UNIVERSITY OF BIRMINGHAM

School of Computer Science

BSc Mathematics and Computer Science with Industrial Year

Universal Method Invocation for Android

Author:
Kelsey McKenna
Student ID: 1412409

Supervisor:
Dr. Dan Ghica

9th April 2018

Abstract

By virtue of being battery-powered and portable, mobile devices suffer from limited battery capacity and computing power. As a result, mobile applications may have processing requirements that are prohibitively high for devices not connected to a power source. In this project, we present a library for the Android platform which can be used to write mobile applications that leverage the processing power of remote systems over the network. This project is an extension of Mokapot, a library for distributed computing which supports stateful computation offloading, where we make the necessary changes to port the JVM-targeted project to the Android runtime. Involvement of remote state presents an additional challenge, particularly for mobile devices: ensuring at-most-once execution. During everyday usage, a mobile device may suffer from network instability. For example, the device may drift into a region not covered by the cellular network, the device may discover a Wi-Fi network and use this instead of the cellular network, or the user may enable flight mode, restricting all network communications. To ensure at-most-once execution, this project implements a failure detection mechanism and a handshake-based recovery protocol, which are necessary to resume execution in those cases where the TCP protocol cannot ensure delivery/acknowledgement of messages. We will show that the solution is (a) highly transparent, enabling swift and easy application development, (b) resilient when encountering network instability and requires no intervention from the application developer, and (c) capable of improving performance and reducing energy consumption of some applications.

Keywords: computation offloading, mobile cloud computing, distributed computing, network recovery.

I would like to thank Dr Dan Ghica for his supervision and Alex Smith, Mokapot's main author, for his assistance throughout the project. All software for this project can be found at `https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2017/kjm409`.

Contents

1 Introduction

The goal of this project was to develop a library that could be used by Android application developers to quickly write distributed applications, especially those that could leverage computation offloading to achieve improved performance and lower energy requirements. We envision a developer who wishes to use a lightweight, transparent, and resilient library to develop distributed mobile applications, which requires only minimal application configuration. To achieve this aim, Mokapot, an existing library written in Java for distributed computation, will provide a perfect foundation. Java's Remote Method Invocation (RMI) system [68] is conceptually similar to Mokapot, but it is not available on the Android platform, so the use of a third-party library, such as Mokapot, is necessary.

The motivation for bringing this functionality to a mobile platform is based on the idea that computation offloading can result in improved performance, both in terms of speed and energy savings [47]. This could open up the possibility for new types of applications or remove impediments from existing services, ones whose performance is hindered by the device itself, or ones whose computation requirements are prohibitively high for a battery-powered device [29]. For example, applications which perform large amounts of image processing, machine learning, or matrix manipulations, etc., have the potential to save significant amounts of battery power if they offload the computations [57]. Similarly, if some specialised hardware is necessary or recommended for swift execution, e.g. machine learning with GPUs [64], applications may benefit from offloading computations to a machine which supports this hardware.

The potential of mobile cloud computing has been discussed in a number of projects, including CloneCloud [16], which demonstrated significant energy savings for a number of applications. Others have observed 45% cost saving in battery usage for complex computations [29]. Some libraries are intrusive, such as Cuckoo which requires much intervention at development time but automates much of the distribution of computation [43], while others automate virtually every aspect of the distribution process but do not give developers fine-grained control [16, 23, 10]. JavaParty provides developers with a similar level of abstraction as Mokapot and targets clusters of workstations, but requires a pre-

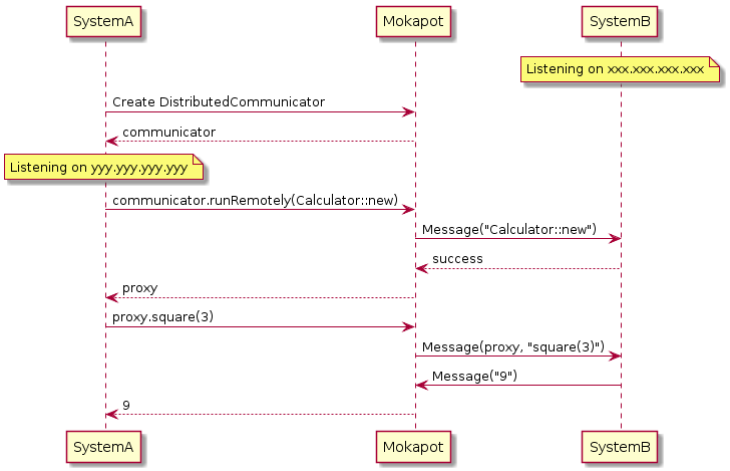

Figure 1: Illustrates Mokapot's communication between two systems, where one system creates a `Calculator` object remotely then invokes a method on the long reference.

processor [53], which introduces complexity in the development process (IDEs will not recognise the language extensions); the advantages of a transparent library over RMI and socket programming are also considered. The recovery protocol implemented in this takes inspiration from algorithms for optimistic recovery in distributed systems [65, 41, 45, 12], which use message sequence numbers, message logging and replaying. Many of these systems use application snapshots, which require extensions to the JVM, which is not desired for this project, and is unsuitable for the Android platform, so this project cannot use an identical approach.

This project is based on a distributed computation library, Mokapot, actively developed at the University of Birmingham under an EPSRC-funded research project. Mokapot allows an application to execute code on other systems. It implements long references, which are references to objects which reside on another system. Method invocations on long references are routed by Mokapot to the actual object, and the results are returned to the caller as short or long references, depending on certain properties of the object, e.g. immutability. For example, to create a `java.util.ArrayList` remotely, one could write

```
List<String> remoteList = DistributedCommunicator.runRemotely(
        () -> new ArrayList<>(), server);
```

The `remoteList` object can be passed around as if it were a regular "short" reference, but method invocations will involve network traffic to the system denoted by `server` in the above example. Figure 1 illustrates an example of how two Mokapot systems would communicate.

There is one main thread in a Java application running in the JVM, but there are n main threads in a distributed system with n systems running Mokapot. Additionally, in a JVM there is a single incarnation of a static field [15], but Mokapot allows access to location-based static fields, where an application can access each system's incarnation of the static field. Therefore, we can consider Mokapot to be an extension of Java.

The key features and changes made in this project are as follows:

- Porting Mokapot, which targets the Java Virtual Machine, to be suitable for deployment in the Android runtime.
- A failure detection mechanism, identifying failures which may not be surfaced as one might expect.
- A recovery protocol to be actioned when network errors are detected or suspected.

In order for a developer to consider this project a feasible option for developing distributed mobile applications, the problem of network instability must be addressed. Since Mokapot (and therefore this project) supports the creation and maintenance of remote objects with state, we must enforce an at-most-once execution model, otherwise remote state may be manipulated twice even though the application code specified only one such manipulation. Mokapot uses the Transmission Control Protocol (TCP) to communicate with other systems. TCP enables highly reliable host-to-host communications, and a connection is based on a pair of sockets, one on each system [54]. However, the guarantees of TCP only apply to its active connections, i.e. a pair of connected sockets. If the connection between the sockets is "broken", i.e. one of the sockets is closed, the IP address of the host system changes, etc., we cannot create a new connection and ask TCP to ensure delivery on this new connection. This requires application-level intervention to ensure that, upon reconnection, undelivered messages are resent and already-delivered messages are not resent. Furthermore, when a TCP connection is broken, the error cannot always be surfaced to the application immediately. For a responsive mobile application, long timeouts are unacceptable and so this project implements a strategy to discover potential failures and recover quickly.

In this report we will discuss related projects in the literature, giving more background to the project. We include an analysis of the problem and a specification of the functional and non-functional aspects of potential solutions. Next we discuss the structure of the chosen solution, including the algorithms used, its implementation, how it was tested, and the project management tactics used to bring the project to completion. Finally, we evaluate the final product, in particular we discuss the suitability of the project for the Android platform and future work that could be undertaken as an extension of this project.

2 Literature Review

Offloading computation and storage of objects to another system has the potential to improve performance, particularly on mobile devices. By allowing the device to simply forward method calls to another system, and then wait for a response, the device is freed from the burden of performing the computation, at the expense of additional network traffic. Offloading is beneficial when large amounts of computation are needed with relatively small amounts of communication [46]. An important aspect of any potential solution is the effort required to implement — or convert — an application to leverage computation offloading. If the library requires the programmer to change every line of code that needs to be run remotely, then it will be difficult to justify the effort required. Some existing technologies for distributed computation, such as Java RMI (Remote Method Invocation) [68], force the programmer to handle a `RemoteException` every time a method is called remotely. Others require fewer boilerplate changes, but still need input from the programmer in the form of annotations [23]. Ideally, the library should give the programmer the choice of granularity: the library can make decisions about how and where to run code, or the programmer can declare everything explicitly. Furthermore, depending on the particular use case of the application or library, the programmer may want support for callbacks, remote manipulation of state, consistent semantics, and certain levels of security.

This project adapts Mokapot for use on the Android operating system, including improving network resiliency and adding mobile-specific functionality. Importantly, the integration of Mokapot with the Android platform introduces support for certain use cases not available in existing technologies, such as CORBA [21], Java RMI or Web APIs. Research done in this area has found approaches to solving some of these problems, but many of the solutions discussed in this review do not manage remote state.

2.1 Related work

There are many existing technologies for facilitating distributed computing. Some technologies support collections of heterogeneous systems, such as CORBA (Common Object Request Broker Architecture) [21] and RESTful web APIs [11], while others make platform-specific assumptions about the communicating systems, such as Java RMI (Remote Method Invocation) [68], which requires both sides of the communication to be running inside a JVM. Web APIs provide a simple method for transferring data and indicating actions, e.g. via HTTP methods, but this approach is limited since all type information is lost.

As an overview of how Java RMI works [6, 66], a server maintains a registry of references to objects accessible for remote method invocations. Clients can obtain references to these objects and then invoke methods over the network. The

communication between client and server is transparent to the user, i.e. method invocations on a "remote" object look like invocations on a regular object. The "remote" object on the client side is called a *stub* and is responsible for several things, including marshalling arguments, transmitting data to the remote system, and unmarshalling the response. The actual object resides on the server side, but has a *skeleton* in front of it, which is responsible for unmarshalling the arguments sent by the client, invoking the method on the actual object, and returning the result. Both systems can act as servers, allowing for callbacks and other features. Some setup is required by the user, including obtaining references to remote objects and defining interfaces extending `java.rmi.Remote`, where each method in the interface declares `throws java.rmi.RemoteException`, in order to invoke a method on a remote object.

CORBA is an "open, vendor-independent specification for distributed computing" [22]. It achieves this loose coupling by using the CORBA IDL (Interface Definition Language). IDL files are not written in Java, but in a style similar to C++. CORBA has the same advantage of transparency, where the code to trigger "remote" invocations looks just like regular invocations. The Object Request Broker is responsible for forwarding requests from the client to the remote object. "Java contains an implementation of the ORB that communicates by using IIOP [Internet Inter-ORB Protocol]" [22].

Now we will discuss research relating to MCC (mobile cloud computing), which has its own unique challenges compared with the distributed computing technologies above. There are several recurring themes in the MCC literature. Satyanarayanan [59] groups the results achieved in mobile cloud computing into the following:

1. mobile networking
2. mobile information access, e.g. disconnected operation
3. support for adaptive applications, e.g. adaptive resource management
4. system-level energy saving techniques
5. location sensitivity

This project focuses on points 2, 3 and 4.

2.1.1 Computation offloading for energy saving and performance

One of the most obvious motivations for leveraging distributed computation on a mobile device is to save energy. Mobile devices are battery-powered, and thus have a limited life. When considering the energy costs associated with a local computation, we can calculate this as a function of the number of computing cycles, but with remote computations, we must consider the number of computing cycles, the number of bytes transmitted to/from the client device, and the power consumed while the device is waiting for the response [46]. For small

computations, for example finding the sum of two integers, local execution is preferred, since the overhead of transmitting the information about the computation to another system consumes more energy than simply performing the computation locally. For complex computations, e.g. reducing large matrices to row echelon form by Gaussian elimination, remote execution may be preferred. The less data that needs to be sent to the remote system, the more offloading is preferred. For this reason, having the ability to store objects remotely and invoke methods on remote objects allows for more opportunities to offload computations.

One can imagine making offloading decisions during development, but in some use cases the developer cannot know the optimal configuration beforehand, so having a runtime system which determines where/when computations should run remotely is beneficial. CloneCloud is a project which implements this idea: "CloneCloud uses a combination of static analysis and dynamic profiling to partition applications automatically at a fine granularity while optimizing execution time and energy use for a target computation and communication environment" [16]. By dynamically profiling the application to determine partition points, the library can determine the optimal strategy for energy saving, since "what might be the right split for a low-end mobile device with good connectivity may be the wrong split for a high-end mobile device with intermittent connectivity" [16]. When a piece of code needs to run remotely, the thread running on the local system migrates to the remote system and then migrates back to the local system when the remote computation is finished. Since CloneCloud needs to migrate the entire thread to the remote system, the data transmission cost is high. However, CloneCloud does have the advantage that it "takes the programmer out of the business of application partitioning" [16], so the programmer can focus on business logic, and can rely on the library to determine the optimal configuration at runtime. Evaluating CloneCloud, researchers found that there were significant energy savings for virus scanning, image search, and behavior profiling applications. This demonstrates the potential for energy savings even when a relatively large number of bytes need to be transmitted, and so there may be further energy savings if the bytes transmitted per remote call are reduced.

Some of the benefits of computation offloading, with respect to energy savings, are discussed in [29]. In particular, there is discussion of the experiments in [57], which show up to 45% cost saving in battery usage for Gaussian elimination on large matrices, for example. Another experiment in [57] based on text formatting using LaTeX showed that remote computation made little difference to the battery consumed, and actually worsened for certain sizes of data. They explain that the most obvious reason for this is because the original computation already had minor power consumption, and so "adding anything that itself consumes significant power (such as moving the result files back to the portable computer over the wireless link) is likely to have a major impact on the total power consumption" [57]. So again, the size of the data transmitted must be considered in order to determine whether or not offloading the computation will be effective.

A technique to determine costs of these processes involves constructing a cost graph, which takes account of the bandwidth, data transmitted, and power consumed for calls between systems. The branch-and-bound algorithm can then be used to determine where computations should be executed [48].

2.1.2 Resource management

As well as determining *if* a computation should run remotely, we also need to know *where* it will run. The device needs to know the address of the system(s) it will use for remote computation. These addresses can be hardcoded or specified at runtime. For example, Cuckoo [43] displays a QR code on the screen which can be used by the mobile device to register the address of the remote system at runtime. Cuckoo implements a resource manager, which maintains important information about potential *surrogates* (remote, powerful machines) [10], such as which surrogates are reachable. By having multiple surrogates, we improve the likelihood that our computation will even be able to run remotely. Quasi-parallel execution can also be performed: send the computation to all (or a subset) of the surrogates and use the result from the system that responds first, and discard all other results. This takes advantage of the fact that the cost of running the computation on the surrogates is negligible. If the number of bytes transmitted is small, the application might as well take advantage of the full fleet of surrogates. While Cuckoo provides support for remote resources, its heuristic for determining when to offload computation is quite primitive: "For now, we use the very simple heuristic to always prefer remote execution. The only context information we use is to check whether the remote resource is reachable." A more sophisticated heuristic could be implemented using ideas from [46] to determine energy cost locally versus remotely, and the authors mention extending the model to include these ideas.

