RICE UNIVERSITY

Distributed Communication Middleware for an Selector Model

by

Bing Xue

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

Master of Science

APPROVED, THESIS COMMITTEE:

Vivek Sarkar
Professor of Computer Science
E.D. Butcher Chair in Engineering

David B. Johnson
Professor of Computer Science
Professor of Eelcctorcal and Computer Engineering

Lin Zhong
Professor of Electrical and Computer Engineering
Professor of Computer Science

Houston, Texas

August, 2017

ABSTRACT

Distributed Communication Middleware for an Selector Model

by

Bing Xue

The problem sizes that the community is dealing with today in both scientific research and day-to-day use computing exceed the capacity of modern shared-memory systems. With the increasing prevalence of powerful multi-core/heterogenous processors on portable devices and cloud computing clusters, the demand for portable mainstream programming models supporting scalable, portable and extensible distributed computing is also rapidly growing.

In this dissertation, we present the distributed selector model enabled distributed programming runtime library: cluster-based Habanero Java Distributed Selector and the mobile platform based Distributed Actor Model for Mobile Platforms by extending the HJDS implementation. This work focuses on enabling distributed message passing through building the communication middleware for a actor/selector model by supporting a fully actor-based runtime communication layer on clusters and a highly decoupled and customizable communication middleware and publish-subscribe enabled application-level runtime event handling on mobile devices that addresses the need for an easy-to-use, portable, reusable and scalable framework for small to medium sized distributed applications. We demonstrated the scalability of computationally intensive applications using distributed cluster-based and mobile-based platforms, and discuss the future steps for expanding the HJDS and DAMMP framework.

Acknowledgments

Firstly, I would like to express my deepest gratitude and appreciation to my advisor Prof. Vivek Sarkar for his invaluable support and guidance throughout this journey. I'm extremely grateful for the opportunity to be part of the Habanero Extreme Scale Research Group.

I would also like to thank my committee member, Prof. Dave Johnson and Prof. Lin Zhong for their insightful and inspirational advice and feedback for this project.

I want to thank all of my co-authors in the works presented in this thesis, Arghya Chatterjee, Zoran Budimlić, Branko Gvoka, Shams Imam and Srdjan Milaković. This work would not have been possible without teamwork.

I would like to thank my colleagues and friends for their continuous support and inspiration. Special thanks to Prasanth Chatarasi for his advice and feedback, to Wanjia Liu, Ziguang Niu and Mengchen Tang for their support and friendship throughout my time here at Rice.

Finally, I would like to thank my parents for their continuous support throughout the years and their belief in me.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Motivation

The problem sizes that the community is dealing with today in both scientific research and day-to-day use computing exceed the capacity of modern shared-memory systems. With the increasing prevalence of multi-core/heterogeneous processors on portable hand-held devices and the proliferation of cloud computing clusters, the demand for mainstream programming models supporting scalable, portable and extensible distributed computing is also rapidly growing. Along with these trends, distributed software systems and applications are shifting towards service oriented architectures (SOA) that consist of largely decoupled, dynamically replaceable components and connected via loosely coupled, interactive networks that may exhibit more complex coordination and synchronization patterns. These trends have encouraged the adoption of distributed computing systems composed of computational nodes with internal parallelism that cooperate by communicating over a network.

Low-level message-passing based protocols have matured over the past couple of decades for distributed software design and many domain-specific high-level abstractions such as MapReduce[1] and distributed file systems, as well as more general purposed large-scale distributed application frameworks like Apache Thrift[2] and Apache Spark[3] are on the rise. However, there still seem to a gap to be filled for a set of general purpose, easy-to-use, unified easy-to-use unified distributed program-

ming paradigm to enable reusable and portable small to medium scale distributed application development. Moreover, despite the popularization of hand-held computing devices through smartphones and wearable devices, there still lacks a reasonable parallel and distributed application framework and most applications remain single-device with shared-memory.

With mobile computing seeing a trend towards miniaturization and energy savings for a number of years, the available hardware parallelism in mobile devices has at the same time continued to increase. Overall, mobile devices still remain resource constrained physically in terms of power consumption and thermal dissipation. Combining the computing capabilities of multiple mobile devices in a distributed and dynamic setting can open up the possibilities for performance improvements, longer aggregated battery life for traditionally single-node applications as well as dynamic mobile network based applications. However, when distributed system and application developers build on clusters through RPCs, message passing and event-streams with enormous scales, mobile developers tend to be limited by the hardware and have to rely largely on Bluetooth or server-end coordination for peer-to-peer information exchange. The discrepancy in technologies and low-level protocols have prevented small and medium scale distributed applications become portable and reusable across different platforms. Therefore, an efficient runtime system that supports both single-node and distributed multi-node systems that can bridge between cluster-based environments and mobile platforms is a highly desirable goal, since such a paradigm can ease the transition from single-node to multi-node parallelism.

The Actor Model, formally described by Gul Agha [4] and having appeared in many modern programming languages and frameworks, is a go-to option as a basis for building distributed software with numerous dynamically interacting units.

The Selector Model (SM) [5] is an extension of the Actor Model and represents isolated units of a distributed system that interact only through message passing with multiple guarded mailboxes. It can be viewed as a generalization of actors and is an expressive primitive for concurrency and distribution and can be a platform-agnostic model for both clusters and mobile-based parallelism. Selectors allows synchronization patterns that relies on different messages types and the order of messages received, such as join patterns and bounded buffers. The expressiveness selectors grants is ideal as a foundation towards building more complex synchronization primitives and higher-level parallel abstractions.

The Selector Model has been integrated as a part of Habanero Java Runtime Library, a successful shared-memory parallel library that implements multiple parallel primitives in Habanero Execution Model such as the Async-Finish primitives [6], generalized barriers and point-to-point synchronization with phasers [7], mutual exclusion synchronization etc., many of which are also desirable in a distributed environment as unified programming primitives for parallel and distributed applications. Therefore, the Habanero Java Runtime Library is an ideal subject to expand to the distributed world as a portable, reusable and easy-to-use runtime system with an unified parallel programming model (Habanero Execution Model) that can be built on top of a selector model enabled distribution and message passing.

In this thesis, we design and implement the communication middle-wares for Habanero Java Distributed Selector (HJDS)[8] and Distributed Actor Model for Mobile Platforms (DAMMP), two distributed runtime environment based on adaptations of the HJlib selectors, each on cluster-based and mobile-based platforms, experiment with and analyze the performance of these two runtimes and discuss future distributed extensions of the HJlib based on the selector-enabled distributed runtimes.

1.2 Thesis statement

Distributed communication for the selector model can be implemented productively and efficiently on both servers and devices

1.3 Contributions

This thesis makes the following contributions:

- Implementations of distributed selector-based runtime for clusters (HJDS) and mobile Android devices (DAMMP), developed jointly with co-authors [9]
- Light-weight Selector reference design enabling better object encapsulation and location agnostic referencing
- Cluster-based run-time communication layer (Proxy Actor) design that enables location transparent selector creation and message exchange
- Android-based run-time communication middleware (IMobileCommunication-Manager) design and implementation, fully-decoupled and readily replaceable
- Experimental evaluation of performance benefits with distributed selectors on clusters, and experimental evaluation of performance benefits and thermal effects of distributed selectors on mobile hand-held devices

1.4 Organization

This thesis is organized as follows:

- Chapter 2 introduces the Selector Model, Habanero Java Runtime Library, and wireless communication technologies.

- Chapter 3 describes how location transparent selector creation and message exchange is achieved on the cluster-based runtime through the communication middleware. This chapter focuses on the design of the Proxy Actor, a system service actor that coordinates the communication for a single node in a distributed selector system.

- Chapter 4 describes how location transparency is preserved in the mobile implementation of distributed selectors and introduces a fully decoupled, readily replaceable communication middleware design. This chapter also introduces two off-the-grid wireless communication layer implementations based on Wi-Fi and Wi-Fi Direct technology.

- Chapter 5 evaluates the performance and scalability of our runtime on cluster-based and mobile-based implementations.

- Chapter 6 discusses related efforts in distribution of actor-based runtime/applications.

- Chapter 7 summarizes the work and contribution in this thesis and discusses next steps in our work.

Chapter 2

Background

This chapter introduces the Habanero Selector model in Section 2.1, explains the joint work on distributed adaptation of selector based on Habanero Runtime Library and summarizes the work by [9] on distributed bootstrap and termination in Section 2.3

2.1 Selector Model

The Selector model [5] is an extension of the Actor programming model [4]. Instead of a single mailbox like Actors, selectors have multiple guarded mailboxes that can be disabled or enabled individually and each mailbox has a priority associated with it. It directly inherits the start and exit semantics of the Actor model, and can be viewed as a generalization of the actor model. The Selector model was created to overcome difficulties in actor synchronization and coordination patterns.

Proposed by Carl Hewitt et al. [10] in 1973 as part of a research on artificial intelligence agents, the original Actor Model was designed to address decentralized agent interactions and has since been developed to be a powerful abstraction for structuring highly concurrent programs that scales to many-core processes as well as clusters and the cloud. Actors. An actor has a *mailbox* where it stores incoming messages. Communication between actors is purely asynchronous and non-blocking. An actor also maintains a local state which is initialized during creation. An actor can only process at most one message at a time which allows actors to avoid data races and

the need for synchronization. When an actor receives a message it may perform any of the following operations: *a*) Send message to another actor asynchronously, *b*) Create a new actor by passing the parameters from the message and *c*) Modify internal state of the actor, which may affect how subsequent messages are processed [11].

The Actor Model, since its first adaptation in Scala standard library in 2006 [12, 13], has been adopted in multiple forms, including actor-based languages like SALSA [14] and runtime library implementations in and Habanero Java library [15] in the Akka event-driven middleware [16].

The first Selector Model implementation was introduced in the Habanero Java Runtime Library [15]. The Habanero Selector implementation provides `selectors` as an asynchronous concurrency primitive that operates on message passing among multiple selectors. The results in [5] show that Selectors can also be implemented efficiently, since that work includes performance comparisons with Scala, Akka, Jetlang, Scalaz, Functional-Java and Habanero actor libraries. However, the implementation described in [5] focused on a single-node (shared-memory) implementation of the Selector model.

A Distributed Selector runtime library for Java based applications (HJDS) based on the shared-memory implementation of Habanero runtime library (HJlib) is presented in [8] and shows a unified runtime that supports both shared-memory and distributed multi-node execution of a program. The HJDS runtime allows the programmer to focus on implementing algorithms for solving a problem without worrying about whether the application should run on a shared-memory or distributed-memory system. The runtime also provides automated system bootstrap and distributed global termination by terminating the distributed system when all tasks of the user code have been successfully executed.

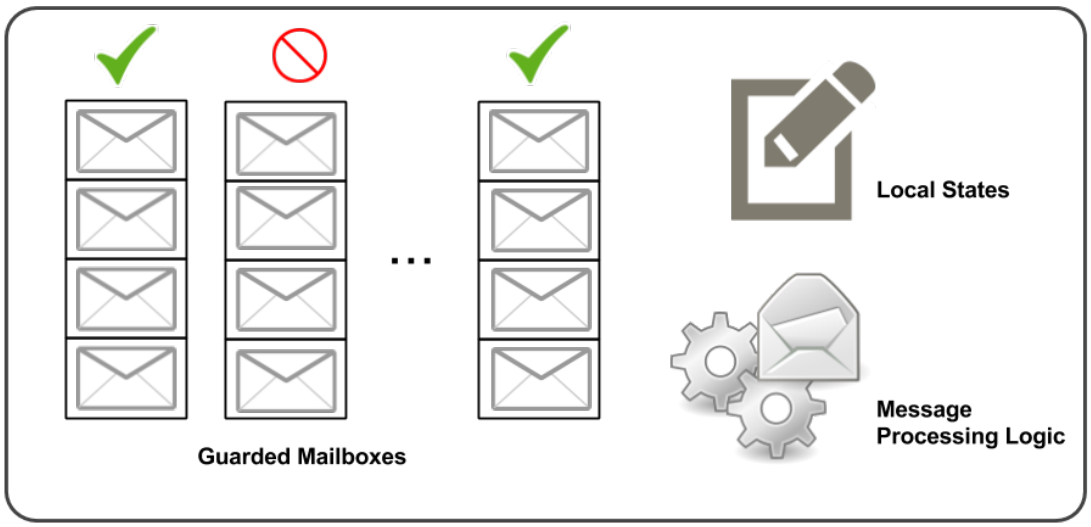

Figure 2.1 : Internals of a selector. A selector contains multiple guarded mailboxes, which can be enabled or disabled. A processing logic that handles the incoming messages in all enabled mailboxes, and a collection of local states retained throughout the selector lifetime.

