nd row (middle)
return 2
end
end

function UTILS.GetPositionColumn(position)
if position==1 then
return 1--1st column
elseif position==2 or position==3 or position==4 then
return 2--2nd column
elseif position==5 or position==6 or position==7 then
return 3--3rd column
elseif position==8 or position==9 or position==10 then
return 4--4th column
else--pos=11
return 5--5th (last) column
end
end

function UTILS.CopyBoardArray(board)
local copy = {}
for i=UTILS.firstBoardIndex, UTILS.lastBoardIndex do
copy[i] = board[i]
end
return copy
end

function UTILS.BoardStringToIntArray(boardString)
local boardArray = {}
local boardCount = 0
for char in boardString:gmatch"." do
boardArray[boardCount] = tonumber(char)
boardCount = boardCount + 1
end
return boardArray
end

function UTILS.IntArrayToBoardString(boardArray)
```

```
local boardString = ""
for i = UTILS.firstBoardIndex, UTILS.lastBoardIndex do
boardString = boardString..tostring(boardArray[i])
end
return boardString
end

function UTILS.GetOccupiedPositions(boardArray)
local isOccupied = {}
for i = UTILS.firstBoardIndex, UTILS.lastBoardIndex do
local pos = i+1
if boardArray[i] ~= 0 then
table.insert(isOccupied, pos)
end
end
return isOccupied
end

function UTILS.GetValidPositionsHare(position)
if position==1 then
return {2,3,4}
elseif position==2 then
return {1,3,5,6}
elseif position==3 then
return {1,2,4,6}
elseif position==4 then
return {1,3,6,7}
elseif position==5 then
return {2,6,8}
elseif position==6 then
return {2,3,4,5,7,8,9,10}
elseif position==7 then
return {4,6,10}
elseif position==8 then
return {5,6,9,11}
elseif position==9 then
return {6, 8, 10,11}
elseif position==10 then
return {6,7,9,11}
```

```
else--position=11
return {8,9,10}
end
end

function UTILS.GetValidPositionsHound(position)
if position==1 then
return {2,3,4}
elseif position==2 then
return {3,5,6}
elseif position==3 then
return {2,4,6}
elseif position==4 then
return {3,6,7}
elseif position==5 then
return {6,8}
elseif position==6 then
return {5,7,8,9,10}
elseif position==7 then
return {6,10}
elseif position==8 then
return {9,11}
elseif position==9 then
return {8, 10,11}
elseif position==10 then
return {9,11}
else--position=11
return {}
end
end

function UTILS.GetValidPositions(pos, type, boardArray)
local occupied = UTILS.GetOccupiedPositions(boardArray)
local validPositions
if type == UTILS.hareInt then
validPositions = UTILS.GetValidPositionsHare(pos)
else
validPositions = UTILS.GetValidPositionsHound(pos)
end
```

```
validPositions = UTILS.removeAll(validPositions, occupied)
return validPositions
end

function UTILS.ai_is_game_over(boardArray)
local hare = -1
local h1 = -1
local h2 = -1
local h3 = -1

--retrieve player positions
for i = UTILS.firstBoardIndex, UTILS.lastBoardIndex do
local pos = UTILS.specialOrder[i]
local value = boardArray[pos-1]
if value == UTILS.emptyInt then
--continue
else
if value == UTILS.hareInt then
hare = pos
end
if h1 == -1 then
h1 = pos
elseif h2 == -1 then
h2 = pos
else
h3 = pos
end
end
end

local hareColumn = UTILS.GetPositionColumn(hare)
local hound1Column = UTILS.GetPositionColumn(h1)
local hound2Column = UTILS.GetPositionColumn(h2)
local hound3Column = UTILS.GetPositionColumn(h3)

if hareColumn <= hound1Column and hareColumn <= hound2Column
and hareColumn <= hound3Column then
return UTILS.hareInt--hare wins
end
```