Spectra [10, 32] uses a similar approach to determine the optimal strategy for executing a piece of code by measuring the "supply and demand for many different resources such as bandwidth, file cache state, CPU, and battery life." Similar to Cuckoo, Spectra (and its extension Chroma) executes the same computation on multiple systems in parallel, using the result from the fastest system, which means "even if one server is experiencing transient load, other servers may be unloaded and will return a result faster than the loaded server." However, rather than using QR codes to allow identification of surrogates, it uses service discovery techniques. Its versatile surrogate discovery service (VERSUDS) provides support for JINI [8], which is described as a "network architecture concept that Sun Microsystems calls *spontaneous networking*." This means surrogates can be "plugged" into the network, and when the framework determines that some code should be executed remotely, VERSUDS' service discovery can help identify potential candidates for execution.

2.1.3 Programming style, development and deployment

The transparency of the API for a distributed library will depend on the use case and the target audience. If the library is designed for casual/one-off usage, a target application should require minimal setup and application code should not be contaminated by library-specific code. The user should not need to be fully aware that the data or computations they are working with are going to be executed remotely — the library should handle this interaction. If performance gains can be achieved with fine-grained configuration, the developer will need to choose a library that allows this level of control. Control over the distributed application could involve automatically dispatching commands to surrogates [43, 10, 32], controlling where data/objects are stored and dealing with coherence [48], blocking behaviour during remote calls, etc.

Cuckoo [43] requires the programmer to define an AIDL (Android Interface Definition Language) interface in order to forward method invocations to another system. These methods are invoked via proxies, which are generated by the Android pre-compiler. The framework requires the programmer to implement a *remote* version of the interface, as well as the existing local implementation, the idea being that the programmer may wish to leverage multithreading on a multi-core surrogate. While Cuckoo provides the programmer with a degree of flexibility in terms of how code is executed, it places additional burden on the programmer. It requires code to be written outside the primary language in the form of AIDL files, and forces the user to create an interface for every class that contains a method to be executed remotely. This certainly makes the library awkward to use, especially if the user wishes to invoke methods on a class from the standard library remotely, which would require the user to create an interface which wraps every single method of the target class. Additionally, the developer will write a large amount of duplicate code in order to provide the framework with a remote implementation of the interface. Cuckoo also does not provide support for callbacks, which can be viewed as server-to-client communication, although the authors mention that this is supported by AIDL and is part of their future work. The build process also needs to be tailored to work with Cuckoo: "The first Cuckoo builder is called the Cuckoo Service Rewriter and has to be invoked after the Android Pre Compiler, but before the Java Builder". The framework provides some help to automate this process via Eclipse [31] integrations and an Ant [7] file, but this either ties the developer to a particular IDE or a particular build tool, creating a tighter coupling between the application and its dependencies. The rewriter which generates the remote implementation to be used on the remote system packages these files in a JAR file, which should be installed on the remote system, i.e. added to the classpath. Cuckoo also supports runtime installation of these JAR files, where the server requests that the client device sends the JAR file to some predesignated port. As the authors point out: "although the installation of a service introduces the overhead of sending the jars to the remote server, this occurs only once per service, while many method invocations can be done." This provides the devel-

opment team with deployment flexibility: if the client source files are known at deployment, they can be included on the server system, thereby avoiding the initial overhead, or if a surrogate is intended to be part of a cluster of public worker nodes, clients have an on-demand facility to install necessary files.

CloneCloud [16] requires minimal input from the developer, and uses static analysis and dynamic profiling to determine when code should run remotely. This is convenient from the standpoint of casual use and/or general performance gains, but it takes away a degree of control from the programmer. The programmer must act entirely as if there is no library manipulating the application, and cannot, for example, refer to remote objects for storage. CloneCloud, as well as Cuckoo, lacks support for core classes: "we restrict these partitioning points to methods of application classes as opposed to methods of system classes (e.g., the core classes for Java) or native methods." The user may only wish to run a particular method of a core class remotely, but with this restriction we would be unable to guarantee remote execution. CloneCloud does not require any client files on the classpath of the server: "when execution of the process on the mobile device reaches a migration point, the executing thread is suspended and its state (including virtual state, program counter, registers, and stack)". This simplifies deployment in some respects, but introduces the restriction that both the client and server must use the same virtualized hardware, i.e. the remote system must be a device clone. It also means every single remote call incurs the overhead of migrating all data of a thread, rather than overcoming an initial penalty and then using lightweight communication for the remainder of the application lifetime. MAUI [23] provides a similar runtime system for determining partitioning based on profiling. However, it requires the developer to annotate code in order to be eligible for offloading: '[developers] should mark as remoteable all code that meets the criteria."

2.1.4 Remote state

Cuckoo [43] does not support remote state, "although the programming model does not forbid a service to maintain internal state." This is because "Cuckoo can arbitrarily change from local to remote execution or from one remote resource to another without transferring state." This prevents the user from, for example, making incremental changes to large objects held remotely, so the device must store and transmit this data for each computation, rather than simply passing a pointer to the data.

Spectra [32] attempts to combat the issue of coherence between client and server manipulation of state by using the Coda [44] file system, which "enables a client to continue accessing critical data during temporary failures of a shared data repository." Essentially, this means the client will only block when necessary in order to achieve data is consistency. By supporting a disconnected mode of operation, users of the device will not suffer from loss of data or functionality when remote resources are unavailable or unresponsive.

Shared data is also discussed in [48], where two methods of achieving consistency are described: push and pull:

> The push method sends a piece of modified data to the opposite cluster after its modification. The pull method lets the intended receiver make the request for the modified data before the data is used.

Javascript Meteor [18] allows applications to work with distributed collections, where the server maintains the actual collection but the client has a client-side cache. However, it requires some intervention from the developer with publications and data loading, Meteor-specific code in the application, etc.

2.1.5 Recovery

The Oz language [74], implemented in the Mozart Programming System, provides transparent mechanisms for detecting and handling faults. Temporary network inactivity between a pair of sites if considered a fault state. The advantages of transparent failure handling, rather than exception handling, in distributed systems is explained in [19]:

> Synchronous failure handlers are natural in single-threaded programs because they follow the structure of the program. Exceptions are handy in this case because the failures can be handled at the right level of abstraction. But the failure modes can become very complex in a highly concurrent application. Handlers for the same entity may exist in many threads at the same time, and those threads must be coordinated to recover from the failure.

Mozart actively attempts to re-establish a connection, especially if TCP drops the connection, and ensures consistency of stateful entities in the distributed system.

Optimistic recovery is an application-independent technique for transparent recovery in distributed systems [65, 41, 45, 12]. It defines recovery units, which can fail and be recovered, and communicate via message passing. It explains why message numbers are necessary for recovery. Message logging and checkpointing are used to provide this fault tolerance. The half-session algorithm described in [65] inspires the handshake protocol implemented in this project, with the use of sequence numbers maintained on several systems, and comparing sequence numbers to determine how to recover. The recovery protocol implemented in this project is Mokapot-specific, but as with optimistic recovery techniques, is application independent. Checkpointing, i.e. taking a snapshot of the JVM for rollback purposes, is not supported in standard JVM implementations and so requires the use of an extension, so is not suitable for Android devices.

2.2 Conclusion

The literature demonstrates that there is the potential for much improved performance on mobile devices by leveraging computation offloading. Most existing approaches focus on offloading heavy computations to a set of stateless surrogate machines. This project will gather ideas from these projects, taking inspiration from the most useful features and filling important gaps in their capabilities. When the project is evaluated, the usage of the library versus other technologies, such as web APIs, should be justified.

3 Analysis and Specification

3.1 Overall Description and Scope

The purpose of this project is to provide mobile application developers with a simple, transparent tool for writing new distributed applications or introducing distributed computing into existing applications. The library should be easy to integrate both with respect to the build process and the development process. This project will use Mokapot as a foundation for providing the developer with transparent distributed code, i.e. method invocations which are executed on other systems will be written as regular code, but this project will be responsible for providing transparent management of network recovery. Currently, Mokapot will throw an exception if network errors are encountered. This is a reasonable action because Mokapot does not attempt to recover broken network connections; without throwing an exception, the application would simply block forever. Furthermore in practice, assuming Mokapot is running on a system physically connected to a stable network, network errors are generally irrecoverable, e.g. a system has crashed — errors due to changing IP addresses, network instability, etc. are rare (some cloud hosting providers have 99.99% uptime SLAs [5]). For mobile devices, however, this assumption is much weaker; network connectivity will often be unstable, and disconnections may be frequent in certain regions with low network coverage, for example.

The likelihood of network errors in mobile applications presents problems. The developer must be aware that code which would normally not throw an exception, e.g. calculating the size of a list, may fail and so the developer must handle these exceptions in order for their application to continue execution. This can result in complicated exception handling, ad hoc retry strategies that are fragile and incorrect — code may be executed twice on the remote system instead of once. To see how this is the case, imagine a scenario where a connection is closed after sending a message to the remote system: did the remote system receive and begin processing the message and then the connection was closed, or was the connection closed before the message reached the remote system? For developers to consider Mokapot a feasible technology for the Android plat-

form, it must be capable of transparently handling network errors, otherwise the burden on the developer will be too high.

This project will implement a recovery protocol which guarantees at-most-once execution of code that is intended to be run remotely. In order to initiate the recovery protocol, the system must first detect a failure. The mobile device will have a recovery system which initiates a recovery protocol if, during its periodic checks, it detects a potential failure. Executing this protocol with another system will ensure that any "lost" messages are re-sent to their recipient, and any already-sent messages are not resent. This guarantees at-most once execution, and if the recovery eventually takes place we will have achieved exactly-once execution. The mobile device will continue attempting to recover with a disconnected remote system forever. This is an optimistic approach which assumes that the remote system will eventually become available for recovery, but of course the user of the mobile application can always close the application if it becomes unresponsive — a course of action more suitable to handheld personal devices than servers in the cloud.

However, the device will make further attempts to recover remote objects which satisfy certain properties: *renewable* objects. These are objects where, for example, state is not critical, the state of other objects is not manipulated, e.g. a calculator object with a single method that doubles the input integer. These objects will be *renewed* on another available system (including the local system if no other remote systems are available) and any unanswered method invocations for the original object will be reinvoked on the renewed object. The developer must explicitly declare classes as renewable, and must accept that the objects of that class may exhibit unexpected behaviour if the class doesn't satisfy the documented constraints.

3.2 Product Perspective

The system will be split into three main parts: the central Mokapot implementation, the network management components, and system-specific modules. The existing implementation of the Mokapot library is responsible for offloading computation to other systems, maintaining remote objects etc. The network management components, added as part of this project, are responsible for improving the resiliency of applications which use Mokapot by implementing failure detection and recovery strategies. The system-specific modules are responsible for providing implementations of Mokapot classes which are specific to the system on which Mokapot is running — this will allow for the differences between ART (Android runtime) and the JVM (Java Virtual Machine) to be accounted for. The application developer only needs to specify whether or not the network management components should be enabled, they do not (and cannot) interact with them any further. To use Mokapot, the developer should add a dependency on the appropriate module for their system. For example, if the developer is

writing the server-side application they will likely use the *mokapot-java* library, while the mobile application will use the *mokapot-android* library.

3.3 Constraints

The core libraries used by ART and the JVM are very similar but not identical. This means some classes have different APIs[1], while other classes are available on one platform but not the other[2]. In practice this should rarely be a problem as the platforms are very similar, but this limitation may prohibit certain applications from being written if the application includes dependencies which cannot be rewritten to circumvent these differences.

3.4 System Requirements

Mokapot uses classes introduced in Android API level 26, which is available only in Android 8.0 (Oreo) and above, so mobile applications that wish to use this project are restricted to API level 26 and above. Furthermore, at the time of writing, Android Oreo is still being rolled out to many devices [63]. During the rollout, applications using this project will be restricted to devices with the upgrade, and even after the rollout older devices will never receive the upgrade [37]. This means that if a developer wishes to use this project to improve application performance on older devices, he must wait until today's phones are considered old. The latest Android APIs are available on Android emulators [58], which can be used by developers instead of physical devices.

3.5 Functional Requirements

ID: FR1
Description: Potential network errors should be actively detected.
Reason: Although TCP ensures reliable host-to-host communications, certain types of errors will not be surfaced for a long time. For example, if the IP address of the remote system changes, the sending system will not receive TCP acknowledgements for its messages, and so it will continue retransmitting [54]. However, the number of retries can correspond to durations of approximately 13 to 30 minutes [67]. In order to resume execution as soon as possible, Mokapot should actively detect *potential* network errors and then initiate some form of reconnection. A "potential" network error should be recorded when there is an abnormal event, e.g. a request hasn't received a response within a predefined period of time.
Dependencies: None

[1] java.math.BigInteger has a different API on Android compared with Java SE.
[2] The entire java.rmi package is not available on the Android platform.

ID: FR2
Description: When a potential network error is detected, a recovery protocol should be initiated.
Reason: The detection of a potential network error suggests that the system should attempt to reconnect to the remote system. To ensure that the systems are in a consistent state and can resume execution they should coordinate any necessary retransmissions.
Dependencies: FR1

ID: FR3
Description: If the local system determines that a remote system is probably unreachable, any renewable objects on the remote system should be renewed on another system.
Reason: By allowing the developer to declare classes as renewable, we improve the chances that an application can recover when confronting network errors. Furthermore, it gives developers the freedom to write applications which leverage remote systems *if they are available*, but can run entirely locally otherwise.
Dependencies: FR1

3.5.1 Non-functional requirements

ID: NFR1
Description: The additions made to Mokapot in this project should make it easy for a developer to integrate Mokapot with their Android applications. Furthermore, the developer should find the library transparent from the point of view of managing network resiliency (and of course from the point of view of distributed computation, since this is what Mokapot provides).
Reason: There are always alternative ways to emulate some form of distributed computation. For example, a developer could simply create a web service with a defined API that their mobile application could use via HTTP. The web service could provide many of the same features as Mokapot. However, this is not a simple setup, since it requires a well defined API, marshalling/unmarshalling frameworks to be used on the client and server, type information is lost, callbacks are tricky to setup, network resiliency must be handled manually, etc. In the spirit of providing developers with a solution which alleviates the developer as much as possible of the burdensome elements of writing distributed code, the final solution should mask the low-level handling of network errors, recovering when resources are no longer available, etc.