Figure 2.1 shows the decomposition of a selector. A selector consists of multiple guarded mailboxes, which are either enabled or disabled. Messages can enter an enabled mailbox and be processed by the selector, while a disabled mailbox continues to receive and buffer messages directed to it but is unavailable for the processing logic to pick up any message to process. The selector contains a processing logic that specifies how each type of messages are processed, and can be viewed as the "main" method for the selector process. A selector in HJlib also allows local states, which are persistent states throughout the selector's lifetime.

A selector's life-cycle consists of three stages: 1. **created**: the selector has been created in the system, but does not process any messages while able to receive and buffer all incoming messages 2. **started**: the selector is started and is able to process

messages as well as receiving and sending messages 3. **exited**: the selector is exited, unable to process or receive any message sent after its call to `exit()`, however the selector instance is subject to regular Java GC and can be inspected for record keeping and cleanups The HJlib also implements API such as `pause()`, `resume()` to allow for halting the whole selector and resuming execution, and `preStart()`, `postExit()` for initialization and bookkeeping.

2.2 Habanero Java Runtime Library

Habanero-Java (HJLib), developed at Rice University, implements the Habanero execution model [6]. Habanero Java Runtime Library APIs include `async`, `forasync`, `asyncAt`, `asyncPut`, `asyncGet`, and `asyncAwait`, and `finish` as general primitives for creating and awaiting the completion of asynchronous computation and data transfer tasks. These Async-Finish primitives enable any (block) statement to be executed as a parallel task, including for-loop iterations and method calls [6]. HJLib also supports generalized barriers and point-to-point synchronization with phasers [7], mutual exclusion synchronization within tasks using delegated isolation [17] and object-based isolation [18] and shared-memory implementation of actors and selectors [5].

These primitives can be used to obtain programmability, parallelism, and scalability benefits across a wide range of task-level parallel algorithms. HJLib implements a priority-based lock-free work-stealing algorithm [19] with multiple thread pools to support priority scheduling of tasks.

2.3 Bootstrap and Global Termination in Distributed Selector Model

The design of our Distributed Selector (DS) model is based on the Habanero Java Runtime Library (HJlib) [6]. We expand the shared-memory implementation of the Selector Model to achieve remote message passing, remote selector creating and bounded global termination in a transparent manner. The DS model refers to each single HJ runtime instance as a `place`. A physical computing node can serve as a single or multiple places, given each place has its own logically isolated address space. In general, selectors are located at the same `place` to show logical affinity and/or to exploit data locality

The global termination is initiated by the master node and is performed in stages to detect when the user program becomes quiescent. Stage 1, starts the termination process when all connected nodes report idleness. In Stage 2, the master node passes a signal to all nodes to verify whether a node is still idle or active. If active, Stage 2 is repeated until all nodes are in idle state. Finally, in Stage 3, when all nodes are guaranteed to be idle, the master node initiates the shutdown process of all nodes and finally terminates itself. The authors claim to guarantee shutdown of the system successfully provided the application developers shutdown the respective Actor/ Selector instances on each node. The design and implementation of automated bootstrap and global termination of HJDS is described by joint-author of the HJDS [8] in [9].

2.4 Wireless Communication

As the IEEE 802.11 standard has become one of the most successful wireless protocols to access the Internet, the Wi-Fi technology extends itself to accommodate P2P device connections beyond the traditional requirement of the presence of an Access Point (AP). The Wi-Fi Direct technology is developed by the Wi-Fi Alliance to expand the use cases for Wi-Fi technology to device-to-device communication. It builds upon the IEEE 802.11 infrastructure mode and uses devices as logical SoftAP (software-enabled access point) for connectivity, without relying on external AP support as in the ad-hoc mode [20]. Device-to-device communication in a typical Wi-Fi network has to be supported by the external APs. However, in a Wi-Fi Direct P2P network, the logical role of AP is specified as *dynamic* and can exist simultaneously on a client device. Devices with Wi-Fi Direct capabilities can communicate by forming P2P groups.

The group formation process has several phases. Before group establishment, devices are in a *discovery* phase, which is performed by a traditional Wi-Fi scan. A device can either discover an existing P2P Group or a few devices can discover each other. When a device discovers an existing P2P group, it may choose to query the set of current services on the group and join based on the information. A device that did not discover any existing P2P group, or other devices to form a group can *autonomously* create a group and become the Group Owner (GO). When devices discover each other, they may enter a *negotiation* phase to determine which device would be the Group Owner (GO) and serve as a Soft-AP [21]. The negotiation process is dependent on individual implementations. Once the GO role is established, clients can choose to join the P2P group through discovery of the GO.

Comparing to traditional device-to-device connectivity technologies such as Blue-

tooth and ZigBee with nominal ranges from 10 meters to 100 meters, and transfer speed between 250 kbps to 25 Mbps, Wi-Fi Direct inherits all the capabilities from IEEE 802.11 standards, and claims to provide nominal range up to 200 meters and transfer speed up to 250 Mbps [22, 23]. With its inherited power saving support and extended QoS capabilities, Wi-Fi Direct can be considered one of the most promising candidate for wide range device-to-device communication, and suitable for our purpose of distribution across mobile devices.

Android 4.0 and versions after, complies with the Wi-Fi Direct certification program and allows applications to interact inter-device without an external network connection [24]. The Android Wi-Fi Direct interface (`WifiP2pManager`) allows developers to discover, request and connect to peers and provides listeners that detect the success or failure of connect, dropped connections and newly discovered peers. The Android API does not implement any specific GO negotiate algorithm and each client can only belong to one P2P group at a given time.

After the maturation of client-server based wireless communication, energy efficient and high bandwidth direct device-to-device communication will be the new challenge that mobile platforms face in creating more secure and more accessible distributed applications and systems. With its inherited power saving support and extended QoS capabilities, Wi-Fi Direct can be considered one of the most promising candidates for wide range device-to-device communication, and suitable for our purpose of distribution across mobile devices. Because Wi-Fi Direct technology is not widely readily available at the implementation level, in this work we also consider Wi-Fi Hotspot (Soft AP) on Android as a proxy due to their common hardware support.

Chapter 3

Communication Middleware for clusters

3.1 Distribution for Habanero Java

3.1.1 HJDS Design Overview

The design of the Habanero Java Distributed Selector Runtime is based on the Habanero Java Runtime Library (HJlib). We expand the shared-memory implementation of the Selector Model to achieve remote message passing, remote selector creation and bounded global termination in a transparent manner [9].

A HJDS runtime instance consists of multiple JVM instances, each referred to as a `place`, typically located on multiple physical nodes, with at least one place on a single physical node.

Figure 3.1 shows a system overview with four places. The internals of a single `place` consists of the developer view which contains user-defined selectors, and the runtime system, which is managed by the `System Actor` and the `Proxy Actor`. The `System Actor` monitors and maintains place status and is responsible for global bootstrap and global termination of the system [8, 9], the `Proxy Actor` is responsible for coordinating message passing between remote places, and also mediates between the HJlib runtime and the communication layer instantiated outside the HJlib runtime as detailed in Section 3.1.3. The low-level communication layer is instantiated out side the HJlib runtime as detailed in Section 3.2

In a HJDS runtime instance, a Master Place maintains the internal state of the

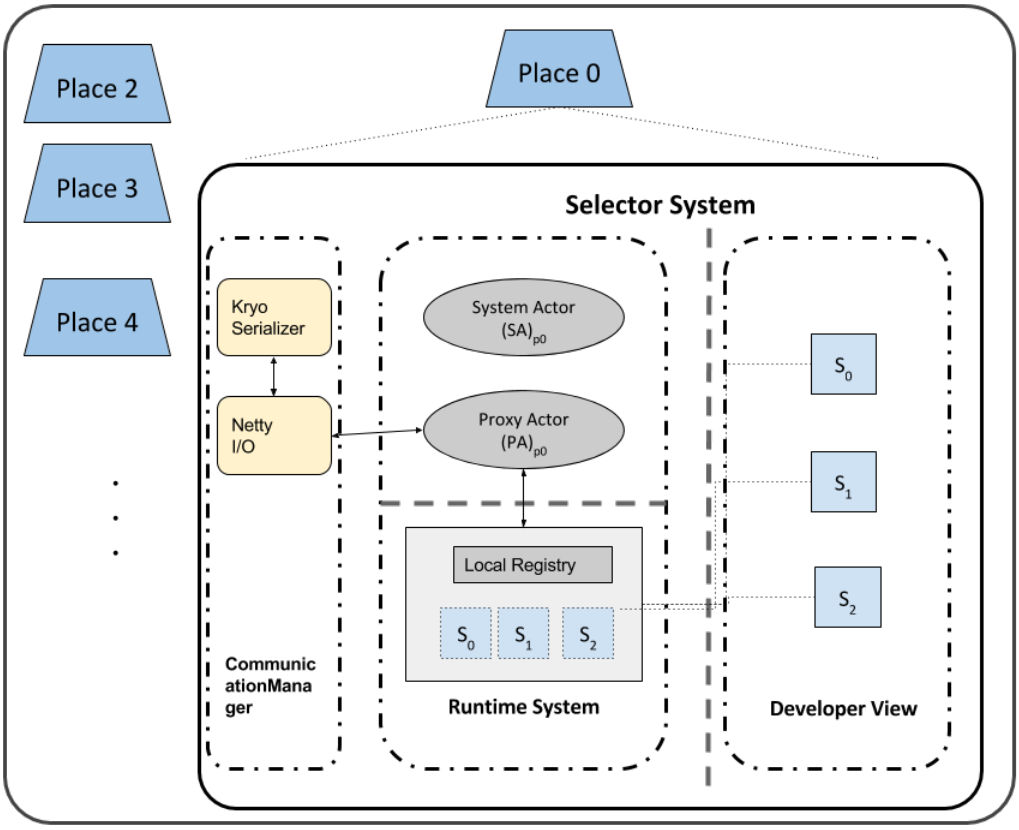

Figure 3.1 : An overview of the Habanero Java Distributed Selector runtime system. A distributed selector system consists of multiple places (four in this figure), each place has a developer view which contains the user-level selectors, and a runtime system view where a system actor and a proxy actor manages the place status and message exchange between remote places.

system in its System Actor and is responsible for the bootstrap and global termination of the system, while each place maintains its internal state with the individual System Actors and complies to the bootstrap and global termination protocols detailed in [9, 8]. Each Proxy Actor manages all communications to and from its residing place to other places in the runtime and is responsible for maintaining local user-level selector

states. For a HJDS based application, user provides a configuration file that specifies available physical hosts as shown in Figure 3.2, where IP addresses (or host names) and ports for all computing nodes are specified. If two places are assigned the same node.

```
1  selectorSystem {
2    init {
3      place : p0,
4      hostname:cn1.davinci.rice.edu,
5      port: 5000,
6        }
7    remote : [
8     {place : p1, hostname:cn2.davinci.rice.edu, port: 5001},
9     {place : p2, hostname:cn2.davinci.rice.edu, port: 5002},
10    {place : p3, hostname:cn3.davinci.rice.edu, port: 5000},
11    ]
12 }
```

Figure 3.2 : Sample configuration file for a HJDS system. The init place is the designated master place of the system, and will start the global bootstrap. Note in this example, a single physical node can host more than one places.