```
local hareValidMoves = UTILS.GetValidPositions(hare,
UTILS.hareInt, boardArray)
if not UTILS.isEmpty(hareValidMoves) then --hare still has
moves
return 0--no winner
end

return UTILS.houndInt
end

function UTILS.GetCurrentHarePosition(boardArray)
local pos
for i=UTILS.firstBoardIndex, UTILS.lastBoardIndex do
pos = i+1
if boardArray[i] == UTILS.hareInt then
return pos
end
end
return pos
end

function UTILS.GetCurrentHoundPositionsAiSpecialOrder(boardArray)
local houndPositions = {}
local hountCount = 0
for i=UTILS.firstBoardIndex, UTILS.lastBoardIndex do
local pos = UTILS.specialOrder[i]
if boardArray[pos-1] == UTILS.houndInt then
houndPositions[hountCount] = pos
hountCount = hountCount + 1
end
end
return houndPositions
end

return(UTILS)

--*--*--*--*--*--*--*--*--*--*--|#/\#|--*--*--*--*--*--*--*--*--*--
--*--*--*--*--*--*--*--*--*--*--|#\/#|--*--*--*--*--*--*--*--*--*--
```

C.3
Aplicativo de compartilhamento de fotos

– **serialize&send.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local selectedPicsSepBySemicolon = arg[1]
local numworkers = arg[2] or 4
local host = arg[3] or "37.187.106.16" --http://test.mosquitto.org
local port = arg[4] or "1883"
local client_id = arg[5]

if selectedPicsSepBySemicolon == nil then
error("No pictures were received by script.\nUsage:
<pictures> [<num-workers> <host> <port> <id>]")
end

print("selectedPicsSepBySemicolon: "..selectedPicsSepBySemicolon)

luaproc.setnumworkers(numworkers)
print("Luaproc worker count: "..tostring(luaproc.getnumworkers()))

print("host: "..host)
print("port: "..port)
print("client_id: "..tostring(client_id))

local serial_code = [=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
b64FileUtils = require "image_tests.b64FileUtils"
string = require "string"
table = require "table"
io = require("io")

config = {}
config.host = ']=].. tostring(host) ..[=['
```

```
config.port = ']=].. tostring(port) .. [=['
config.client_id = (']=].. tostring(client_id) ..[=[' ~=
'nil' and ']=].. tostring(client_id) .. [=[' or nil)

print("luaproc.mqconnect:")
retcode, errmsg = luaproc.mqconnect( config )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

function string:split(sep)
local sep, fields = sep or ":", {}
local pattern = string.format("([^%s]+)", sep)
self:gsub(pattern, function(c) fields[#fields+1] = c end)
return fields
end

local selectedPicsSepBySemicolon = ']=].. selectedPicsSepBySemicolon
..[=['
local pictures = selectedPicsSepBySemicolon:split(";")

for index, value in ipairs(pictures) do
--serialize picture
print("photoPath: "..value)
msg = b64FileUtils.FilePathContentsToB64(value)

--send picture
print("luaproc.mqsend:")
topic = "topic_Mqtt_PicShareApp_jpg"
retcode, errmsg = luaproc.mqsend( topic, msg )
print("    topic: "..tostring(topic))
print("    msg(Enc64): "..tostring(msg))
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))
end
]=]

local parallel_code = [=[

lualogcatlib = require "lualogcatlib"
```

```
print = lualogcatlib.print
string = require "string"
table = require "table"
io = require("io")

config = {}
config.host = ']=].. tostring(host) ..[=['
config.port = ']=].. tostring(port) .. [=['
config.client_id = (']=].. tostring(client_id) ..[=[' ~=
'nil' and ']=].. tostring(client_id) .. [=[' or nil)

print("luaproc.mqconnect:")
retcode, errmsg = luaproc.mqconnect( config )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

function string:split(sep)
local sep, fields = sep or ":", {}
local pattern = string.format("([^%s]+)", sep)
self:gsub(pattern, function(c) fields[#fields+1] = c end)
return fields
end

local selectedPicsSepBySemicolon = ']=].. selectedPicsSepBySemicolon
..[=['
local pictures = selectedPicsSepBySemicolon:split(";")

luaproc.newchannel( "finished_send_channel" )
for index, value in ipairs(pictures) do

luaproc.newproc( [[
lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
b64FileUtils = require "image_tests.b64FileUtils"

--serialize picture
local photo = ']].. value ..[[']
print("photoPath: "..photo)
local msgEnc64 = b64FileUtils.FilePathContentsToB64(photo)
```

```
--send picture
print("luaproc.mqsend:")
local topic = "topic_Mqtt_PicShareApp_jpg"
retcode, errmsg = luaproc.mqsend( topic, msgEnc64 )
print("    topic: "..tostring(topic))
print("    msg(Enc64): "..tostring(msgEnc64))
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

luaproc.send( "finished_send_channel", "done" )
]])

end

--wait for child processes to finish
for index, value in ipairs(pictures) do
luaproc.receive( "finished_send_channel" )
end
]=]

--process to connect and send selected images
--luaproc.newproc(serial_code)
luaproc.newproc(parallel_code)
luaproc.wait()

--process to send finish message and disconnect
luaproc.newproc( [=[

--send "finished" msg
print("luaproc.mqsend:")
topic = "topic_Mqtt_PicShareApp_jpg"
msg = "ok_done"
retcode, errmsg = luaproc.mqsend( topic, msg )
print("    topic: "..tostring(topic))
print("    msg: "..tostring(msg))
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

print("luaproc.mqdisconnect:")
```

```
retcode, errmsg = luaproc.mqdisconnect( config )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))
]=])
luaproc.wait()

print("Done calling '"..arg[0].."'")
```