ID: NFR2
Description: The library should be easy to deploy and integrate into the build process.
Reason: Similar to NFR1, the spirit of Mokapot is to manage as much of the distribution process as possible. Since computation offloading/distributed computing will often not be a necessity, but rather an opportunity to attain improved performance, it is crucial that the overhead of using the library with new or existing applications is minimal. Otherwise, the benefits a developer might achieve will be dwarfed by the struggles encountered. Furthermore, deploying a system that uses Mokapot should be easy. The deployment will be more involved than other deployment scenarios, since both the client and server require some coordination in terms of their classpaths, etc.

ID: NFR3
Description: The library should improve the performance of some computation-heavy applications and save energy.
Reason: If the library cannot yield noticeable results from offloading computations to another system, then developers will be unlikely to adopt this project. There will always be a trade-off between network and CPU activity, and so we will only consider cases where the transmission time is negligible compared with the computation time.

3.6 Verification

This project will use acceptance tests and proofs to verify the software. Acceptance tests will demonstrate that, from the developer's point of view, the software exhibits normal behaviour even when the network is unstable, e.g. sockets are closed, messages are lost, etc. Mokapot already has unit and integration tests, which will ensure regressions are not introduced by the changes in this project. To show that the recovery protocol is *correct* we will construct a model of the existing protocols and prove that, under this model, the proposed changes will ensure that all lost messages are recovered (if possible) and messages are not retransmitted unnecessarily.

4 Design

In this section we will discuss the design of the features which contribute to network resiliency and renewable types. We will also discuss the reasons for such design decisions and why they must be implemented at the application level. We begin this section with an analysis of the differences between the Android runtime and the Java Virtual Machine, as this must be understood before undertaking any design decisions and before implementing the Android-compatible version.

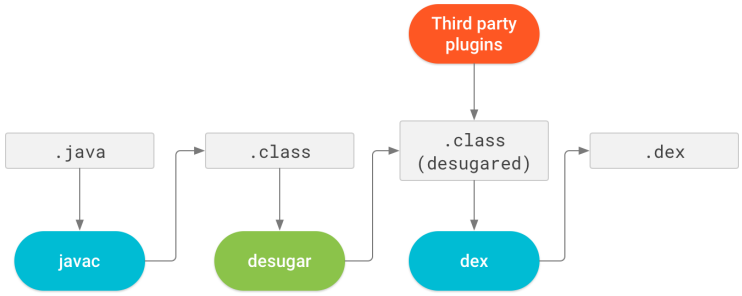

Figure 2: The Android toolchain with support for Java 8 language features [72].

4.1 ART vs JVM

4.1.1 Lambdas expressions and Android's toolchain

To run code remotely without invoking a method on a long reference, the developer must write code similar to the following:

```
List<String> ls1 = DistributedCommunicator.runRemotely(
    () -> new ArrayList<>(), serverAddress);
List<String> ls2 = DistributedCommunicator.runRemotley(
    new CopiableSupplier<>() {
        @Override
        public List<String> get() {
            return new ArrayList<>();
        }
    }, serverAddress);
```

In the above example, the code used to create `ls1` and `ls2` is semantically equivalent, but `ls1` uses a lambda expression while `ls2` uses an anonymous inner class. The developer community is continuing to adopt Java 8 [52], and with lambda expressions cited as one of the reasons for its popularity [56], the style of code written for `ls1` will be preferred by many developers. The Android platform supports Java 8 language features [72] through its default toolchain, as shown in figure 2; previously, other tools, such as the Jack toolchain or Retrolambda, were necessary.

In order for a remote system to execute the code specified by a lambda expression, the lambda must be serialized, and therefore must be serializable. Mokapot uses the `writeReplace` and `readResolve` methods of lambdas, which are exposed by the `Serializable` interface, for serialization and deserialization respectively. The `writeReplace` method produces a `SerializedLambda` while `readResolve` constructs a lambda from a `SerializedLambda` [17] [38]. Java's strategy for lambda deserialization, which attempts to mitigate potential security vulnerabilities [35], involves the use of a synthetic method, `$deserialize$`,

generated by the compiler in the capturing class, which is used by `readResolve` [17]. For example, using the `javap` tool to disassemble a class file compiled in the early stages of the build process shows the following methods are present:

```
Compiled from "MainActivity.java"
public class MainActivity extends android.support.v7.app.
    AppCompatActivity {
  public MainActivity();
  private void startCommunicator() throws java.io.
      IOException, java.security.KeyStoreException, java.
      security.KeyManagementException;
  private static ClientTable setup(xyz.acygn.mokapot.
      CommunicationAddress);
  protected void onCreate(android.os.Bundle);
  private static java.lang.Object
      deserializeLambda(java.lang.invoke.SerializedLambda);
  private void lambda$onCreate$1(ClientTable);
  private ClientTable lambda$onCreate$0(java.lang.String)
      ;
}
```

However, Android does not support the serialization of lambda expressions [72]. Instead, lambdas are transformed as part of the desugaring process into a combination of a method call and a separate class, with a name such as `MyClass$$Lambda$0.class`. This means that, although the developer may write code using lambdas, they are compiled and executed without any need for the lambda deserialization mechanisms in the JVM. The built-in desugaring process cannot desugar dependencies which use classes not available on the Android platform, e.g. classes in the `javafx` package. In these cases, Retrolambda can be used to "transform Java 8 compiled bytecode so that it can run on an older Java runtime" [55], which means lambda expressions are transformed in a similar fashion as in the Android toolchain but does not encounter the same issues as the desugaring process. Retrolambda has limitations, and the Android community recommends using the default toolchain if possible [72], but is an alternative if necessary.

It is necessary to understand the build process and how the Android toolchain compiles and transforms source code because Mokapot requires code which is executed remotely to be present on the classpath of both systems. If the build process for Android did not generate bytecode that could be used on the classpath of an application running in the JVM, then we could not use Mokapot without some fundamental changes to Android's toolchain or Mokapot's mechanism of executing code remotely. By using Gradle to build Android applications, intermediate artifacts as well as an installable APK are generated [20]. These artifacts include the class files generated immediately after the desugar trans-

formations[3], which can be packaged in a JAR file as part of the build and then added to the classpath of the remote system.

For serialization/deserialization purposes, Mokapot treats lambdas differently than other objects. Since Android transforms all lambdas during desugaring, this project should ensure that Mokapot running on Android always knows that the objects at runtime are never *actually* lambdas. This must be done carefully, since static analysis may suggest that the objects *are* lambdas, e.g. `Class.isSynthetic` will be true, the class name will use the `$$Lambda$N.class` convention, etc.

4.1.2 Proxy creation

A *long reference* in Mokapot is a reference to an object which resides on another system. Dynamic, i.e. runtime-generated, proxies [30] are used to implement this behaviour, where method invocations on a long reference are actually proxied, and the invocation is routed to the system where the object resides. Java's built-in proxy features are not suitable for Mokapot's purposes, so Javassist is used instead to create proxies. Javassist uses bytecode manipulation to achieve the behaviour of proxies [40] . This presents a challenge for porting to the Android platform: the bytecode formats used in Android and the JVM are different [24, 50, 25]. Therefore the port must use an alternative library for proxy creation. Byte Buddy [13] and Dexmaker [28] are libraries compatible with Android and its bytecode, which allow developers to manipulate bytecode at runtime to produce dynamic proxies. To port Mokapot to Android, an Android-compatible alternative to Javassist, such as Byte Buddy or Dexmaker, must be used. To avoid creating an entirely new project, this project should implement a strategy pattern in which Mokapot calls out to some other module to create a proxy, where the module is suitable for the runtime system.

4.1.3 Class differences

A surprising, and virtually undocumented, fact about Android's `Object` class is that it has hidden fields:

```
private transient java.lang.Class java.lang.Object.shadow$_klass_
private transient int java.lang.Object.shadow$_monitor_
```

Commit messages in the Android source code repositories [1, 2] suggest that these were added to support Brooks pointers [3, 33], however the behaviour is disabled by default. The presence of these objects is a problem for Mokapot because the fields of objects serialized by Android will be different from those expected by an application running in the JVM. Therefore, this project will

[3]In the directory build/intermediates/transforms/desugar/debug/0/com/...

need to ensure that Mokapot is able to handle these fields in an appropriate fashion.

This mechanism is used as a garbage collection *optimization*, but are disabled by default. Furthermore, in many cases the number of objects to which long references exist will be negligible compared with the number of local objects, and if the system on which the referent resides is running Mokapot in the JVM then we do not need this optimization. Therefore, in this project, we will ignore transient fields, and therefore the hidden fields above, during serialization and deserialization. Java serialization ignores transient fields, but Mokapot uses a custom serialization algorithm so we must handle this manually.

4.1.4 Classpath differences

As mentioned, Mokapot requires code that will be executed remotely to be available on the classpath of both systems. However, there are classes available on the Android runtime which are not available in the JVM, and vice versa. This means that the JVM will encounter problems if an Android device attempts to run a block of code remotely which contains a method invocation on a object of a class which is available only on the Android platform. The classloader on the JVM will not be able to load the Android-only class, and will therefore throw an exception. However, this is not much of a restriction: why would a developer attempt to run Android-only code on a non-Android system?

There are some cases where the developer might accidentally encounter these issues. For example, consider the following two programs, which are semantically equivalent:

```
class MyClass extends android.app.Activity {
  void foo() {
    CommunicationAddress serverAddress = ...;
    int x = DistributedCommunicator.getCommunicator()
        .runRemotely(() -> 5 + 1, serverAddress);
  }
}
```

```
class MyClass extends android.app.Activity {
  int increment(int i) {
    return i + 1;
  }

  void foo() {
    CommunicationAddress serverAddress = ...;
    int x = DistributedCommunicator..getCommunicator()
        .runRemotely(() -> increment(5), serverAddress);
  }
}
```

The first program will be able to run `() -> 5 + 1` on the remote system, while the second, which uses a method call, will not. This is because, in the first example, Android can compile and transform the lambda into a separate class similar to the following:

```
class MyClass$$Lambda$0 implements CopiableSupplier {
  public Integer get() {
    return 5 + 1;
  }
}
```

That is, the transformed class is standalone and independent of `MyClass.java`. However, in the second example, the lambda must capture the `this` pointer (in order to invoke the `increment` method), and so the transformed class will be similar to the following:

```
class MyClass$$Lambda$0 implements CopiableSupplier {
  private final MyClass arg$1;

  public Integer get() {
    return arg$1.lambda$foo$MyClass();
  }
}
```

where `lambda$foo$MyClass` is the synthetic method generated when transforming the lambda `() -> increment(5)`. To execute the "lambda" remotely, the remote system must load `MyClass$$Lambda$0`, but since the `arg$1` field extends `android.app.Activity`, if the application running in the JVM, rather than the Android runtime, then the classloader will throw an exception.

Therefore application developers must be careful to ensure that their applications are well architected and use design patterns that ensure separation of Android-specific/GUI code and remote computation. Mokapot has a partner project called *Millr* which can rewrite the user's code so that it is compatible

with Mokapot. An extension to Millr could involve creating wrapper classes to avoid `CopiableSupplier`s capturing fields which are Android-specific.

4.1.5 Security

Mokapot has several security mechanisms, such as encrypted connections and whitelists of permitted systems. The Android platform provides various security protections [9], and a important feature for Mokapot is private internal application storage. This means we can safely create proxies and save them on disk so they can be loaded by the classloader. Android provides developers with access to internal storage through `android.content.Context`'s `getFilesDir` and `getCacheDir` methods [26].

An application running in the JVM can set a security manager which specifies and restricts actions that are unsafe or sensitive [70]. Since Mokapot entails the execution of code specified by other systems, Mokapot requires a security manager to be present, otherwise a `java.lang.SecurityException` will be thrown. However, the Android's security model works differently, and in fact the `java.lang.SecurityManager` is considered legacy security code and is unsupported [61]. Instead, Android's permission model requires developers to specify in a manifest file which protected APIs should be accessible, including camera functions, location data, and network/data connections [9]. Apps are automatically given permission to read and write to their private internal storage [60], so Mokapot requires only the `android.permission.INTERNET` permission. Applications can then add their own permissions as necessary.

4.1.6 Networking

In the JVM there are no restrictions on which threads can perform network operations. In the Android runtime, however, if the main thread of an application attempts to access the network an `android.os.NetworkOnMainThreadException` is thrown. This is done because long-running operations, such as network operations, should be performed in a worker thread in order to keep the application responsive [42]. This means that invoking a method on a long reference, such as `List.size`, may result in an exception being thrown, even though this would never happen for a short reference. Although this changes the semantics of the code, this project will not attempt to circumvent this restriction. Mokapot could, for example, run network requests in a different thread if it detects that the current thread is the main thread. However, this introduces added complexity for developers attempting to debug their applications, as there are additional threads involved, and it also masks the fact that the developer is running a (somewhat) long-running operation on the main thread, impairing the responsiveness of their application. Therefore we leave the responsibility with the developer, who will be aware that their application is using Mokapot,

and so should design their application so that remote computation is executed in a separate thread.

4.2 Network Resiliency

Since a mobile device is typically connected to the Internet wirelessly, it is more prone to losing network connectivity than their desktop counterparts, which can be connected to the Internet via a wired connection, which is more reliable. A mobile device running Mokapot may become disconnected from the network during idle periods, or in the worst cases while the device is actively communicating with another system. This highlights two important issues Mokapot must handle:

1. If a system has established a connection with another system running Mokapot, but this connection is lost or appears to be broken, the system should attempt to reconnect to the remote system so that distributed computations can resume. This is discussed in section 4.2.2.

2. Upon re-establishing a connection with another system, a handshake should take place in order to synchronise the state of distributed threads handled by these systems. For example, a message sent by the first system may never have reached the second system, but the first system has no way of knowing whether or not this is the case. The first system cannot safely resend the message, since processing the message may have had side effects. To ensure a consistent state is established, the systems will need to follow the handshake protocol. The handshake protocol is discussed in section 4.2.3.

The fundamental reason for identifying and dealing with the above cases is because Mokapot is *stateful*. If Mokapot were stateless, systems could simply retry the sending of messages until a response is received. Furthermore, a human user cannot determine by inspection if a message has been lost because requests are not necessarily triggered by user input and may not be visible to the user.

While two systems are "connected", TCP will ensure reliable communication. The concept of a TCP connection is defined in terms of sockets [54]:

> To allow for many processes within a single Host to use TCP communication facilities simultaneously, the TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the internet communication layer, this forms a socket. A pair of sockets uniquely identifies each connection. That is, a socket may be simultaneously used in multiple connections.

Importantly, if a system's network or host address changes, its socket in the socket pair will no longer receive data. Since a pair of sockets uniquely identifies each connection, and (at least) one socket is no longer valid, the connection is no longer valid. In this scenario, TCP will not ensure message delivery, so

this must be handled by some other protocol. To recover from lost data, TCP assigns a sequence number to each octet transmitted and requiring a positive acknowledgement (ACK) from the receiving TCP [54]. Even if it were possible to access this information at the application level, it would not solve the problem: if an ACK is not received for a certain octet, this does not mean that the receiving TCP never received the octet, it just means no ACK has been received at the sending TCP, i.e. the ACK might arrive later or not at all. In the scenario we are considering, the sending system's network and/or host address may have changed, so the octet may have reached the other system while the sending system's address changed, in which case, without any additional information, the sending system has no way of knowing if the octet was received (or processed).