When the application launches, the system will run on multiple JVM instances (using different ports) on the same node. The `init` keyword specifies the bootstrap master node, while the `remote` keyword indicates other predefined places in the bootstrap. The runtime reads information from the configuration file and boots up the system. By ensuring that the master node has the program executable and SSH access to all places specified in the configuration file, the runtime stages all executable on all `remote` nodes and initiates the bootstrap sequence. The runtime will exit the program after all user created selectors have safely terminated. The current implementation requires user to set the user name and password for remote hosts as

environment variables. The runtime can be extended to support more configurations like memory usage limit and thread count in the bootstrap Config files.

3.1.2 Selector Interface

Figure 3.3 shows the design for the new ISelector interface for distribution. Both `hj.distributed.Select orHandle` and `hj.distributed.DistributedSelector` inherits from the `hj.distributed.ISelector` interface. The `hj.distributed.SelectorHandle` is the single point of access to a Selector object in user programs, while `hj.distributed.DistributedSelector` extends its shared-memory predecessor but remains exclusive to access internally to the package.

```
1  public class HJSelector{
2    pubic SelectorHandle newSelector(Class<T> classType,
3        Object... args);
4    pubic SelectorHandle newSelector(int placeId,
5        Class<T> classType, Object... args);
6  }

8  public interface ISelector{...}

10 public abstract class DistributedSelector<MessageType>
11     extends Selector<MessageType> implements ISelector {
12       private SelectorHandle _handle;
13       public final void send(int mailboxId,
14       final MessageType message);
15       public final void start();
16       public final void exit();
17 }
```

Figure 3.3 : The HJ Distributed Selector class hierarchy. The `DistributedSelector` class is not accessible to users

A user program can use the factory method `hj.distributedHJSelector.newSelector` to obtain a SelectorHandle instance. The factory method abstracts away the difference between creating a selector locally or at a remote location by allowing the user to omit the location of the selector to be created. The introduction of `hj.distributed.SelectorHandle` eliminates any possibility of sharing state by disallowing the user to directly interact with Selector references. More importantly, the separation of Selector object and the access handle gives a lightweight vehicle of communicating Selector object information across the distributed system, as well as routing messages when needed. Given the lightweight handle, programmers will not have the need to deal explicitly with the low-level complexities of distributed coordination.

The HJDS interface provides users with a Selector Handle as the access point to a selector object. A Selector Handle is designed to be lightweight for sending across the network. It consists of a globally unique identifier for the selector object, and method handles for sending messages to the selector. Since we do not differentiate between selectors that are created to reside locally or on remote places, the selector will need an identifier that can be constructed upon the request of creation and unique across the entire distributed system. Selectors can only be accessed through these handles by invoking the `send` method, shown in Figure 3.4 A Selector Handle contains a globally unique identifier for the selector object and method handle for sending messages to the selector to act as a global reference for the selector instance. No differentiation is made between selectors that are created to reside locally or on remote places in the user application and the selector handle GUID encodes the actual location of the selector instance.

The Selector Handle GUID is constructed upon a request for selector creation and

is unique across the entire distributed system. The GUID is a 32-bit integer that encodes three pieces of information: *1*) an 8-bit value encoding the *place* p on which the selector is created; *2*) an 8-bit value encoding the *place* q on which the selector instance resides; and *3*) a 16-bit integer value representing a unique identifier for the selector on p. The length of the GUID is chosen arbitrarily for small networks, however it can be elongated to allow secure distributed assignment of selector GUID and place UID.

```
2 public class SelectorHandle<MessageType>
3 implements ISelector, Serializable {
4   private long _UID;
5   public void send(final int mailboxId,
6   final MessageType message);
7   public long getUID();
8 }
```

Figure 3.4 : The HJDS Selector Handle Class. The Selector Handle consists of a GUID that encodes the location of the selector instance and send method for communicating with the selector instance.

3.1.3 Proxy Actor

In an HJDS application, one Proxy Actor exists for each place in the system. The Proxy Actor is responsible for coordinating messages among local and remote selectors. The Proxy Actor passes messages to remote selectors to the correct place, keeps track of status of all user-level selectors to update the System Actor and handles remote selector creation.

For any application-level message sent to a selector through its selector handle, the selector handle passes the messages and its selector GUID to its local proxy actor. The proxy actor, upon receiving a message, decodes the GUID to get the selector's residence place p. If p is a remote place, the proxy actor sends the message and selector GUID directly to place p. If p is the local place id, the proxy actor checks its local registry for the 16-bit ID to get the local selector instance and passes the message. In practice, we try to minimize the number of hash map look-up by adding a transient field to a selector handle that directly references its local selector instance. However the method does not eliminate the need for proxy actor local registry because a selector handle to a local selector can be obtained through remote messaging, where the local selector reference would be lost.

The proxy actor may not be able to find a local instance for a selector message that is directed to a local selector. In this case, the proxy actor assumes the selector is to be created on local place and creates a mailbox instance as a buffer to place the message.

For any system-level message directed to the system actor (e.g. place status updates, bootstrap and termination protocol messages), the proxy actor directly passes the messages to the System Actor.

3.1.3.1 Remote Selector Creation

When creating a new selector, the factory method *newSelector* in Figure 3.3 checks the residence place. If the selector is to be created locally, the factory methods creates the local selector instance and returns the selector handle directly. If the selector is to be created on a remote place, the factory method creates a selector handle instance with its GUID, constructs a system-level message for remote selector creation, passes the message to the proxy actor and returns the selector handle. The system-level remote selector creation message contains a message type field and the selector handle and is sent to the residence place through proxy actor. The proxy actor keeps a record of all sent remote creation messages until a confirmation is received.

Figure 3.5 shows the decomposition of a remote selector creation in a simple ping-pong program.

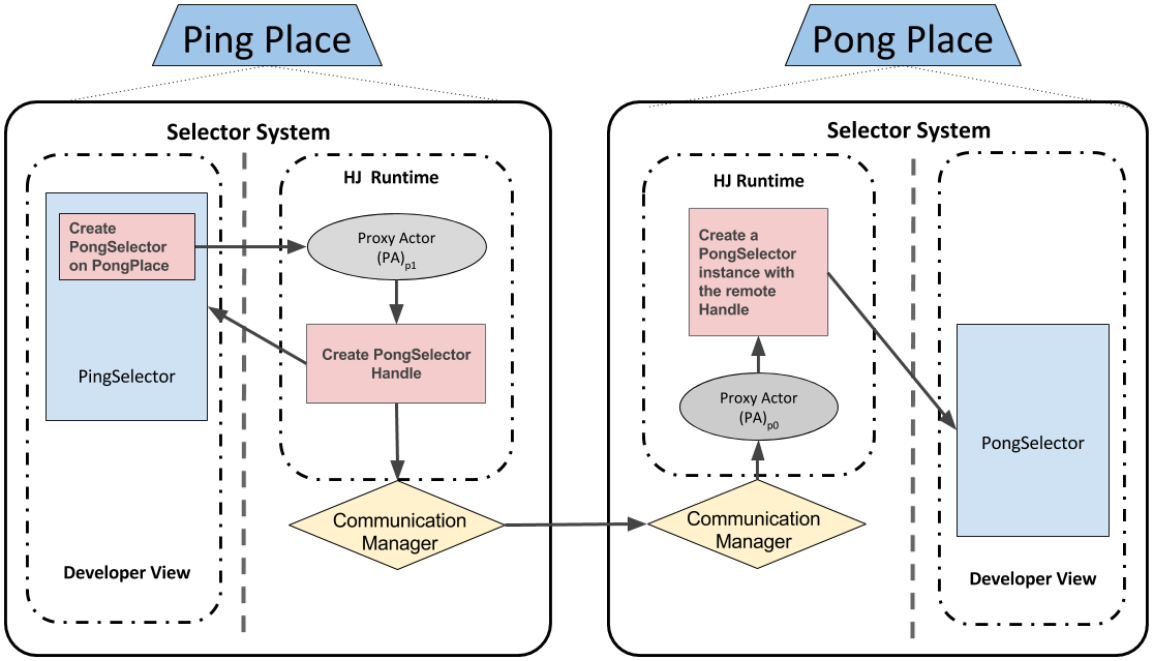

Figure 3.5 : In a simple ping-pong program. The ping selector on p_0 tries to create a new pong selector on p_1, the request is sent to the Proxy Actor, which creates a local handle and returns the handle to ping selector, as well as sending the handle along with a remote creation request to p_1, the communication manager decodes the command and the proxy actor creates the pong selector instance based on the handle information and sends back a confirmation reply.

3.1.3.2 Message exchange between remote places

Upon receiving a selector creation message, the proxy actor checks if a selector instance with the GUID already exists in its local registry. If present, the proxy actor sends a reply confirming creations. If not, it creates a local selector and maps the selector GUID to the instance in its local registry. The proxy actor also checks its local mailbox buffers for the selector GUID, if found the mailbox buffer associated with the GUID is attached to the local selector instance. After the creation com-

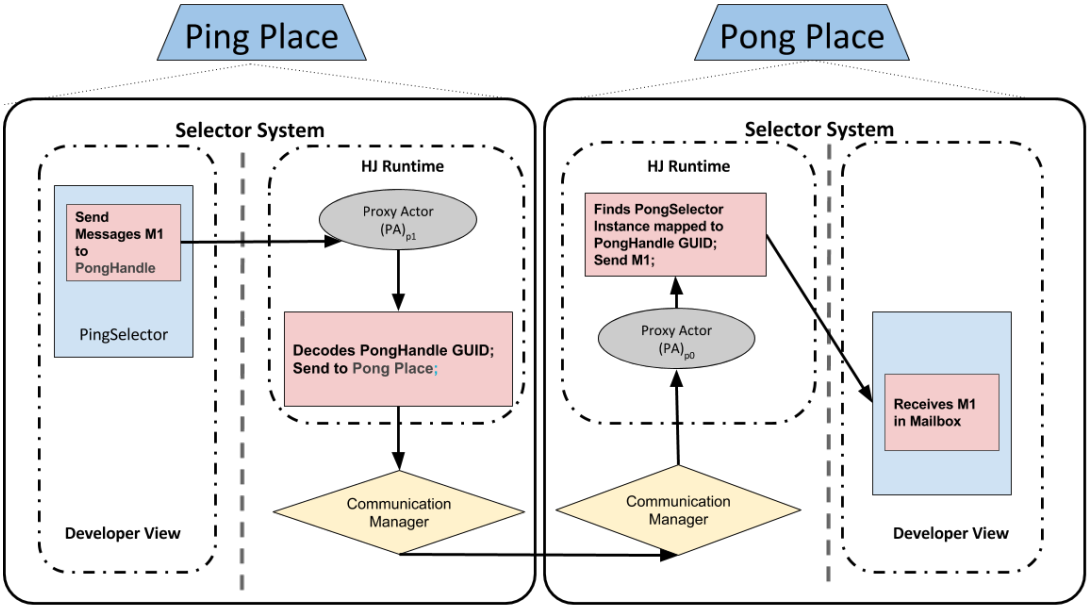

Figure 3.6 : In a simple ping-pong program. The ping selector on p_0 sends a message to the pong selector on p_1. The message goes through the proxy actor who identifies the message to be on a remote place by decoding the pong selector handle GUID, and sends to p_1 through the communication manager. On p_1, the proxy actor finds the destination through its local registry and sends the message to pong selector.

pletes, the proxy actor constructs a reply messages confirming creation. A selector starts processing messages right after its creation. Figure 3.6 shows the decomposition of message exchange between remote places through the proxy actor in a simple ping-pong program.