– **receive&reconstruct.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local outputFolder = arg[1]
local numworkers = arg[2] or 4
local host = arg[3] or "37.187.106.16" --http://test.mosquitto.org
local port = arg[4] or "1883"
local client_id = arg[5]

if outputFolder == nil then
error("No output folder defines.\nUsage: <output-folder>
[<num-workers> <host> <port> <id>]")
end

luaproc.setnumworkers(numworkers)
print("Luaproc worker count: "..tostring(luaproc.getnumworkers()))

print("host: "..host)
print("port: "..port)
print("client_id: "..tostring(client_id))

--process to connect and receive images in selected folder
luaproc.newproc([=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
b64FileUtils = require "image_tests.b64FileUtils"
string = require "string"
table = require "table"
```

```
io = require("io")

config = {}
config.host = ']=].. tostring(host) ..[=['
config.port = ']=].. tostring(port) .. [=['
config.client_id = (']=].. tostring(client_id) ..[=[' ~=
'nil' and ']=].. tostring(client_id) .. [=[' or nil)

print("luaproc.mqconnect:")
retcode, errmsg = luaproc.mqconnect( config )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

print("luaproc.mqregister:")
topic = "topic_Mqtt_PicShareApp_jpg"
retcode, errmsg = luaproc.mqregister( topic )
print("    topic: "..tostring(topic))
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

local picIndex = 0
while true do
--Receive picture
print("luaproc.mqreceive:")
recmsg, errmsg = luaproc.mqreceive( topic )
print("    topic: "..tostring(topic))
print("    recmsg(Enc64): "..tostring(recmsg))
print("    errmsg: "..tostring(errmsg))

if recmsg == nil then
print("ERROR receiving pics...")
break
end

if recmsg == "ok_done" then
print("Finished receiving pics...")
break
end
```

```
--Save (reconstruct) picture locally
photoFolder = ']=]..outputFolder..[=['
print("photoFolder: "..photoFolder)
outfile = photoFolder .. "/temp_copy_"..picIndex..".jpg"
print("outfile: "..outfile)

b64FileUtils.WriteB64ToFilePath(recmsg, outfile)
print("outfile: copy complete")

picIndex = picIndex + 1
end

print("luaproc.mqdisconnect:")
retcode, errmsg = luaproc.mqdisconnect( config )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))
]=])
luaproc.wait()