We propose an application-level recovery protocol, which will ensure recovery (assuming the two systems are able to reconnect) in the scenarios identified above.

4.2.1 Identifying network errors

A system must identify that the connection between itself and another system is suspicious before it can take steps to re-establish a new connection. We propose a ping-based detection mechanism, wherein a system decides that a connection is suspicious if it has not received a reply to a ping after a timeout period. Eventually the local system may observe some sort of socket exception, e.g. a timeout exception because the local system refuses to retransmit packets, but retransmission can continue for a long time (approximately 13 to 30 minutes) [67] — an amount of time we consider unreasonably long for a responsive mobile application. Instead, we desire a fail-fast detection mechanism, allowing prompt recovery. The application could also use a thrown exception as indication that the connection should no longer be used. However, exceptions may not be thrown, e.g. in the case of change of network/host address, and so this is certainly not a reliable method in all cases. Instead of sending pings, the application could infer separation of sender and recipient by timing the existing network requests. This is prone to error since different requests will take different amounts of time to receive a response. For example, a request that entails a large amount of processing may breach the timeout, and in fact it may never be able to complete this task successfully if Mokapot continually decides the connection on which this request was issued is suspicious. Furthermore, on its own, it denies an application the opportunity to proactively recover during idle periods since there are no requests to supervise/time.

A ping is a lightweight message which requires a lightweight response. By sending regular pings, and setting a timeout on the response, we can be confident that latency of the response is due to the network rather than processing time. The timeout could be adjusted to take account of the average round-trip time so that a slow network connection is given a longer timeout than a high-speed network. Solutions exist for adaptive timeout strategies [14, 49], but in this project

we will simply provide the developer with an option to specify a timeout period, which should be short enough so that the application can recover quickly but long enough for slow networks. An application can proactively detect suspicious connections with pings, as they will be sent even during idle periods.

A response to a ping is an implicit confirmation of receipt of all previously sent data. This is because in TCP, "at the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates" [54]. This means that if a response is received to a ping, we know that data sent before the ping reached the receiving TCP, which ensured correct ordering of data before passing the data onto the "end user" (the application reading from the socket). Simply put, Mokapot cannot process the ping message, and therefore send a response, until the previously sent messages/data has been received. This means we don't have to worry about not detecting a lost message through pings, i.e. we cannot receive a response to a ping until a "lost" message has been recovered (retransmitted and received) by TCP.

Mokapot also sends pings for other reasons. For example, lifetime manager-location manager pairs will send pings to inform the other system that a reference still exists to a particular object so that it is not deallocated by the garbage collector. An optimisation to reduce network traffic would require Mokapot to intelligently send pings as needed, i.e. send a new ping to detect suspicious connections only if a certain amount of time has passed since the last ping (of any type) or receipt of a message. However, at the time of writing the other ping messages are not implemented, so this optimisation is not in place, but should be added in the future.

Although we cannot always detect failures in this way, if an exception *is* thrown when attempting to send a message then we should consider the connection to be suspicious. If an exception is encountered, Mokapot should record the message it intended to send, and then the recovery mechanisms can treat these "sent" messages as if they were actually sent but the recipient never received them, and so the message will be recovered during the handshake.

4.2.2 Establishing a new connection

Once a system determines that a connection to another system is suspicious, it should attempt to close the connection and establish a new one. Any data that would have arrived via the connection at a later time will be recovered through steps at the application level. When a system decides that a connection is "suspicious", it has decided that it is unlikely that future communications on that connection will succeed, possibly because the system's address has changed, therefore closing the connection, rather than keeping it alive with a hope that we will receive already-sent data, is consistent with the determination of the connection being suspicious.

Closing the socket *may* result in the remote socket closing as well, but since the

connection is suspicious, we suspect that the termination of this socket will not be known by the remote system via TCP. This means the remote system needs to recognise when a system is reconnecting rather than connecting for the first time. When a system reconnects, all existing sockets communicating with that system should be closed (so that no more data will be received from the stale connection) and distributed threads being managed by both systems should be tied together. This means each system must be identified by a unique ID, rather than by a combination of IP address and port number (otherwise when a system's IP address changes the other system won't know with which system it is communicating). The order of these events is important: the system which detects a suspicious connection must close its socket *before* establishing new one, and the other system should immediately close its corresponding socket when it recognises the connecting system. This ensures that, during the recovery process, no messages are sent between the systems other than those required for the recovery process itself.

4.2.3 Synchronising after reconnection

After two systems re-establish their connection, they will need to perform a handshake to ensure a consistent state is reached. The problem here is that one system may have sent a message after the connection was invalidated but before the connection was re-established. This can happen when, for example, a system's address changes but its socket is not closed.

Each system may have a number of distributed threads which are blocked waiting for a response from another system. A system will only wait for an answer on a distributed thread if it asked a question on that distributed thread. After re-establishing the connection, the system can collate a list of distributed threads for which it is waiting for a response. However, in general, the system does not know the reason for the lack of response: is network latency causing a delay but the response will be received later, or does the request require more time to fulfil, or was the message lost?

Definition 4.1. Let X be a system, then Θ_X denotes the set of distributed threads on which X has ever sent or received a message.

Definition 4.2. Let m be a message, then m_Ω denotes the system which is intended to receive m.

Definition 4.3. Let X be a system, and let $t \in \Theta_X$, then $\pi_X(t)$ denotes the most recent message processed, i.e. received or sent, by X on t.

Definition 4.4. Let X be a system, and let $t \in \Theta_X$, then $M_X(t)$ denotes the set of all messages processed by X on t, and $M_X = \{M_X(t) : t \in \Theta_X\}$.

Definition 4.5. Let m be a message, then $\theta(m)$ denotes the distributed thread on which m is processed. For a lightweight message, m', we define $\theta(m') = \theta_0$.

Definition 4.6. Let m be a message, then $\Pi(m)$ denotes the ordered sequence of systems involved in processing this message on a single thread to "completion", i.e. such that no more messages are required to be sent to fulfill the obligations of m. For example, if m is sent by system A, received by system B, which contacts system C to process the message, which responds to B and then B responds to A, we have $\Pi(m) = ABCBA$.

Definition 4.7. Let m be a message, then $\mathrm{Ack}(m)$ denotes the acked state of m, i.e. true or false.

Each system should record the last message sent on each distributed thread so that if the message is lost then the intended recipient can ask for the message to be re-sent. But is this enough information for the systems to reach a consistent state, i.e. a state in which both systems know which messages have been sent by the other system and lost messages are then re-sent? No. Consider two systems, A and B, say, which are running Mokapot and communicating with each other. However, a message is lost, i.e. a system attempted to transmit it but it was not received by the other system, and the systems need to coordinate the resending of the lost message. We have the following possible scenarios:

1. A sends a message m_1 to B, where $\Pi(m_1) = ABABA$. B receives m_1 and sends m_2, where $\Pi(m_2) = BAB$, but A never receives m_2. Now A is waiting for a response to m_1, and B is waiting for a response to m_2.
2. A sends a message m_1 to B, where $\Pi(m_1) = ABABA$. B receives m_1 and sends m_2, where $\Pi(m_2) = BAB$. A receives m_2 and sends m_3, where $\Pi(m_3) = AB$, but B never receives m_3. Now A is (still) waiting for a response to m_1, and B is waiting for a response to m_2.

Notice that $\theta(m_1) = \theta(m_2) = \theta(m_3)$. Consider the information available when attempting to coordinate which system should resend a message. In both scenarios presented above, A is waiting for a response from B and vice versa, and therefore the last message sent on the distributed thread from A is to B and vice versa. Therefore, without any other information, the systems cannot determine which system is responsible for resending its message. In the first scenario, B is responsible for resending its message, and in the second scenario A is responsible for resending its message.

In the scenarios, we have smuggled in the solution: label each message with a sequence number. Suppose the systems now know the sequence number for each message sent/received.

Definition 4.8. Let m be a message, then $\sigma(m)$ denotes the sequence number of m.

Definition 4.9. Let X be a system, and let $t \in \Theta_X$, then $\sigma_X(t) = \sigma(\pi_X(t))$.

Definition 4.10. Transition rules for a system X:

$$\texttt{receive m} \Longrightarrow [\theta(m) \notin \Theta_X \to \Theta_X \coloneqq \Theta_X \cup \{\theta(m)\}] \wedge [\sigma_X(\theta(m)) \coloneqq \sigma(m)]$$
$$\texttt{send m} \Longrightarrow [\sigma(m) \coloneqq \sigma_X(\theta(m)) + 1] \wedge [\sigma_X(\theta(m)) \coloneqq \sigma_X(\theta(m)) + 1]$$

To expand on the definition above, when a system receives a message m on a thread t, this is the most recent event on this distributed thread and so the system should update its view of the sequence number for t. When a system sends a message on a thread, it should increment its sequence number so that the recipient sees a unique sequence number (on this thread) — this distinguishes the message from others sent on the same thread. The message sent will have the incremented sequence number, and the sending system, X, will record the incremented sequence number in $\sigma_X(\theta(m))$.

In the first scenario, $\sigma_A(t) = 1 < 2 = \sigma_B(t)$, so B knows that it is "ahead" of A, and therefore should resend m_2. Conversely, in the second scenario, $\sigma_A(t) = 3 > 2 = \sigma_B(t)$ so B is "behind" and should ask A to resend m_3. This needs to be clarified a little bit further. In the scenarios presented, A is the system responsible for establishing a new connection with system B, and therefore the inequalities compare $\sigma_A(t)$ and $\sigma_B(t)$. However, if there is a third system, $C \neq A$, we must be careful to only consider resending messages to the intended recipient. Consider the following scenario: A sends a message m_1 to B, where $\Pi(m_1) = ABCBA$. B receives m_1 and sends m_2 to C, where $\Pi(m_2) = BCB$. Suppose at this point, A establishes a new connection and initiates the synchronisation process with B (and suppose C will take a long time to complete its processing of m_2). Here we have $\sigma_A(t) = 1 < 2 = \sigma_B(t)$ — should we "resend" m_2 from B to A? No — we never sent m_2 to A in the first place! Instead, B should recognise that $(\pi_B(t))_\Omega \neq A$, and so there is nothing to do. Similarly, if $\sigma_A(t) = \sigma_B(t)$, then there is nothing to do.

As an optimisation, during the handshake the systems should acknowledge messages received by the other system, so that in future handshakes these messages are not sent; this will reduce network traffic.

The handshake for systems A and B proceeds as follows after the connection is re-established:

1. System A sends $R_A = \{(t, \sigma_A(t)) : t \in \Theta_A, (\pi_A(t))_\Omega = B \text{ and } \neg\mathrm{Ack}(\pi_\mathrm{A}(\mathrm{t}))\}$ to system B (note that since $(\pi_A(t))_\Omega = B$, $\pi_A(t)$ must have been *sent* to B).
2. System B replies with $S_A = \{t : (t, \sigma_A(t)) \in R_A, t \notin \Theta_B \text{ or } \sigma_B(t) < \sigma_A(t)\}$, the list of threads on which A should resend messages, and $R_B = \{(t, \sigma_B(t)) : t \in \Theta_B, (\pi_B(t))_\Omega = A \text{ and } \neg\mathrm{Ack}(\pi_B(t))\}$.
3. System A replies with $S_B = \{t : (t, \sigma_B(t)) \in R_B, t \notin \Theta_A \text{ or } \sigma_A(t) < \sigma_B(t)\}$ and $U_B = R_B \setminus S_B$.

4. For each $t \in S_A$, A resends $\pi_A(t)$ to system B. For each $t \in R_A \setminus S_A$, system A sets $\text{Ack}(\pi_A(t)) = \text{true}$.

5. For each $t \in S_B$, system B resends $\pi_B(t)$. For each $t \in U_B$, system B sets $\text{Ack}(\pi_B(t)) = \text{true}$.

We will now prove correctness of the handshake. The following assumes that there are no errors during the recovery process (if errors are encountered, the recovery process restarts after a period of time).

Lemma 4.11. Let X be a system and let $m \in M_X(t), t \in \Theta_X$. If m has not been received at m_Ω then $\pi_X(t) = m$.

Proof. X cannot send another message on t until it receives a response since the notion of a distributed thread entails one operation at a time. Similarly, another system cannot send a message on t until it has received a message on t, thereby giving that system control of the thread. Therefore X has sent no additional messages on t, and cannot receive any additional messages on t since only one system at a time has control of t (the next system to gain control should be m_Ω), so $\pi_X(t) = m$, as required. □

Lemma 4.12. Let A be a system and let $m \in M_A(t), t \in \Theta_A$. If m has not been received at m_Ω then either $t \notin \Theta_{m_\Omega}$ or $\sigma_{m_\Omega}(t) < \sigma_A(t)$.

Proof. Let $B = m_\Omega$. If $t \notin \Theta_B$, i.e. B has never processed a message for t, then we are done. Otherwise, $\sigma_B(t) = \sigma(\pi_B(t))$. The sequence number of each message sent on a distributed thread is unique and since B has not received m, $\sigma_B(t) \neq \sigma(m) = \sigma_A(t)$. A is the most recent system to have sent a message on t, so we have that $(\pi_B(t))_\Omega \neq B$, i.e. $\pi_B(t)$ was *sent* (not received) by B (since B relinquished control of the thread at some point). Since A sent m *after* B sent $\pi_B(t)$, $\sigma_A(t) \geq \sigma_B(t)$. This combined with $\sigma_A(t) \neq \sigma_B(t)$ gives us $\sigma_B(t) < \sigma_A(t)$, as required. □

Theorem 4.13. Let A, B be systems. If $\exists m \in M_A$ such that $m_\Omega = B$, but $m \notin M_B$, m will be re-sent by A to B during the handshake.

Proof. $m \in M_A$, hence $m \in M_A(t)$, for some $t \in \Theta_A$. Since $m \notin M_B$, m has not been received at B, so by lemma 4.11 we have $\pi_A(t) = m$. We also know that $\neg\text{Ack}_A(m)$, otherwise $\sigma_B(t) \geq \sigma_A(t)$, but by lemma 4.12 $\sigma_B(t) < \sigma_A(t)$. Therefore $(t, \sigma_A(t)) = (t, \sigma(m)) \in R_A$. By lemma 4.12, since B never received m, either $t \notin \Theta_B$ or $\sigma_B(t) < \sigma_A(t)$, therefore $t \in S_A$. By step 3 of the handshake, we have that $\pi_A(t) = m$ will be re-sent from A to B. □

Theorem 4.14. Let A, B be systems. If $\exists m \in M_B$ such that $m_\Omega = A$, but $m \notin M_A$, m will be re-sent by B to A during the handshake.