3.1.4 System bookkeeping

The proxy actor is also responsible for keeping track of selector status on its place. The proxy actor keeps a counter for the number of active local selector instances, incremented at each selector instance creation and decremented at each selector in-

stance exit. When there is no active local selector instance, no mailbox buffers for expected selector creation and no remote creation messages waiting for a reply, the proxy actor decides that the place is "idle". Otherwise the place is regarded as "not idle". Upon any place status change, the proxy actor notifies the System Actor with a system message, which the System Actor may use to determine whether to initiate the global termination protocol or not.

3.2 Communication Middleware

For a HJDS application, the communication middleware for serialization and data transmission are instantiated outside the HJlib runtime.

3.2.1 JVM Serializers

We choose the Kryo serialization framework [25], which has been shown to be faster than the Java serializer [26]. As a Java-oriented framework, it is better suited for our purpose than other high-performance serialization tools such as Google's Protocol Buffers [27], Apache Avro [28], or Apache Thrift [2], which work across multiple languages and platforms and have more restrictions on the data that can be sent [29]. Most cross-platform serialization tools ask for primitive types (sometimes with the addition of nested arrays) in serializable datagrams, however, a selector should be able to send and receive any message that implements `java.serializable`.

Our system does not limit the message types that could be sent across the selectors. While Kryo provides compression to reduce the size the serialized object, it will be up to the developer to decide whether to include objects that may result in massive data transfer.

3.2.2 Netty

We choose to use *The Netty Project* [30] to provide asynchronous communication between places. *The Netty Project* provides a unified interface for blocking and non-blocking socket communications that simplifies implementation of stream-based data transport, and is popular among client-server based network applications. It is also used by AKKA Cluster for its distributed actor framework.

The Netty communicator is instantiated with the start of each place, and its reference is kept by the Proxy Actor to send and receive messages to/from other places. For each place, a different channel is created and is kept open until global termination completes. Netty provides a multi-threaded even loop that handles I/O operation to manage multiple open channels.

Chapter 4

Communication Middleware on Android

4.1 DAMMP design overview

The Distributed Actor Model for Mobile Platform (DAMMP) is an extension to the HJDS runtime as described in Chapter 3. The DAMMP design addresses the unpredictable nature of connectivity with mobile devices by introducing a separate communication middleware for mobile devices shown in Section 4.2 to provide flexibility for wireless connections. The automated bootstrap and global termination features from HJDS is replaced with manual operations to allow for decentralized application design on *volatile* mobile networks. Instead, we introduce a publish-subscribe interface to communication system events between the runtime and application, thus allowing application specific control on intermittently connected mobile networks. An system overview is shown in Figure 4.1, the major difference from Figure 3.1 is the inclusion of `MobileCommunicationManager` for a more complex communication layer in mobile platforms.

The DAMMP design supports all selector features from HJDS including remote selector creation. Due to the limited computing capabilities of mobile devices, DAMMP assumes a single physical device to host only one place. Places can dynamically join and leave the network to allow for dynamic topological changes and reconfiguration. A seamless computational offloading model using the master-worker paradigm with heterogeneous devices is introduced with the DAMMP design and is detailed in [9].

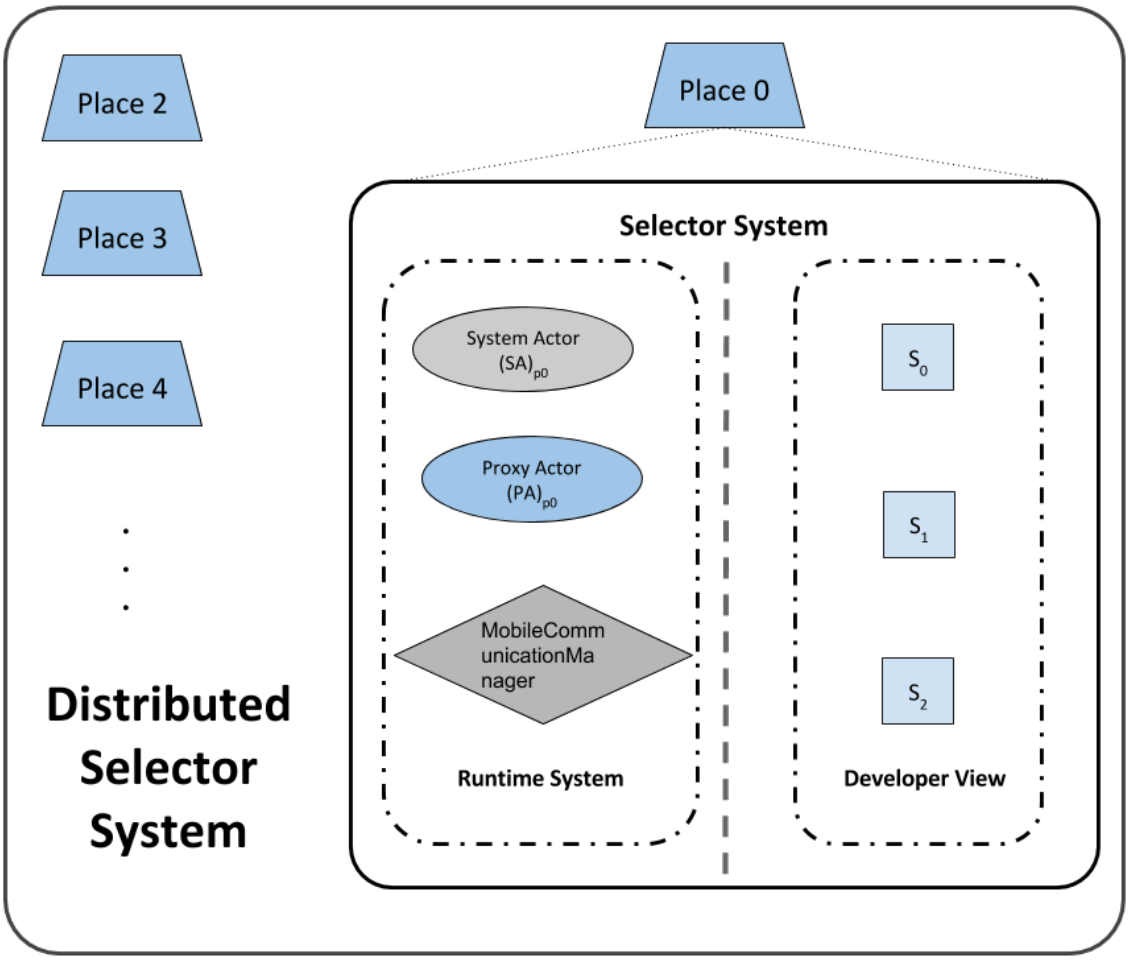

Figure 4.1 : An overview of the DAMMP distributed selector system. The runtime system is now responsible for lower-level communication manager.

4.2 Mobile Communication Layer design

The HJDS implementation is intended for cluster-based distribution and assumes network connection to be stable with low transmission costs using asynchronous TCP socket servers. In a mobile network with often intermittent wireless connection and a much higher data transmission cost, however, the communication middleware becomes a main source of system overhead and needs to be readily replaceable to meet

the needs of specific hardware with different wireless modules. In the DAMMP design, we introduce the `IMobileCommunicationManager` interface in Figure 4.2 as a stand alone communication layer, readily to be replaced by any customized wireless communication middleware. On top of the communication middleware, the system actor and proxy actor on a single place will behave the same as described in Chapter 3, apart from omitting the global bootstrap and termination protocol.

The `IMobileCommunicationManager` interface defines the DAMMP communication middleware and has three methods: start() to initiate the communication manager, stop() to end communication manager, and send(place, message) for sending any message to a target place. After the stop() method is called, a mobile communication manager closes all existing connection and leaves the wireless network. This process cannot be reversed, but the IMobileCommunicationManager object is still subject to normal garbage collection and can be used for bookkeeping. Alternatively a set of pause/resume function can be implemented with specific wireless modules, however our interface does not require such functions. It also includes a callback handle for communication with the local runtime system. Figure 4.3 shows an example to create a selector system by passing in the WifiCommunicationManager our DAMMP implementation provides. The `ISystemCallback` handle is installed at application bootstrap, and is to be invoked for any change in the network status.

At the creation time of a selector system instance, the selector system calls `start()` to initiate the communication manager, and after initialization, the communication manager calls `ISystemCallback.onConnectionReady` once the device is ready to join a network. To guarantee asynchronous data transmission, the communication manager is forced to run on a separate thread at creation time of a selector system instance.

```
1     public interface IMobileCommunicationManager {

3         interface ISystemCallback {
4             void onConnectionReady(Place localNode);
5             void onMessage(Message message);
6             void onPlaceJoin(Place place);
7             void onPlaceLeft(Place place);
8         }

10        void start();
11        void stop();

13        boolean send(Place place, Message message);
14        void setSystemCallback(ISystemCallback callback);

16    }
```

Figure 4.2 : The communication API for mobile platform. The IMobileCommunicationManager interface defines the communication middleware for device-to-device communication in a DAMMP application. A mobile communication manager implements this interface and initializes its wireless connection when start() is invoked. The mobile communication manager routes a message to place i when send() is invoked. The mobile communication manager terminates all connections and stops routing messages to remote places when stop() is invoked, the communication manager cannot be resumed after. Depending on specific wireless modules, a separate set of pause/resume functions can be implemented, but is not required for our interface

When a neighbor leaves the network, the communication manager notifies the local selector system by invoking `ISystemCallback.onPlaceLeft`, where the selector system will notify the application with a system status message through the publish-subscription mechanism described in Section 4.3. When a new neighbor

joins the network, the communication manager notifies the local place by invoking `ISystemCallback.onPlaceJoin`, where the selector system will notify the application with a system status message. The `ISystemCallback.onMessage`

A place can, in principle, join the network again after leaving without terminating its local place. In such case, the mobile communication manager is responsible for identifying if the joining place has already been assigned a place id in the current session through implementation specific techniques. The DAMMP implementation provides two implementations of `IMobileCommunicationManager` that can achieve this as detailed in Section 4.4.1 and Section 4.4.2.

In essence, the introduction of `IMobileCommunicationManager` interface with the omission of global bootstrap and termination asks for the application to be partially responsible of system resilience and recovery, resulting in a decentralized distribution that differs from the monolithic approach on HJDS where network stability is assumed.

4.3 Network status control at Application level

In the DAMMP design we introduce network status control at application level through a publish-subscribe mechanism. As described in Section 4.2, by omitting the global bootstrap and termination features available at a stable network environment like HJDS assumes, the communication manager notifies the application level of network status change including the availability of the mobile connections and the join/leave of neighbor places.