print("Done calling '"..arg[0].."'")
```

C.4
Aplicativo de busca e reconhecimento facial

– **sendpicforsearch.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"
require"string"
require"table"

local refPhotoPath = arg[1]
local outputPhotoPath = arg[2]
local host = arg[3] or "37.187.106.16" --http://test.mosquitto.org
local port = arg[4] or "1883"
local client_id = arg[5]

if refPhotoPath == nil then
error("Usage: <ref_image_path>")
```

```
end

print("refPhotoPath: "..refPhotoPath)

--process to connect, register and send the ref image for
searching
luaproc.newproc([[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
b64FileUtils = require "image_tests.b64FileUtils"

config = {}
config.host = ']].. tostring(host) ..[['
config.port = ']].. tostring(port) .. [['
config.client_id = (']].. tostring(client_id) ..[[' ~=
'nil' and ']].. tostring(client_id) .. [[' or nil)

print("luaproc.mqconnect:")
retcode, errmsg = luaproc.mqconnect( config )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

print("luaproc.mqregister:")
retcode, errmsg = luaproc.mqregister( "onlinetopic" )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

local anyUserOnline = false
while not anyUserOnline do
recmsg, errmsg = luaproc.mqreceive("onlinetopic", true)
if recmsg ~= nil then --someone is online
anyUserOnline = true
end
end

--serialize picture
photoPath = ']]..refPhotoPath..[['
print("photoPath: "..photoPath)
```

```
msg = b64FileUtils.FilePathContentsToB64(photoPath)

--send picture
print("luaproc.mqsend:")
topic = "topic_jpg"
retcode, errmsg = luaproc.mqsend( topic, msg )
print("    topic: "..tostring(topic))
print("    msg(Enc64): "..tostring(msg))
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

]])
luaproc.wait()

-- process to recieve response
luaproc.newproc([=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
b64FileUtils = require "image_tests.b64FileUtils"
string = require "string"
table = require "table"

print("luaproc.mqregister:")
topic = "found_jpg_in_gallery"
retcode, errmsg = luaproc.mqregister( topic )
print("    topic: "..tostring(topic))
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

print("luaproc.mqreceive:")
recmsg, errmsg = luaproc.mqreceive( topic )
print("    topic: "..tostring(topic))
print("    recmsg(Enc64): "..tostring(recmsg))
print("    errmsg: "..tostring(errmsg))

if recmsg ~= "not_found" then

--Save (reconstruct) picture locally
```

```
local outpusPhotoPath = ']=]..outputPhotoPath..[=['
print("outpusPhotoPath: "..outpusPhotoPath)
outfile = outpusPhotoPath .. "/temp_found_copy.jpg"
print("outfile: "..outfile)
b64FileUtils.WriteB64ToFilePath(recmsg, outfile)
print("outfile: copy complete")

print("Person was (possibly) found!")

end
]=])
luaproc.wait()

--process to disconnect (search finished)
luaproc.newproc([[
lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print

print("luaproc.mqdisconnect:")
retcode, errmsg = luaproc.mqdisconnect( )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))
]])
luaproc.wait()

print("Done calling '"..arg[0].."'")
```