Proof. We prove this in the exact same way as above, replacing occurrences of A with B and vice-versa. □

Since a "lost" message is one that was sent by one system but not received by the other, we conclude the desired result that any lost message sent by one system is re-sent as part of the handshake. This proof is based on a successfully implemented model of messaging passing, correct state transitions, avoidance of deadlocks, correct synchronisation etc.

4.3 Renewable Types

4.3.1 Description and usage

FR3 specifies that if a system holds long references to renewable types on a remote system then these objects should be "renewed" on another system if the local system determines that the remote system is (probably) unreachable. This gives systems a better chance to recover, and especially for those applications that wish to use remote computation as an optimisation rather than a strict requirement, providing renewable types caters for more use cases. Once the object is renewed, all method invocations on the long reference will be routed to the renewed object on the new host system rather than the original object. However, there are a number of restrictions the developer should keep in mind when declaring a class as renewable.

- Mokapot does not backup/persist the state of remote objects, so in most cases, to ensure applications behave as expected, renewable classes should be stateless, as the renewed object's state will be unaware of any changes to the original object's state. The developer may wish to create a stateless object remotely because it computes some complex function, or acts as a connector to a database, or downloads all data from the web, for example. If semantics are consistent even after state is reset then declaring the class to be renewable is also valid. For example, a class may use an in-memory cache to improve the performance of future method invocations, but will still work even if the cache is cleared.
- Renewable objects will *always* be renewed on another remote system, if one is known, or on the local system if no other remote system is known.
- Although renewable objects will be renewed on another system, if the local system is blocked waiting for the return value from a non-renewable object on the unreachable system, then the system will continue waiting for the return value. The whole point of renewable objects is to allow a blocked thread that is waiting on the return value from a method invocation on a *renewable* object to resume execution, but if the thread is waiting on the return value from the method invocation on a non-renewable object, the

thread will continue to block until the system, and therefore the object, becomes reachable again.

- Renewable objects will only be renewed if they were created by calling `DistributedCommunicator.runRemotely`, and the renewed object will be created using the `CopiableSupplier` used to create the original object. Therefore, if the `CopiableSupplier` captures any long references to objects on the unreachable system, then the object cannot be renewed until the system is reachable again.

If any method invocations or changes should be applied to the object upon renewal, these should be performed in the `CopiableSupplier` used to construct the original object. For example, if the object should have a flag set before any method invocations, a developer should write the following:

```
DistributedCommunicator.getCommunicator()
    .runRemotely(() -> {
        MyClass x = new MyClass();
        x.setFlag(true);
        return x;
    }, serverAddress);
```

rather than

```
MyClass x = DistributedCommunicator.getCommunicator()
    .runRemotely(MyClass::new, serverAddress);
x.setFlag(true);
```

In the former case, the renewed object will have its flag set to true, but in the latter case the object will have the default value of the flag because future method invocations are not recorded and replayed.

4.3.2 Design

To declare that a class is renewable, a developer should declare that the class implements the empty interface `Renewable`. When a long reference to an object is created using the `DistributedCommunicator.runRemotely` method, Mokapot will check if the object implements the `Renewable` interface. If the object is indeed renewable then the `CopiableSupplier`, the long reference and the host address are registered with `PingAndRecoveryManager`. If, in the future, a system is determined to be unreachable, all renewable objects on that system will be renewed. Renewing an object involves three steps:

1. Re-running the `CopiableSupplier` used to construct the original object on the new system. If the new system is remote then the code is executed remotely using the `DistributedCommunicator.runRemotely` method. Otherwise the new object is constructed by invoking the `CopiableSupplier.get` method on the local system, thereby obtaining

a short reference to the object. `PingAndRecoveryManager` knows which renewable objects are on which systems because they are registered by `DistributedCommunicator` when they are created, and it knows how to reconstruct the objects because the `CopiableSupplier`'s are also recorded.

2. Any outstanding method invocations, i.e. method invocations on which the local system is blocking because the return value has not been given, must be re-executed on the renewed object so that the local system's thread may resume execution. If the documented restrictions of renewable objects are followed, the semantics should be unchanged by these *re*invocations. Each time a method is invoked on a long reference to a renewable object the following information is recorded:

   - The ID of the thread
   - The method being invoked
   - The arguments to the method being invoked

   Once the long reference has the return value, the method invocation is no longer outstanding and is therefore removed from the list of those method invocations to be reinvoked in the case of renewal. A reinvocation will not take place on the *actual* thread that performed the invocation, since it is blocked waiting for a response, but instead by one of `PingAndRecoveryManager`'s threads. Although the *actual* thread is not the same as the thread that executed the original invocation, the thread ID used in the reinvocation process will be the ID the original thread, not the ID of the thread reinvoking the method.

3. Re-routing future invocations on the long reference to the new system. The application should be able to continue using the long reference as if were a short reference, and so only the underlying behaviour should change, i.e. the application shouldn't be responsible for obtaining a new reference.

5 Implementation

In this section we will discuss some of the challenging aspects of this project's implementation. Extending an existing project rather than starting from scratch is also challenging because the new features must be compatible with the existing ones, conventions must be followed, and one must understand the existing project quite well to navigate the codebase. In particular, for a distributed computing library, it is important to ensure thread safety, to maintain garbage collection behaviour, etc.

5.1 Module Splitting

Several code blocks in the core Mokapot library have been refactored into separate classes. For example, Mokapot checks the following condition to determine if a class `about` is actually a lambda expression:

```
about.getName().contains("$$Lambda$")
```

There is a note to improve this check to use static analysis. In JVM-only setups this check will usually be sufficient because developers are unlikely to include `$$Lambda$` in their class names, and, unless they are using a library such as Retrolambda, the only classes that should satisfy the condition should be actual lambdas. However, as explained in section 3, Android transforms and removes lambdas in the build process, and so the condition above is not sufficient. There is no official way to determine whether or not an object is actually a lambda or just a class with a name that looks like a lambda at runtime [34], and so we need a different approach depending on the system.

Now, there is the core Mokapot library, *mokapot*, two system-dependent libraries, *mokapot-java* and *mokapot-android*, and an annotation processor library used by the system-dependent libraries, *mokapot-annotation-processor*. Places where Mokapot requires a system-specific implementation, the code will be refactored (using the extract method operation). The new class will have an empty implementation — the implementation will be provided by the other libraries. For example, the above condition to check if an object is a lambda is refactored to a method call: `LambdaAnalyser.isLambda`, where `LambdaAnalyser` is implemented (in Mokapot) as follows:

```
abstract class LambdaAnalyser {
    private static LambdaAnalyser implementation;
    static boolean isLambda(Class<?> about) {
        return implementation.isLambdaImpl(about);
    }
    abstract boolean isLambdaImpl(Class<?> about);
}
```

Mokapot is responsible for injecting the appropriate implementation by setting the `implementation` field at runtime. Similarly, proxy creation has been refactored into methods in an abstract class `ProxyCreator`, and instead of checking to see if a security manager is installed, Mokapot now calls the method `SecurityAccess.checkAccess`, where *mokapot-java* checks for a security manager and *mokapot-android* does nothing.

The JVM-specific implementation of `ProxyCreator`, for example, is called `JavaProxyCreator` and is annotated with the `@SystemSpecific` annotation, which is supplied by the *mokapot-annotation-processor* library. When building *mokapot-java* or *mokapot-android*, the `SystemSpecificAnnotationProcessor` will be provided with the classes that are annotated with the `@SystemSpecific`

annotation. At compile time, this class will generated a new class called `SystemSpecificImplementationInjector`, which has a single method called `injectSystemSpecificImplementations`. This method will set the `implementation` field of each of the superclasses of the `@SystemSpecific` classes. For example, the `implementation` field of `ProxyCreator` will be set to `new JavaProxyCreator()` in the *mokapot-java* library. Then at runtime Mokapot can trigger the injection of these implementations by executing the following:

```
Class.forName(
    "xyz.acygn.mokapot.SystemSpecificImplementationInjector");
    .getDeclaredMethod("injectSystemSpecificImplementations")
    .invoke(null);
```

Reflection is necessary here because the class is not available in the core Mokapot library but is generated, and we cannot trigger any code in *mokapot-java* until one of its classes are incarnated. Typical classpath-scanning libraries, such as Spring [62], are not compatible with the Android runtime, and classpath scanning on Android is an expensive operation [71], so the approach used was necessary to achieve an appropriate level of generality and performance.

Since Mokapot uses Javassist to create proxies — not Java's built-in proxy creation tools — this project must use an Android-compatible proxy creation tool with the same proxy-creation features as Javassist. For this purpose, Byte Buddy and Dexmaker are very similar. Furthermore, both libraries are licensed under Apache Licence 2.0, and are available through Android's build management system (Gradle). We have chosen Dexmaker, as its style of usage is very similar to Javassist's. *mokapot-java* will continue to use Javassist, but *mokapot-android* will use Dexmaker.

5.2 Deadlock Recovery

The system which initiates a connection with another system establishes a connection by creating a socket and then sends a `ReverseConnectMessage`, which tells the other system to only use the socket created by the initiating system (the system may not be publicly routable, for example, in which case the other system has no other way of communicating with it). However, the thread establishing the connection holds the monitor for connections to remote systems, and so until the remote system responds to the `ReverseConnectMessage`, the current system cannot communicate with any other system. This is a problem for the `PingAndRecoveryManager`, which must have the ability to cancel running tasks so that it can establish new connections, resend messages, etc. The `PingAndRecoveryManager` cannot just interrupt the thread holding the monitor and then attempt to acquire the monitor itself, because if other threads are also blocked waiting for the monitor then they might obtain the monitor first and the `PingAndRecoveryManager` will block waiting for the monitor, likely resulting in

a deadlock situation.

To recover from/avoid deadlocks, this project implements a new type of lock called a `ForcefullyAcquirableLock`. This lock has similar semantics as the `Semaphore` class in the Java library, but with two key additions: the methods `isActiveThread` and `acquireForcefully`. Threads that previously synchronized on the connections to other systems instead acquire and then release the lock. Additionally, before the thread performs a "final" operation, e.g. an assignment after a blocking operation that could have caused a deadlock, it must check to see if it still holds the lock, as it may have been acquired forcefully by another thread. If the thread still holds the lock it should continue as normal, otherwise it should exit without modifying any state. A thread can forcefully acquire the lock, which will also interrupt the thread, if any, that holds the lock. This avoids the deadlock scenarios described above since the `PingAndRecoveryManager` can forcefully gain access to the critical sections, and can interrupt itself so ensure it does not block forever.

5.3 Recovery

The `PingAndRecoveryManager` class uses a `ScheduledExecutorService` in order to schedule the transmission of ping messages to remote systems, with a fixed delay between each round trip. Each ping message is timed, so that if a response is not received within a specific period of time, the thread blocking on the response is interrupted and the current system places the remote system in recovery mode. Systems can also be placed in recovery mode if an exception is thrown when attempting to send a message to the remote system. Once an address is in recovery mode, the next scheduled event is to initiate the recovery process with the system from which a response was not received. First, the thread forcefully obtains the lock for the collection of connections to remote systems. Next, it sends a `SynchronisationMessage` to the remote system, which contains information about threads on the local system which last sent a message to the remote system. The remote system calculates its reply, then sends a `SynchronisationMessage.CounterpartInfo` message with information about which threads have lost messages and includes its own information about threads that last sent to the local system. The time taken to receive a response to the `SynchronisationMessage` is measured. For the sending of this message, we must use the thread that forcefully acquired the lock for the collection of connections, otherwise the thread will not be able to send the message. This means we must use a separate thread to interrupt the thread sending the message. If the request finishes promptly, the thread with the lock can cancel/interrupt the supervising thread, otherwise the supervising thread should interrupt the thread waiting for a response:

```
Thread requesterThread = Thread.currentThread();

final Future<?> interrupter = pool.submit(() -> {
```

```
    try {
        Thread.sleep(unit.toMillis(timeout));
    } catch (InterruptedException e) {
        // The request must have completed quickly enough
        return;
    }

    // The request was too slow, so interrupt to cancel it
    synchronized (PingAndRecoveryManager.class) {
        requesterThread.interrupt();
        LOGGER.log(Level.SEVERE, "Ping/recovery task timed out.");
        throw new RuntimeException("Ping/recovery task timed out.");
    }
});

final T result;
try {
    result = supplier.get();
} catch (Throwable t) {
    // We don't want the interrupter to interrupt us in the future,
    // so cancel now and rethrow the exception
    interrupter.cancel(true);
    throw t;
}

// We have finished quickly enough (since we weren't interrupted),
// so stop the interrupter and return the result
synchronized (PingAndRecoveryManager.class) {
    interrupter.cancel(true);
    return result;
}
```

Next, the local system sends an asynchronous `RequestResendMessage` message, informing the remote system of which messages have not been received. In processing this message, the remote system will resend (asynchronously) the appropriate messages on the specified threads. The local system then asynchronously resends the messages that were not received by the remote system. If no exceptions are thrown and all requests finish promptly, the address will then leave recovery mode and the local system will begin sending ping messages as normal.

The messages above contain lists of thread IDs. List implementations, e.g. `java.util.ArrayList`, are usually `NonCopiable`, so when a remote system obtains a reference to the list, it is a long reference. Since the remote system must access every element of the list, to reduce network traffic this project implements a `CopiableList`, which is an immutable list that is `Copiable`, so remote systems get the entire contents of the list when it is sent, rather than just a long

reference.

5.4 Renewable Types

`LocationManager` has a static field declared as follows:

```
Map<GlobalID, Map<Thread, InvocationInformation>>
        lastInvokedRenewableMethod = new ConcurrentHashMap<>();
```

This map is used to record information about invocations on renewable objects, so that if an object is renewed on another system, any invocations on that object that have not received a response can be reinvoked on the new object. The map is keyed on the `GlobalID` of the renewable objects, so that for a given object ID we can use the `Map.get` method to find all invocation information about that object. Since the object may have been invoked by more than one thread, the values of this map are maps keyed on threads. So, for a given object and thread, we can identify the outstanding invocation. In the course of processing a method invocation for a particular object on a particular thread, further invocations may occur. The renewal process only needs to know the original invocation that triggered the sequence of invocations on the object, and so we do not override the value with the intermediate invocations. When reinvoking a method, the `GlobalID` of the thread which originally invoked the method is provided so that the object's `LocationManager` can send an asynchronous message which specifies the thread ID for the other system to use.

Usually, if the object managed by a `LocationManager` is held locally, the method will be invoked on the object directly and the results returned. However, when reinvoking a method we must re-trigger Mokapot's message handling so that the original invocation will actually receive a response (since the original thread will be blocking waiting for a response, from its perspective, over the network). Future invocations on the renewed object will short-circuit and target the locally held object, but the initial "reinvocations" are really resending the messages that were sent in the initial invocation. This means that if the destination address is equal to the current address, Mokapot should directly invoke `DistributedCommunicator`'s `handleArrivingMessage` method, which will take the thread ID of the message and carry on as normal.