The DAMMP system utilizes the availability of multiple mailboxes in the selector model, and proposes a publish-subscribe mechanism for delivery of system messages to applications that is non-blocking and even-driven. We introduce the `Subscription`

```
1   /* A Wi-Fi based implementation for mobile communication middleware*/
2   public class WiFiCommunicationManager implements ↩
        IMobileCommunicationManager{...}

4   /* Application that creates a DAMMP selector system */
5   public class MainActivity extends AppCompatActivity {
6       ...

8       launchHabaneroApp(new HjSuspendable() {
9               public void run() throws SuspendableException {

11                  ...
12                  final SelectorSystem example =
13                      SelectorSystem.createInstance(new ↩
                            WifiCommunicationManager());
14                  example.start();
15              }
16          });
17  }
```

Figure 4.3 : An example on how to start a selector system by specifying a mobile communication manager on Android. A mobile communication manager is passed to the Selector System at creation time, and the system callback handle for the local place is set. Upon any network status change, the mobile communication manager should notify the runtime through the system callback handle.

class decorator. For any selector class it could choose to subscribe to system messages with a designated mailbox. Currently available system notification classes are `NodeJoined` and `NodeLeft`. The local selector system registers all subscription by associating the subscribed mailbox instances at start time, when network status change is propagated by the mobile communication manager through system callback handle, the selector system will produce the corresponding system messages and sends the

message to all subscribed mailboxes. We build on the Selector model, which supports multiple mailboxes for each selector and can also assign processing priority to each mailbox. Specific applications could choose different mailboxes based on its priority of network status change and failure model. The mailbox guards can be turned on/off and multiple mailboxes can subscribe to different system message class, giving more flexibility to application level network status control.

As shown in Figure 4.5, applications subscribe to different system messages (as `Topics`) with designated mailboxes. The example shows messages to be processed with top priority, but applications can choose to react to different categories of run-time and communication events in different ways. By assigning mailbox priorities, developers can implement application-specific resilience models without changing the underlying actor-based program semantics. Figure 4.4 shows how a network change (a node leaving the network) travels to application-level selectors through publish-subscribe pattern.

4.4 Off-the-grid mobile network formation

We introduce the two `IMobileCommunicationManager` implementations based on Wi-Fi and Wi-Fi Direct technology detailed in Section 2.4 the current DAMMP implementation provides.

4.4.1 Wi-Fi enabled mobile communication manager

The Wi-Fi based implementation for communication manager requires an external access point to aid the cluster formation. In most cases, one of the mobile devices to initiate a DAMMP application (often the master device) will serve as a SoftAP for the group before launching the application. Our implementation is based on the

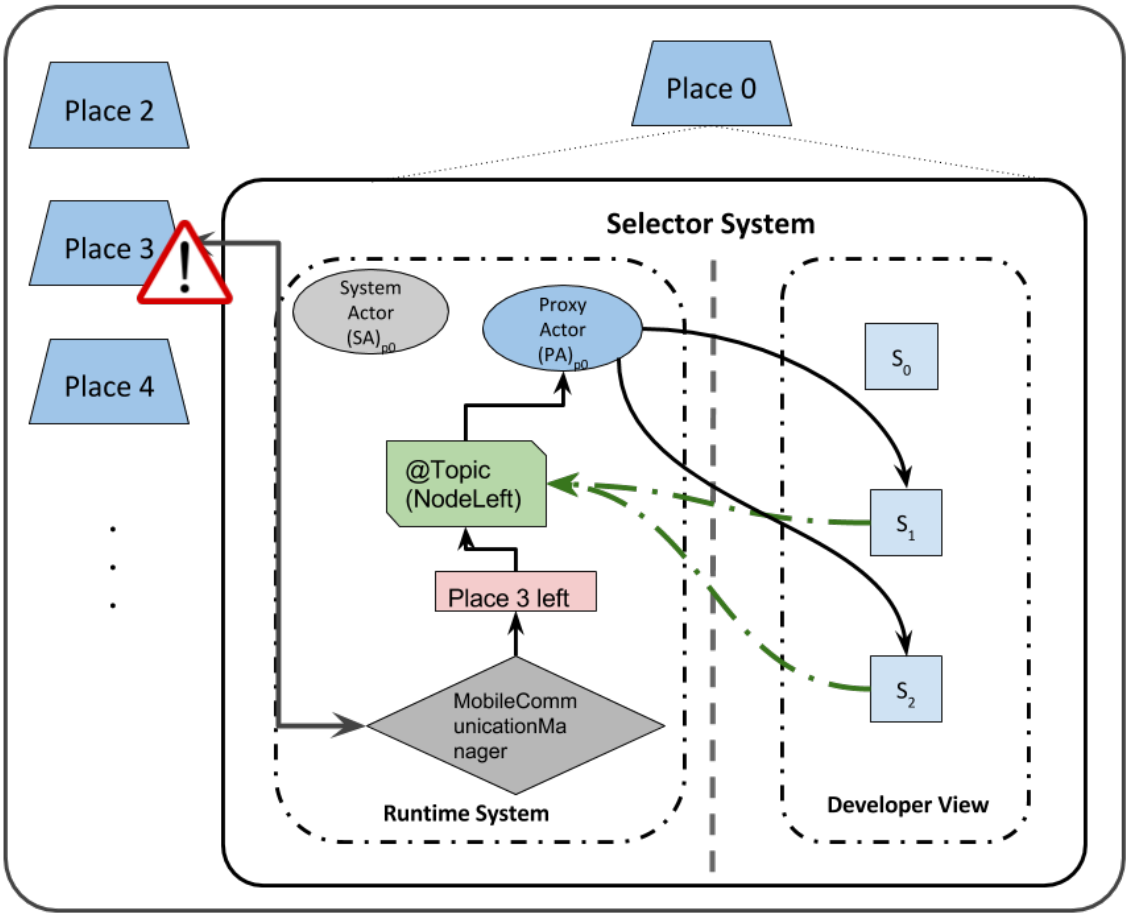

Figure 4.4 : An example of publish-subscribe enabled system notification traveling from the communication middleware to application-level selectors subscribed to the message. Place 3 have left the network, which is detected by the MobileCommunicationManager. The communication middleware sends a message to the topic (NodeLeft), which selectors S_1 and S_2 are subscribed to. The topic distributes a copy of the messages to every selector subscribed to it, which proxy actor directly delivers to the instances.

SoftAP interface on Android and asks the master place act as the access point.

To prepare for application launch, the master broadcasts a service name as its SoftAP SSID. The service name consists of the application name and a session ID

```
1 /**
2  *MySelector subscribes to NodeJoined and NodeLeft messages with its mailbox 0
3  */
4  @Subscription(topics = {
5     @Topic(messageClass = NodeJoined.class, mailbox = 0),
6     @Topic(messageClass = NodeLeft.class, mailbox = 0)
7     })
8  public class MySelector extends DistributedSelector {...}
```

Figure 4.5 : A selector class can subscribe to different alerts from runtime.

generated from current date. Nearby mobile devices will be able to join the network through recognizing the application name from the SSID. Upon connection to the access point, the master place obtains an IP address for the device and opens a socket connection. The master place then send a list of currently available IP address in the network for the newly joined device to connect, and sends the newly joined device IP address to all other devices in the network. After establishing connection to all available places in the network, the newly joined device sends a `PlaceReady` message to the master place, and becomes available for computation offloading.

The master place, unlike in the cluster HJDS implementation where it has to wait for a fixed set of places to bootstrap, can start the user code manually at any given time. As places dynamically join or leave the network, the application has to rely on obtaining the list of available place ids from the proxy actor to create remote actors.

4.4.2 Wi-Fi Direct enabled mobile communication manager

The Wi-Fi direct based implementation for communication manager does not require external access points [20]. By using the Wi-Fi Direct interface, a mobile device is designated as group owner by the application. In our implementation the Master

Place in a distributed selector system is the default Wi-Fi Direct group owner.

The master place at launch time will start a Wi-Fi Direct group and designate itself to be the group owner. The master place broadcasts its service ID consisting of the application name and its session ID. Mobile devices that are not the master place, at application launch time, search for a nearby Wi-Fi Direct service, where the service ID corresponds to its application name and tries to join the network as a group member. Upon joining the Wi-Fi Direct group, the group member is assigned a new place id and can obtain a list of all other available places in the group from the group owner, then establish socket-level connection to all the places. Upon joining the Wi-Fi Direct group, the group member remembers the application session ID that associates with its place ID assignment. A system message `PlaceReady` is then sent to the master place, indicating the new group member is ready for computation offloading.

Any device that disconnects from the group but have not exit the application will be able to rejoin with the same place ID after matching the cached session ID associated its assigned place ID and resume computation as the application specifies.

Chapter 5

Experimental Evaluation

5.1 Evaluation on Cluster Implementation

5.1.1 Hardware Setup

The experiments were conducted on 12 core, 2.8GHz Westmere nodes with 48GB of RAM per node (4 GB per core), running Red Hat (RHEL 6.5).On each node equal number of selectors are created. Each benchmark was run 20 times, and we report the mean and the best execution times across these runs for a given number of nodes.

5.1.2 Microbenchmarks

We designate each physical computing node to be a single `place`, and each place to have 12 workers (equal to the number of cores on each node). All implementations uses multiple mailboxes to differentiate between control messages and actual computational tasks where control messages (start and end messages for each task) are placed in the mailbox of highest priority.

5.1.2.1 Trapezoidal Approximation

The `Trapezoidal Approximation` benchmark approximates an integral function over an interval using the trapezoidal approximation algorithm [31, 32]. The implementation is adapted from the Savina benchmark [33]. The approximation is calculated with a master-worker based parallelism by dividing the large interval of the approxi-

mation task into a fixed number of small intervals. Each worker is an application-level selector that computes the integral approximation of each small interval in parallel and send their results back to the master selector. The master selector collects the results from all the worker selectors, adds them up, terminates all worker selectors after all works are completed, and displays the final result before terminating itself (thus the whole program).

The communication between works and the master selector in this benchmark is fairly limited. The amount of work for each worker is divided up among workers with configuration parameters from the master along with their activation signal. Each worker processes all computation tasks assigned to it to report to the master, and the master signals its workers to exit after collecting all results. In such scenario, the system is less affected by the overhead resulting from system setup and remote message transfer; thus we set up the experiment to explore the weak scaling property by keeping the computation effort constant on each node. The result in Figure 5.1 shows steady scalability over 2 to 12 nodes with increasing workload under minimal communication among selectors.

5.2 Evaluation on Mobile Implementation

The DAMMP runtime is implemented on Android 5.1.1 at API level 22. The DAMMP implementation includes both standard Wi-Fi enabled and Wi-Fi Direct based communication layer. The Wi-Fi enabled communication layer requires one of the devices in the network to act as SoftAP for the cluster formation. External access points could also be used under this mode. Due to the limitations of Android Wi-Fi Direct interface, only a single group owner is allowed for any Wi-Fi Direct groups.

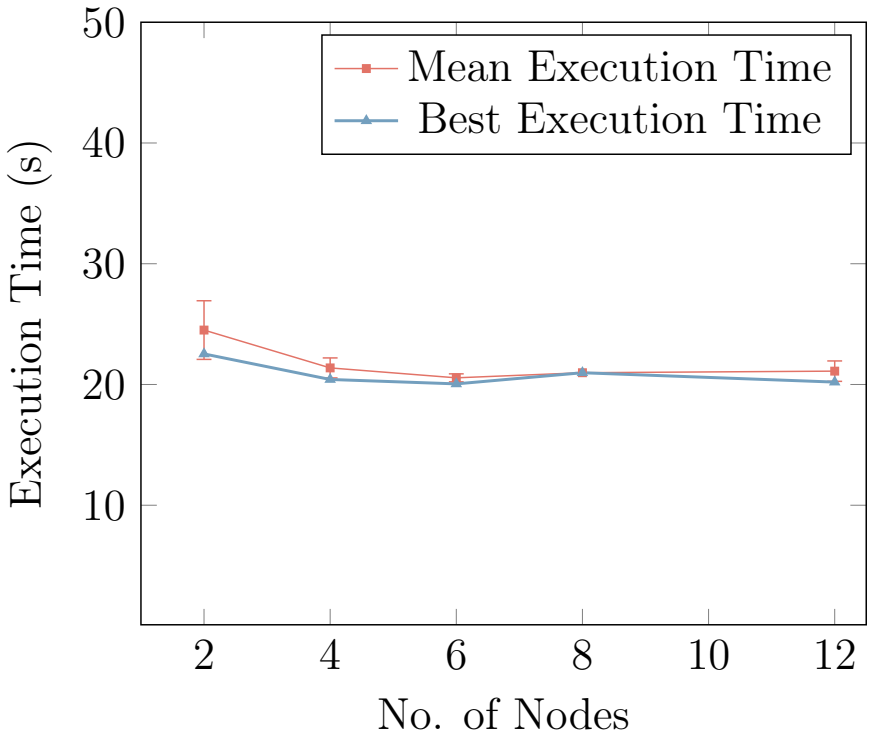

Figure 5.1 : Trapezoidal Approximation : Shows weak scaling computing approximation with 10,000,000 pieces to calculate for each worker. Number of workers per node is constant (12), and as the number of nodes increases we increase the total number of pieces to keep the amount of work on each node constant. Mean Execution time in milliseconds from 20 iterations. The error bar on mean execution time plot shows the standard deviation of the 20 iterations.

