mHot

Mobile Here or There - A framework for runtime offloading of computation from mobile devices to the cloud.

by

William Joseph Gaudet

B.ScEng, The University of New Brunswick, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTERS OF APPLIED SCIENCE

in

The Faculty of Electrical and Computer Engineering

(Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

June 2011

Abstract

Today the average north american carries a computer several orders of magnitude more powerful than the computer used to guide the Apollo space craft to the moon and safely back. However, with this dramatic increase in both ubiquity and power of personal mobile computing, has come a host of applications that require significantly more computational power than is available in today's smart phones. Additionally, battery life of mobile devices is still a significantly limiting factor when doing large quantities of computation on mobile platforms.

With these factors in mind this research investigates the gains in performance, quality of computation, and battery life which can be made possible by real time offloading of computation from a mobile device to a server platform.

Furthermore, it details the design and implementation of a framework for offloading computation from mobile devices which has at its core the goal of limiting the cognitive overhead for the developer to conduct such an offload.

Table of Contents

Appendices

List of Tables

List of Figures

List of Programs

Acknowledgements

Dedication

Chapter 1

Introduction

Chapter 2

Background

Chapter 3

Literature Review

Chapter 4

Design and Modelling

4.1 Overview

4.2 A Framework for Data Migration

4.2.1 Dynamic Proxies

The primary mechanism for abstracting the decision of where to execute a method (in the cloud or on the mobile device) is the Java Dynamic Proxy class [citation]. The dynamic proxy class enables runtime proxing of any object provided it implements an interface. A dynamic proxy object then implements the interface of the object to be proxied at runtime, and then routes all method calls through its invoke method. It is in this invoke method that the decision to either package the object graph and ship it to the server for execution or execute locally is made.

The dynamic proxy class does have some shortcomings, it requires developers to create interfaces for the methods they want to to be remotable, additionally the developer must explicitly create the proxy objects for use in their code, and finally dynamic proxies do incur a performance overhead. Simple running time experiments show that the average time cost of a regular Java method invocation is approximately 60ns while the average time cost of the same method invocation done through a dynamic proxy object is around 5ms. However with these costs in mind a developer can make architectural decisions to help mitigate these performance overheads. The shortcomings with respect to the use of interfaces is actually viewed as a positive as it forces the developer towards best practices of programming to interfaces and not implementation. Finally steps have been taken to mitigate developer pain with regards to creation and use of dynamic proxies, by way of the Java Generics system every object which descends from the AbstractRemoteObject has a factory method which will wrap the object in a proxy.

Remote Method Invocation

If mHot determines that a method should be executed in the cloud rather than locally on the mobile device, it must conduct a remote method invocation. While it takes its name from the Java RMI system, the remote method invocation object is only related to RMI in the fact that it allows for a developer to remotely invoke methods on an object. When mHot was first designed RMI was investigated as a potential solution to the problem of object transmission and method invocation, however the lack of support for native RMI in the Android platform was seen as an insurmountable barrier. As such, a custom remote invocation methodology was devised with a goal in mind to minimize both the amount of computation required to marshal the objects between the compute platforms and limit the amount of data that would be transmitted over the wire—this is particularly important due to battery life constraints and the substantial energy costs of the high bandwidth radios (3g / WiFi).

A single remote invocation is a collaboration between three objects, the target remoteable object, the argument array, and the result. Each object plays a particular role in the method invocation process, and together make up the composition of a RemoteInvocationMethod object. The target remoteable object is the object against which the method will be invoked, the argument array contains facilities for serializing and deserializing the arguments the method takes—these objects must be either primitive data types, or implement RemoteableObject otherwise an exception will be thrown—, finally the result is container for the returned result of the method—presently the method should not mutate the target object's state as mHot has no facilities for returning the object once it's been shipped to the remote server, thus the developer must currently take that into account when designing their apis.

Data Serialization

When a dynamic proxy object receives a method invocation that has been tagged as remotable, if the framework has determined that the request should be executed remotely, the proxied object must first be serialized before it can be shipped to the server. The proxied object can take on one of two forms: a simple object composed only of primitive data types (integers, floating point numbers, booleans etc) or a complex object graph composed of zero or more primitive types and one or more references to other objects. These other objects may in turn also contain references to additional objects

both complex and simple as well as cyclical references between objects.

Given the cost associated with object serialization both in terms of computation (due to memory allocation and object graph graph traversal) and in terms of communications bandwidth the task of the object serializer becomes two fold. Most importantly it must correctly serialize the object graph preserving any and all object references (including circular ones); additionally the serializer must attempt to minimize the size of the transmitted object graph.