6 Testing

This project has extended Mokapot's existing test suite to include acceptance tests that simulate unstable network conditions, under which Mokapot should successfully recover connections and continue execution. Assertions are made in the tests to ensure that a consistent state is reached after the various network errors, e.g. if a remote object is initialised with a value of 0 and then incremented 20 times, the final value should be 20. If not, perhaps a method was

invoked twice or not at all, which would suggest that the recovery protocol implemented is not correct. To simulate unstable network conditions, we will use the `ChaosMonkey` class, which was created in the spirit of the "Chaos Monkey" tool created by Netflix, which randomly disables production instances to ensure that their system is fault tolerant and helps identify potential weaknesses for future outages [69, 51]. Our `ChaosMonkey` will periodically perform a chaotic action, such as the following:

- Close the input/output stream of a socket
- Change a socket's timeout value
- Silently shut down an input/output stream of a socket
- Swallow any data Mokapot attempts to write to the output stream of a socket

We can configure how often these chaotic actions will happen, which means we can simulate a very unstable network where connections are constantly dropped.

We will also use the `UnpluggableDataOutputStream` class, which implements the `DataOutput` interface, but can be "unplugged". Every method invocation after calling the `unplug` method will do nothing and return, which allows us to simulate a scenario in which one of the two system's IP address has changed, since the data written to the output stream will never actually reach its expected destination.

6.1 Test Cases

Class name: ClientToServerTest
Test name: clientToServerRandomNetworkInstability
Test description: The client remotely creates a `Holder` object which initially holds an integer value of 0. The client invokes the `map` method on the long reference 20 times, incrementing the value each time: `x -> x + 1`. A `ChaosMonkey` performs a chaotic action every six seconds. The client waits one second between each invocation. At the end of the test, the test asserts that the value held by the `Holder` object is 20.
Reason: The wait time between each invocation ensures that the `ChaosMonkey` will tamper with the connection a few times during the test. Mokapot will be forced to deal with socket errors and will reconnect with the remote system and execute the recovery protocol. This test ensures that Mokapot can establish a new connection and synchronise with the other system to ensure (in this case) code is executed precisely once on the remote system.

Class name: ClientToServerTest
Test name: clientToServerMissingMessages
Test description: Same as above but the chaotic action is always the "unplugging output stream" action, which means all data written to the output stream

is ignored. This simulates a scenario where one of the system's IP address has changed.
Reason: Same as above, but ensures Mokapot will recover even if data transmission is failing silently.

Class name: ServerToClientTest
Test name: serverToClientRandomNetworkInstability
Test description: Same as ClientToServerTest.clientToServerMissingMessages, but the `Holder` object is on the client, and the modifications are executed on the server with the `DistributedCommunicator.runRemotely` method.
Reason: With the above tests the client could potentially detect an error without using ping messages, since the requests could be timed, exceptions could be detected, etc. With this test, the test code is not requesting any computations to be run remotely. Other than the rare scenario in which the connection is tampered with immediately before the client responds, the client can only detect a problem connection by sending ping messages (and the remote system does not have a `PingAndRecoveryManager`).

Class name: ServerToClientTest
Test name: serverToClientMissingMessages
Test description: Same as above but the chaotic action is the "unplugging output stream" action.
Reason: Same as above, but ensures Mokapot will recover even if data transmission is failing silently.

Class name: ClientToFromServerTest
Test name: clientToFromServerRandomNetworkInstability
Test description: The client creates local and remote `BackAndForth` objects, and sets the remote object as the partner of the local object and vice versa. The client invokes the `count` method with a random integer between 1 and 50 on the local object on the even iterations and the remote object on the odd iterations. The test asserts that the return value of each invocation is correct.
Reason: This tests the correctness of the handshake process. The systems are involved in many back-and-forth communications, and neither system knows which system sent the last message and how to resume execution correctly so that the systems are consistent. The recovery will take place at different levels of recursion, and sometimes the client will be responsible for resending its last message and sometimes the server will be responsible.

```
private static class BackAndForth implements NonCopiable {
    private BackAndForth partner;

    void setPartner(BackAndForth partner) {
        this.partner = partner;
    }

    int count(int times) throws InterruptedException {
        if (times == 0) {
```

```
            return 0;
        }
        Thread.sleep(100);
        return 1 + partner.count(times - 1);
    }
}
```

Class name: ClientToFromServerTest
Test name: clientToFromServerMissingMessages
Test description: Same as above but the chaotic action is the "unplugging output stream" action.
Reason: Same as above, but ensures Mokapot will recover even if data transmission is failing silently.

Class name: RenewableTest
Test name: renewObjectRemotely
Test description: Two servers are running, and the client creates a renewable object on one of these servers. We shut down the server holding the renewable object, triggering the recovery/renewal process. The client then checks that the object has been renewed on the other server and that method invocations on the original long reference work as expected.
Reason: To check that Mokapot will renew objects on another server when the host system becomes unreachable.

Class name: RenewableTest
Test name: renewObjectLocally
Test description: Same as above but only one server is known to the client. The client asserts that the object is renewed on the local system (rather than on another server).
Reason: Same as above, but ensures that renewable objects are renewed even if no other remote systems are available.

6.2 Verification

Software verification involves asking the question "are we building the system right?" This project has achieved its goals using the intended techniques, including using ping messages to identify suspicious connections, a handshake recovery protocol to ensure system consistency, and, most importantly, porting Mokapot to the Android platform to provide distributed computing abilities. The proof of correctness presented in the design section and the automated acceptance tests verify that the software behaves in the intended manner and achieves the goals in the correct way. The existing unit tests verify that regressions are not introduced.

6.3 Validation

Software validation involves asking the question "are we building the right system?" In section 8 we discuss sample applications in which we manually test and observe Mokapot's behaviour in unstable network conditions, and check that it is able to recover lost messages. We also observe that renewable objects are renewed on another system if a system is unreachable. And, of course, all extensions and features are suitable for the Android platform.

7 Project Management

Throughout the project I had regular meetings with my supervisor, which helped me stay on track and stay focused on achieving my goals. Outside meetings, I would post updates in the messaging channel for Mokapot, which meant I could get feedback from Mokapot's main author about whether my changes might interfere with the existing implementation, or if it could be more elegantly handled in the core Mokapot library. This meant we could avoid conflicts and duplicate work, where I could focus on the changes relevant to my project, and changes that would benefit Mokapot's existing features and support this project could be prioritized.

The project was version controlled with Git, and updates were regularly committed, with descriptive log messages to help write this report. Furthermore, all design decisions and thought processes were recorded in a logbook so that I could fully articulate (and remember) the reason for each major decision made.

8 Results and Evaluation

In the analysis and specification section we specified the following functional requirements:

- **FR1**: Potential network errors should be actively detected.
- **FR2**: When a potential network error is detected, a recovery protocol should be initiated.
- **FR3**: If the local system determines that a remote system is probably unreachable, any renewable objects on the remote system should be renewed on another system.

and non-functional requirements:

- **NFR1**: The additions made to Mokapot in this project should make it easy for a developer to integrate Mokapot with their Android applications. Furthermore, the developer should find the library transparent from the

point of view of managing network resiliency (and of course from the point of view of distributed computation, since this is what Mokapot provides).

- **NFR2**: The library should be easy to deploy and integrate into the build process.
- **NFR3**: The library should improve the execution time for heavy computations.

Ideally actual developers and users would use the software and provide feedback so that this evaluation could use real-world users' opinions. Unfortunately, due to specific hardware requirements, setup required, privacy issues for the research project's source code, requirements for a private server, etc., this is not feasible, and so this evaluation will be restricted to evaluating those criteria that can be measured reasonably objectively. Since the purpose of this project is to port Mokapot to the Android platform, rather than build a new software system with a graphical user interface, we will present the results of the work by looking at how the software behaves in a simulated unstable network.

8.1 Comparison with Specification

The design, implementation and testing sections describe how the functional requirements have been met. Here, we evaluate how the non-functional requirements have been met.

8.1.1 NFR1

The first non-functional requirement is to ensure easy integration of this project with Android applications. We will show that the steps required to integrate Mokapot with an Android application do not place much of a burden on the developer and are typical steps necessary when adding any build dependency [4]. The typical Android developer workflow involves setting up the workspace, writing an app, then building and running [27]. Android Studio uses Gradle for dependency management; this means that adding this project as a dependency to a developer's project is as easy as adding the following line to their `build.gradle` file:

```
implementation 'xyz.acygn:mokapot-android:1.0-SNAPSHOT'
```

Effectively this adds the `mokapot-android` library (which includes Mokapot's core functionality) to the application's classpath. Next, so that Dexmaker can physically write class files for proxy creation to a private directory, the following line must be added:

```
System.setProperty("dexmaker.dexcache",
        .getApplicationContext().getFilesDir().toString());
```

The developer can then use Mokapot in their application by starting a `DistributedCommunicator`, looking up a remote system and then requesting remote execution.

To summarise: to integrate this project with an Android application, the developer must add *mokapot-android* as a build dependency and then set a configuration property.

8.1.2 NFR2

The second non-functional requirement to ensure the project is easy to integrate into the build process and easy to deploy. Once the developer has added *mokapot-android* as a build dependency, the only other addition which may be necessary is Retrolambda, which allows the developer to backport other dependencies to earlier target versions of Java. Otherwise, assuming the `minSdkVersion` is at least 26, the mobile application can be built as expected. In order to deploy the application from the server side, the class files compiled during Android's build process must be copied to the server. This can be automated by creating a new Gradle task which bundles the correct class files into a JAR file, which can be copied to the server:

```
task createJar(type: Zip) {
    from('build/intermediates/transforms/desugar/debug/0')
    extension = 'jar'
    include('**/*.class')
    classifier('android')
}
```

This task can be run manually within Android studio, from the terminal with `./gradlew createJar`, or can be run after every build [39]. The server can then run the application with these class files on its classpath with a command similar to the following:

```
echo "$password" | java \
    -Djava.security.manager \
    -Djava.security.policy=security.policy \
    -cp mokapot.jar:android-application.jar \
    DistributedServer endpoint.p12 15238 123.123.123.123
```

The Android application can be be assembled, installed and run in the usual fashion. This demonstrates that most of the build and deploy process is, or can be, automated and adds only a few extra steps compared with device-only applications/systems.

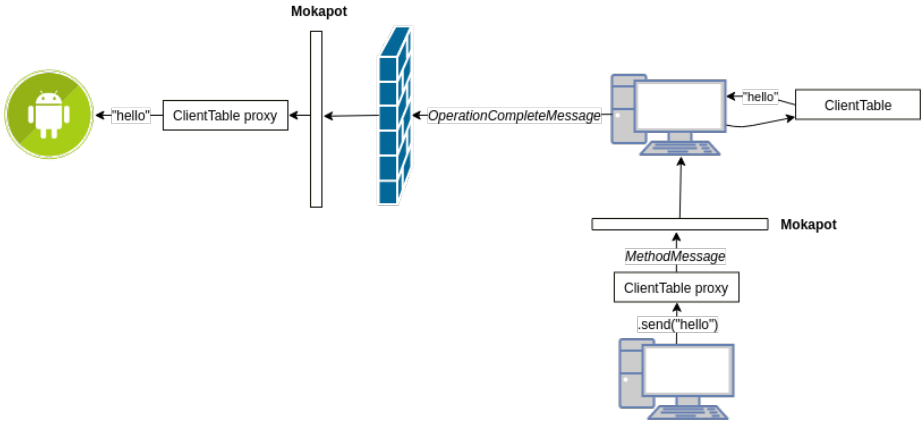

Figure 3: Architecture of messaging demo application using Mokapot.

8.1.3 NFR3

The third non-functional requirement is that the library should improve the execution time for heavy computations. See section 8.3 for a discussion about the performance of this project on an Android device.

8.2 Robustness and Reliability

We can evaluate the functional requirements by executing the project in real-world and simulated environment. To demonstrate how these requirements are met in this project, we will discuss the observed behaviour in several sample applications in which we simulate network connection problems.

For one of the sample applications, to simulate network connection problems we will simply terminate Mokapot on one of the servers. In the other sample application, and in the tests, we use a class called `ChaosMonkey` to simulate unstable networks (see section 6 for more information). The first sample application we will discuss is a messaging application, in which clients can send messages to each other through a middle Mokapot server. Each client obtains a long reference to a `ClientTable` object which resides on the server system. Each client has a username. To send a message to another client, the application invokes the `ClientTable.sendMessage` method on the `ClientTable` long reference, specifying the recipient and the message to send. The server will then place the message on the message queue for the recipient. For a client to read its messages, it should repeatedly invoke `ClientTable.getMessage`, which will block until the client's message queue has a message. Figure 3 illustrates the architecture of the messaging application.

The communicator on each client has a `PingAndRecoveryManager` which re-

quires responses to network requests within 2 seconds:

```
new PingAndRecoveryManager.RecoveryConfig(2, TimeUnit.SECONDS)
```

and we employ a chaos monkey which will perform a chaotic action every two seconds:

```
comm.setChaosMonkey(new ChaosMonkey(1.0, 2000, seed, null));
```

Then the two clients begin sending messages to each other. Observing the applications' behaviour we see that every few seconds the applications stop receiving messages (because the chaos monkey has tampered with the network connection), but a few seconds later every single message in the backlog of messages sent by the other client are displayed in the messaging window. This demonstrates that even under extremely unstable conditions the network recovery mechanisms and strategies are able to reconnect to the remote system and recover any lost messages. Below is an excerpt of the log messages captured during execution from one client:

```
CHAOS MONKEY: closing input stream
Sending ping message to localhost/127.0.0.1:15238
Exception encountered while sending message to localhost/127.0.0.1:15238.
Will now place address in recovery mode.
Socket is closed
Ping/recovery task timed out.
Ping to localhost/127.0.0.1:15238 failed or timed out. Address is now in
    ↪ recovery mode.
Closing communication with localhost/127.0.0.1:15238
Finished closing communications with localhost/127.0.0.1:15238
Attempting recovery for localhost/127.0.0.1:15238
Establishing new connection with localhost/127.0.0.1:15238
Registered new receive connection for localhost/127.0.0.1:15238
Handling message on thread 3@localhost/127.0.0.1:15240 with sequence number 2
Sending message on thread 3@localhost/127.0.0.1:15240 with sequence number 3
Handling message on thread 3@localhost/127.0.0.1:15240 with sequence number 4
Recovery for localhost/127.0.0.1:15238 complete, address is now in ping mode.
```

The next sample application demonstrates how Mokapot now supports renewable types. We will have three systems involved: one client and two servers. The client will create two renewable objects remotely, one on each system, and maintain long references to these objects. The objects are `ChangingColor` objects, which implement the `Runnable` interface, and their task is to change their colour every millisecond to an interpolated colour between configured start and end colours. The tasks performed by these objects represent long running tasks that a client may wish to execute remotely and on different systems, e.g. one system may have hardware to support machine learning while the other may be memory optimized. The client has long references to red and blue `ChangingColor` objects. The client application can interact with the objects by invoking the `ChangingColor.setMessage` method. Figure 4 illustrates the initial setup of this system.