5.2.1 Hardware Setup

Our experiments are conducted on five Nexus 5 devices, each with Quad-core 2260 MHz Krait 400 processor and a Qualcomm Snapdragon 800 MSM8974 system chip, and three Nexus 4 devices, each with a Quad-core 1500 MHz Krait processor and a Qualcomm Snapdragon S4 Pro APQ8064 system chip.

5.2.2 Benchmarks

5.2.2.1 Cannon's Algorithm

Dense matrix multiplication is one of the most basic operations in scientific computations and is often at the core of many image processing algorithms. `Cannon's Algorithm` is a memory-efficient distributed matrix multiplication usually implemented on toroidal mesh interconnections [34]. It is designed for execution on a virtual $N \times N$ grid of processors, where matrices A and B are mapped onto the processors in a block-based fashion, with sub-blocks A_{ij} and B_{ij} mapped to processor p_{ij}.

The algorithm executes in two phases. In the first phase, the sub-blocks are aligned through an initial skew, where each sub-block A_{ij} is shifted left by some number of positions along the row and each sub-block B_{ij} is shifted up by some number of positions along the column. Each processor p_{ij} receives $A_{i,(j+i)modN}$ and $B_{(i+j)modN,j}$. The second phase is a series of circular shift by one processor and computation of partial results. During each step the sub-blocks are shifted one processor up or left, each processor multiplies the newly received sub-blocks and add the results to the sub-block C_{ij}, maintained at processor p_{ij}.

The loose coupling among processors and the message-passing nature of the algorithm make it a natural candidate for an actor-based implementation. Our implementation of the Cannon's algorithm uses one actor to represent one independent processor. The scaling experiment is done on one to eight phones, with a matrix size of 1680×1680. The first five devices are Nexus 5s, with three Nexus 4 devices added after that. To achieve a $N \times N$ actor grid, each device hosts the same number of actors and to the number of devices in the network (e.g., for a three device network,

each device hosts three actors). These results were obtained using the Wi-Fi Soft AP based communication layer, and only a Nexus 5 device (not a Nexus 4 device) was used as the Soft AP for all configurations.

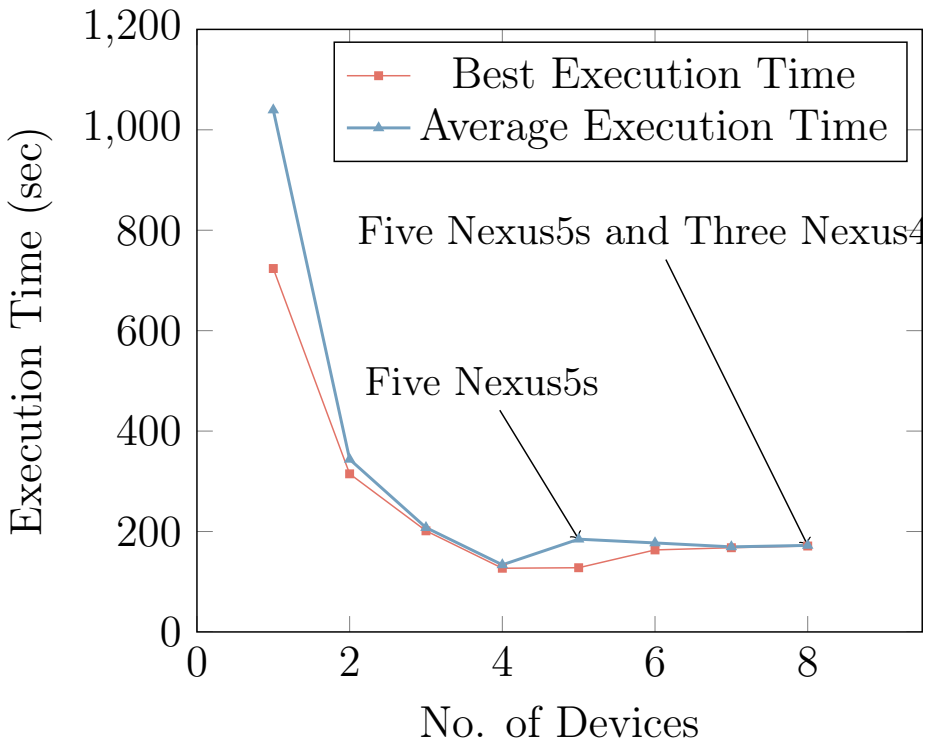

Figure 5.2 : Cannon's algorithm: Shows the best and average time (over 20 executions) to multiply two matrices of size = 1680×1680. For each experiment with N devices, each device hosts N actors (processors). The first five devices are Nexus 5s, and three Nexus 4 devices are added thereafter, as in Figure 5.3.

Figure 5.2 shows the experimental results for a matrix size of 1680×1680 on one to eight mobile devices. For each experiment with N devices, each device hosts N actors to make an $N \times N$ processor grid in the original Cannon's algorithm. For one device, there will be a single processor, making the matrix multiplication serial in effect.

We can observe that the two device experiment achieves a 4x speedup, and the four device experiment achieves a 8x speedup compared to the serial execution. This is due to the fact that multiple actors on the same device also utilize intra-device multi-core computing power. As more devices are added to the network the performance improvement diminishes, due to the increased synchronization from the quadratically increasing number of processors. Further optimization can be made with a more generalized matrix multiplication algorithm.

5.2.2.2 Pi Precision

The `Pi Precision` benchmark computes the value of Pi to a specified precision using a digit extraction algorithm. This benchmark uses a master-worker pattern with dynamic work distribution much like the `Trapezoidal Approximation` mentioned in Section 5.1.2.1. The implementation is also an adaptation from the Savina Benchmark [33]. Unlike the `Trapezoidal Approximation` the amount of communication between the master and its workers are much more frequent where the master sends more work (if available) to a worker that sends a reply with results.

Figure 5.3 shows the experimental results for calculating Pi to the 15,000 decimal place on increasing number of Android devices. For each devices there are two worker actors and the total amount of work remains constant, giving each devices less work as the number of devices increases. The experiments are done with Wi-Fi Hotspot based `IMobileCommunicationManager` implementation where a Nexus 5 devices (the master place in all experiments) also serve as a SoftAP.

The chart in Figure 5.3 starts with a single Nexus 5 device running the Pi precision approximation, and each data point shows the execution time after adding one more device. After five Nexus 5 devices are added, we add one Nexus 4 device incrementally

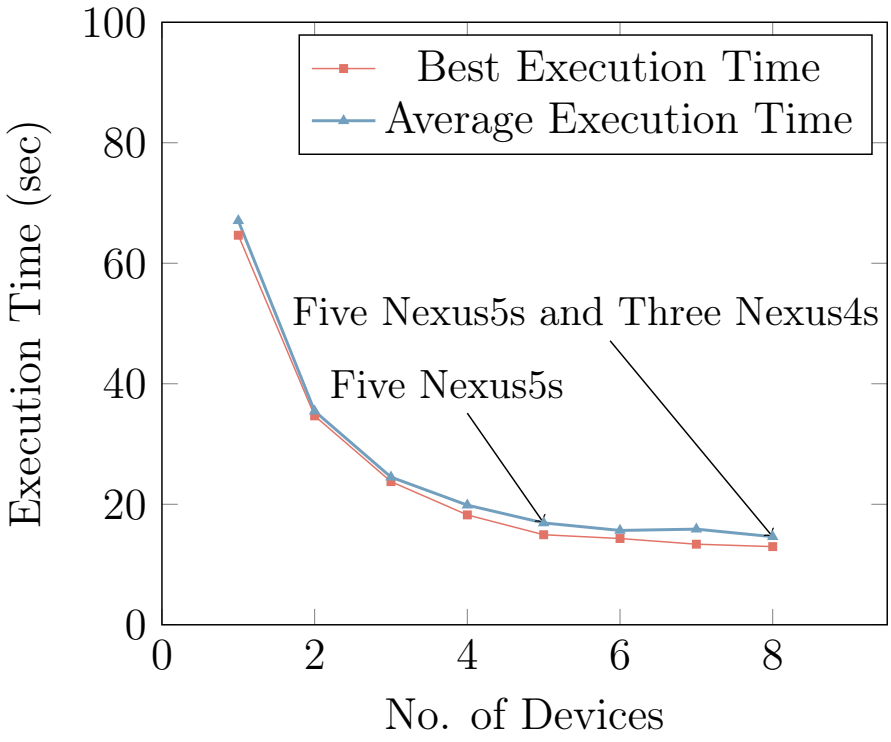

Figure 5.3 : Pi Precision Computation: Shows the best and average time (over 20 executions) to compute the value of Pi to 15,000 decimal points . The x-axis shows the number of devices used for the computation, the y-axis shows the execution time. Each device runs two worker actors, and only one device also runs a master actor. From single device to five devices the results are obtained using Nexus 5, from six to eight devices the additional devices are Nexus 4.

for each of the remaining data points. We can observe the near linear scaling effect with the first five Nexus 5 devices, with the scaling effect slowing down after that. This is due to the fact that Nexus 4 devices are only half as powerful as Nexus 5 devices, adding modest increase in computing power, while still increasing the communication traffic to the AP host device. In spite of the limited computing capability of Nexus 4s, due to the dynamic nature of the generated work and the effective load balancing

technique implemented by the application, the total execution time is still improved by adding slower Nexus 4's to the computation.

5.2.3 Thermal effect of task offloading

We conduct an experiment with the Pi Precision benchmark introduced in Section 5.2.2.2 without temperature control in a more realistic usage scenario for mobile hand held devices. While the experiment explores the thermal dissipation impact with the master-worker task offloading mechanism, we also discuss the implications on power consumption. Figure 5.4 shows the average execution time for the benchmark under room temperature without a device cooling system against the average execution time of the benchmark in temperature controlled environment. The large discrepancy in total execution time is partially attributed to the large amount of heat dissipation.

Table 5.1 shows the thermal difference on the master device for five iterations of the Pi precision benchmark, with the same configuration as Section 5.2.2.2 for each experiment. The total execution time for five iterations are recorded and the temperature difference shown in the table. The temperature data are taken at the begin and end of each experiment on the device that hosts the Soft AP and master actor, on the 11 on-chip thermal sensors in the devices. The arithmetic mean is calculated with the difference between average of 11 sensors at begin and end of the experiment, the maximum and minimum temperature difference is calculated with individual sensor data. All of the devices are fully charged and at room temperature at the beginning of each experiment.