– **searchgallery.lua**

```
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luaproc = require "luaproc"

local wholeGallerySepBySemicolon = arg[1]
local modelsFolderPath = arg[2]
local numworkers = arg[3] or 4
local host = arg[4] or "37.187.106.16" --http://test.mosquitto.org
local port = arg[5] or "1883"
local client_id = arg[6]
```

```
if wholeGallerySepBySemicolon == nil or modelsFolderPath ==
nil then
error("Usage: <whole_gallery_image_paths> <models_folder_path>")
end

print("wholeGallerySepBySemicolon: "..wholeGallerySepBySemicolon)
print("modelsFolderPath: "..modelsFolderPath)

luaproc.setnumworkers(numworkers)
print("Luaproc worker count: "..tostring(luaproc.getnumworkers()))

--process to connect, register and recieve ref image to
search in gallery
luaproc.newproc([=[

lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
b64FileUtils = require "image_tests.b64FileUtils"
string = require "string"
table = require "table"
io = require("io")

config = {}
config.host = ']=].. tostring(host) ..[=['
config.port = ']=].. tostring(port) .. [=['
config.client_id = (']=].. tostring(client_id) ..[=[' ~=
'nil' and ']=].. tostring(client_id) .. [=[' or nil)

print("luaproc.mqconnect:")
retcode, errmsg = luaproc.mqconnect( config )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

print("luaproc.mqregister:")
topic = "topic_jpg"
retcode, errmsg = luaproc.mqregister( topic )
print("    topic: "..tostring(topic))
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))
```

```
--notify i'm online
print("luaproc.mqsend:")
retcode, errmsg = luaproc.mqsend( "onlinetopic", "connected"
)
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

--Receive picture
print("luaproc.mqreceive:")
topic = "topic_jpg"
recmsg, errmsg = luaproc.mqreceive( topic )
print("    topic: "..tostring(topic))
print("    recmsg(Enc64): "..tostring(recmsg))
print("    errmsg: "..tostring(errmsg))

--Save (reconstruct) picture locally
photoFolder = ']=]..modelsFolderPath..[=['
print("photoFolder: "..photoFolder)
outfile = photoFolder .. "/temp_copy.jpg"
print("outfile: "..outfile)

b64FileUtils.WriteB64ToFilePath(recmsg, outfile)
print("outfile: copy complete")

----------------------SEARCH----------------------

local modelsFolderPath = photoFolder
local wholeGallerySepBySemicolon = ']=]..wholeGallerySepBySemicolon..[=['

function string:split(sep)
local sep, fields = sep or ":", {}
local pattern = string.format("([^%s]+)", sep)
self:gsub(pattern, function(c) fields[#fields+1] = c end)
return fields
end

local gallery = wholeGallerySepBySemicolon:split(";")

luaproc.newchannel( "finished_channel" )
```

```
for index, value in ipairs(gallery) do
luaproc.newproc([[
local lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print
local luafacefinderlib = require "luafacefinderlib"
local io = require "io"
b64FileUtils = require "image_tests.b64FileUtils"

local refPhotoPath = ']]..outfile..[['
local value = ']]..value..[['
local modelsFolderPath = ']]..modelsFolderPath..[['
local index = ']]..index..[['

print("Calling face_find for gallery["..tostring(index).."]='"..tostring(value)
local retbool, errmsg = luafacefinderlib.find_face(refPhotoPath,
value, modelsFolderPath)
print("     retbool(was face found): "..tostring(retbool))
print("     errmsg: "..tostring(errmsg))
if retbool == true then
print("Person was FOUND in picture("..tostring(value)..")!!!
LP=["..index.."]")

local enc64Photo = b64FileUtils.FilePathContentsToB64(value)
local topic = "found_jpg_in_gallery"
print("luaproc.mqsend:")
print("    topic: "..tostring(topic))
print("    msg: "..tostring(enc64Photo))
local retcode, errmsg = luaproc.mqsend( topic, enc64Photo )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))

local resultfilepath = modelsFolderPath .. "/temp_found_result.txt"
local resultfile = io.open(resultfilepath, "w")
resultfile:write("true")
print("wrote on 'temp_found_result.txt' file: true")
else
print("Person is NOT in picture("..tostring(value)..")!!!
LP=["..index.."]")
end
```

```
luaproc.send( "finished_channel", "im_done" )
]])
end
-------------------END SEARCH-------------------

for index, value in ipairs(gallery) do
luaproc.receive( "finished_channel" )
end

]=])
luaproc.wait()

--create file to signal the search has ended (in append
mode as not to overwrite if existing)
local resultfilepath = modelsFolderPath .. "/temp_found_result.txt"
io.open(resultfilepath, "a")
local resFile = io.open(resultfilepath, "r")
local fileText = resFile:read("*a")
print("fileText: "..tostring(fileText))
if fileText ~= "true" then --Not in folder
luaproc.newproc([[
lualogcatlib = require "lualogcatlib"
print = lualogcatlib.print

print("luaproc.mqsend (for 'Not in folder'):")
local msg = "not_found"
local topic = "found_jpg_in_gallery"
local retcode, errmsg = luaproc.mqsend( topic, msg )
print("    topic: "..tostring(topic))
print("    msg: "..tostring(msg))
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))
]])
luaproc.wait()
end

--process to disconnect (search finished)
luaproc.newproc([[
lualogcatlib = require "lualogcatlib"
```

```
print = lualogcatlib.print

print("luaproc.mqdisconnect:")
retcode, errmsg = luaproc.mqdisconnect( )
print("    retcode: "..tostring(retcode))
print("    errmsg: "..tostring(errmsg))
]])
luaproc.wait()

print("Done calling '"..arg[0].."'")
```