To simulate a network failure, we will stop Mokapot on the system holding the red `ChangingColor` object. Mokapot will detect that the system is no longer reachable and will then attempt to renew its objects on the other system. The

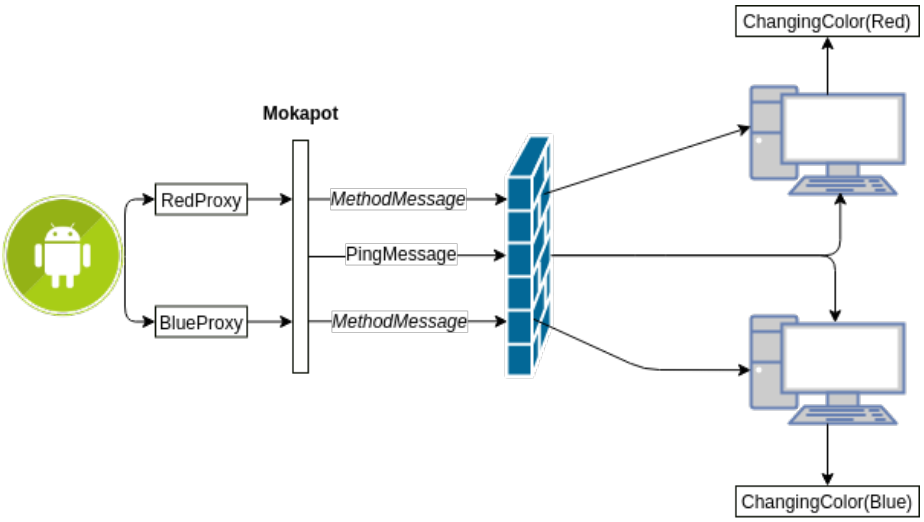

Figure 4: Android client application holds a long reference to a `ChangingColor` object on each system.

renewal is entirely transparent to the client. The client application can continue invoking methods on the existing long reference (to change the object's message, for example) and the task will now be running on the other system. Figure 5 shows the state of the system after Mokapot has renewed the red `ChangingColor` object on the other system.

This sample application demonstrates that the third functional requirement is met and allows distributed applications to recover renewable objects.

8.3 Performance

We will demonstrate that using Mokapot on a mobile device has the potential to improve performance and save energy/extend battery life. We will write an application which sorts 50,000 lists, each with 50,000 elements, ten times. In one experiment, this will be executed locally on an Android device, and in another we will use Mokapot to create the task runner remotely and therefore the tasks will be executed remotely. The experiment will run on a Google Pixel 2 running Android Oreo 8.0 with the following specifications:

- 4 GB LPDDR4x RAM
- 2.35 Ghz + 1.9 Ghz, 64 Bit Octa-Core

We will use a server in the cloud with the following specification

- 1GB RAM

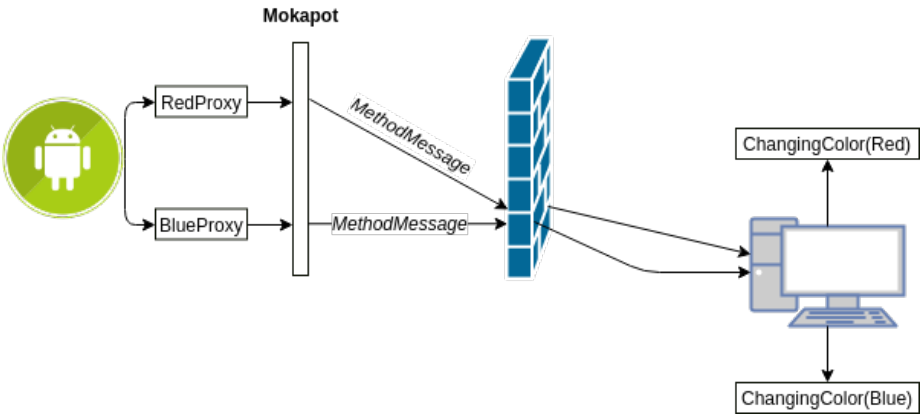

Figure 5: `ChangingColor(Red)` is renewed on the other system, and client's long references now point to two objects on the same system.

- Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz

We will fully charge the device before each experiment and measure the time taken to complete the experiment and the battery capacity at the end of the experiment. In order to observe Mokapot's overhead, we will run a third experiment in which the device is idle with Wi-Fi enabled and the screen is on. Figure 6 shows the time taken and the end battery capacity in the three experiments. As the graph illustrates, the experiment in which Mokapot is used to offload computation finishes 5 minutes earlier and with 7% more battery capacity than the local experiment. We can see that the overhead of running Mokapot over 44 minutes results in an additional 0.54% reduction of battery capacity — effectively negligible.

8.4 Comparison with Existing Work

By extending Mokapot, this project supports remote state, which is a feature not present in many of the projects discussed in section 2. Although it does not use the same static and dynamic analysis techniques as CloneCloud [16], Spectra [10, 32] and MAUI [23], the high level of transparency afforded by Mokapot means the developer has similar experience of transparency. Through *renewable* types we get similar features as the Cuckoo project, which manages a set of remote resources/systems for computation, and the network resiliency strategies discussed above match some of the the fault tolerance guarantees of Mozart. In contrast with simple web APIs, porting Mokapot to the Android platform allows developers to write distributed applications with type information, callbacks, management of remote state, and network recovery protocols.

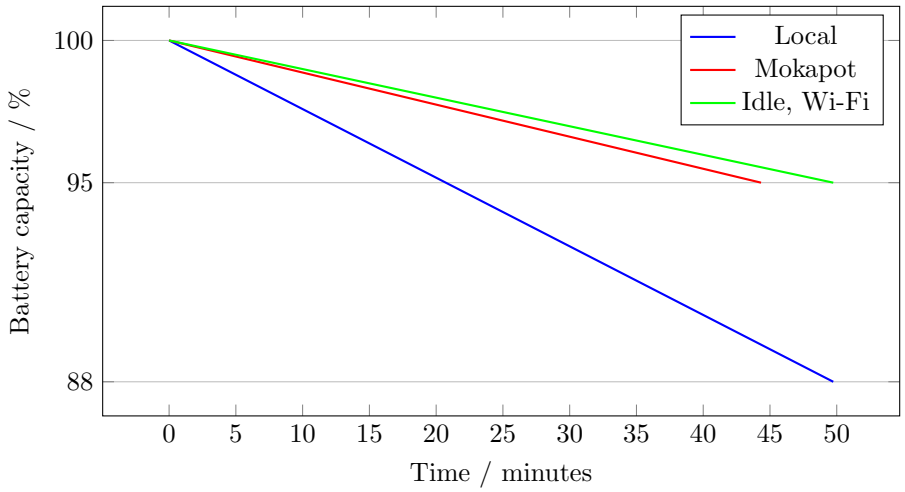

Figure 6: Graph showing the battery capacity of the Pixel 2 with local vs remote execution.

9 Discussion

In this project Mokapot has been successfully ported to the Android platform. It was not self-evident that this would be possible — security restrictions, bytecode differences, availability of proxy libraries, etc., placed doubt on the success of porting Mokapot to Android. The final software product maintains the high level of transparency found in Mokapot, but extends the features under Mokapot's management to include network resiliency mechanisms and strategies. This opens up the opportunity for developers to write stateful distributed applications for the Android platform with the confidence that Mokapot will manage system consistency in unstable network conditions.

However, reflecting on what has been accomplished reveals some deficiencies. For example, although the main advantages of Mokapot's transparent programming style are preserved, the differences between the Android runtime and the JVM force the developer to architect their solution with wrapper classes and extract methods into new classes. For example, if the remote object imports a class that is not available on the Android platform then it must wrap this functional or introduce an intermediate class so that this incompatibility is not a problem. Similarly, if Android application code to be executed remotely captures an object of a class that is not available in the JVM (usually the `this` pointer) then the code must be refactored, usually the call site must be moved to a different class. Mokapot's partner project *Millr* could be used for this purpose. Millr can rewrite application code to ensure that it is compatible with Mokapot. This could be extended to automatically refactor the user's code to avoid the problems mentioned.

The renewable types introduced in this project give applications a better chance of resuming execution in unstable networks. However, the properties that a class must satisfy in order to be renewed successfully are quite restrictive. For some applications, renewable types may provide the resiliency enhancements an application needs, while others may find that this feature is not suitable for their application. Renewable types, and the recovery features in general, could be improved by implementing additional fault tolerance features. For example, Mokapot could support distributed transactions, where operations are indivisible and the results of execution must be stored on several systems before continuing. If a system becomes unreachable then the other systems can surface the backup data and continue communicating, with invocations on long references re-routed to the new system holding the remote objects.

Although the aim was to implement fully transparent management, application developers don't even have the *option* to inspect the state of Mokapot. For example, an application developer cannot display a useful message to the user if Mokapot failed to reconnect with a remote system a certain number of times. Additionally, the developer has no control to specify a preferred system for renewal; renewable objects will always be renewed even if the user does not want this to happen because, for example, the application will use more battery on the local device.

It was mentioned in the analysis and specification section that Android's `Object` class has hidden fields, apparently used to optimize the garbage collection process. These fields are ignored by Mokapot, and so any garbage collection optimizations will no longer work. Ideally, Mokapot would handle these fields in the correct manner, and would populate them correctly so that Android's garbage collection works just as if Mokapot were not being used. Since the usage of these fields was not clear at the time of writing, we have assumed that they are purely an optimization and, based on commit messages, are disabled by default.

Looking to the future, there are many extensions to this project, and in fact work is still being done on the core Mokapot library. The sample applications presented in the evaluation section used a QR code scanner to scan the QR code displayed by the server application and decode this to obtain the server's IP address, taking inspiration from the Cuckoo project. Other than hard-coding the IP address, techniques like this are the only elegant way of obtaining the IP address of remote systems from the client device. The Android platform allows applications to access services on the local network through Network Service Discovery (NSD) [73]. By adding NSD to Mokapot, systems running Mokapot could be used as kiosks deployed in public areas, where mobile applications could discover these systems and then offload computation to prolong their battery life and improve application performance. This would also be more elegant than scanning a QR code every time a device wanted to use a new system, especially if the system does not have a visual display. Ideas from Spectra could be incorporated here.

Android also supports Wi-Fi Peer-to-Peer interaction, which allows devices to

connect directly without an intermediate access point [75]. Mokapot provides an abstraction that allows developers to ignore low-level network programming and focus on application logic. By extending Mokapot to support this feature, developers who wish to write applications that use the Wi-Fi peer-to-peer feature can do so and avoid socket programming.

Elaborating on the idea of renewable types and the kiosk usage model, Mokapot could reinforce its fault tolerance by implementing load balancing of requests. Since the state of renewable types is not critical, it should not matter to the caller if the actual object that runs the requests is the different each time. This allows Mokapot to route requests to particular systems with the aim of balancing the load on a cluster of systems. This could also apply to `DistributedCommunicator.runRemotely` requests, which could be routed to the most appropriate system, instead of explicitly specifying the target system. Quasi-parallel execution could be achieved with renewable types by using ideas from the Cuckoo project.

As mentioned, only the latest versions of Android (API level 26+) are capable of running Mokapot. This means that developers cannot write or enhance applications that target cheap/old/spare devices, such as Haven [36], a security application, which would benefit most from computation offloading and energy savings because they have older hardware and batteries. The parts of Mokapot that require the latest Android API might be able to be rewritten to be compatible with earlier versions of Android, allowing more users and developers to use Mokapot.

One of the features to be implemented in the core Mokapot library is migration of objects, in which objects (and their state) are migrated to the system that uses them most. For example, a quadtree data structure can be used to efficiently detect collisions between objects in a 2D surface. Each quadrant of the quadtree could be managed by a different Mokapot system. Each system would be responsible for detecting the collisions in its quadrant, which means invoking methods on the objects in its quadrant. Once an object moves into a different quadrant, it should be migrated to the system responsible for that quadrant, which means there will be less network traffic because the system can invoke methods locally, rather than routing them over the network. Ideally, migration can be triggered manually or automatically based on which system uses an object most frequently. At the time of writing, the migration feature is not ready for use, and so there is no integration between the recovery strategy for renewable types and the migration strategy described above. Once the migration feature is fully implemented, the recovery of renewable objects should be reviewed and, if possible and appropriate, aligned with the migration strategy.

Conclusion

In this project we have implemented an extension of Mokapot which provides mobile application developers with the features necessary to support distributed mobile computing. We have evaluated its performance, behaviour and efficacy under several scenarios, demonstrating that it has the potential to assist distributed applications. There are many improvements that can be made, which have been discussed in section 9.

References

[1] URL: https://android.googlesource.com/platform/art/+/9d04a20bde1b1855cefc64aebc1a44e253 (visited on 31st Mar. 2018).

[2] URL: https://android.googlesource.com/platform/libcore/+/a7c69f785f7d1b07b7da22cfb9150c584ee143f4 (visited on 31st Mar. 2018).

[3] URL: https://android-review.googlesource.com/c/platform/libcore/+/83484 (visited on 31st Mar. 2018).

[4] *Add Build Dependencies*. URL: https://developer.android.com/studio/build/dependencies.html (visited on 2nd Apr. 2018).

[5] *Amazon Compute Service Level Agreement*. URL: https://aws.amazon.com/ec2/sla/ (visited on 27th Mar. 2018).

[6] *An Overview of RMI Applications*. URL: https://docs.oracle.com/javase/tutorial/rmi/overview.html (visited on 16th Nov. 2017).

[7] *Apache Ant*. URL: http://ant.apache.org (visited on 16th Nov. 2017).

[8] *Apache River*. URL: https://river.apache.org (visited on 15th Nov. 2017).

[9] *Application security*. URL: https://source.android.com/security/overview/app-security (visited on 31st Mar. 2018).

[10] Rajesh Balan et al. 'The case for cyber foraging'. In: *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM. 2002, pp. 87–92.

[11] Robert Battle and Edward Benson. 'Bridging the semantic Web and Web 2.0 with representational state transfer (REST)'. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 6.1 (2008), pp. 61–69.

[12] Bharat Bhargava and Shu-Renn Lian. 'Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach'. In: *Reliable Distributed Systems, 1988. Proceedings., Seventh Symposium on*. IEEE. 1988, pp. 3–12.

[13] *Byte Buddy*. URL: http://bytebuddy.net (visited on 30th Mar. 2018).

[14] Jing Cai et al. 'An adaptive timeout strategy for profiling UDP flows'. In: *Networking and Computing (ICNC), 2010 First International Conference on*. IEEE. 2010, pp. 44–48.