We can observe the temperature increase goes down for the host device, as more devices are added to the network for work offloading, showing that the offloading

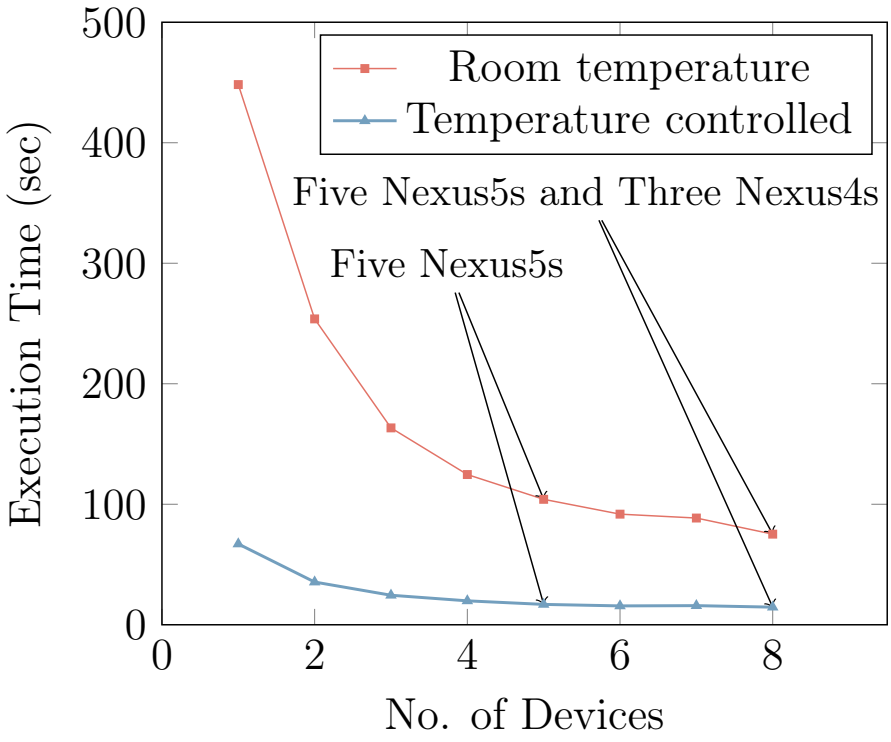

Figure 5.4 : Pi Precision Computation: Shows the best and average time (over 20 executions) to compute the value of Pi to 15,000 decimal points . The experiment under room temperature has a significantly longer execution time than in temperature controlled environment overall. The graph shows the average execution time for realistic room temperature environments and the average execution time for the same benchmark under The x-axis shows the number of devices used for the computation, the y-axis shows the execution time. Each device runs two worker actors, and only one device also runs a master actor. From single device to five devices the results are obtained using Nexus 5, from six to eight devices the additional devices are Nexus 4.

pattern reduces the heat dissipation of such intensive computation to some degree. We also observe a steadily increasing execution time for each iteration of the benchmark, as the temperature increases on the device, showing that the on-device OS thermal

Nexus 5	Nexus 4	Total Exec. Time on Host (sec)	Temperature increase ()		
			Arith. Mean	Max	Min
1	0	448.259	16.82	N/A	N/A
2	0	253.818	13.205	22.18	12.91
3	0	163.376	15.850	22.27	13.91
4	0	124.659	16.113	19.73	13.73
5	0	104.100	16.144	20.64	14.73
5	1	91.793	14.940	19.64	12.00
5	2	88.566	13.817	19.45	10.27
5	3	75.202	12.906	15.00	10.45

Table 5.1 : Pi Precision benchmark under a realistic usage scenario. Each experiment is executed with 5 iterations and total execution time is recorded on the Soft AP host device (Nexus 5). The temperature of each device is recorded at the beginning and at the end of the experiment with the 11 on-chip thermal sensors. The average temperature difference is calculated with the arithmetic average of the 11 sensors, while the best and worst temperature difference is calculated based on the individual sensor data.

management may be limiting the processor frequency. As [35] discussed, the thermal behavior of the mobile SoC demonstrates complex behavior affected by both the application processor and the battery. The power consumption on mobile devices, in

reverse, can also be heavily impacted by the thermal dissipation [36]. As multiple scheduling and power management techniques are developed with the thermal limit in consideration, the effort concentrates on thermal dissipation control through software based task scheduling [37], and architecture based improvements [38].

Our experiment shows a new possibility for thermal aware applications in user software by offloading computationally intensive tasks to nearby mobile devices with a small communication cost.

Chapter 6

Related Work

The Actor Model was a large influence in distributed software design ever since its introduction. With the powerful abstraction of asynchronous processing logic and its advantages in structuring highly concurrent complex systems with inherent scalability, the Actor Model has been a go-to option for building large-scale distributed applications and platforms for decades. The Habanero Java Runtime Library, the Scala language[13] and the Akka event-driven middleware[16] are all known long-term efforts to bring the actor model to main-stream shared-memory parallel programming. In recent years, Actors have been introduced to distributed computing by multiple efforts including the Virtual Actors by Microsoft Orleans project[39](Section 6.1.2), the Akka cluster Module[40](Section 6.1.1), and mobile-platform based efforts such as the AmbientTalk language[41] (Section 6.2.1) and the ActorNet[42] project for wireless sensor networks (Section 6.2.2).

6.1 Actor-based distribution on clusters

6.1.1 Akka Cluster

Akka is an open-source toolkit and runtime using the *Actor Model* and at its core for building highly concurrent, fault-tolerant and distributed systems on the JVM, available for both Scala and Java. Akka is known as one of the first attempts to bring actors to mainstream programming languages, and have been one of the most widely

used actor-based frameworks.

The Akka runtime consist of a system of actors and requires users to explicitly terminate each subsystems. For distributed application, Akka provides the Akka.cluster module, which is dedicated to enabling actor-based distributed programming and achieves location transparency with a strict adaptation of the Actor Model [40]. Akka actors also partially support priority-enabled mailbox by allowing the Akka prioritized mailbox to associate with a specific message class or value to a predefined priority. The priority processing of messages is limited on generic-typed messages with specific binding to an actor. While still maintaining a single mailbox, Akka runtime effectively changes the order of received messages in an actor's mailbox based on predefined priorities and have been shown to be have some inefficiencies. In our selector model, it is more flexible to implement message prioritization and the model permits applications to pass the same message type to different mailboxes with different priorities, rid of any hard-defined association between the mailboxes and the messages types they hold. The distributed selector model, therefore, reduces the complexity to implement sophisticated synchronization and interaction patterns between actors [5].

6.1.2 Microsoft Orleans

Project "Orleans" is an open-source .NET framework built at Microsoft Research that specifically targets actor-based distributed applications. The project was initialled developed to aid the development of streaming applications focusing on high scalability and low latency[39].

Project "Orleans" utilizes the Actor Model, and introduces the *Virtual Actor abstraction*. While adapting a traditional approach with actor life-cycles and sequential processing for actor instances, it introduces Actors as virtual entities, rather than

physical ones. An Orleans actor *always* exists virtually. It cannot be explicitly instantiated or destroyed, and transcends the lifetime of any particular server or host.

An Orleans application automatically instantiate actors, while the programmer access a virtual abstraction of the actor, without explicit knowledge of the number of instances available. The instantiation of an actor can be triggered by a message sent to it, and an unused instance is automatically reclaimed as part of the runtime resource management. An actor thus never fails in this sense: a message to an actor on a failed server only triggers the instantiation on another. By using such indirection, the runtime addresses actor placement, load balancing, migration that must be otherwise explicitly handled by developers. The indirection is supported by a distributed directory mapping from virtual actor to its instances.

The Orleans API is used extensively in the Halo franchise on Microsoft Azure since 2011, and is provided as an Actor library in the Microsoft Azure Service Fabric as Service Fabric Reliable Actors[43].

6.2 Actor-based projects on mobile platform

6.2.1 AmbientTalk

The AmbientTalk language, an actor-based programming language, is specifically designed to run on `mobile ad hoc networks` [44]. Its grammar inherits from Erlang and features distributed implementation of λ calculus based functional elements and reduction computations, and object-oriented elements including isolated objects with pass-by-value semantics and regular objects with pass-by-reference semantics. The AmbientTalk language bases its semantics for concurrent and distributed programming soley on the `Actor Model`.

In the AmbientTalk semantics, actors are used as containers of a set of regular objects. Each VM instance hosts multiple number of actors running concurrently, where each actor is treated as an event loop that uses the *run-to-completion* semantics to invoke methods on its hosted objects. The AmbientTalk actor model is mostly compatible with the classic actor model, though it allows the use of *far reference* (inherited from the E language [45]), which can break the pure message-passing semantics in traditional actors.

The AmbientTalk language provides a cluster-based implementation and a preliminary Android-based implementations that focus on high-level abstractions for distributed programming based on distributed reduction through actors and distributed method invocation through far references. Our work on distributed selectors focus on supporting a pure actor/selector model at the high-level, with distributed mechanisms supported in configurations and decoupled communication middleware and separated from the programming application logic. By enabling multiple guarded mailboxes the distributed selector can achieve elegant distributed reduction implementation through master-worker paradigm without limiting to it as the single distribution pattern. Further, our model does not limit ourselves to *mobile ad-hoc networks* as the distributed runtime design supports communications within and across mobile devices and server devices.

6.2.2 ActorNet

The ActorNet project implements an actor-based mobile agent platform for wireless sensor networks (WSNs) based on the `Scheme` language [42]. ActorNet aims to use high-level abstractions for concurrent and asynchronous programming on WSNs specifically designed for the limited hardware resources on each mobile sensor. It im-

plements a `Scheme` interpreter optimized for wireless sensors to maximally utilize the limited processing power and memory available on-chip. For its version of distributed actor-based programming, ActorNet introduces new language primitives for actor message passing, queries and program continuation access. Comparing to the specific emphasis of ActorNet on optimization for limited hardware resources on WSNs, the work in HJDS and DAMMP addresses the portability and versatility of mobile applications on a wider range of platforms through a more generalized model. The HJDS and DAMMP design goes beyond mobile devices with limited hardware resources, and aim to support versatile combinations of both mobile-based and cluster-based networks. It should be noted that the distributed selector work is intended for modern day powerful consumer devices such as tablets and smartphones and can not fulfill the specific needs on distributed actor-based applications on WSNs.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation we address the communication and location transparency challenges in HJDS in both cluster-based and mobile-based designs by providing a fully decoupled, readily replaceable communication middleware and introduce a lightweight Selector reference with global unique identifier that encodes its location.

We have developed the cluster-based HJDS runtime library that allows programmers to focus on implementing the algorithm for solving the problem their application is trying to solve, without worrying whether their application will run on a shared-memory or distributed-memory system. Our runtime implementation supports Selectors (a strictly more powerful version of Actors) on both shared-memory and distributed-memory systems. This framework provides automated system bootstrap and global termination, unlike any other distributed approaches. Our experimental evaluation using the Savina benchmark suite shows promising strong scaling results, making a strong case for the DS model as a viable alternative to the existing, much harder to program and port, parallel programming models.

We have developed a mobile platform based Java runtime library (DAMMP) by extending the HJDS implementation. Our work focuses on decentralized distributed applications using the actor/selector model by supporting a highly decoupled and customizable communication middleware and publish-subscribe enabled application-

level runtime event handling. We provide a hierarchical, heterogeneous concurrency and distribution model by extending the actor model in HJDS for shared-memory and distributed parallelism. We presented a task offloading pattern based on the selector model and the Master-Worker paradigm.

We demonstrated the scalability of computationally intensive applications using distributed mobile platforms, examined the message passing overheads with two promising off-the-grid wireless communication technologies, and showed decreased thermal dissipation while maintaining scalability for compute-intensive applications in a realistic ad-hoc mobile network environment. Our empirical results expose some of the limitations of the current state-of-the-art in device-to-device wireless connectivity This work also provides network researchers an intuitive and easy to use platform for connectivity experiments.