This is primarily done by the encapsulation of an object's fields in container object which implements the RemotableField¡T¿ interface. [CODE LISTING] The interface exposes several values about the encapsulated object all of which are required to correctly and efficiently (in both space and time) serialize the object graph, the more relevant methods to the serialization process are the following:

- int size() - Returns the size in bytes required to serialize this field (in the implemented case of a RemoteFieldObject this is 4 bytes for the reference to the object)
- ¡T¿ get() - Returns the object which is encapsulated in this field, in the case of a primitive value it is wrapped in it's Java object container type (Integer for int) etc, due to performance issue with the autoboxing system [CITATION NEEDED] where T is either a primitive data type or an implementation of RemoteObject
- boolean isDirty() - Returns true if this object has had it's value set since initialization or last serialization - this is used to minimize the amount of data that must be transmitted to the remote server.
- void Serialize(ByteBuffer) - Serializes the field to the provided ByteBuffer
- void Deserialize(ByteBuffer) - Deserializes the field from the provided ByteBuffer
- FieldVisitor acceptVisitor(FieldVisitor) - Accepts the field visitors which are used to conduct a variety of tasks in the graph serialization process.

The Serialization Process

The object serialization process is a modified depth first search. In effect the process visits every node in the graph twice, once to compute the overall

size required to encode the graph as a binary representation, and a second time to perform the serialization it self. [This is a potential future work situation, some of this could probably be improved].

When a object in the graph is visited by the size computation pass, each of the object's remoteable fields are interrogated for their serialization size—It should be noted that the fields for each object class are stored in a memoization cache to prevent recalculation of the classes' remoteable fields through the use of the Java reflection APIs which are not very performant [Maybe this should be a footnote probably needs a citation]. If the field encapsulates a primitive field it simply looks up the serialization cost for that field which is it's data size in addition to the field reference, however, if the field encapsulates a remotable object it either adds 8 bytes if it has been visited once before—the byte cost of referencing an object which has already been visited—or it adds 8 bytes plus the serialization cost of that object, which is computed immediately—A complete list of default data type serialization costs is shown in 1. The size cost of an object is computed by adding the length of the fully qualified class name (eq: com.someCompany.somePackage.someClass) to the size of the serialized data including referenced object sizes to 8 bytes—4 for the entire schema size, 4 for the first object size.

On the second traversal of the object graph each object is serialized by calling each of it's field's serialize method. This method creates a binary representation of the data which can later be decoded to reproduce the same values—The put and get methods on the Java ByteBuffer class handle the serialization of primitive data types into their byte representations. A primitive field is serialized as a tuple of the hash value of the string representation of it's name and it's value. The hash value is used to reduce the size and complexity of the encoding. Given that a string of arbitrary length will have a unique hash code, this means that every field can be key value encoded with an integer for a key. This means the field can be encoded as the length of the field representation (including the length of the length), the hash code of the field name, and the value of the field. ObjectFields—that is fields which reference remote objects—are represented as a length, field name hash code, and the hash code of the object they reference, the object it self is added to a queue of objects to be serialized. The serialized object representation after the serialization has completed is shown in Figure 4.1

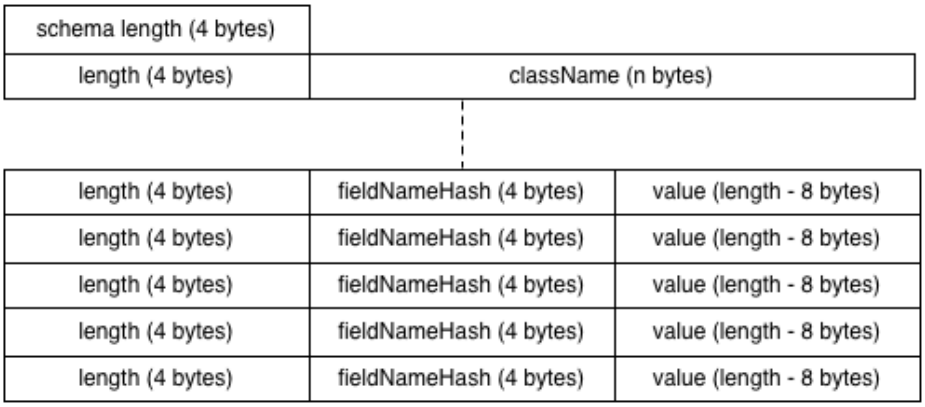

Figure 4.1: The data outline for the serialized data

4.2.2 Result Serialization

4.3 Programmer Responsibilities

4.3.1 Annotation

4.3.2 Parallelization

4.4 The Model

4.5 Exercising the Framework

Chapter 5

Experimental Results

Chapter 6

Conclusion

6.1 Summary of Contribution

6.2 Future Work

First Appendix

Type	Data Size	Effect On Schema
byte or Byte	1	5
char or Character	2	6
short or Short	2	6
float, Float, int, or Integer	4	8
double, Double, long, or Long	8	12
byte[length]	1 * length	1 * length + 4
char[length]	2 * length	2 * length + 4
float[length] or int[length]	4 * length	4 * length + 4
double[length] or long[length]	8 * length	8* length + 4
object (not visited)	object.size()	object.size() + 8
object (visited)	object.size()	8

Table 1: Table of field sizes

Second Appendix

Here is the second appendix.