[15] *Chapter 8. Classes*. URL: https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.1.1 (visited on 6th Apr. 2018).
[16] Byung-Gon Chun et al. 'Clonecloud: elastic execution between mobile device and cloud'. In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 301–314.
[17] *Class SerializedLambda*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/SerializedLambda.html (visited on 30th Mar. 2018).
[18] *Collections and Schemas*. URL: https://guide.meteor.com/collections.html (visited on 8th Apr. 2018).
[19] Raphaël Collet and Peter Van Roy. 'Failure handling in a network-transparent distributed programming language'. In: *Advanced topics in exception handling techniques*. Springer, 2006, pp. 121–140.
[20] *Configure Your Build*. URL: https://developer.android.com/studio/build/index.html (visited on 27th Mar. 2018).
[21] *CORBA*. URL: http://www.corba.org (visited on 16th Nov. 2017).
[22] *CORBA*. URL: https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.aix.70.doc/diag/understanding/orb_corba.html (visited on 16th Nov. 2017).
[23] Eduardo Cuervo et al. 'MAUI: making smartphones last longer with code offload'. In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM. 2010, pp. 49–62.
[24] *Dalvik Bytecode*. URL: https://source.android.com/devices/tech/dalvik/dalvik-bytecode (visited on 31st Mar. 2018).
[25] *Dalvik Bytecode Generation*. URL: https://bravenewgeek.com/dalvik-bytecode-generation (visited on 31st Mar. 2018).
[26] *Data and File Storage Overview*. URL: https://developer.android.com/guide/topics/data/data-storage.html (visited on 31st Mar. 2018).
[27] *Developer Workflow Basics*. URL: https://developer.android.com/studio/workflow.html (visited on 2nd Apr. 2018).
[28] *Dexmaker*. URL: https://github.com/linkedin/dexmaker (visited on 16th Nov. 2017).
[29] Hoang T Dinh et al. 'A survey of mobile cloud computing: architecture, applications, and approaches'. In: *Wireless communications and mobile computing* 13.18 (2013), pp. 1587–1611.
[30] *Dynamic Proxy Classes*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html (visited on 31st Mar. 2018).
[31] *Eclipse*. URL: http://www.eclipse.org (visited on 16th Nov. 2017).
[32] Jason Flinn, SoYoung Park and Mahadev Satyanarayanan. 'Balancing performance, energy, and quality in pervasive computing'. In: *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*. IEEE. 2002, pp. 217–226.
[33] Christine Flood and Roman Kennke. Red Hat. URL: https://archive.fosdem.org/2016/schedule/event/shenandoah2016/attachments/

slides/1268/export/events/attachments/shenandoah2016/slides/1268/Shenandoah2016.pdf (visited on 31st Mar. 2018).

[34] Brian Goetz. *How to correctly determine that an object is a lambda?* URL: https://stackoverflow.com/a/23870800/5510565 (visited on 4th Apr. 2018).

[35] Brian Goetz. *Translation of Lambda Expressions The Java.* URL: http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html (visited on 30th Mar. 2018).

[36] *Haven: Keep Watch.* URL: https://guardianproject.github.io/haven/ (visited on 3rd Apr. 2018).

[37] Chris Hoffman. *Why Your Android Phone Isn't Getting Operating System Updates and What You Can Do About It.* URL: https://www.howtogeek.com/129273/why-your-android-phone-isnt-getting-operating-system-updates-and-what-you-can-do-about-it/ (visited on 28th Mar. 2018).

[38] *Interface Serializable.* URL: https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html (visited on 30th Mar. 2018).

[39] *Interface Task.* URL: https://docs.gradle.org/current/javadoc/org/gradle/api/Task.html#finalizedBy-java.lang.Object...- (visited on 2nd Apr. 2018).

[40] *Javassist.* URL: http://jboss-javassist.github.io/javassist (visited on 16th Nov. 2017).

[41] David B Johnson and Willy Zwaenepoel. 'Recovery in distributed systems using optimistic message logging and checkpointing'. In: *Journal of algorithms* 11.3 (1990), pp. 462–491.

[42] *Keeping Your App Responsive.* URL: https://developer.android.com/training/articles/perf-anr.html (visited on 1st Apr. 2018).

[43] Roelof Kemp et al. 'Cuckoo: A Computation Offloading Framework for Smartphones.' In: *MobiCASE*. Springer. 2010, pp. 59–79.

[44] James J Kistler and Mahadev Satyanarayanan. 'Disconnected operation in the Coda file system'. In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 3–25.

[45] Richard Koo and Sam Toueg. 'Checkpointing and rollback-recovery for distributed systems'. In: *IEEE Transactions on software Engineering* 1 (1987), pp. 23–31.

[46] Karthik Kumar and Yung-Hsiang Lu. 'Cloud computing for mobile users: Can offloading computation save energy?' In: *Computer* 43.4 (2010), pp. 51–56.

[47] Karthik Kumar et al. 'A Survey of Computation Offloading for Mobile Systems'. In: *Mobile Networks and Applications* 18.1 (Feb. 2013), pp. 129–140. ISSN: 1572-8153. DOI: 10.1007/s11036-012-0368-0. URL: https://doi.org/10.1007/s11036-012-0368-0.

[48] Zhiyuan Li, Cheng Wang and Rong Xu. 'Computation offloading to save energy on handheld devices: a partition scheme'. In: *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2001, pp. 238–246.

[49] Lavy Libman and Ariel Orda. 'Optimal retrial and timeout strategies for accessing network resources'. In: *IEEE/ACM Transactions on Networking (TON)* 10.4 (2002), pp. 551–564.

[50] Tim Lindholm et al. *The Java® Virtual Machine Specification. Java SE 8 Edition*. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html (visited on 31st Mar. 2018).

[51] *Netflix/chaosmonkey*. URL: https://github.com/Netflix/chaosmonkey (visited on 2nd Apr. 2018).

[52] Eugen Paraschiv. *Java in 2017 Survey Results*. URL: http://www.baeldung.com/java-in-2017 (visited on 30th Mar. 2018).

[53] Michael Philippsen and Matthias Zenger. 'JavaParty - Transparent Remote Objects in Java'. In: *Concurrency Practice and Experience* 9.11 (1997), pp. 1225–1242.

[54] Jon Postel. 'Transmission control protocol'. In: (1981).

[55] *RetroLambda*. URL: https://github.com/orfjackal/retrolambda (visited on 30th Mar. 2018).

[56] Simon Ritter. *4 Reasons Why Java is Still #1*. Ed. by Azul Systems. URL: https://www.azul.com/4-reasons-java-still-1/ (visited on 30th Mar. 2018).

[57] Alexey Rudenko et al. 'Saving portable computer battery power through remote process execution'. In: *ACM SIGMOBILE Mobile Computing and Communications Review* 2.1 (1998), pp. 19–26.

[58] *Run Apps on the Android Emulator*. URL: https://developer.android.com/studio/run/emulator.html (visited on 28th Mar. 2018).

[59] Mahadev Satyanarayanan. 'Pervasive computing: Vision and challenges'. In: *IEEE Personal communications* 8.4 (2001), pp. 10–17.

[60] *Save Files on Device Storage*. URL: https://developer.android.com/training/data-storage/files.html#WriteInternalStorage (visited on 1st Apr. 2018).

[61] *SecurityManager*. URL: https://developer.android.com/reference/java/lang/SecurityManager.html (visited on 1st Apr. 2018).

[62] *Spring*. URL: https://spring.io (visited on 9th Apr. 2018).

[63] Jeff Springer. *Always-Updated List of Phones That Will Get Android Oreo*. Ed. by Gadget Hacks. URL: https://android.gadgethacks.com/news/always-updated-list-phones-will-get-android-oreo-0181976/ (visited on 28th Mar. 2018).

[64] D. Steinkraus, I. Buck and P. Y. Simard. 'Using GPUs for machine learning algorithms'. In: *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*. Aug. 2005, 1115–1120 Vol. 2. DOI: 10.1109/ICDAR.2005.251.

[65] Rob Strom and Shaula Yemini. 'Optimistic recovery in distributed systems'. In: *ACM Transactions on Computer Systems (TOCS)* 3.3 (1985), pp. 204–226.

[66] *Stubs and Skeletons*. URL: https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-arch2.html (visited on 16th Nov. 2017).

[67] *TCP*. URL: http://man7.org/linux/man-pages/man7/tcp.7.html (visited on 27th Mar. 2018).
[68] *The Java Remote Method Invocation API (Java RMI)*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html (visited on 16th Nov. 2017).
[69] *The Netflix Simian Army*. URL: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116 (visited on 2nd Apr. 2018).
[70] *The Security Manager*. URL: https://docs.oracle.com/javase/tutorial/essential/environment/security.html (visited on 31st Mar. 2018).
[71] Tyler Treat. *Implementing Spring-like Classpath Scanning in Android*. URL: https://bravenewgeek.com/implementing-spring-like-classpath-scanning-in-android/ (visited on 4th Apr. 2018).
[72] *Use Java 8 Language Features*. URL: https://developer.android.com/studio/write/java8-support.html (visited on 30th Mar. 2018).
[73] *Using Network Service Discovery*. URL: https://developer.android.com/training/connect-devices-wirelessly/nsd.html (visited on 3rd Apr. 2018).
[74] Peter Van Roy. 'On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart'. In: *Fourth International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99), Tohoku University, Sendai, Japan, World Scientific*. 1999.
[75] *Wi-Fi Peer-to-Peer*. URL: https://developer.android.com/guide/topics/connectivity/wifip2p.html (visited on 3rd Apr. 2018).

10 Appendix

10.1 Directory Structure

The source code for the main Mokapot project, and the extensions introduced in this project, are in the `src/main/java` folder, and the tests are in the `src/test/java/xyz/acygn/mokapot` folder. The *mokapot-java*, *mokapot-android*, and *mokapot-annotation-processor* modules are in the top-level directory. The dependencies for each of these modules is declared in their `pom.xml` files. The *android-demos* folder contains several demo applications for Android, including *factorial-demo*, *messaging-demo*, *migrationdemo*, and *performance*. The corresponding server-side applications for the Android demos are in the *mokapot-messaging-demo* and *mokapot-migration-demo* folders. Classes annotated with "@author Kelsey McKenna" are a good place to start to see the changes made in this project, `git blame` can be used to find changes made in other classes, such as `DistributedCommunicator`

A Mokapot `DistributedCommunicator` requires a whitelist of systems per-

mitted to communicate with it. You can create a whitelist using the scripts in the *scripts* folder. First, create a whitelist controller with the `makewhitelistcontroller.sh` script, and then create a whitelist for each system that will run Mokapot using the `addwhitelistentry.sh` script. Place each whitelist in a location accessible to each Mokapot system, e.g. use the `scp` command to copy the file to the servers that will run Mokapot.

10.2 Building Modules

We use Maven and Gradle for build and dependency management. To build Mokapot, simply run `maven clean install` from the command line. You can then build *mokapot-annotation-processor*, *mokapot-java*, *mokapot-android*, *mokapot-messaging-demo*, and *mokapot-migration-demo* (in that order) in a similar way. The build order is important because some these modules depend on others. You must change directory into the module before running the Maven build command.

There are three Android demos: *factorial-demo*, *messaging-demo*, and *migrationdemo*. There is also a performance module called *performance*. Each of these modules is in the *android-demos* folder, and can be built by changing into the directory of the module and running `../gradlew assembleDebug`, or `../gradlew installDebug` if you wish to build and install the application on the connected device or emulator. You can execute the *createLibraryJar* task to build a JAR file that can be added to the classpath of a Mokapot server with the `../gradlew createLibraryJar` command; the resulting file can be found in the *build/distributions* folder. Each application requires a Mokapot server to be running with the distribution JAR on the server's classpath.

10.3 Running Mokapot

To run a "Mokapot server", use the following command from the *mokapot-java* module:

```
java \
-Djava.security.manager \
-Djava.security.policy=security.policy \
-cp target/mokapot-server-jar-with-dependencies.jar:OTHERJAR.jar \
xyz.acygn.mokapot.DistributedServer \
WHITELIST.p12 -d 15238 123.123.123.123
```

replacing `security.policy` with an appropriate security policy (examples can be found in the *src/main/resources* folder), `OTHERJAR.jar` with the necessary additions to the classpath, e.g. the Android application distribution JAR, `WHITELIST.p12` with the whitelist for the system, and 123.123.123.123 with the IP address of the system running Mokapot, e.g. `localhost` if running locally.

10.4 Running Demos

The *factorial-demo* application simply requires a Mokapot server to be running with the `factorial-demo-android.jar` file on its classpath.

The *messaging-demo* application requires a Mokapot server to be running with both `messaging-demo-android.jar` and `mokapot-messaging-demo-1.0-SNAPSHOT.jar` (generated by building the *mokapot-messaging-demo* module) on its classpath. First, start the server, then start the mobile application, and then start the desktop application (the other participant in the messaging demo) by running the `run.sh` script in the *mokapot-messaging-demo* module, replacing the placeholders with appropriate values for your system.

The *migrationdemo* requires two Mokapot servers to be running, each with `migrationdemo-android.jar` on the classpath.

The *performance* module includes the source code for the performance test discussed in section 8, and requires a Mokapot server running with `performance-android.jar on its classpath.`

10.5 Glossary

Term	**Definition**
User	Someone who uses a mobile device, and mobile applications in particular
Developer	Someone who writes mobile applications for users
Mokapot	The distributed computing library on top of which this project is built
Client	The system initiating a sequence of requests to run code remotely. In many cases, it may be necessary for the other system to run code on the client before it can return the final result. In this case, the other system is not called a 'client' with respect to the first client, because it did not initiate the *sequence* of requests; it just initiated one of many requests in this sequence. In cases where there would otherwise be ambiguity surrounding which system is the client and which is the server, more context will be given in order to resolve this ambiguity.
Server	The system being contacted by a client. The client will be asking to the server to run code and respond with the result of running the computation.

`DistributedCommunicator`	Conceptually: the entry and exit point for all of Mokapot's communications. Once a `DistributedCommunicator` has "started communicating", we say that Mokapot is running, and when it has stopped communicating, we say that Mokapot has stopped running. Responsible for coordinating requests to run code across different resources.
Communicator	Refers to an instance of `DistributedCommunicator`.
`Copiable`	A type whose instances can safely be replaced by copies without changing the semantics of the program. For example, immutable types, such as `java.lang.String` are Copiable.
`NonCopiable`	A type whose instances cannot safely be replaced by copies without changing the semantics of the program or *should* be created remotely, even if they could be created locally. For example, `java.util.ArrayList` is a `NonCopiable` type, since if a copy of the list is made instead of the original list, and an element is added to the copy, it will not be added to the original.
Long Reference	Similar to a Java reference, but the referent is not necessarily held on the same resource as the reference. Mokapot will forward invocations on long references to the resource which holds the object to which the long reference refers.
Renewable	A type whose instances can be safely re-instantiated. If a system becomes unreachable, all renewable objects on that system will be "renewed" on another system, including the local system if no other remote systems are available.

Note: some of these definitions paraphrase details from Mokapot's Javadoc.