7.2 Future Work

In this section we describe the future works to be explored based on the next stage of the distributed selector-based runtime adaption of the Habanero Java Runtime Library.

7.2.1 Selector Migration

In the current HJDS design, selectors are created either locally or remotely through a selector creation message and remains on the same place throughout its lifetime. Many actor-based distributed system, on the other hand, introduces the ability to migrate selector instances from node to node enabled by providing a global naming service[32][40]. Migration of selector instances provides further benefits including aids on software resilience and failure recovery design and allowing dynamic load-

balancing for system task scheduling. The HJDS does not provide a separate global naming services like most actor-based distributed design, but encodes the location information in the selector reference (SelectorHandle) as a GUID. To enable selector instance migration and reserve the location-transparent selector semantics in HJDS, we can utilize the fact that a physical node can host more than one place and extend the definition of a *place* to allow a single place to be "virtually hosted" on multiple physical places.

As described in Section 3.1.2, a selector GUID keeps the place Id which requested the creation of the selector, the place Id on which the selector instance resides, and a unique Id for the selector instance on the creation place. To migrate a selector instance from its residing place p to a different place q, place q sends a remote creation message to place q giving the selector handle as described in Section 3.1.3, where the selector handle's residence place id is changed from p to q. The selector GUID will be able to remain unique because the combination creation place id and an unique identifier on the creation place guarantees the global uniqueness of selector GUID. Place p, after sending the selector creation message, disables all mailboxes on the current selector to stop message processing and changes the selector instance in its local registry to map to the new selector handle, therefore forwarding all messages to this selector directly to place q. Place p, after receiving confirmation on the new selector creation on place q, sends the selector's mailboxes over to place q, which associates the mailboxes to the new selector instance.

For this initial design, further mechanism need to be developed to reserve the selector messaging semantics where the message sequence between the same pair of sender and receiver are sorted in the message sending order, and since the old residence place will be responsible for a lot of message forwarding when messages are sent to

an old selector handle with GUID before migration, a better mechanism needs to be developed for propagating the selector migration to all existing selector handle instances.

7.2.2 Distributed extensions of the Habanero Exectuion Model

The shared-memory HJLib (discussed in Section 2.2 implementation includes multiple high-level parallel abstractions from the Habanero Execution Model [6]. It implements the Async-Finish model, in which `async` represents a general primitive for creating asynchronous computation and data transfer tasks. HJLib APIs include `async`, `forasync`, `asyncAt`, `asyncPut`, `asyncGet`, and `asyncAwait`, and `finish` as general primitives for creating and awaiting the completion of asynchronous computation and data transfer tasks. These Async-Finish primitives enable any (block) statement to be executed as a parallel task, including for-loop iterations and method calls. The shared-memory HJLib also supports generalized barriers and point-to-point synchronization with phasers [7], mutual exclusion synchronization within tasks using delegated isolation [17] and object-based isolation [18]. The rich concurrency and parallelism abstractions can be used to obtain programmability, parallelism, and scalability benefits across a wide range of task-level parallel algorithms.HJLib implements a priority-based lock-free work-stealing algorithm [19] with multiple thread pools to support priority scheduling of tasks and integrates with the Actor Model [46].

Our work in HJDS lays the foundation for distributed extensions of these task-parallel abstractions by coordinating tasks through anonymous selector instances across physical nodes and multiple places. The first few task-parallel concurrent abstractions we plan to implement are data driven futures[15] and phasers[7], both fine-grained synchronization primitives that can be implemented with the synchro-

nization and join patterns in the Selector Model.

7.2.3 Thermal aware task offloading on heterogeneous devices

The DAMMP design is created with the intention of mitigating the hardware limitations of hand-held mobile devices in mind. As battery capacities remain as a key physical constraint for mobile devices, energy efficiency is an important software design consideration. While distributed programming abstractions are becoming a prevalent integrated part of mobile software platforms, the energy consumption characteristics are not well understood yet often is one of the major issues concerning distributed mobile applications. Thermal emission often pose the most threat to battery over-consumption and is especially common in distributed applications due to intensive usage of the wireless modules. Thermal emission both causes faster battery drain and can trigger cpu thermal-throttling that causes inefficient computation. Yet, developers in many cases have to rely on trial-and-error for coordinating data exchange and computation on such limited resources.

The effort of HJDS has provided a flexible and expressive high-level abstraction for efficient distributed software development, and with the Android-based design in DAMMP providing readily replaceable communication middleware we believe there is a potential to enable research on thermal aware distributed software design on mobile platform with much faster turnaround time. The Android-based implementation allows testing with most main-stream devices and wireless modules and the readily replaceable communication middleware gives software portability across different hardware.

As shown in Section 5.2.3 distribution on multiple mobile devices can shown observable reduction on thermal emission for each device, we'd like to continue this

study further to see the effect of computation offloading to heterogeneous devices. Since the communication middleware is fully decoupled in both cluster-based and Android-based implementation for HJDS and the runtime system control remain the same for proxy actor and system actor with exception for global bootstrap and termination, the cluster-based system would be able to seamlessly interact with the mobile version given manual bootstrap and termination. It would be interesting to research the impact on thermal emission and battery drain when tasks are offloaded to nearby desktops or tablets with more computational capabilities, and require less data exchange resulted from multi-device coordination as well as less wait time on the master device. Further evaluation on different wireless technologies can also provide beneficial heuristics for mobile development.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[2] "Apache thrift."

[3] "Apache spark."

[4] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[5] S. M. Imam and V. Sarkar, "Selectors: Actors with multiple guarded mailboxes," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents and Decentralized Control*, AGERE! '14, (New York, NY, USA), pp. 1–14, ACM, 2014.

[6] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: The new adventures of old x10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, (New York, NY, USA), pp. 51–61, ACM, 2011.

[7] J. Shirako and V. Sarkar, "Hierarchical phasers for scalable synchronization and reductions in dynamic parallelism," in *in 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[8] A. Chatterjee, B. Gvoka, B. Xue, Z. Budimlic, S. Imam, and V. Sarkar, "A Distributed Selectors Runtime System for Java Applications," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, (New York, NY, USA), pp. 3:1–3:11, ACM, 2016.

[9] A. Chatterjee, "Enabling distributed reconfiguration in an actor model," April 2017.

[10] C. Hewitt, P. Bishop, and R. Steiger, "Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence." Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Stanford, CA.

[11] N. Raja and R. K. Shyamasundar, "Actors as a Coordinating Model of Computation," in *Proceedings of the 2nd International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pp. 191–202, Springer-Verlag, 2004.

[12] P. Haller and F. Sommers, *Actors in Scala*. USA: Artima Incorporation, 2012.

[13] "The Scala Programming Language." Home page.

[14] T. Desell and C. A. Varela, "A Performance and Scalability Analysis of Actor Message Passing and Migration in SALSA Lite," in *Agere Workshop at ACM SPLASH 2015 Conference*, 2015.

[15] S. Imam and V. Sarkar, "Habanero-java library: A java 8 framework for multicore programming," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, (New York, NY, USA), pp. 75–86, ACM, 2014.

[16] Typesafe Inc., "Akka framework."

[17] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar, "Delegated isolation," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, (New York, NY, USA), pp. 885–902, ACM, 2011.

[18] S. Imam, J. Zhao, and V. Sarkar, "A composable deadlock-free approach to object-based isolation," in *Euro-Par 2015: Parallel Processing* (J. L. TrÃd'ff, S. Hunold, and F. Versaci, eds.), vol. 9233 of *Lecture Notes in Computer Science*, pp. 426–437, Springer, 2015.

[19] S. Imam and V. Sarkar, "Habanero-java library: A java 8 framework for multicore programming," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pp. 75–86, ACM, 2014.

[20] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano, "Device-to-device communications with wi-fi direct: overview and experimentation," *IEEE Wireless Communications*, vol. 20, pp. 96–104, June 2013.

[21] K. Jahed, O. Farhat, G. Al-Jurdi, and S. Sharafeddine, "Optimized group owner selection in wifi direct networks," in *2016 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 1–5, Sept 2016.

[22] "Wi-Fi Direct | Wi-Fi Alliance." Available at http://www.wi-fi.org/discover-wi-fi/wi-fi-direct.

[23] J. S. Lee, Y. W. Su, and C. C. Shen, "A comparative study of wireless protocols: Bluetooth, uwb, zigbee, and wi-fi," in *IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society*, pp. 46–51, Nov 2007.

[24] "Wi-Fi Peer-to-Peer | Android Developers." Avaialble at https://developer.android.com/guide/topics/connectivity/wifip2p.html.

[25] Esoteric Software, "Kryo : Graph serialization framework for Java ," 2015. [latest commit 31-Oct-2015].

[26] Esoteric Software, "Kryo: V1 Benchmarks," 2012. [Online; accessed 3-April-2012].

[27] G. Inc., "Protocol buffer."

[28] "Apache avro."

[29] Eishay Smith, "Object graph serializers - performance evaluation ," 2015. [Online; accessed 11-Aug-2015].

[30] "Netty project."

[31] J. Ayres and S. Eisenbach, "Stage: Python with Actors," in *Proceedings of IWMSE '09*, (Washington, DC, USA), pp. 25–32, IEEE Computer Society, 2009.

[32] C. Varela and G. Agha, "Programming Dynamically Reconfigurable Open Systems with SALSA," *ACM SIGPLAN Notices*, vol. 36, pp. 20–34, Dec. 2001.

[33] S. Imam and V. Sarkar, "Savina - An Actor Benchmark Suite," in *Proceedings of the 4th International Workshop on Programming based on Actors, Agents, and Decentralized Control*, AGERE! 2014, October 2014.

[34] H. Gupta and P. Sadayappan, "Communication efficient matrix-multiplication on hypercubes.," Technical Report 1994-25, Stanford Infolab, 1994.

[35] Q. Xie, J. Kim, Y. Wang, D. Shin, N. Chang, and M. Pedram, "Dynamic thermal management in mobile devices considering the thermal coupling between battery and application processor," in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, (Piscataway, NJ, USA), pp. 242–247, IEEE Press, 2013.

[36] U. Y. Ogras, R. Z. Ayoub, M. Kishinevsky, and D. Kadjo, "Managing mobile platform power," in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, (Piscataway, NJ, USA), pp. 161–162, IEEE Press, 2013.

[37] T. Chantem, R. P. Dick, and X. S. Hu, "Temperature-aware scheduling and assignment for hard real-time applications on mpsocs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, (New York, NY, USA), pp. 288–293, ACM, 2008.

[38] O. Khan and S. Kundu, "Hardware/software co-design architecture for thermal management of chip multiprocessors," in *2009 Design, Automation Test in Europe Conference Exhibition*, pp. 952–957, April 2009.

[39] Microsoft Research, "Microsoft Orleans," 2015.

[40] Typesafe Inc., "Akka Cluster Documentation," 2014.

[41] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-oriented programming," in *Companion to the 20th Annual ACM SIG-*

PLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, (New York, NY, USA), pp. 31–40, ACM, 2005.

[42] Y. Kwon, S. Sundresh, K. Mechitov, and G. Agha, "Actornet: An actor platform for wireless sensor networks," in *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '06, (New York, NY, USA), pp. 1297–1300, ACM, 2006.

[43] Microsoft, "Service Fabric Reliable Actors," 2017.

[44] T. Van, C. Christophe, S. D. Harnie, and W. D. Meuter, "An operational semantics of event loop concurrency in ambienttalk."

[45] M. S. Miller, E. D. Tribble, and J. Shapiro, *Concurrency Among Strangers*, pp. 195–229. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[46] S. Imam and V. Sarkar, "Integrating Task Parallelism with Actors," in *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 753–772, ACM, 2012.